This table attempts to cover the semantics of each element (class, attribute and property) in the CAMEL meta-model. Apart from this, it also supplies the concrete syntax of each element by using a particular symbolism. This symbolism is explained at the end of this document. A more detailed inspection of the concrete syntax of CAMEL can be performed over the `CamelDsl.xtext` file in the `eu.paasage.camel.dsl` directory of the CAMEL distribution. For a more structured inspection of the language semantics, the CAMEL meta-model definition itself can be found at the camel.ecore file in the `eu.paasage.camel.dsl` directory of the CAMEL distribution.

Table 1. The CAMEL language semantics

Class Name	Attribute / Property Name	Concrete Syntax Element	Semantics
CAMEL Meta-Model			
`Model`			An abstract class representing any kind of model
	name		The name of the model
	importURI		The URI of a model to import. This attribute has been introduced due to a technical requirement of the concrete textual editor of CAMEL.

`CamelModel`		`camel model` *name* `{`
		inner def ...
		`}`
			A concrete subclass of *Model*. It represents the overall CAMEL model acting as a container for all the other kinds of models that can be specified.

	actions	(action def ...)*	Containment reference to a set of actions
	applications	(application def ...)*	Containment reference to a set of applications modelled by the user
	deploymentModels	(deployment model def ...)*	Containment reference to all deployment models defined in this CAMEL model
	executionModels	(execution model def ...)*	Containment reference to all execution models defined in this CAMEL model
	locationModels	(location model def ...)*	Containment reference to all location models defined in this CAMEL model
	metricModels	(metric model def ...)*	Containment reference to all metric models defined in this CAMEL model
	organisationModels	(organisation model def ...)*	Containment reference to all organisation models defined in this CAMEL model
	providerModels	(provider model def ...)*	Containment reference to all provider models defined in this CAMEL model
	requirementModels	(requirement model def ...)*	Containment reference to all requirement models defined in this CAMEL model
	scalabilityModels	(scalability model def ...)*	Containment reference to all scalability models defined in this CAMEL model
	securityModels	(security model def ...)*	Containment reference to all security models defined in this CAMEL model
	typeModels	(type model def ...)*	Containment reference to all type models defined in this CAMEL model

	unitModels	(unit model def ...)*	Containment reference to all unit models defined in this CAMEL model

Action		action *name* {
		inner def ...
		}
			A class representing any action that can be defined by the user. Actions can be used for the modelling of scalability rules but also in order to define access control policies in organisation models.

	name		The name of the action
	type	type: *ActionType*	The type of the action taking values from the *ActionType* enumeration
ActionType		EVENT CREATION or SCALE DOWN or SCALE IN or SCALE OUT or SCALE UP	An enumeration listing all possible concrete actions that can be defined by the user. Current content mainly reflects scaling and access control actions.

Application		application *name* {
		inner def ...
		}
			A class representing an application of the user. A CAMEL model can include the specification of multiple applications for the same user.

	name		The name of the application that should be unique.
	version	version: 'string'	The version of the application. This along with the name attribute constitute a unique identifier that characterises the current state of an application. In principle, in the same CAMEL model, the same application can be associated to different versions. This is modelled by specifying as many *Application* instances as the number of the application versions. In this kind of modelling, then other CAMEL models can reference exactly the version of the application that they concern. For instance, a deployment model can refer to a specific version of an application which might also include even different components from a previous version of that application.
	description	description: 'string'	A textual description of the application which could cover, e.g., its overall functionality and goals.
	owner	owner: entityRef	A reference to the owner of the application that can be an *Entity*. Entities can be either organisations or users (defined in an organisation model).
	deploymentModels	deploymentModels: [depModelRef(,depModelRef)*]	A reference to a set of deployment models that have been defined for the application (and its corresponding version).
LayerType		BPM or SCC or SaaS or PaaS or IaaS	An enumeration that list the layers on which an application can reside. Such a list can be used to associate specific CAMEL elements to a certain level. For example, a metric can be

			mapped to a certain layer, e.g., IaaS, to denote that is used to measure application objects/components (in the overall application dependency hierarchy) at that layer. The list includes the typical cloud layers as well as higher-level ones mapping to service compositions and business processes.
Deployment Meta-Model			
DeploymentElement			An abstract class representing any deployment element in the deployment meta-model.
	name		The name of the deployment element

DeploymentModel		deployment model *name* {
		inner def ...
		}
			A concrete subclass of *Model*. It represents the model of the application deployment acting as a container of different deployment elements

	internalComponents	(internal component def ...)*	A containment reference to all *InternalComponent*s defined in this model
	internalComponentInstances	(internal component instance def ...)*	A containment reference to all *InternalComponentInstance*s defined in this model
	vms	(vm def ...)*	A containment reference to all *VM*s defined in this model
	vmInstances	(vm instance def ...)*	A containment reference to all *VMInstance*s defined in this model
	communications	(communication def ...)*	A containment reference to all *Communication*s defined in this model
	communicationInstances	(communication instance def ...)*	A containment reference to all *CommunicationInstance*s defined in this model
	hostings	(hosting def ...)*	A containment reference to all *Hosting*s defined in this model
	hostingInstances	(hosting instance def ...)*	A containment reference to all *HostingInstances* defined in this model
	vmRequirementSets	(vm requirement set def ...)*	A containment reference to all *VMRequirementSet*s defined in this model
	globalVMRequirementSet	global vmRequirementSetRef	A reference to a global VMRequirement set, i.e., a set of requirements that holds for all VMs defined in this model. This set should be included in the *vmRequirementSets* containment reference list.
Component			A class that represents any component, either internal or external to the user application. Internal components are usually normal application components while external components are usually VMs.
	providedCommunications	(provided communication	A containment reference to all

		port def ...)*	communication ports provided by this component.
	providedHosts	(provided host port def ...)*	A containment reference to all hosting ports provided by this component.
	configurations	(configuration def ...)*	A containment reference to all configurations that can be applied for that component to handle its lifecycle management.

InternalComponent		internal component *name* {
		inner def ...
		}
			This class represents an internal component of a user application. Such a component usually maps to a software artefact that has to be downloaded, installed and executed as well as deployed in a certain VM. To this end, to assist in its lifecycle management, a configuration can be defined for that component (inherited from the *configurations* containment reference from the Component parent class).

	compositeInternalComponents		A containment reference to composing internal components for this internal component. This can enable to define a hierarchy of internal application components.
	requiredCommunications	(required communication port def ...)*	A containment reference to required communication ports expressing an actual requirement to communicate with other components via these ports.
	requiredHost	(required host port def ...)*	A containment reference to a required hosting port expressing a requirement for being hosted via this port by another component (e.g., servlet container or VM).

VM		vm *name* {
		inner def ...
		}
			A class representing a VM as a container of user (deployment) requirements. A mapping of an internal component to a VM via a *Hosting* indicates that the component is coupled with such deployment requirements. Such requirements can then be exploited to discover the actual VM offering to be instantiated to host this internal component.

	vmRequirementSet	requirement set vmRequirementSetRef	A reference to a VMRequirementSet, i.e., a set of VM requirements spanning quantitative or qualitative hardware, OS or image, location and provider requirements. This reference might not be provided, depending on whether the reference to a global VM requirement set has been given for the deployment model containing this VM. Such global VM requirement set can be complete enough such that there is no need to specify an additional VM requirement set at the local level of a VM. Global and local VM requirement sets can play a complementary role: i.e., global sets can

			define (partial) global requirements that should hold for all VMs defined in the current deployment model while local sets define (partial) requirements specific to a certain VM, where partial means that not all kinds of requirements are specified in such sets. When a global and a local VM requirement set is overlapping (i.e., there is at least one common kind of requirement being specified in them), then the semantics should be that the local VM requirement set takes priority over the global one. However, such a modelling should be avoided as it can potentially lead to logical errors.

VMRequirementSet		requirement set *name* {
		inner def ...
		}
			This class represents a set of VM requirements of different kinds, including quantitative/qualitative hardware, OS/Image, location and provider requirements. At least one kind of requirements should be specified for this set; otherwise, it is not meaningful to model it.

	locationRequirement	location: locationRequirementRef	A reference to a location requirement that is specified in a certain requirement model of the overall CAMEL model. Via this requirement, we can restrain the placement of a VM (local scope) or sets of VMs (global scope) to a one or more geographical or cloud-specific locations.
	providerRequirement		A reference to a provider requirement that is specified in a certain requirement model of the overall CAMEL model. Via this requirement, we can restrain the placement of a VM (local scope) or sets of VMs (global scope) to occur only in the cloud(s) of one or more specific cloud providers.
	qualitativeHardwareRequirement	qualitative hardware: qualitativeHardwareRequirementRef	A reference to a qualitative hardware requirement that is specified in a certain requirement model of the overall CAMEL model. Via this requirement, we can restrain a VM (local scope) or sets of VMs (global scope) to be instantiated only based on VM offerings that match this requirement.
	quantitativeHardwareRequirement	quantitative hardware: quantitativeHardwareRequirementRef	A reference to a quantitative hardware requirement that is specified in a certain requirement model of the overall CAMEL model. Via this requirement, we can restrain a VM (local scope) or sets of VMs (global scope) to be instantiated only based on VM offerings that match this requirement.
	osOrImageRequirement	os: osRequirementRef or image: imageRequirementRef	A reference to an OS or Image requirement that is specified in a certain requirement model of the overall CAMEL model. Via this

			requirement, we can restrain a VM (local scope) or sets of VMs (global scope) to have a specific OS or be instantiated based on a particular image. Such a requirement can also be used to filter the VM offering / provider space based on the OS and/or images that can be supported by the cloud providers modelled.

`Configuration`		`configuration` *`name`* `{`
		`  inner def ...`
		`}`
			This class represents a configuration element, i.e., a container of configuration information that can be used to handle the lifecycle of an application (internal) component. Currently, each lifecycle activity covered maps to a certain OS-specific command that has to be executed in order to realise this activity. This means that, for instance, the download of the binaries (or source code) of an application component can be supported via the execution of a *downloadCommand*). Please note that depending on the particular situation, not all different commands kinds need to be provided. For instance, a database component could start running immediately after its installation, such that a *startCommand* does not need to be provided for it.

	downloadCommand	`download: 'string'`	An OS-specific command, represented as a String, that can be executed to download the code (binaries or source code) of an internal component.
	uploadCommand	`upload: 'string'`	An OS-specific command, represented as a String, that can be executed to upload the code (binaries or source code) of an internal component.
	installCommand	`install: 'string'`	An OS-specific command, represented as a String, that can be executed to install an internal component in a corresponding VM.
	configureCommand	`configure: 'string'`	An OS-specific command, represented as a String, that can be executed to configure the internal component before executing/starting it.
	startCommand	`start: 'string'`	An OS-specific command, represented as a String, that can be executed to initiate the running of an internal component.
	stopCommand	`stop: 'string'`	An OS-specific command, represented as a String, that can be executed to stop an internal component from executing.

`Communication`		`communication` *`name`* `{`
		`  inner def ...`
		`}`
			A class representing a communication between two internal components. The communication is established by connecting

			the provided and required communication ports of the components to be connected. The provided and required communication ports can also be configured to better support the establishment & management of the respective communication by associating them to a certain *Configuration*.
	type	`type: CommunicationType`	An attribute that characterises the type of the communication taking values from the *CommunicationType* enumeration. If the type is LOCAL, then the two components should be placed in the same VM. If the type is REMOTE, then the two component should always be placed in a different VM. Finally, if the type is ANY, then there is flexibility in the placement of the communicating components as both placement/deployment options are possible (mapping to the previous two communication type values).
	providedCommunication	`to` providedCommunicationPortRef	A reference to the provided communication port of one component from the communicating components pair.
	requiredCommunication	`from` requiredCommunicationPortRef	A reference to the required communication port of one component from the communicating components pair.
	providedPortConfiguration	`provided port` configuration def ...	A containment reference to a *Configuration* of the provided communication port to support the lifecycle management of the communication.
	requiredPortConfiguration	`required port` configuration def ...	A containment reference to a *Configuration* of the required communication port to support the lifecycle management of the communication.
`CommunicationType`		`ANY or LOCAL or REMOTE`	An enumeration which specifies the kind of communication that can be established between two internal components. The semantics of the enumeration members have been explained in the *type* attribute of the *Communication* class.
`CommunicationPort`			A class representing the port of a communication. A port can be either provided or required for a certain component.
	portNumber	`port:` 'integer'	The number of the port as an integer attribute

`ProvidedCommunication`		`provided communication` *`name`* `{`
		inner def ...
		`}`
			A sub-class of *CommunicationPort* which represents a provided communication port.

`RequiredCommunicati`		`required communication`	A sub-class of *CommunicationPort* which

on		*name* {
		inner def ...
		}
			represents a required communication port.

Hosting		hosting *name* {
		inner def ...
		}
			A class that represents the hosting of one internal component by another component which can be either internal (e.g., servlet container) or a VM. A hosting is established by connecting the provided hosting port of the latter component with the required hosting port of the former component (i.e., the internal one).

	providedHost	to providedHostPortRef	A reference to the provided host of the hosting component.
	requiredHost	from requiredHostPortRef	A reference to the required host of the internal component to be hosted.
	providedHostConfiguration		A containment reference to the *Configuration* of the provided host to support the lifecycle management of the *Hosting*.
	requiredHostConfiguration		A containment reference to the *Configuration* of the required host to support the lifecycle management of the *Hosting*.
HostingPort			A class that represents a hosting port. Such a port can be required or provided by a certain component. In contrast to the *CommunicationPort*, this class does not need to be associated with a port number as hosting ports can be considered as virtual (thus not mapping to real ports).
ProvidedHost		provided host *name*	A sub-class of *HostingPort* which denotes a hosting port provided by a component.
RequiredHost		required host *name*	A sub-class of *HostingPort* which denotes a hosting port required by an internal component.
ComponentInstance			This class represents an instance of a component (by following the type-instance pattern based on the models@runtime approach). Such an instance signifies a real instance of a component and not an abstract specification of such a component (which is represented by the instance type, i.e., the component itself).
	type		A reference to the type of the component instance, i.e., a certain *Component*.
	providedCommunicationInstances	(provided communication port instance def ...)*	A containment reference to all provided communication port instances of this component instance.
	providedHostInstances	(provided host port instance def ...)*	A containment reference to all provided hosting port instances of this component instance.

	instantiatedOn		A date attribute which explicates when the component has been instantiated.
	destroyedOn		A date attribute when the component instance has been destroyed.

`InternalComponentInstance`		`internal component instance` *name* `typed` internalComponentRef `{`
		inner def ...
		`}`
			This class represents an instance of an internal component. Similarly to the case of an internal component, this instance is associated to instances of required communication and hosting ports of its type.

	requiredCommunicationInstances	`(`required communication port instance def ...`)*`	A containment reference to the instances of required communication ports of this internal component instance.
	requiredHostingInstances	`(`required host port instance def ...`)*`	A containment reference to the instances of required hosting ports of this internal component instance.

