Metric Modelling

Current monitoring systems can be classified as either static or dynamic. The static systems are able to monitor a fixed set of non-functional properties (e.g., CPU utilisation) and require a great re-engineering effort in order to extend this set. On the other hand, dynamic systems are able to monitor any non-functional property through the ability to parse certain specifications of how such a property can be measured and then spawn the respective monitoring infrastructure needed. CAMEL aims at supporting dynamic monitoring systems through the supply of the right abstractions that are required for the measurement of non-functional properties. All such abstractions are included in a metric model whose content is analysed below.

A metric model encompasses various kinds of monitoring elements. The most important of such kinds map to the CAMEL notions of metric and metric variable. A metric encapsulates all the needed details for the measurement of a non-functional, measurable property. Such a metric can be either raw or composite. A raw metric (e.g., raw CPU utilisation) can be measured directly from sensors. On the other hand, a composite metric (e.g., average CPU utilisation) can be measured through the computation of mathematical formulas expressed over other metrics (e.g., mean(raw CPU utilisation) for average CPU utilisation). A metric can be restrain via a constraint which constraints the measurement values that it can take. Such a constraint can be used for definiting both Service Level Objectives (SLOs) as well as non-functional events in scalability rules.

Metric variables are configuration measures, i.e., they apply over properties spanning configuration elements. These configuration elements can concern either the current application configuration (e.g., current number of cores for a certain application component) or the current candidate solution produced by the (application deployment) Reasoner (e.g., number of cores mapping to the offering selected for the same application component in the candidate solution). Similarly to metrics, metric variables can be simple (i.e., directly mapping to a configuration element as in the case of current number of cores) or composite. In the second case, they encapsulate a formula that can be used for their computation spanning other metrics or metric variables (e.g., the max over the number of cores spanning all candidate offerings for an application component). Metric variables can be used for specifying either utility functions or SLO constraints. In the first case, a utility function could span both