`VMInstance`		`vminstance` *name* `typed` vmRef `{`
		inner def ...
		`}`
			An instance of a particular VM. This instance is associated via its properties to a particular VM flavour of the provider model of the selected cloud provider. This means that this element is a real instance of this VM flavour deployed in the cloud of the selected cloud provider.

	vmType	`vm type:` featureRef	A reference to an *Attribute* of the VM *Feature* of the selected cloud provider's model which represents a VM flavour/type characteristic (i.e., a certain provider-specific classification of the different kinds of VMs offered).
	vmTypeValue	`vm type value:` attributeRef	A reference to the value of the VM flavour attribute which corresponds to the concrete VM flavour/offering, supplied by the selected cloud provider, which has been instantiated.
	ip	`ip:` 'string'	This attribute represents the IP of the VM instance that has been generated.
`CommunicationInstance`		`connect` requiredCommunicationPortInstanceRef `to` providedCommunicationPortInstanceRef `typed` communicationRef	This class represents an instance of a communication between two components, i.e., the actual communication between two instances of these components.
	type		A reference to the type of this communication instance, i.e., the actual *Communication*.
	providedCommunicationInstance		A reference to the communication port instance provided by one component instance in the communicating component instance pair of this communication instance.
	requiredCommunicationInstance		A reference to the communication port instance required by one component

			instance in the communicating component instance pair of this communication instance.
`CommunicationPortInstance`			A class that represents an instance of a *CommunicationPort*.
	type		A reference to the type of the communication port instance, i.e., the actual *CommunicationPort* that is instantiated.
`ProvidedCommunicationInstance`		`provided communication instance` *`name`* `typed` providedCommunicationPortRef	A sub-class of *CommunicationPortInstance* representing an instance of a provided communication port.
`RequiredCommunicationInstance`		`required communication instance` *`name`* `typed` requiredCommunicationPortRef	A sub-class of *CommunicationPortInstance* representing an instance of a required communication port.
`HostingInstance`		`host` requiredHostPortInstanceRef `on` providedHostPortInstanceRef `typed` hostingRef	A class that represents an instance of a *Hosting* between two components.
	type		A reference to the type of the hosting instance, i.e., the actual *Hosting* instantiated.
	providedHostingInstance		A reference to the instance of the provided hosting port of this communication instance.
	requiredHostingInstance		A reference to the instance of the required hosting port of this communication instance.
`HostingPortInstance`			A class that represents an instance of a *HostingPort*.
	type		A reference to the type of this hosting port instance, i.e., the actual *HostingPort* instantiated.
`ProvidedHostingInstance`		`provided host instance` *`name`* `typed` providedHostPortRef	A sub-class of *HostingPortInstance* that represents an instance of a provided hosting port.
`RequiredHostingInstance`		`required host instance` *`name`* `typed` requiredHostPortRef	A sub-class of *HostingPortInstance* that represents an instance of a required hosting port.
Requirement Meta-Model			
`RequirementModel`		`requirement model` *`name`* `{` inner def ... `}`	A concrete subclass of *Model* that represents a requirement model, i.e., a container of different kinds of requirements. The logical combination of requirements is handled by another element, the *RequirementGroup* which enables to form hierarchies of requirements partitions.
	requirements	`(`requirement def ...`)*`	A containment reference to all requirements that have been specified in this model.
`Requirement`			An abstract class that represents any kind of

			requirement.
	name		An attribute that represents the name of the requirement.

RequirementGroup		group *name* {
		inner def ...
		}
			This class represents a logical group of requirements on which a particular logical operator can be applied. As this class is a sub-class of *Requirement*, this means that a requirement group can contain requirement sub-groups thus enabling to capture whole requirement group hierarchies/trees.

	requirements	requirements [requirementRef (, requirementRef) *]	A reference to the requirements being grouped.
	application	application [applicationRef (, applicationRef) *]	A reference to the applications on which the requirement group applies. This does not mean that each requirement can apply to all applications. On the contrary, the reference can be considered as a collection point for all the applications that are associated with the requirements of a certain group.
	requirementOperator	operator: *RequirementOperatorType*	An attribute that points to the logical operator that is used for the grouping of the requirements. This attribute has as its domain a corresponding enumeration called *RequirementOperatorType*.
RequirementOperatorType		AND or OR or XOR	An enumeration that represents a list of all logical operators that can be used for the logical grouping of requirements (into requirement groups).
HardRequirement			An abstract sub-class of *Requirement* which represents a hard requirement, i.e., a requirement that needs to be guaranteed by the platform at all costs.
SoftRequirement			An abstract sub-class of *Requirement* which represents a soft requirement, i.e., a requirement that the platform should attempt to satisfy in a best-effort basis.
	priority	priority: double	An attribute of type double which represents the priority or preference over a specific soft requirement. This enables to support a prioritisation of soft requirements which can enable some kind of flexibility in the platform towards the satisfaction of this kind of requirements. This maps to the policy that the higher is the priority of a soft requirement, the higher will be its satisfaction degree.

ServiceLevelObjective		slo *name* {
		inner def ...
		}
			This is a sub-class of a hard requirement which represents a Service Level Objective (SLO), i.e., a non-functional requirement.

			Such a requirement is expressed via a condition over a non-functional property (e.g., cost) or metric (e.g., mean response time).
	customServiceLevel	`service level:` conditionRef	The condition of the SLO which can be either a *MetricCondition* or a *PropertyCondition*.
`OptimisationRequirement`		`optimisation requirement` *name* `{` inner def ... `}`	This is the sole sub-class of *SoftRequirement* which represents an optimisation requirement, i.e., a requirement to either minimise or maximise the value of a certain non-functional property or metric. Apart from the element to be optimised, this requirement should be associated to the context of that element for application measurability / monitoring reasons.
	optimisationFunction	`function:` *OptimisationFunctionType*	This attribute represents the optimisation function of the optimisation requirement which is captured by the *OptimisationFunctionType* enumeration. Obviously, such a function can take only two possible values, i.e., MINIMISE or MAXIMISE, reflecting the actual content of the aforementioned enumeration.
	metric	`metric:` metricRef	A reference to the metric whose values need to be optimised.
	property	`property:` propertyRef	A reference to the (non-functional) property whose values need to be optimised.
	application	`application:` applicationRef	A reference to the application on which this optimisation requirement applies.
	component	`component:` componentRef	A reference to the component on which this optimisation requirement applies.
	metricContext	`metric context:` metricContextRef	A reference to the context of the metric to be optimised, which can supply more details to assist in the actual monitoring of this metric.
`HardwareRequirement`			An abstract sub-class of hard requirement which represents a hardware requirement. Two alternative kinds of hardware requirements can be modelled: qualitative and quantitative.
`QualitativeHardwareRequirement`		`qualitative hardware` *name* `{` `benchmark:` double `..` `(`double`)?` `}`	A concrete sub-class of *HardwareRequirement* which represents a qualitative requirement. Qualitative here means non-numeric but linguistic. For instance, based on the classification of VMs, the capabilities of VMs according to the computation aspects could be specified via the values of "small", "medium" and "large". Then, based on that classification, a qualitative requirement could specify that there is a need only for a "large" VM or a

			need for a VM which is "medium" or "large". Such a classification of VMs can be produced via different techniques, including benchmarking.
	minBenchmark	The first double - see above	This property of type double represents the minimum linguistic value for this qualitative requirement.
	maxBenchmark	The second double - see syntax at class level	This property of type double represents the maximum linguistic value for this qualitative requirement.

QuantitativeHardwareRequirement		quantitative hardware *name* {
		(core: int .. (int)?)?
		(ram: int .. (int)?)?
		(storage: int .. (int)?)?
		(cpu: double .. (double)?)?
		}
			This concrete sub-class of *HardwareRequirement* represents a quantitative requirement over the characteristics of a VM. Such characteristics span the number of cores, the CPU frequency, the size of main memory and the storage size. Via such a requirement, ranges over these characteristics can be specified.

	minCPU		An attribute of type double which represents the minimum value of the CPU frequency of the required VM. A value of 0.0 signifies that there is no lower bound on this VM characteristic.
	maxCPU		An attribute of type double which represents the maximum value of the CPU frequency of the required VM. A value of 0.0 signifies that there is no upper bound on this VM characteristic.
	minCores		An attribute of type integer which represents the minimum value of the number of cores of the required VM. A value of 0 signifies that there is no lower bound on this VM characteristic.
	maxCores		An attribute of type integer which represents the maximum value of the number of cores of the required VM. A value of 0 signifies that there is no upper bound on this VM characteristic.
	minRAM		An attribute of type integer which represents the minimum value of the RAM size of the required VM. A value of 0 signifies that there is no lower bound on this VM characteristic.
	maxRAM		An attribute of type integer which represents the maximum value of the RAM size of the required VM. A value of 0 signifies that there is no upper bound on this VM characteristic.
	minStorage		An attribute of type integer which represents the minimum value of the storage size of the required VM. A value of 0 signifies that there

			is no lower bound on this VM characteristic.
	maxStorage		An attribute of type integer which represents the maximum value of the storage size of the required VM. A value of 0 signifies that there is no upper bound on this VM characteristic.

ProviderRequirement		provider requirement *name* {
		inner def ...
		}
			A concrete sub-class of *HardRequirement* which represents a requirement over the cloud provider(s) that can be selected for a certain VM or a specific set of VMs. This requirement can actually specify not only one but multiple cloud providers that can be preferred by the user.

	providers	providers: [providerRef(, providerRef)*]	A reference to the cloud providers (see *CloudProvider* class in organisation meta-model) that are preferred for the deployment of a VM or set of VMs.
OsOrImageRequirement			An abstact sub-class of *HardRequirement* which specifies an OS or image requirement. This is a convenience class representing a exclusive OR combination of such requirements as one from these two kinds of requirements is usually expressed, as users either just need a specific OS for their VMs or also a certain image for instantiating these VMs.

OSRequirement		os *name* {
		inner def ...
		}
			A concrete sub-class of *OsOrImageRequirement* which represents a specific OS requirement for one or more VMs in a deployment model. Such a requirement demands the modelling of two attributes: the actual OS and whether it should be 64-bit or not.

	os	os: 'string'	A String attribute representing the actual OS required.
	is64os	('64os')?	A boolean attribute indicating whether the OS is 64-bit or not. By default, the value of true is assumed.

ImageRequirement		image *name* {
		inner def ...
		}
			A concrete sub-class of *OsOrImageRequirement* which represents a requirement over the image of one or more VMs in a deployment model.

	imageId	imageId: 'string'	A String attribute which specifies the id of the image required. Please note that the image can be either cloud-specific (i.e., mapping to image templates offered in a certain cloud) or user-specific (i.e., mapping to images prepared by the user him/herself).
SecurityRequirement		security requirement *name* { inner def ...	A concrete sub-class of *HardRequirement* which represents a high-level security requirement. Such a requirement explicates

		}	the set of security controls that have to be already implemented for a cloud provider in order to be selected (in particular, for selecting the VM offerings of this cloud provider).
	securityControls	controls: [securityControlRef (, securityControlRef)*]	A reference to the security controls (see *SecurityControl* class in security meta-model) that have to be realised by a selected cloud provider.
	application	application: applicationRef	A reference to the application over which the security requirement applies. This means that when this reference is applied, then this requirement applies to all the components of the referenced application and thus to each cloud provider that has been selected to host these components.
	component	component: internalComponentRef	A reference to an internal application component on which the security requirement should hold. This means that only the selected provider on which the component will be hosted should comply to this requirement.
LocationRequirement		location requirement *name* { inner def ... }	A concrete sub-class of *HardRequirement* which represents a requirement over the placement of one or more VMs in a deployment model. The requirement maps to expressing a set of preferred locations over which such placement is required to be performed.
	locations	locations: [locationRef (, locationRef)*]	A reference to the preferred locations over which the VM placement can be performed. Both cloud-specific and physical locations can be referred to here.
ScaleRequirement			An abstract class of *HardRequirement* which represents a scale requirement over an application component or a VM.
HorizontalScaleRequirement		horizontal scale requirement *name* { component: internalComponentRef instances: int .. int }	A concrete sub-class of ScaleRequirement which represents a horizontal scaling requirement. Such a requirement expresses a range over the number of instances that can be generated for a certain application component.
	minInstances	See first integer above	An integer attribute that represents the minimum number of instances that have to be generated for the application component referenced. A positive value should be provided for this attribute.
	maxInstances	See second integer in the syntax at the class level.	An integer attribute that represents the maximum number of instances that have to be generated for the application component referenced. Either a positive or a value of -1

			should be provided for this attribute. In the former case, this value should be greater or equal to the one provided for the *maxInstances* attribute. In the latter case, this value represents positive infinity signifying that the user does not impose an upper bound on the number of instances of the referenced component.
	component		A reference to the internal application component on which the scale requirement is specified.

`VerticalScaleRequirement`		`vertical scale requirement` *name* `{`
		`vm:` vmRef
		`(core:` int `..` `(`int`)?)?`
		`(ram:` int `..` `(`int`)?)?`
		`(storage:` int `..` `(`int`)?)?`
		`(cpu:` double `..` `(`double`)?)?`
		`}`
			A concrete sub-class of *ScaleRequirement* which represents a requirement over the size of a VM, i.e., over the values that the VM characteristics can take. To this end, min and max values are allowed to be specified for all 4 characteristics of a VM, i.e., the CPU frequency, the number of cores, the RAM size and the storage size. At least one bound over at least one VM characteristic should be provided in order for an instance of a *VerticalScaleRequirement* to be meaningful and valid.

	minCPU		An attribute of type double which represents the minimum value of the CPU frequency of the VM referenced by this vertical scale requirement. The value of this attribute should be greater or equal to 0.0 (where 0.0 indicates that we do not care for the value of this attribute).
	maxCPU		An attribute of type double which represents the maximum value of the CPU frequency of the VM referenced by this vertical scale requirement. The value of this attribute should be either positive or equal to -1. A positive value should also be greater or equal to the value of the *minCPU* attribute. A value of -1 denotes positive infinity and signifies that the user is open for the upper limit of this VM characteristic.
	minCores		An attribute of type integer which represents the minimum value of the number of cores of the VM referenced by this vertical scale requirement. The value of this attribute should be greater or equal to 0.0 (where 0.0 indicates that we do not care for the value of this attribute).
	maxCores		An attribute of type integer which represents the maximum value of the number of cores of the VM referenced by this vertical scale requirement. The value of this attribute should be either positive or equal to -1. A

			positive value should also be greater or equal to the value of the *minCores* attribute. A value of -1 denotes positive infinity and signifies that the user is open for the upper limit of this VM characteristic.
	minRAM		An attribute of type integer which represents the minimum value of the RAM size of the VM referenced by this vertical scale requirement. The value of this attribute should be greater or equal to 0.0 (where 0.0 indicates that we do not care for the value of this attribute).
	maxRAM		An attribute of type integer which represents the maximum value of the RAM size of the VM referenced by this vertical scale requirement. The value of this attribute should be either positive or equal to -1. A positive value should also be greater or equal to the value of the *minRAM* attribute. A value of -1 denotes positive infinity and signifies that the user is open for the upper limit of this VM characteristic.
	minStorage		An attribute of type integer which represents the minimum value of the storage size of the VM referenced by this vertical scale requirement. The value of this attribute should be greater or equal to 0.0 (where 0.0 indicates that we do not care for the value of this attribute).
	maxStorage		An attribute of type integer which represents the maximum value of the storage size of the VM referenced by this vertical scale requirement. The value of this attribute should be either positive or equal to -1. A positive value should also be greater or equal to the value of the *minStorage* attribute. A value of -1 denotes positive infinity and signifies that the user is open for the upper limit of this VM characteristic.
	vm		A reference to the VM on which the vertical scale requirement applies.
`OptimisationFunctionType`		MIN or MAX	An enumeration which represents the concrete optimisation functions (i.e., MINIMISE and MAXIMISE) which should be used in an *OptimisationRequirement*.
Metric Meta-Model			
`Condition`			An abstract class that represents a condition over a non-functional term, either a *Metric* or a *Property*. Conditions are building constructs for modelling service level objectives and events for scalability rules.

	name		A String attribute representing the name of the condition
	comparisonOperator	`comparisonOperator: ComparisonOperatorType`	An attribute that represents a comparison operator. Such an operator is used to compare the threshold provided (see next property) with the actual value of the non-functional term concerned. It takes values from the *ComparisonOperatorType* enumeration.
	threshold	`threshold:` double	An attribute of type double which represents the actual threshold that needs to be imposed over the values of the non-functional term referenced.
	validity	`validity:` date	An attribute of type Date which represents the final date over which the condition specified is valid.
`MetricCondition`		`metric condition name { inner def ... }`	A concrete sub-class of *Condition* which represents a condition over a *Metric*. As many measurement details are needed for enabling the evaluation of such a condition, this class is associated with the context of the metric (see *MetricContext* class later on).
	metricContext	`context:` metricContextRef	A reference to the context of the metric on which the condition applies.
`PropertyCondition`		`property condition name { inner def ... }`	A concrete sub-class of *Condition* which represents a condition over a non-functional property. As additional details have to be specified for enabling the appropriate evaluation of such a condition, this class is associated with the context of the property (see *PropertyContext* class later on) on which the condition applies.
	propertyContext	`property context:` propertyContextRef	The context of the property on which the condition applies.
	unit	`unit:` unitRef	The unit of measurement of the non-functional property. Such a unit is supplied in order to cover the variability in measurement of such a property by different monitoring systems.
	timeUnit	`time unit:` timeIntervalUnitRef	----------TO BE REMOVED----------
`ComparisonOperatorType`		= = or < or < = or <> or > or >=	An enumeration that list all possible comparison operators that can be used in (non-functional term) conditions.
`MetricInstance`			An abstract class that represents an instance of a *Metric*. Such an instance represents a concrete monitoring entity which can be attached to a specific application object that is to be measured. Depending on the kind of the metric concerned, there can be instances

			of raw and composite metrics.
	name		A String attribute that represents the name of the metric instance
	metric	`metric:` metricRef	A reference to the type of this metric instance, i.e., a specific *Metric*.
	schedule	`schedule:` scheduleRef	A reference to the measurement schedule that applies for this metric instance highlighting how often to measure it (see also *Schedule* class later on)
	window	`window:` windowRef	A reference to the measurement window (see also *Window* class later on) that applies for the metric instance which specifies how many values need to be considered within a specific window of a certain size in order to be aggregated. Thus, this applies mainly for instances of composite metrics.
	objectBinding	`object binding:` metricObjectBindingRef	A reference to the object binding that applies for this metric instance. Such a binding associates this instance to a certain execution context (see *ExecutionContext* class) as well as to the instance of the object (e.g., internal component or VM) that is being measured.
	metricContext	`context:` metricContextRef	A reference to the context of the type of this metric instance which provides appropriate details for its measurement.

`CompositeMetricInstance`		`composite metric instance` *name* `{`
		inner def ...
		`}`
			A concrete sub-class of *MetricInstance* which represents an instance of a composite metric. As such a metric is composed of other metrics, the instance of this metric is associated with the instances of all the composing metrics of this metric.

	composingMetricInstances	`composing metric instances: [` metricInstanceRef`(,` metricInstanceRef`)*` `]`	A reference to the composing metric instances of this composite metric (i.e., the instances of metrics which are components of this instance's metric).

`RawMetricInstance`		`raw metric instance` *name* `{`
		inner def ...
		`}`
			A concrete sub-class of *MetricInstance* which represents an instance of a *RawMetric*. As raw metrics are usually measured via sensors, this class is associated with the sensor used to measure this metric instance.

	sensor	`sensor:` sensorRef	A reference to the sensor which is used to measure this raw metric instance.

`MetricFormulaParameter`		`parameter` *name* `{`
		inner def ...
		`}`
			A class which represents a parameter which can be used as input in the formula of a *CompositeMetric*. Different kinds of metric formula parameters can be specified (e.g., metrics or formulas). In addition, a metric formula parameter can represent a certain value/constant (see *SingleValue* class).

	name		A String attribute that represents the name of the metric formula parameter.
	value	`value`: single value inner def ...	A reference to a *SingleValue* which represents a constant metric formula parameter.
`MetricFormula`		`metric formula` *name* { `function arity:` *MetricFunctionArityType* `(function pattern:` *FunctionPatternType*`)?` *MetricFunctionType* (metricFormulaParameterRef (, metricFormulaParameterRef)*) }	A concrete sub-class of *MetricFormulaParameter* which represents a particular formula via which a composite metric can be measured. A formula just represents the application of a function over a set of input metric formula parameters. Thus, as a metric formula is also such a parameter, a hierarchy of metric formulas can be expressed even for a single composite metric mapping to the expression of advanced, complicated metric aggregation computations.
	function	See syntax at class level	An attribute that represents the actual concrete function that is applied in the formula of a composite metric. The type of this attribute maps to the *MetricFunctionType* enumeration that lists all possible metric formula functions.
	functionArity	See syntax at class level	This attribute represents the arity of the function that is applied in the formula of a composite metric. Such an arity is associated with the size of the input parameters set that should be exploited in the metric formula computation. Depending on the function exploited by a metric formula, a different arity might apply. For instance, the MEAN function maps to an UNARY arity meaning that one input parameter should be involved in the composite metric formula. The type of the attribute relates to the *MetricFunctionArityType* enumeration that specifies all possible kinds of arities that can be involved.
	functionPattern	See syntax at class level	This attribute signifies what is the pattern of the formula function exploited and takes values from the *FunctionPatternType* enumeration. Two kinds of patterns apply here: MAP and REDUCE. The MAP pattern indicates that the values of the input parameters are mapped to another value (via typical mathematical operators), while the REDUCE pattern indicates that the values of usually one input parameter are reduced/aggregated into one (via the use of statistical functions/operators).
	parameters	See syntax at class level	A reference to the set of input parameters used in the metric formula.

Metric			An abstract sub-class of the *MetricFormulaParameter* which represents a metric, i.e., a conceptualisation of various monitoring details for the measurement of a particular non-functional property. Two kinds of metrics can be expressed: (a) raw metrics that are directly measured via sensors; (b) composite metrics which are computed from formulas applied on the values/measurements of other metrics.
	description	description: 'string'	A String attribute that can be used to specify a textual description of the metric targeting human users.
	valueType	value type: valueTypeRef	A reference to the domain of values / value type (see *ValueType* class) of the metric. Such a domain is a restriction over the possible measurement values that a metric can take.
	valueDirection	value direction: short	An attribute of type short which indicates the preferred direction of values for a certain metric. A value of 1 indicates that the metric is positively monotonic while a value of 0 indicates that the metric is negatively monotonic. Positive monotonicity favours greater values / measurements for a metric (i.e., the greater is the value, the better) while negative monotonicity favours smaller metric measurement values (i.e., the smaller is the value, the better).
	unit	unit: unitRef	A reference to the unit of measurement (see *Unit* class) for this metric.
	layer	layer: LayerType	An attribute that represents the layer on which the metric applies which takes values from the *LayerType* enumeration. A metric at the IaaS layer, for example, would measure elements/objects at that layer, i.e., VMs.
	property	property: propertyRef	A reference to the non-functional (MEASURABLE) property that is measured by this metric.
	isVariable	(variable)?	A boolean attribute which indicates whether the metric can be considered also as a variable in the formulation of a constraint model that can be used to solve a deployment reasoning problem for the application components.

CompositeMetric		composite metric *name* {
		inner def ...
		}
			A concrete sub-class of *Metric* representing a composite metric, i.e., a metric which is computed via the application of a certain formula over the values / measurements of other metrics.

	formula	metric formula def ...	A containment reference to the metric formula from which the composite metric is

			computed.

RawMetric		raw metric *name* {
		inner def ...
		}
			A concrete sub-class of *Metric* representing a raw metric, i.e., a metric which is directly measured via a sensor.

MetricFunctionAritytype		UNARY or BINARY or N_ARY	An enumeration which explicates the exact kinds of arities that can apply to metric functions (which are used in the formulas of composite metrics). Three kinds of arities are modelled as members of this enumeration: (a) UNARY: this means that the function should take only one parameter as input; (b) BINARY: this indicates that the function should take exactly two parameters as input; (c) N_ARY: this signifies that the function should take at least two parameters as input.
MetricFunctionType		COUNT or DERIVATIVE or DIV or MAX or MEAN or MEDIAN or MIN or MINUS or MODE or MODULO or PERCENTILE or PLUS or STD or TIMES	An enumeration that list all functions that can be used in the computation of values for a composite metric. Such functions, members of this enumeration, span typical mathematical operators (e.g., TIMES) as well as statistical operators (e.g., MEAN).
MetricObjectBinding			An abstract class that represents the binding between an instance of a metric and the instance of the object (e.g., internal component or VM) that is being measured. Such a binding takes place under a certain execution context, i.e., a specific deployment episode involving the user application at hand.
	name		A String attribute representing the name of the binding.
	executionContext	execution context: executionContextRef	A reference to the execution context (application deployment episode) under which the binding holds.

MetricApplicationBinding		application binding *name* {
		inner def ...
		}
			A concrete sub-class of *MetricObjectBinding* which represents the binding of a metric instance to a specific user application. As the execution context, referenced and inherited by the parent class is associated with the application at hand, no other information needs to be specified for this class.

MetricComponentBinding		component binding *name* {
		inner def ...
		}
			A concrete sub-class of *MetricObjectBinding* which represents the binding of a metric instance to an instance of a certain application component.

	vmInstance	vm instance: vmInstanceRef	A reference to the instance of a VM on which the bound instance of the application component has been deployed. This reference does not need to be always modelled. This depends on whether this constitutes important information for the

			monitoring system.
	componentInstance	component instance: componentInstanceRef	A reference to the instance of the (application) component that is bound by this metric (instance) binding.

MetricVMBinding		vm binding *name* {
		inner def ...
		}
			A concrete sub-class of *MetricObjectBinding* which represents a binding of a specific VM instance to a certain metric instance.

	vmInstance	vm instance: vmInstanceRef	A reference to the instance of the VM that is bound on the corresponding metric instance.

Property		property *name* {
		inner def ...
		}
			A class that represents a non-functional property. A property can be measurable or abstract. A measurable property can be measured via one or more metrics. An abstract property cannot be measured but comprises more concrete properties that could be measured instead.

	name		A String attribute that represents the name of the non-functional property
	description	description: 'string'	A String attribute that can be used to provide a textual description of the property for human consumption reasons.
	type	type: *PropertyType*	An attribute that denotes the type of the property. This attribute takes values from the *PropertyType* enumeration. As already indicated, a property can be measurable or abstract. To this end, two corresponding members of the aforementioned enumeration have been modelled to be able to specify these two alternatives.
	subProperties	sub-properties [propertyRef (, propertyRef)*]	A reference to the sub properties of an abstract property.
	sensors	sensors [sensorRef (, sensorRef)*]	A reference to a set of sensors that can be used in the measurement of corresponding raw metrics that can measure the current property.
PropertyType		ABSTRACT or MEASURABLE	An enumeration that lists the two main kinds of a non-functional property (MEASURABLE and ABSTRACT).

Schedule		schedule *name* {
		inner def ...
		}
			A concrete class which represents the measurement schedule for a metric. Such a schedule might have a starting and ending time and it can be repeated with a particular time frequency. The number of repetitions can also be restrained, if needed. Such a schedule can be provided for both raw and composite metrics and is coupled to them via their (metric) context. In case such a schedule is not provided, then the corresponding measurement system should

			be able to derive it somehow in the most appropriate way.
	name		A String attribute representing the name of the schedule
	start	`start:` date	A date attribute which denotes the starting time point for the schedule.
	end	`end:` date	A date attribute which denotes the ending time point for the schedule
	type	`type:` *ScheduleType*	An attribute which denotes the type of the schedule, taking values from the *ScheduleType* enumeration. In case the value of this attribute equals the SINGLE_EVENT member of that enumeration, this means that the schedule maps to the occurrence of just one measurement and not to the repeated production of multiple measurements for a certain metric. As such, no further details need to be provided for this schedule (with respect to the values of the rest of the attributes apart from the *name* one).
	unit	`unit:` timeIntervalUnitRef	A reference to a *TimeInternalUnit* which indicates the unit of time for the frequency / internal of this schedule.
	repetitions	`repetitions:` int	An attribute of type integer that denotes the number of times a measurement for the metric must be produced (according to the time interval specified - see next attribute)
	interval	`interval:` long	An attribute of type long that represents the internal between the time points involved in the production of a measurement for the metric correlated to this schedule.
`ScheduleType`		FIXED_RATE or FIXED_DELAY or SINGLE_EVENT	An enumeration that list all kinds of *Schedule*s that can be modelled.

`Sensor`		`sensor` *name* `{`
		inner def ...
		`}`
			A concrete class that represents a sensor. Such a sensor is able to produce measurements for raw metrics. A sensor can be seen as a software component that is configurable and can be installed either internally to a measurement system (either at the cloud or the system domain) or can be external (e.g., a user-specific sensor which runs at the user domain).

	name		A String attribute that represents the name of the sensor.
	configuration	`configuration:` 'string'	A String attribute that encapsulates the configuration of the sensor. For instance, in the PaaSage platform such an attribute contained two pieces of configuration information: (a) the name of the metric to

			report; (b) the (Java) class implementing the sensor logic (not needed to be provided when the sensor is external to the monitoring system).
	isPush	(push)?	A boolean attribute that indicates whether the sensor should push the measurements to the monitoring system or the system should otherwise pull the generated measurements from the sensor.

Window		window *name* {
		inner def ...
		}
			A concrete class that represents a measurement window mainly utilised for measurement aggregation purposes (i.e., for composite metric computation). There can be different clusterings of windows according to their size and type. Concerning the size, windows can be either time-based (i.e., mapping to a time-based window size which highlights that the window becomes full when this time period is passed), measurement-based (i.e., mapping to a particular measurement size for the window), or mixed spanning first-match (i.e., either one of the two aforementioned kinds of size is reached to consider that the window is full) or both-match (both sizes need to be reached to consider that the window is full). Concerning the type, only two kinds of windows can be modelled: (a) sliding meaning that the values stored in the window slide as soon as new values are entered and the window is already full; (b) fixed meaning that the window is re-initialised when it becomes fully occupied. Depending on the value designated for these two dimensions for a certain window, different values should be provided for the rest of the window attributes and parameters (please see analysis below per each attribute/property of this class).

	name		A String attribute representing the name of the sensor.
	unit	unit: timeIntervalUnitRef	A reference to a *TimeIntervalUnit* which represents the unit of time used to denote the time extent of the window. For instance, if the unit is seconds and the value of *timeSize* attribute is 5, then this means that the window has as its extent a period of 5 seconds. This reference should not be given if the size type of the window is MEASUREMENTS_ONLY.
	windowType	window type: *WindowType*	An attribute that represents the type of the window taking its values from the

			WindowType enumeration. As stated, two types of window have been captured, SLIDING and FIXED.
	sizeType	`size type: WindowSizeType`	An attribute that represents the size type of the window taking its values from the *WindowSizeType* enumeration. Four members of this enumeration have been modelled as already indicated in the description of the semantics of the Window class: TIME_ONLY, MEASUREMENTS_ONLY, FIRST_MATCH and BOTH_MATCH.
	measurementSize	`measurement size:` long	The size of the window in terms of the maximum number of measurements that it can store. This attribute should not be given any value when the size type of the window is TIME_ONLY.
	timeSize	`time size:` long	The time size of the window, i.e., the extent or period of time considered until the window is regarded to be full. This attribute should not be given a value when the size type of the window is MEASUREMENTS_ONLY. When a value for this attribute is given, then also a reference to a time internal unit should be supplied via the *unit* property.
`WindowSizeType`		MEASUREMENTS_ONLY or TIME_ONLY or FIRST_MATCH or BOTH_MATCH	An enumeration representing the types of window size and having as members the aforementioned size types: MEASUREMENTS_ONLY, TIME_ONLY, FIRST_MATCH and BOTH_MATCH.
`WindowType`		FIXED or SLIDING	An enumeration representing the different kinds of a window and having as members the values: SLIDING and FIXED.
`ConditionContext`			An abstract class that represents the context for a particular condition. Such a context encapsulates various kinds of information that need to be utilised to support the proper measurement of the metric involved in the condition as well as the evaluation of the condition itself. Such information spans the object being measured, the non-functional term (metric or property) involved in the measurement, as well as restrictions at the instance level which explicate for how many instances of the referenced object need to be measured and evaluated against the condition threshold in order to consider that the condition has been violated. Depending on the kind of non-functional terms, two different kinds of condition context can be modelled: *MetricContext*s and *PropertyContext*s.

name		A String attribute representing the name of the condition context.
component	`component: componentRef`	A reference to an internal (application) component as the object that is being measured.
application	`application: applicationRef`	A reference to the application being measured. If this property is given but not the component one, this signifies that the measurement is performed for the whole user application and not one of its components. This leads to concluding that the object being measured is the application itself, in this case.
quantifier	`quantifier:` *`QuantifierType`*	An attribute taking values from the *QuantifierType* enumeration which represents a quantifier. This quantifier indicates whether the condition is regarded as violated when its threshold is surpassed by the measurement(s) of all (logical AND semantics), some or just one (logical OR semantics) application component instance.
minQuantity	`quantity: double .. (double)?` Note: also referring to next attribute in this syntax	An attribute of type double which indicates the minimum number of component instances whose measurements lead to the surpassing of the condition threshold in order to consider that the condition is violated. To be provided only when the *quantifierType* equals to SOME.
maxQuantity	See syntax of minQuantity attribute	An attribute of type double which indicates the maximum number of component instances whose measurements lead to the surpassing of the condition threshold in order to consider that the condition is violated. To be provided only when the *quantifierType* equals to SOME.
isRelative	`(relative)?`	A boolean attribute which indicates whether the values of the *minQuantity* and *maxQuantity* attributes are relative or absolute. Relative means that the values of these attributes represent a percentage of the overall number of instances of a certain component/measurement object. Absolute means that the values of these attributes map to the actual number of instances of the component / measurement object. For instance, suppose that the value of this attribute is true and that minQuantity and maxQuantity are 0.2 and 0.6. This means that when the measurements between 20% and 60% percent of the measured component instances surpass the condition threshold, then the condition is considered to be

			violated.
QuantifierType		ALL or SOME or ANY	An enumeration that represents the different kinds of a quantifier (ALL, ANY or SOME) for a *ConditionContext*.

MetricModel		metric model *name* {
		inner def ...
		}
			A concrete subclass of *Model*, representing a metric model which acts as a container for all measurement elements that have to be specified, such as metrics, properties, formulas, conditions and contexts.

	contexts	(condition context def ...) *	A containment reference to all the condition contexts defined in this metric model
	metricInstances	(metric instance def ...) *	A containment reference to all metric instances defined in this metric model
	conditions	(condition def ...) *	A containment reference to all conditions defined in this metric model
	properties	(property def ...) *	A containment reference to all properties defined in this metric model
	bindings	(metric binding def ...) *	A containment reference to all metric object bindings defined in this metric model
	windows	(window def ...) *	A containment reference to all windows defined in this metric model
	schedules	(schedule def ...) *	A containment reference to all schedules defined in this metric model
	parameters	(metric formula parameter def ...) *	A containment reference to all metric formula parameters defined in this metric model
	sensors	(sensor def ...) *	A containment reference to all sensors defined in this metric model
	units	(unit def ...) *	A containment reference to all units defined in this metric model
MetricContext			An abstract sub-class of *ConditionContext* which specifies the context for a certain metric. Such a context refers to the metric itself as well as provides measurement details with respect to the metric scheduling and window of measurement.
	metric	metric: metricRef	A reference to the metric of this *MetricContext*.
	schedule	schedule: scheduleRef	A reference to the schedule of the metric concerned. When this reference is not provided, then either a default schedule is computed by the monitoring system or it is considered that the metric is measured on demand. This depends on the nature of the metric. For instance, for raw metrics, usually the schedule needs to be derived. For composite metrics, this depends on the formula function. If this function maps to a

			typical mathematical operator, then the metric should be computed on demand (as soon as new values for the component metrics are produced). Otherwise, a certain schedule for the composite metric should be derived by the monitoring system.
	window	`window:` windowRef	A reference to the measurement window of the metric concerned. If this reference is not provided, then a default window should be calculated by the monitoring system.

`CompositeMetricContext`		`composite metric context` *name* `{`
		inner def ...
		`}`
			A concrete sub-class of *MetricContext* which represents the context for a composite metric. Apart from the information provided by the parent class, this class also refers to the context of the composing metrics of this composite metric. This is essential in order to be able to provide all necessary measurement details not only for the top composite metric but also for its descendant (composing) metrics.

	composingMetricContexts	`composing metric contexts [` metricContextRef `(,` metricContextRef`)*]`	A reference to the context of the composing metrics of the composite metric concerned.

`RawMetricContext`		`raw metric context` *name* `{`
		inner def ...
		`}`
			A concrete sub-class of *MetricContext* which represents the context of a raw metric. As raw metrics are produced from sensors while all other needed measurement information is inherited from the parent class, this class only refers to the sensor exploited for producing the measurements of the raw metric concerned.

	sensor	`sensor:` sensorRef	A reference to the *Sensor* that is used for producing the measurements of the raw metric concerned.

`PropertyContext`		`property context` *name* `{`
		inner def ...
		`}`
			A concrete sub-class of ConditionContext which represents the context of a non-functional property. As such, as all measurement details are inherited by its parent class, this class only refers to the property being concerned.

	property	`property:` propertyRef	A reference to the property associated with this property context.
`FunctionPatternType`		MAP or REDUCE	An enumeration which represents all kinds of function patterns (MAP & REDUCE in particular - see also analysis of the *MetricFormula* class).
Scalability Meta-Model			

`ScalabilityModel`		`scalability model` *name* `{`
		inner def ...
		`}`
			A concrete subclass of *Model*, representing a scalability model which acts as a container for all elements required for specifying

			scalability rules, including the scalability rules themselves.
	rules	(scalability rule def ...) *	A containment reference to all scalability rules defined in this scalability model
	events	(event def ...) *	A containment reference to all events defined in this scalability model
	eventInstances	(event instance def ...) *	A containment reference to all event instances defined in this scalability model
	actions	(action def ...) *	A containment reference to all actions defined in this scalability model
	patterns	(event pattern def ...) *	A containment reference to all event patterns defined in this scalability model
	timers	(timer def ...) *	A containment reference to all timers defined in this scalability model
	scaleRequirements	(scale requirement def ...) *	A containment reference to all scale requirements defined in this scalability model
`BinaryPatternOperatorType`		AND or OR or XOR or PRECEDES or REPEAT_UNTIL	An enumeration of binary event pattern operators, i.e., kinds of operators used for composing two events into an event pattern. Such operators include typical logical operators (e.g., AND) as well as time-based operators which include PRECEDES (i.e., the first event in the pattern precedes the second one) and REPEAT_UNTIL (repetition of first event in the pattern, possibly constrained via a range over the minimum and maximum occurrence number of this first event, until a specific (second) event occurs or a timer is fired (denoting the end of a specific time period).
`Event`			An abstract class that represents an event that can lead to the triggering of a scalability rule. Events can be simple or composite (i.e., event patterns).
	name		A String attribute that represents the name of the event.
`EventPattern`			An abstract sub-class of *Event* that represents an event pattern (a combination of one or more events). An event pattern might include a *Timer*, an entity which can express the occurrence of a time event which can be correlated with other, normal events in the event pattern.
	timer	`timer:` timerRef	A reference to a *Timer* element that can be associated to this event pattern.
`BinaryEventPattern`		`binary event pattern` *`name`* `{` inner def ... `}`	A concrete sub-class of *EventPattern* which represents a pattern or composition of either two events or one event and a *Timer*. The two component elements of a pattern can be

			combined via the use of a binary event pattern operator (see *BinaryPatternOperatorType* enumeration).
	leftEvent	`left event:` eventRef	A reference to the first event in the event pattern. If this reference is not given, this means that the first event is represented by the *Timer* element. Please note that as the *Timer* can occupy one from the two possible event positions in a pattern, this means that at least one of *leftEvent* or *rightEvent* references needs to be always provided.
	rightEvent	`right event:` eventRef	A reference to the second event in the event pattern. If this reference is not given, this means that the second event is represented by the *Timer* element. Please note that as the *Timer* can occupy one from the two possible event positions in a pattern, this means that at least one of *leftEvent* or *rightEvent* references needs to be always provided.
	lowerOccurrenceBound	`lower occurrence bound:` int	An attribute of type integer which denotes the lower bound on the occurrence of an event in a pattern composition that is governed by the REPEAT_UNTIL binary event pattern operator. Such a lower bound can also be zero (default value) which can either mean that the bound is not considered at all (operator different than REPEAT_UNTIL) or that a non-occurrence of the first event in the pattern is also allowed (operator is REPEAT_UNTIL).
	upperOccurrenceBound	`upper occurrence bound:` int	An attribute of type integer which denotes the upper bound on the occurrence of an event in a pattern composition that is governed by the REPEAT_UNTIL binary event pattern operator. Such an upper bound can also be zero (default value) which can either mean that the bound is not considered at all (operator different than REPEAT_UNTIL) or that the upper bound is open (operator is REPEAT_UNTIL). Please note that if this bound is positive, then it should always be greater or equal to the lower occurrence bound.
	operator	`operator:` *`BinaryPatternOperatorType`*	An attribute taking values from the *BinaryPatternOperatorType* enumeration representing the actual binary event pattern operator that is exploited for composing the component events (or event plus *Timer* elements) of the event pattern.
`UnaryEventPattern`		`unary event pattern` *`name`* `{` inner def ...	A concrete sub-class of *EventPattern* which represents a unary event pattern. Such a pattern always involves one event as well as

		}	a unary pattern operator, and in one case (i.e., when the WHEN operator is used) also a Timer element.
	event	`event: eventRef`	A reference to the sole event involved in the event pattern.
	occurrenceNum	`occurrence num: int`	An attribute of type integer which denotes the number of occurrences of the sole event referenced. A positive value for this attribute should only be used when the unary pattern operator is REPEAT denoting an exact repetition in number of the event concerned.
	operator	`operator: BinaryPatternOperatorType`	An attribute taking values from the *UnaryPatternOperatorType* enumeration which represents the kind of unary pattern operator utilised in the event pattern. Four values can be provided for this attribute mapping to a corresponding member of this enumeration: (a) EVERY: has the semantics of "each", meaning that we consider here each occurrence of an event in an individual manner; (b) NOT: classical unary logical operator which in this case denotes the non-occurrence of the event referenced; (c) REPEAT: denotes an exact number of occurrences of the event referenced; (d) WHEN: denotes that the referenced event needs to occur within a specific time period expressed via a *Timer* element.
SimpleEvent			An abstract sub-class of *Event* which represents a single and thus not composite event (either combined with a unary operator or participating in a binary event composition). Simple events can be either functional or non-functional.
FunctionalEvent		`functional event name { inner def ... }`	A concrete sub-class of *SimpleEvent* which represents a functional event. Such an event can map to a certain fault, e.g., characterising the whole application or one of its components (for instance a permanent component failure or a temporary bug that can be usually resolved by restarting that component) or even the infrastructure hosting the application components (e.g., VM failure).
	functionalType	`functional type: 'string'`	A String attribute which represents the type of the functional event. We consider here that a customised taxonomy of functional events can be built such that it can be referenced by this attribute. This is a rather different way of modelling such an attribute with respect to other CAMEL classes which is due to the fact that the taxonomy here is not

			generic and thus amenable for being modelled via a certain enumeration.

Non-FunctionalEvent		non-functional event *name* {
		inner def ...
		}
			A concrete sub-class of *SimpleEvent* which denotes a non-functional event, i.e., an event that is not directly related to the functionality of the application. Non-functional events are usually expressed via a reference to a metric condition (see *MetricCondition* class) along with the specification of an additional boolean attribute which signifies whether the violation of the condition should trigger the generation of the event or the satisfaction of this condition.

	metricCondition	metric condition: metricConditionRef	A reference to a metric condition whose violation or satisfaction leads to the generation of this non-functional event.
	isViolation	(violation)?	A boolean attribute which signifies that either the violation of the metric condition (true value) should lead to the generation of this non-functional event or the satisfaction of this metric condition.

EventInstance		event instance *name* {
		inner def ...
		}
			A concrete class that represents an instance of a specific event. Such an instance is generated by evaluating the metric condition over a measurement produced for a certain instance of the metric at hand.

	name		A String attribute representing the name of the event instance.
	status	status: *StatusType*	An attribute that represents the exact status of the event and takes values from the *StatusType* enumeration. The latter indicates whether an event instance is FATAL (e.g., application cannot be executed), CRITICAL (e.g., an SLO has been violated), WARNING (e.g., we are near a SLO condition threshold) or SUCCESS (e.g., this represents a normal and thus non-problematic instance of an event).
	layer	layer: *LayerType*	An attribute that refers to the layer (please see LayerType class) that applies for this event instance.
	event	event: eventRef	A reference to the type of this event instance, i.e., the actual event concerned.
	metricInstance	metric instance: metricInstanceRef	A reference to the metric instance that led to the measurement which triggered the generation of the event instance. Please note that the metric of that instance should have been involved in the metric condition referenced by the event/type of this event instance.

ScalabilityRule		scalability rule *name* {
		inner def ...
		}
			A concrete class that represents a scalability rule. Such a rule is a mapping from an event to one or more actions (usually scaling ones) which can enable the adaptive reconfiguration of an application in a local / cloud-based manner.

	name		A String attribute that represents the name of a scalability rule.
	event	event: eventRef	A reference to the event (simple or event pattern) that leads to the triggering of the scalability rule.
	actions	actions [actionRef (, actionRef)*]	A reference to a set of actions (usually scaling ones) that need to be executed when the scalability rule is triggered.
	entities	entities [entityRef (, entityRef)*]	A reference to the entity (user or application) that has designed the scalability rule.
	scaleRequirements	scale requirements [scaleRequirementRef (, scaleRequirementRef)*]	A reference to a set of scale requirements (see *ScaleRequirement* class) that need to be obeyed during the execution of the scalability rule. Actually such requirements constraint the way the scalability rule can be executed. For instance, a horizontal scale requirement constraints the number of instances that can be created for a certain application component.
ScalingAction			An abstract sub-class of *Action* that denotes a scaling action which can be either horizontal or vertical. In both cases, there is a need to reference the VM concerned. This is needed for an horizontal scaling action in order to clarify the respective hosting, i.e., to which exactly VM on which the component to be scaled has been deployed, something essential if an application component is deployed on multiple VMs within the same deployment episode. Trivially a vertical scaling action need to refer to the scaling of a certain VM.
	vm	vm: vmRef	A reference to the VM concerned.

HorizontalScalingAction		horizonal scaling action *name* {
		inner def ...
		}
			A concrete sub-class of *ScalingAction* which represents a horizontal scaling action. Such an action leads to scaling out or in a certain component according to a specific number of instances.

	count	count: int	An attribute of type integer that represents the number of instances to be added or removed for the internal component concerned, depending on the kind of the horizontal scaling action.
	internalComponent	internal component:	A reference to the *InternalComponent* to be

		internalComponentRef	scaled.

`VerticalScalingAction`		`vertical scaling action` *name* {
		inner def ...
		}
			A concrete sub-class of *ScalingAction* that denotes a vertical scaling action. Such an action indicates whether a certain VM should be scaled up (acquire more resources) or down (acquire less resources). To this end, respective attributes have been modelled which represent the difference in the amount of a specific VM characteristic (e.g., number of cores) that needs to take effect.

	memoryUpdate	`memory update:` int	An attribute of type integer that represents the amount of main memory that has to be additionally reserved (positive value) or released (negative value) for the VM concerned. A value of 0 means that this kind of VM characteristic remains untouched.
	CPUUpdate	`cpu update:` double	An attribute of type double that represents the amount of CPU frequency that has to be additionally reserved (positive value) or released (negative value) for the VM concerned. A value of 0 means that this kind of VM characteristic remains untouched.
	coreUpdate	`core update:` int	An attribute of type integer that represents the amount of cores that has to be additionally reserved (positive value) or released (negative value) for the VM concerned. A value of 0 means that this kind of VM characteristic remains untouched.
	storageUpdate	`storage update:` int	An attribute of type integer that represents the amount of storage that has to be additionally reserved (positive value) or released (negative value) for the VM concerned. A value of 0 means that this kind of VM characteristic remains untouched.
	ioUpdate	`io update:` int	An attribute of type integer that represents the amount of IO throughput that has to be additionally accommodated (positive value) or decommissioned (negative value) for the VM concerned. A value of 0 means that this kind of VM characteristic remains untouched.
	networkUpdate	`network update:` int	An attribute of type integer that represents the amount of network throughput that has to be additionally accommodated (positive value) or decommissioned (negative value) for the VM concerned. A value of 0 means that this kind of VM characteristic remains untouched.

`Timer`		`timer` *name* {
		inner def ...
		}
			A concrete class that represents a timer. Such a timer represents certain time periods that either need to be passed (having as type the value of INTERVAL) or within which one or

			more events need to take place (having as type the value of WITHIN or WITHIN_MAX). The difference between the WITHIN and WITHIN_MAX kinds of timers is that in the second case there should be a maximum number of respective event(s) that need to take place within the time period defined. If this maximum number is reached, the respective pattern is considered to be satisfied and the timer finishes. If it is not, then the timer finishes when the time period defined is ended.
	name		A String attribute that represents the name of the timer.
	type	`type:` *TimerType*	An attribute taking values from the *TimerType* enumeration that denotes the exact kind of the timer concerned (see also discussion in *Timer* class).
	timeValue	`time value:` int	An attribute of type integer that denotes the time interval size for the time period pertaining to the timer.
	maxOccurrenceNum	`max occurrence num:` int	An attribute of type integer that denotes the maximum number of occurrences of the event correlated to this timer that need to take place before the timer can be finished, unless the time period defined by the timer expires first. The value of this attribute should be only positive instead of 0 when the type of the timer is WITHIN_MAX.
	unit	`unit:` timeIntervalUnitRef	A reference to a *TimeInternalUnit* for the time period of the timer.
`TimerType`		WITHIN or WITHIN_MAX or INTERVAL	An enumeration that lists the kinds (INTERVAL, WITHIN, WITHIN_MAX) of a timer. See also description of these kinds in the analysis of the *Timer* class.
`UnaryPatternOperatorType`		EVERY or NOT or REPEAT or WHEN	An enumeration that lists all possible kinds (EVERY, NOT, REPEAT, WHEN) of unary pattern operators. See also description of these kinds in the analysis of the *UnaryEventPattern* class.
`StatusType`		CRITICAL or WARNING or SUCCESS or FATAL	An enumeration that all possible status values (CRITICAL, WARNING, SUCCESS, FATAL) for an event instance. See also description of these kinds in the analysis of the *EventInstance* class.
Security Meta-Model			

`SecurityModel`		`security model` *name* `{`
		inner def ...
		`}`
			A concrete subclass of *Model* that represents a security model, i.e., a container of different security elements that can be used to define security requirements and capabilities.

	securityControls	(security control def ...) *	A containment reference to all security controls that can be defined in this security model.
	securityRequirements	(security requirement def ...) *	A containment reference to all security requirements that can be defined in this security model.
	securityProperties	(security property def ...) *	A containment reference to all security properties that can be defined in this security model.
	rawSecurityMetrics	(raw security metric def ...) *	A containment reference to all raw security metrics that can be defined in this security model.
	compositeSecurityMetrics	(composite security metric def ...) *	A containment reference to all composite security metrics that can be defined in this security model.
	rawSecurityMetricInstances	(raw security metric instance def ...) *	A containment reference to all raw security metric instances that can be defined in this security model.
	compositeSecurityMetricInstances	(composite security metric def ...) *	A containment reference to all composite security metric instances that can be defined in this security model.
	securityDomains	(security domain def ...) *	A containment reference to all security domains that can be defined in this security model.
	securityCapabilities	(security capability def ...) *	A containment reference to all security capabilities that can be defined in this security model.
	securitySLOs	(security slo def ...) *	A containment reference to all security SLOs that can be defined in this security model.

SecurityDomain		security model *id* {
		  inner def ...
		}
			A concrete class that represents a security domain (e.g., Application and Interface Security). A security domain can be further separated into security sub-domains (e.g., Application Security).

	id		A String attribute that defines the unique code for the security domain (e.g., AIS).
	name	name: 'string'	A String attribute that defines the name of the security domain (e.g., Application and Interface Security)
	subDomain	sub-domains [securityDomainRef (, securityDomainRef) *]	A reference to all security sub-domains that belong to this domain.

SecurityControl		security control *name* {
		  inner def ...
		}
			A concrete class that represents a security control, i.e., a high-level security capability or requirement. Such a security control is associated to other security elements (i.e., security properties and metrics) which enable associating it to more fine-grained

			security capabilities or requirements (i.e., security SLOs). Moreover, the monitoring of the satisfaction of the fine-grained security requirements that are associated with a security control can enable to also to derive the level of satisfaction of that security control. Such a monitoring can then enable the system to take adaptation / corrective actions to remedy for the non-satisfaction of a certain security control.
	name		A String attribute that represents the name of the security control.
	domain	`domain:` securityDomainRef	A reference to the domain in which the security control belongs.
	subDomain	`sub-domain:` securityDomainRef	A reference to the sub domain in which the security control belongs.
	specification	`specification:` 'string'	A String attribute used for textually describing the security control for human consumption purposes.
	securityProperties	`security properties [` securityPropertyRef `(,` securityPropertyRef`)*]`	A reference to all security properties that are associated with this security control.
	rawSecurityMetrics	`raw security metrics [` rawSecurityMetricRef `(,` rawSecurityMetricRef`)*]`	A reference to all raw security metrics that are associated with this security control.
	compositeSecurityMetrics	`composite security metrics [` compositeSecurityMetricRef `(,` compositeSecurityMetricRef`)*]`	A reference to all composite security metrics that are associated with this security control.

`RawSecurityMetricInstance`		`raw security metric instance` *`name`* `{`
		inner def ...
		`}`
			A concrete sub-class of *RawSecurityMetricInstance* that represents a raw security metric instance which inherits all respective attributes and properties of its parent class. More importantly, the reference to the respective sensor is inherited, which can be used to provide the measurements for this security metric instance.

`RawSecurityMetric`		`raw security metric` *`name`* `{`
		inner def ...
		`}`
			A concrete sub-class of *RawSecurityMetric* that represents a raw security metric which inherits all respective attributes and properties of its parent class, such as the direction of values and the unit of measurement. The rationale here is that any security metric is a special kind of metric which would then be associated to all measurement-oriented information that can be used to support its monitoring.

`SecurityProperty`		`property` *`name`* `{`	A concrete sub-class of *Property* which

		inner def ... }	represents a security property. Such a property is also associated to its respective security domain. This class could be instantiated to represent security properties that are abstract as otherwise there is another security property class which represents a measurable security property.
	domain	domain: securityDomainRef	A reference to the security domain to which the security property belongs

Certifiable		certifiable *name* {
		inner def ...
		}
			A concrete subclass of *SecurityProperty* which represents a measurable / certifiable security property that must be also associated to a certain security metric.

SecuritySLO		security slo *name* {
		inner def ...
		}
			A concrete subclass of *ServiceLevelObjective* representing security SLOs. Such SLOs are a special kind of SLOs which should be associated to conditions that only involve security metrics or properties.

SecurityCapability		security capability *name* {
		inner def ...
		}
			A concrete class that represents a high-level security capability, i.e., a certain capability of a cloud service provider to support a specific set of security controls.

	name		A String attribute that represents the name of the security capability.
	securityControls	controls [securityControlRef (, securityControlRef)*]	A reference to a set of security controls that are supported via this security capability.
	dataCenter	data center: dateCenterRef	A reference to a data centre of the cloud provider in which the security controls of the respective capability have been realised. As different capabilities might be implemented in different data centres of the same cloud provider, it is important to also keep an account of the particular data centre in which the security capability holds.

CompositeSecurityMetric		composite security *name* {
		inner def ...
		}
			A concrete sub-class of a *CompositeMetric* which represents a composite security metric. Such a metric inherits more importantly from its parent class the metric formula that can be exploited for its computation. Such formula will normally involve only composing security metrics.

CompositeSecurityMetricInstance		composite security metric instance *name* {
		inner def ...
		}
			A concrete subclass of *CompositeMetricInstance* which represents a composite security metric instance. Such an instance, through its parent class, is also associated with the metric contexts of the instances of the composing (security) metrics of its type (composite security metric).

Provider Meta-Model			

`ProviderModel`		`provider model` *`name`* `{`
		inner def ...
		`}`
			A concrete subclass of *Model* that represents the provider model of a certain cloud provider. Such a model is a container of all those elements that can be used to denote all the cloud service offerings supplied by this cloud provider.

	constraints	`(`constraint def ...`)*`	A reference to a set of *Constraint*s which represent constraints across the same or different features of the cloud provider that include as sub-constraints restrictions over the values of some attributes of these features.
	rootFeature	`root` feature def ...	A containment reference to the root feature of the cloud provider which could be considered as the overall cloud of that provider under which all offerings are provided.

`Attribute`		`attribute` *`name`* `{`
		inner def ...
		`}`
			A concrete class that represents an attribute of a certain feature. For example, such an attribute could represent the number of cores for a VM feature. Each attribute can have either a single fixed value or a value type denoting that the values of such attribute can vary (or both requiring then that the single value should belong to the value type but still signifying that the value of the *Attribute* is fixed).

	name		A String attribute that represents the name of the attribute.
	value	`value:` single value def ...	A containment reference to a *SingleValue*, i.e., a single fixed value that the *Attribute* takes for the feature that exhibits it. If also a value to the *valueType* attribute is provided, this means that the *value* attribute content (i.e., the single value) would then have to be member of this value type.
	valueType	`value type:` valueTypeRef	A reference to a *ValueType* denoting the domain of values for the *Attribute* concerned. When a value for this attribute is provided while no value is provided for the *value* one, this means that the Attribute's value can vary and it is the role of *AttributeConstraint*s to restrain it under a certain context (e.g., a VM flavour in case of VM features).
	unitType	`unit type:` *`UnitType`*	An attribute taking values from the *UnitType* enumeration which represents all possible concrete units that can be modelled. This attribute, when supplied, enables to associate the *Attribute* concerned with a certain unit of measurement (e.g., a storage

			size *Attribute* would have as a unit the GIGABYTES one)

`AttributeConstraint`		`attribute constraint` *name*
		{
		inner def ...
		}
			A concrete class that represents a constraint over two attributes of the same or different features. Such a constraint represents a mapping between a value from the first attribute to the value of the second. For instance, if both attributes refer to the same VM feature and if the first denotes the VM flavour and the second the VM number of cores, then the value "medium" for the first attribute will be mapped to the value of 2 for the second (highlighting that for the "medium" VM flavour of this provider, the number of cores is always 2).

	name		A String attribute that represents the name of the attribute constraint.
	from	`from:` attributeRef	A reference to the first *Attribute* of the constraint.
	to	`to:` attributeRef	A reference to the second *Attribute* (the mapped one) of the constraint.
	fromValue	`from value:` singleValueRef	A containment reference to the single value of the first attribute (the mapping one). Needless to say that this value should either be the fixed value of that attribute or otherwise belongs to the attribute's value type.
	toValue	`to value:` singleValueRef	A containment reference to the single value of the second attribute (the mapped one). Needless to say that this value should either be the fixed value of that attribute or otherwise belongs to the attribute's value type.
`Cardinality`			An abstract class that represents a cardinality restriction for features or groups of features. Such a restriction takes the form of a range of integers which could be open on its upper limit.
	cardinalityMin	`cardinality:` int .. int Note: syntax also refers to next attribute	An attribute of type integer that represents the lower bound of the *Cardinality*. The lowest possible value for this attribute is, logically speaking 0, indicating that the corresponding feature (or group of features) might not appear always in a certain offering of the cloud provider.
	cardinalityMax	please see above attribute syntax	An attribute of type integer that represents the upper bound of the *Cardinality*. If the value of this attribute is positive, then it should be greater that the value of the *minCardinality* one. On the other hand, a value of -1 signifies that the upper bound is

			open, going to positive infinity.

FeatCardinality		feature cardinality *name* {
		inner def ...
		}
			A concrete subclass of *Cardinality* that represents the default cardinality for a certain feature. This means that unless not otherwise stated, this feature should have this default cardinality (when offered to a certain client).

	value	value: int	An attribute of type integer that represents the actual default value of the feature cardinality. This values should not be less than the min cardinality and should not be greater than the max cardinality, if that cardinality is not open (positive infinity).

GroupCardinality		group cardinality *name* {
		inner def ...
		}
			A concrete subclass of *Cardinality* that represents the cardinality of a group of features.

Clone		clone *name* {
		inner def ...
		}
			A class that represents a clone of a certain feature. Such a clone can have sub clones.

	name		A String attribute that represents the name of a clone.
	subClones	sub-clones [cloneRef (, cloneRef)*]	A containment reference to all sub-clones of this clone.
Constraint			An abstract class that represents a certain constraint across the same or two different features. Different concrete kinds of such a constraint can be modelled. In any case, such a constraint is associated with a set of *Attribute* constraints which restrain the values that some *Attribute*s of the second feature can take depending on the values of some *Attribute*s of the first feature (where both features can also coincide as already stated).
	name		A String attribute that represents the name of the constraint.
	from	from: featureRef	A reference to the first feature of this constraint.
	to	to: featureRef	A reference to the second feature of this constraint.
	attributeConstraints	attribute constraints [attributeConstraintRef (, attributeConstraintRef)*]	A containment reference to a set of attribute constraints over these two features (or a sole one).

Exludes		excludes *name* {
		inner def ...
		}
			A concrete subclass of *Constraint* which denotes that the appearance of the first feature excludes by default the appearance of the second in a certain provider (bundle) offering. In other words, the two features are conflicting and can never be purchased

			together in a certain (bundle) offering of the cloud provider concerned. Needless to say that usually in this case there is no need to provide attribute constraints for this constraint.

Implies		implies *name* {
		inner def ...
		}
			A concrete subclass of *Constraint* which denotes that the appearance of the first feature implies the appearance of the second. In case that these two features are equal, then the implication is trivially moved to the attribute constraints belonging to this constraint.

Requires		requires *name* {
		inner def ...
		}
			A concrete subclass of *Constraint* with more restrictive semantics than *Implies*. In this case, the appearance of the first feature requires the appearance of the second. Additionally, both features are subject to specific cardinality and *Scope* restrictions. Cardinality restrictions would signify that a certain range cardinality for each feature should apply for this kind of constraint. Scope restrictions would signify that the scope of each feature is global or local. Global means that it maps to the level of the feature itself, while local means that it maps to the instance level, thus, holding for each instance of the feature at hand.

	scopeFrom	scope from: scopeRef	An optional reference to the *Scope* of the first feature.
	scopeTo	scope to: scopeRef	An optional reference to the *Scope* of the second feature.
	cardFrom	card from: featCardinalityRef	An optional reference to the cardinality of the first feature.
	cardTo	card to: featCardinalityRef	An optional reference to the cardinality of the second feature.

Functional		functional *name* {
		inner def ...
		}
			A concrete subclass of *Requires* which represents a more restrictive form of this super class. In this form, the appearance of the first feature can lead to applying a certain operator over a specific number of instances (see *value* attribute) of the second feature. Such an operator, belonging to the *Operator* enumeration, could be either *select* (i.e., select a certain number of instances of the second feature), *add* (i.e., require the addition of a certain number of instances of the second feature), *remove* (i.e., require the removal of a certain number of instances of the second feature), multiply (i.e., require the multiplication of the number of instances of the second feature with the *value*

			provided), *divide* (i.e., require the division of the number of instances of the second feature with the *value* provided).
	type	`type: Operator`	An attribute that takes values from the *Operator* enumeration denoting the operator to be applied on the number of instances of the second feature.
	value	`value:` int	An attribute of type integer which represents the second argument given to the operator apart from the current number of instances of the second feature (i.e., operator=add & value=2 would mean that we need to obtain 2 more instances for the second feature).

`Feature`		`feature` *name* `{`
		inner def ...
		`}`
			A concrete class that represents a feature of a cloud provider. Such a feature could represent, for instance, a VM that can be offered by that provider. Each feature is associated to a set of attributes that characterise it (e.g., could represent the number of cores, the memory size, the VM flavour, etc.). In addition, it can contain additional sub-features (e.g., location could be considered such a sub-feature). A feature has a certain feature cardinality which represents its default cardinality as well as the minimum and maximum instances of such a feature that could be exploited by each client of the corresponding cloud provider.

	name		A String attribute that represents the name of the feature.
	attributes	`attributes [` attributeRef `(,` attributeRef`)*]`	A containment reference to the list of *Attribute*s that belong to this feature and characterise it.
	subFeatures	`sub-features [` featureRef `(,` featureRef`)*]`	A containment reference to a list of sub-features of this feature.
	featureCardinality	feat cardinality def ...	A containment reference to the cardinality of this feature.
	clones	`clones [` cloneRef `(,` cloneRef`)*]`	A containment reference to the list of clones of this feature.

`Alternative`		`alternative` *name* `{`
		inner def ...
		`}`
			A concrete subclass of *Feature* that represents an alternative feature with a set of one or more variants (i.e., other features). This means that one or more of these variants could be exploited by a certain client of the cloud provider concerned. When the group cardinality is also specified for such a feature, then the number of variants that can be selected would need to conform to a certain range (i.e., from 3 to 5 variants).

	groupCardinality	feat cardinality def ...	A containment reference to the cardinality of the variants of this alternative feature.
	variants	`variants [` feature def ... `(,` feature def ...`)*]`	A containment reference to the variants of this feature.

`Exclusive`		`exclusive` *name* `{`
		inner def ...
		`}`
			A concrete subclass of *Alternative* that represents an exclusive feature with a set of one or more variants. Differently from an *Alternative* feature, in the case of an exclusive one, only one variant can be exclusively selected by a client of the cloud provider concerned. Needless to say that in this case, the cardinality of the group of variants included is not meaningful to be specified.

`Operator`		select or add or remove or multiply or divide	An enumeration that represents a list of concrete operators that can be applied in a *Functional* constraint over the current number of instances of the second feature and the respective integer value given for this constraint.
`Scope`			An abstract class that represents the scope of a feature. Two main concrete kinds of scope exist: (a) Product and (b) Instance.

`Instance`		`instance {`
		inner def ...
		`}`
			A concrete subclass of *Scope* which represents an instance / local scope. This means that the scope of a feature maps locally to each instance of such a feature.

	feature	`feature:` featureRef	A reference to the feature that maps to this local scope.
`Product`		`product`	A concrete subclass of *Scope* which represents a product / global scope. This means that the scope maps to the level of the feature itself and not to each instance of this feature.
Organisation Meta-Model			

`OrganisationModel`		`organisation model` *name* `{`
		inner def ...
		`}`
			A concrete subclass of *Model* that represents the model of a certain organisation. This model acts as a container of all sorts of information that needs to be modelled for the organisation concerned.

	organisation	`(`organisation def ...`)?`	A containment reference to the organisation concerned that is specified in this organisation model.
	provider	`(`provider def ...`)?`	A containment reference to the cloud provider concerned that is specified in this organisation model. As a cloud provider is considered as a special kind of an organisation, either the *organisation* or *provider* containment reference should be strictly provided in an organisation model.

	externalIdentifiers	(external identifier def ...) *	A containment reference to a set of external identifiers that can be used to distinguish / uniquely identify the users of an organisation in this organisation model.
	users	(user def ...) *	A containment reference to the users specified in this organisation model.
	userGroups	(user group def ...) *	A containment reference to the groups of users specified in this organisation model.
	dataCentres	(date centre def ...) *	A containment reference to the data centres specified in this organisation model. Such data centres should only be provided when a cloud provider is concerned and not any other kind of an organisation.
	roles	(role def ...) *	A containment reference to the roles specified in this organisation model.
	roleAssignments	(role assignment def ...) *	A containment reference to the role assignments specified in this organisation model.
	permissions	(permission def ...) *	A containment reference to the permissions specified in this organisation model.
	securityLevel	security level: *SecurityLevel*	An attribute that takes values from the *SecurityLevel* enumeration which denotes the different security levels that might be applied over the access to the resources of the organisation concerned. A LOW security level would indicate that the organisation enables other users to see most of its models apart from the private ones (i.e., only the organisation model itself for that organisation). A MEDIUM security level would enable the organisation to provide access restrictions to some model kinds that could be considered sensitive. A HIGH security level would indicate that no model of that organisation is visible to any other organisation within any instance of the platform involved.
	resourceFilters	(resource filter def ...) *	A containment reference to the resource filters specified in this organisation model.
Credentials			An abstract class that represents any kind of credentials. For CAMEL, two concrete kinds of credentials have been captured: (a) platform credentials which enable a user to authenticate against the platform concerned; (b) cloud credentials which allow the platform to perform actions over a specific cloud on behalf of a certain user.

CloudCredentials		*name* {
		inner def ...
		}
			A concrete subclass of *Credentials* which represents credentials that can be used to authenticate users over a certain cloud. Such

			credentials can take the form of a login and password or of public and private keys. These credentials can be exploited by the platform in order to handle the cloud-specific deployments of user application components in the cloud of the specific cloud provider concerned.
	name		A String attribute that represents the name of the credentials (as a unique identifier).
	cloudProvider	`cloud provider:` cloudProviderRef	A reference to the cloud provider for which the credentials hold.
	securityGroup	`security group:` 'string'	A String attribute that represents the security group in which the authentication of the user can be performed
	publicSSHKey	`public SSH key:` 'string'	A String representation of the public SSH key of the user in the cloud of the provider concerned.
	privateSSHKey	`private SSH key:` 'string'	A String representation of the private SSH key of the user in the cloud of the provider concerned.
	username	`username:` 'string'	A String representation of the username for authenticating the user in the cloud of the provider concerned.
	password	`password:` 'string'	A String representation of the password for authenticating the user in the cloud of the provider concerned.

`DataCenter`		`data centre` *`name`* `{`
		inner def ...
		`}`
			A concrete class that represents a data centre of the cloud provider concerned. Such a data centre has a specific name and code and is situated in a specific location, which can be physical or cloud-based. Please note that the modelling of this class is essential in order to capture some differentiation in the pricing, offering and security levels across different data centres in the same cloud which can enable the more precise (re-)configuration of an application that can be supported by a certain platform.

	name		A String attribute representing the name of the data centre.
	codeName	`code name:` 'string'	A String attribute representing the code name of this data centre.
	location	`location:` locationRef	A reference to the *Location* of the data centre.
`Entity`			An abstract class that represents any entity which can be either an organisation (and a cloud provider being a subclass of organisation) or a user.
`Organisation`		`organisation` *`name`* `{`	A concrete class of *Entity* which represents a

		inner def ... }	certain organisation. Such an organisation is characterised by various information fields out of which the *name* and *email* are those that can uniquely identify it.
	name		A String attribute that represents the name of the organisation.
	www	`www:` 'string'	A String attribute that represents the URL of the official website of the organisation concerned.
	postalAddress	`postal address:` 'string'	A String attribute that represents the postal address of the organisation (normally of the eadquarters).
	email	`email:` 'string'	A String attribute that represents an email address of that organisation for contact purposes (e.g., contact@<orgname>.com).

`CloudProvider`		`provider` *name* {
		inner def ...
		}
			A concrete subclass of *Organisation* that represents a cloud provider. Such kind of organisation is characterised by additional information which spans the type of the cloud offered, the kinds of cloud services supplied, the provider model and security capabilities exhibited this organisation kind.

	public	(Public)?	A boolean attribute that indicates whether this provider offers a public or a private cloud.
	SaaS	(SaaS)?	A boolean attribute that indicates whether this provider offers SaaS services.
	PaaS	(PaaS)?	A boolean attribute that indicates whether this provider offers PaaS services.
	IaaS	(IaaS)?	A boolean attribute that indicates whether this provider offers IaaS services.
	providerModel	`provider model:` providerModelRef	A reference to the *ProviderModel* of this cloud provider.
	securityCapability	`security capability [` securityCapabilityRef `(,` securityCapabilityRef`)*]`	A reference to the security capabilities realised by this cloud provider (which apply across one or more of its data centres).

`User`		`user` *name* {
		inner def ...
		}
			A concrete subclass of *Entity* which represents a user of the organisation at hand. Such a user is characterised by some unique information like its identification name in the platform and its email as well as by references to particular CAMEL elements that have been either generated / modelled by that user or represent additional identification means for this user (e.g., credentials or external identifiers).

	name		A String attribute that represents a unique identifier in the platform for the user at hand.

	email	email: 'string'	A String attribute that represents the email of the user.
	firstName	first name: 'string'	A String attribute that represents the first name of the user.
	lastName	last name: 'string'	A String attribute that represents the last name of the user.
	www	www: 'string'	A String attribute that represents the URL of the personal web page of the user.
	externalIdentifiers	external identifiers [externalIdentifierRef (, externalIdentifierRef)*]	A containment reference to external identifiers that can distinguish the user.
	requirementModels	requirement models [requirementModelRef (, requirementModelRef)*]	A reference to the *RequirementModel*s that have been specified by this user.
	cloudCredentials	cloud credentials [cloud credentials def ... (,cloud credentials def ...)*]	A containment reference to the cloud credentials of the user which can be exploited by the platform to perform deployments of the user application components in specific clouds.
	deploymentModels	deployment models [deploymentModelRef (, deploymentModelRef)*]	A reference to the *DeploymentModel*s that have been specified by the user.
	paasageCredentials	paasage credentials paasage credentials def ...	A containment reference to the credentials of the user with respect to the platform exploited.

ExternalIdentifier		external identifier *identifier* {
		inner def ...
		}
			A concrete class that represents an external identifier that can be used to distinguish a certain user.

	identifier		A String attribute that represents the value of the external identifier.
	description	description: 'string'	A String attribute that further provides additional information about the identifier.

Permission		permission *name* {
		inner def ...
		}
			A concrete class which represents an access control permission. Such a permission enables a certain role to perform a specific action over a certain resource or sets of resources. The identification of resources is achieved via a *ResourceFilter*. Resources can be models or services offered by the platform. A permission holds for a certain validity period, denoted by a starting and ending date, and then it should be revoked.

	name		A String attribute representing the name of the permission.
	role	role: roleRef	A reference to the *Role*, i.e., the subject of the permission which is allowed to perform the corresponding action to the resources

			identified.
	startTime	`start:` date	A date attribute that denotes the starting time for the validity of the permission.
	endTime	`end:` date	A date attribute that denotes the end time for the validity of the permission which signifies then that the permission should be revoked.
	resourceFilter	`resource filter:` resourceFilterRef	A reference to a resource filter which identifies a set of resources on which the permission holds.
	action	`action:` action def ...	An attribute referring to the action allowed to be performed on the identified resources by this permission. Currently, only the READ and WRITE members of the *ActionType* enumeration are allowed to be provided as values for this attribute mapping to the most widely used action kinds specified in access control permissions.
`SecurityLevel`		LOW or MEDIUM or HIGH	An enumeration that lists concrete security levels that can apply for a particular organisation. Each security level maps to a certain access level over the models owned by the organisation. For instance, a LOW security level enables any external entity to read all models of the organisation except the organisation model one.
`ResourcePattern`		EXACT or TREE	An enumeration that lists certain resource patterns. Such patterns include: EXACT (an exact resource pattern usually denoting a certain path in which the respective resources are situated) and TREE (a tree-based resource pattern which is associated to a certain path - the semantics is that any descendant sub-path from that path, including the directory denoted by that path, is selected such that all resources residing in it apply for the corresponding permission). For instance, when resource path is: "/myOrg" and the pattern is EXACT, then all resources under this path are selected. On the other hand, for the same path, if the pattern is TREE, then the resources on both the path and its descendants (e.g., "/myOrg/applications") are selected.
`ResourceFilter`			An abstract class that denotes a resource filter, i.e., a selection filter for one or more resources that can be exploited in the specification of a certain permission.
	name		A String attribute that represents the name of the resource filter.
	resourcePattern	`pattern:` *`ResourcePattern`*	An attribute that takes values from the

			ResourcePattern enumeration. Such a resource pattern is the main driver for the selection of the resources in the filter along with the respective path or URL that is provided in the concrete kinds of resource filters.

`InformationResourceFilter`		`information resource filter` *name* `{`
		inner def ...
		`}`
			A concrete subclass of *ResourceFilter* which denotes a filter for information resources. Such a filter mainly includes the specification of the resource path in the model repository where the corresponding resources to be selected reside. In case that the inherited resource pattern is TREE, then not only the path's resources are selected but also the resources under all the descendants sub-paths of this path.

	informationResourcePath	`information resource path:` 'string'	A String attribute that represents a resource path (e.g., "/organisations/myOrg") (in a model repository).
	everyInformationResource	`(`all`)?`	A convenience boolean attribute that, when is equal to true, signifies that all information resources are selected (in the model repository). In that case, there is no point in providing a value for the *informationResourcePath* attribute.

`ServiceResourceFilter`		`service resource filter` *name* `{`
		inner def ...
		`}`
			A concrete subclass of *ResourceFilter* which denotes a filter over service resources. Such a filter mainly includes the specification of a (service) URL for which a permission should hold. Such a URL could map to either a specific service method, to the service itself or to all services available in the corresponding server identified by that URL. This depends on the URL provided and how deep it goes in the service tree. In conjunction also with the resource pattern inherited, interesting combinations can be modelled. For instance, suppose that we need to provide a permission for all the services hosted by a specific server. Further suppose that the URL of the server is: "http://www.myorg.com:8080". If then the resource pattern is TREE, all the services hosted by the server are selected (logically speaking being deployed in the servlet container mapping to this URL). As another example, suppose that we need to select a certain method called m1 of service s1. Then, we would need to specify the exact URL of the method: " http://www.myorg.com:8080/s1/rest/m1" as well as the EXACT resource pattern.

	serviceURL	`service url:` 'string'	A String attribute that represents the URL of the service(s) to be selected or its (their) methods.
	everyService	`(`all`)?`	A convenience boolean attribute that denotes that all services should be selected (for a certain platform). In that case, the serviceURL does not need to be provided.

`Role`		`role` *`name`* `{`
		inner def ...
		`}`
			A concrete class that represents a role, i.e., a certain grouping of users that pertains to certain sets of responsibilities and rights. By assigning users to roles, then RBAC-based access control policies can be specified as actually being denoted by the *Permission* class.

	name		A String attribute that represents the name of the role.

`RoleAssignment`		`role assignment` *`name`* `{`
		inner def ...
		`}`
			A concrete class that represents the assignment of a role to a certain user or user group that pertains to a certain validity period after which the assignment needs to be revoked.

	name		A String attribute that represents the name of the role assignment.
	user	`user:` userRef	A reference to the user for which the assignment holds.
	role	`role:` roleRef	A reference to the role assigned to the user or user group.
	userGroup	`user group:` userGroupRef	A reference to the user group for which the assignment holds.
	startTime	`start:` date	An attribute of type date that denotes the starting time of the assignment's validity period.
	endTime	`end:` date	An attribute of type date that denotes the end time of the assignment's validity period. This end time should not be less than the starting time of the assignment, if both dates are provided.
	assignmentTime	`assigned on:` date	An attribute of type date that denotes the exact time that the assignment was submitted to the authentication and authorisation system. Logically speaking, the *assignmentTime* should be less or equal to the *startTime* of the assignment, if both dates are provided.

`UserGroup`		`user group` *`name`* `{`
		inner def ...
		`}`
			A concrete class that represents a group of users on which a certain role could be assigned. As such, this actually represents a convenient and compact way of assigning certain common rules to a specific group of

			users.
	name		A String attribute that represents the name of the user group.
	users	`users [` userRef `(,` userRef`)*]`	A reference to one or more users that belong to this group.

`PaaSageCredentials`		`paasage credentials` *password* `{`
		inner def ...
		`}`
			A concrete subclass of Credentials that represents platform-specific credentials which could be exploited in order to uniquely authenticate the user across all entry points and services offered by the platform.

	password		A String attribute that represents the actual password of the user platform credentials. The logic part of such credentials is the actual name of the user (see *User* class).
Location Meta-Model			

`LocationModel`		`location model` *name* `{`
		inner def ...
		`}`
			A concrete subclass of *Model* that represents a location model. It constitutes a container for all kinds of locations that can be specified by this meta-model.

	cloudLocations	`(`cloud location def ...`)*`	A containment reference to all cloud locations specified in this model.
	countries	`(`country def ...`)*`	A containment reference to all cloud countries specified in this model.
	regions	`(`geographical region def ...`)*`	A containment reference to all regions specified in this model.
`Location`			An abstract class that denotes any kind of location. Two specific kinds of locations can be then specified, being subclasses of this class: geographical regions (continents, subcontinents & countries) and cloud locations.
	id		A String attribute that represents the unique identifier of this location. Such an identifier could, for instance, be an ISO code for a certain country (e.g., GE denoting the country of Germany).

`CloudLocation`		`cloud location` *id* `{`
		inner def ...
		`}`
			A concrete class of *Location* that represents a cloud-specific location. Such a location can also have cloud location descendants thus enabling to specify an hierarchy of cloud locations, like the one that can be defined, e.g., for Amazon AWS. A cloud location within a certain hierarchy can also be associated with a certain physical location represented by the *GeographicalRegion* class. Once this is done, then automatically all descendants of that location are also associated with this physical location.

	isAssignable	(assignable)?	A boolean attribute indicating whether such a location can be assigned to a certain resource.
	subLocations	sub-locations [cloud location def ... (, cloud location def ...)*]	A containment reference to the child cloud locations of this location.
	parent	parent: cloudLocationRef	A reference to the parent of this cloud location, if it exists (i.e., it is not a root / top one).
	geographicalRegion	geographical region: geographicalRegionRef	A reference to the physical location (*GeographicalRegion*) in which the cloud location is situated.

GeographicalRegion		geographical region *id* {
		inner def ...
		}
			A concrete subclass of *Location* that represents a physical location in terms of a geographical region, such as a continent or a sub-continent.

	name	name: 'string'	A String attribute that represents the name of the physical location / geographical region.
	parentRegions	parent regions [geographicalRegionRef (, geographicalRegionRef)*]	A reference to the parent regions of this region, if they exist (thus more prominent for sub-continents which have as a parent whole continents).
	alternativeNames	alternative names ['string' (, 'string')*]	A list of alternative names for this geographical region. Such a list could, for instance, include the name of this region in different spoken languages.

Country		country *id* {
		inner def ...
		}
			A concrete subclass of *GeographicalRegion* that represents a certain country.

Type Meta-Model			

TypeModel		type model *name* {
		inner def ...
		}
			A concrete subclass of *Model* which represents a type model, i.e., a container of values and value types.

	dataTypes	(value type def ...)*	A containment reference to all the value types defined in this model.
	values	(value def ...)*	A containment reference to all the values defined in this model.

Limit		limit {
		inner def ...
		}
			A concrete class that represents an upper or lower bound for a *Range*. For such a bound the actual value is specified along with an indication if this value is to be included or not in the corresponding range.

	included	(included)?	A boolean attribute which indicates whether the value of this limit should be included in the range encompassing this limit.
	value	value def ...	A containment reference to the numeric value of this bound / limit.

TypeEnum		IntType or StringType or BooleanType or FLoatType or DoubleType	An enumeration that lists all basic types for single values (e.g., StringType and IntegerType).
SingleValue			An abstract class that represents a single value. Various concrete kinds of single values are also captured, such as *EnumerateValue*s and *IntegerValue*s.
BoolValue		boolean value boolean value def ...	A concrete class of *SingleValue* that represents a boolean value.
	value		An attribute of type boolean representing the actual boolean value of this single value kind.
EnumerateValue		*name* : *value*	A concrete class of *SingleValue* that represents an enumeration value / member which comprises two main parts: (a) the name of the member and (b) its actual integer value.
	name		A String attribute that represents the name of the enumeration member.
	value		An attribute of type integer representing the value of this enumeration member.
NumericValue			An abstract subclass of *SingleValue* that represents all kinds of numeric values (e.g., *IntegerValue*s)
IntegerValue		int value integer value def ...	A concrete class of *NumericValue* that represents an integer value.
	value		An attribute of type integer representing the actual integer value of this numeric value kind.
FloatsValue		float value float value def ...	A concrete class of *NumericValue* that represents a float value.
	value		An attribute of type float representing the actual float value of this numeric value kind.
DoublePrecisionValue		double value double precision value def ...	A concrete class of *NumericValue* that represents a double (precision) value.
	value		An attribute of type double representing the actual double precision value of this numeric value kind.
NegativeInf		negative infinity	A concrete subclass of *NumericValue* which represents the negative infinity and can be exploited to specify the lower bounds of ranges (i.e., semi- or open ranges).
PositiveInf		positive infinity	A concrete subclass of *NumericValue* which represents the positive infinity and can be exploited to specify the lower bounds of ranges (i.e., semi- or open ranges).
ValueToIncrease		value to increase numeric	A concrete subclass of NumericValue which

		value def ...	represents a value that can be increased.
	value		A reference to the numeric value that can be increased.
`StringsValue`		`string value` string value def ...	A concrete class of *SingleValue* that represents a String value.
	value		An attribute of type String representing the actual String value of this single value kind.
`ValueType`			An abstract class that represents any kind of value type. More concrete value types are also captured which are made subclasses of this class.
	name		A String attribute that represents the name of the value type.
`BooleanValueType`		`boolean value type` *name* `{` inner def ... `}`	A concrete subclass of *ValueType* that represents all booleans. Its primitive type is by default mapped to BooleanType.
	primitiveType	`primitive type:` *TypeEnum*	An attribute that denotes the (fixed) primitive type of this value type.
`Enumeration`		`enumeration` *name* `{` inner def ... `}`	A concrete subclass of *ValueType* that represents an enumeration that comprises a list of enumeration values.
	values	`values [` enumerate value def ... `(,` enumerate value def ...`)*]`	A containment reference to all *EnumerateValue* members of this enumeration.
`List`		`list` *name* `{` inner def ... `}`	A concrete subclass of *ValueType* that represents a list. Such a list can uniformly contain a list of values of the same type, which can be either primitive or a certain value type. By allowing the type of a value to be a *ValueType*, then we enable a list to contain values that conform to such a value type, e.g., a certain (integer) range (rather than taking values from the whole integer set).
	values	`values [` single value def ... `(,` single value def ...`)*]`	A containment reference to all (single) values contained in this list.
	primitiveType	`primitive type:` *TypeEnum*	An attribute taking values from the TypeEnum enumeration, i.e., mapping to a certain primitive type. It should not be used in conjunction with the *type* property.
	type	`type:` valueTypeRef	A reference to the *ValueType* of the values contained in this list. It should not be used in conjunction with the *primitiveType* attribute.
`Range`		`range` *name* `{` inner def ... `}`	A concrete subclass of *ValueType* that represents a (numeric) range. Such a range has an upper and lower bound which can be open (mapping to positive and negative infinity, respectively). Its contained values

			also conform to a certain primitive type.
	lowerLimit	`lowerLimit:` limitRef	A containment reference to the lower *Limit* of this range.
	upperLimit	`upperLimit:` limitRef	A containment reference to the uper *Limit* of this range.
	primitiveType	`primitive type:` *TypeEnum*	An attribute of type *TypeEnum* which denotes the primitive type of the values contained in this range.

RangeUnion		`range union` *name* `{`
		inner def ...
		`}`
			A concrete subclass of *ValueType* that represents a union of (numeric) ranges. Such a union should not be continuous and thus needs to include non-overlapping ranges (i.e., ranges that do not share any value) of the same primitive type.

	ranges	`ranges [` range def ... `(,` range def ...`)*]`	A containment reference to the list of ranges over which the corresponding union is constructed.
	primitiveType	`primitive type:` *TypeEnum*	An attribute of type *TypeEnum* denoting the primitive type of this range union and of all the ranges that are included in it.

StringValueType		`string value type` *name* `{`
		inner def ...
		`}`
			A concrete subclass of *ValueType* that represents all strings. Its primitive type is by default mapped to StringType.

	primitiveType	`primitive type:` *TypeEnum*	An attribute that denotes the (fixed) primitive type of this value type.
Unit Meta-Model			

UnitModel		`unit model` *name* `{`
		inner def ...
		`}`
			A concrete subclass of *Model* which represents a model of units.

	units	`(`unit def ...`)*`	A containment reference to all units defined in this model.
Unit			An abstract class that represents any kind of unit. Subclasses of this class are concrete unit kinds, such as *TimeIntervalUnit*s and *MonetaryUnit*s.
	name		A String attribute that represents the name of the unit.
	unit	`unit:` *UnitType*	An attribute taking values from the *UnitType* enumeration which includes all concrete units that can be exploited, such as SECONDS and BYTES.

CoreUnit		`core unit {`
		name `:` *unit*
		`}`
			A concrete subclass of *Unit* representing the unit of cores. For such a unit, the *unit* attribute can take only the value of CORES (number of cores).

Dimensionless		`dimensionless {`
		name `:` *unit*
		`}`
			A concrete subclass of *Unit* representing a dimensionless unit. For such a unit, the *unit* attribute can take only the value of RATIO

			and PERCENTAGE.

MonetaryUnit		`monetary unit {`
		`  name : unit`
		`}`
			A concrete subclass of *Unit* representing a monetary unit. For such a unit, the *unit* attribute can take only the value of EUROS, DOLLARS and POUNDS.

RequestUnit		`request unit {`
		`  name : unit`
		`}`
			A concrete subclass of *Unit* representing a request unit. For such a unit, the *unit* attribute can take only the value of REQUESTS (i.e., number of requests).

StorageUnit		`storage unit {`
		`  name : unit`
		`}`
			A concrete subclass of *Unit* representing a storage unit. For such a unit, the *unit* attribute can take only the value of BYTES, KILOBYTES, MEGABYTES and GIGABYTES.

ThroughputUnit		`throughput unit {`
		`  name : unit`
		`}`
			A concrete subclass of *Unit* representing a unit of (network or processing) throughput. For such a unit, the *unit* attribute can take only the value of REQUESTS_PER_SECOND, TRANSACTIONS_PER_SECOND and BYTES_PER_SECOND.

TimeIntervalUnit		`time interval unit {`
		`  name : unit`
		`}`
			A concrete subclass of *Unit* representing a time interval unit. For such a unit, the *unit* attribute can take only the value of MILLISECONDS, SECONDS, MINUTES, HOURS, DAYS, WEEKS, and MONTHS.

TransactionUnit		`transaction unit {`
		`  name : unit`
		`}`
			A concrete subclass of *Unit* representing a transaction unit. For such a unit, the *unit* attribute can take only the value of TRANSACTIONS.

UnitType		BYTES or KILOBYTES or MEGABYTES or GIGABYTES or EUROS or DOLLARS or POUNDS or MILLISECONDS or SECONDS or MINUTES or HOURS or DAYS or WEEKS or MONTHS or REQUESTS or REQUESTS_PER_SECOND or TRANSACTIONS or TRANSACTIONS_PER_SECOND or BYTES_PER_SECOND or PERCENTAGE or CORES	An enumeration that list all possible concrete units that can be modelled.
UnitDimensionType			An enumeration that represents all types of unit dimensions (e.g., STORAGE, COST).
Execution Meta-Model			

`ExecutionModel`		`execution model name {`
		`  inner def ...`
		`}`
			A concrete subclass of *Model* that represents an execution model. Such a model acts as a container for all elements that can be specified for modelling the execution history of one or more user applications.

	actionRealisations	`(action realisation def ...)*`	A containment reference to a list of all (scale) action realisations specified in this model.

	eventInstances	(event instance def ...) *	A containment reference to a list of all event instances specified in this model.
	executionContexts	(execution context def ...) *	A containment reference to a list of all execution contexts specified in this model.
	measurements	(measurement def ...) *	A containment reference to a list of all measurements specified in this model.
	sloAssessments	(slo assessment def ...) *	A containment reference to a list of all (scale) SLO assessments specified in this model.
	ruleTriggers	(rule trigger def ...) *	A containment reference to a list of all rule triggers specified in this model.

ActionRealisation		action realisation *name* {
		inner def ...
		}
			A concrete class that represents an instance of a (usually scaling) action that has been executed in the context of a scaling rule triggering. For such an action, it is specified its start and end time as well as the lower level actions implementing it, as it can be differentiated depending on the respective cloud that it is executed.

	name		A String attribute representing the name of the action realisation concerned.
	action	action: *ActionType*	An attribute taking values from the ActionType enumeration thus indicating the exact action that has been executed.
	startTime	start time: date	A date attribute that represents the start time of the action execution.
	endTime	end time: date	A date attribute that represents the end time of the action execution.
	lowLevelActions	low level actions: 'string'	A String attribute that denotes the low level actions at the provider side that have been performed in the context of executing this action.

ExecutionContext		execution context *name* {
		inner def ...
		}
			A concrete class which represents a certain deployment episode for a specific application. Such an episode starts when the application starts being deployed and ends when the application deployment is stopped. This deployment episode is associated with a certain (deployment) cost (which dynamically should change while the application is running). It is also connected with the deployment model actually enforced as well as the requirements that have led to the concretisation of this deployment model. The latter information can be exploited not only for traceability reasons but also for enabling correlating together different deployments of the same or different applications in order to support any kind of related analysis (e.g., best deployment discovery for a certain application). The execution context is also

			related to the actual measurements produced for the deployed application at hand which also facilitates such kinds of analysis. Finally, the execution context is associated with all the rules that have been triggered in the context of the same deployment episode.
	name		A String attribute that represents the name of the execution context.
	application	`application:` applicationRef	A reference to the application correlated to this execution context.
	startTime	`start time:` date	A date attribute that denotes the start time of the execution context.
	endTime	`end time:` date	A date attribute that denotes the end time of the execution context.
	totalCost	`total cost:` double	An attribute of type double that denotes the total cost of the application provisioning (deployment and execution) associated with this execution context. While the application deployment has not been ended, this cost is the running cost. Otherwise, the cost is the final cost of the application provisioning.
	costUnit	`cost unit:` monetaryUnitRef	A reference to the monetary unit of the application provisioning cost associated with this deployment episode.
	deploymentModel	`deployment model:` deploymentModelRef	A reference to the application deployment model in effect for this deployment episode.
	requirementGroup	`requirement group:` requirementGroupRef	A reference to the group of requirements that led to the production of the application's concrete deployment model executed under this deployment episode.
`Measurement`			A class that represents any kind of measurement, such as measurements associated with VMs, internal components, whole applications and component communications. Apart from classical information, such as the value and timestamp of the measurement, this class also requires to specify additional information, such as the execution context in effect and the metric instance involved in the production of this measurement.
	name		A String attribute representing the name of the measurement.
	executionContext	`execution context:` executionContextRef	A reference to the execution context under which this measurement has been produced.
	metricInstance	`metric instance:` metricInstanceRef	A reference to the metric instance that was involved in the production of this measurement.

	value	`value:` double	The actual value of the measurement as a double.
	rawData	`raw data:` 'string'	A String attribute that refers to a pointer to raw (measurement) data which were exploited in the aggregation that resulted in this measurement.
	measurementTime	`measurement time:` date	The actual timestamp of the measurement as an EDate.
	slo	`slo:` serviceLevelObjectiveRef	A reference to the SLO that needs to be evaluated against this measurement - this mainly applies for measurements mapping to metrics that are directly involved in the conditions of these SLOs.
	eventInstance	`event instance:` eventInstanceRef	A reference to the instance of the event that has been generated due to the production of this measurement (e.g., in the case of the violation of a certain metric condition mapping to the production of a corresponding non-functional event instance).

`ApplicationMeasurement`		`application measurement` *`name`* `{`
		inner def ...
		`}`
			A concrete subclass of *Measurement* which represents an application measurement.

	application	`application:` applicationRef	A reference to the application that has been measured.

`InternalComponentMeasurement`		`internal component measurement` *`name`* `{`
		inner def ...
		`}`
			A concrete subclass of *Measurement* which represents an internal component measurement.

	internalComponentInstance	`internal component instance:` internalComponentInstanceRef	A reference to the actual instance of the internal application component that has been measured.

`VMMeasurement`		`vm measurement` *`name`* `{`
		inner def ...
		`}`
			A concrete subclass of *Measurement* which represents a VM measurement.

	vmInstance	`vm instance:` vmInstanceRef	A reference to the actual instance of the VM that has been measured.

`CommunicationMeasurement`		`resource coupling measurement` *`name`* `{`
		inner def ...
		`}`
			A concrete subclass of *Measurement* which represents a communication / network measurement. Differently from the other concrete kinds of measurement, such a measurement needs to be correlated with two instances: an instance of the source and target VM communicating.

	sourceVMInstance	`source vm instance:` vmInstanceRef	A reference to the instance of the source VM involved in the communication.
	destinationVMInstance	`destination vm instance:` vmInstanceRef	A reference to the instance of the destination / target VM involved in the communication.

SloAssessment		slo assessment *name* {
		inner def ...
		}
			A concrete class that represents a certain SLO assessment that has been performed upon the production of a relevant measurement (i.e., mapping to a metric which directly participates in the condition of this SLO). Such an assessment is trivially associated to the corresponding SLO. It is also connected with the execution context in which it has been performed. It is also related with the measurement that led to its production.

	name		A String attribute that represents the name of this assessment.
	slo	slo: serviceLevelObjectiveRef	A reference to the SLO that has been assessed.
	assessment	(violated)?	The result of the assessment as a boolean value that indicates whether the SLO has been violated or not.
	executionContext	execution context: executionContextRef	A reference to the execution context in which the SLO assessment has been performed.
	measurement	measurement: measurementRef	A reference to the measurement that led to the production of this SLO assessment.
	assessmentTime	assessment time: date	A date attribute that denotes the exact time point when this SLO assessment has been performed.

RuleTrigger		rule trigger *name* {
		inner def ...
		}
			A concrete class that represents the actual triggering of a scalability rule within a certain execution context / deployment episode.

	name		A String attribute that represents the name of the rule trigger.
	scalabilityRule	rule: scalabilityRuleRef	A reference to the scalability rule being triggered.
	eventInstances	event instances [eventInstanceRef (, eventInstanceRef)*]	A reference to the event instances that led to the triggering of the scalability rule concerned.
	actionRealisations	action realisations [actionRealisationRef (, actionRealisationRef)*]	A reference to the actual actions that have been executed upon the rule triggering.
	triggeringTime	triggering time: date	A date attribute that represents the exact time point where the scalability rule has been triggered.
	executionContext	execution context: executionContextRef	A reference to the execution context under which the rule triggering has occurred.

Here comes now the explanation of the notation used in this document for referring to the CAMEL textual syntax for all the CAMEL elements analysed.

Table 2. Explanation of the textual syntax notation used in this document

Notation	**Explanation**

class *name* {
inner def ...
}
	The outer definition of a CAMEL class is provided here. "class" maps to the class name in the CAMEL meta-model while *name* refers to the name that is provided by the user when creating an instance of this class. "inner def ..." means that there is an inner definition of that class (instance) which relies on inserting one or more lines that map to the values that the attributes and properties involved in that class take.

class *name* typed class2Ref {
inner def ...
}
	Similar as in the previous case with the sole exception that we also define here the type of the instance-based class specified here. This type maps to referring to a second class (see class2Ref).

element def ...	A containment reference to one element. The definition of that element is provided inline in the definition of the containing element.
(element def ...)*	A containment reference to zero or more elements. The definition of that elements is provided inline in the definition of the containing element.
(element def ...)?	A containment reference to at most one element. The definition of that element is provided inline in the definition of the containing element.
propertyName [elementRef (, elementRef)*]	This represents the values that a certain property named as "propertyName" takes for the current class instance definition. Such values refer to one or more references to a certain element. For better comprehension, let us provide a simple example: event instances [eventInstanceRef (, eventInstanceRef)*]. This represents the values that the "eventInstances" property of the "RuleTrigger" class takes. Such values map to referring to one or more instances of events. A reference to an instance of a certain class named as "element" is represented by "elementRef" in Verdana. The notation (,elementRef)* means that zero or more ",elementRef" values can be provided. For instance, if two need to be supplied for the current example, then the representation of the respective property would take the following form: event instances [eventInstanceRef1, eventInstanceRef2]. [] always denotes a list. Please also note that references in CAMEL textual syntax take the form of globalContainerName.(localContainerName.)*.elementName. For instance, if we need to refer to a certain metric named as ResponseTime and this metric belongs to metric model MM1 of CAMEL model CM, then the reference to that metric will taken the form of: CM.MM1.ResponseTime.
propertyName [element def ... (, element def ...)*]	This is a similar case to the previous one. The sole difference is that now we deal with a containment reference. As such, the definition of each element referenced, denoted by " element def ...", should be inline inside the list definition.
propertyName: elementRef	Simpler case that the two previous ones. Instead of referring to multiple, only one element is referred by

	the current property at hand.
propertyName: element def ...	Similar case to the previous one with the sole exception that we refer to an inline definition of the element referenced as we are dealing here with a containment reference.
(propertyName: elementRef)?	Same case as in "propertyName: elementRef" where now we denote that the property might be defined but may also be not. Symbol ? means 0 to 1 occurrence for the content inside the parenthesis.
(propertyName: element def ...)?	Similar case to the previous one with the sole exception that we refer to an inline definition of the element referenced as we are dealing here with a containment reference.
attribute: basicType	Here we denote that the attribute "attribute" will take a value of basicType (e.g., int, double, or string). For example, an attribute of "description" would take a value of 'string'. As non-alphanumeric characters might be provided, it is better to always enclose the actual string provided with quotes (').
attribute: *ValueEnum*	Here we denote that the attribute "attribute" will take a value from the ValueEnum enumeration.
(attribute: basicType)?	Same as in the case of "attribute: basicType" where we just additionally indicate that this row might or might not be provided.
(attribute: *ValueEnum*)?	Same as in the case of "attribute: *ValueEnum*" where we just additionally indicate that this row might or might not be provided.
attribute [*type*(,type)*]	Here we have the case where we provide multiple values of the "attribute" attribute where "type" can be a basic type or an enumeration. Such multiple values are represented via the [], list symbol, which include a comma separated values conforming to the "type".
attribute: basicType .. basicType	Here we specify a range of values (from lower to higher) for the "attribute" attribute which are of a basicType. For instance, the number of cores required for a VM could be expressed as: "core: 1 .. 3" (i.e., the number of cores should be between 1 and 3.
attribute: basicType .. (basicType)?	Same as in previous case where we also maintain the possibility that the upper value in the range might not be provided.
(name)?	Here we denote that the true value of a boolean attribute, named as "isName", might be provided. Absence of the corresponding value/row means that the attribute takes the false value.
X or Y or Z or ...	Here we just list all possible values for a certain enumeration. Please be aware here of the "or" operator which just indicates that one of the values from the enumeration is always mapped to a certain class property.
name1 : *name2*	This is a case where we define in a single row the values of 2 attributes (named as name1 and name2 in this example). As already indicated, such values can be either of a basic type or map to a certain member of an enumeration.
Please note that in some cases, we provide the definition of the whole class, including its contained attributes and properties. This is more convenient in case that it is not possible to put each (required) attribute or property in a single row as multiple attributes might be mentioned even in the same row.	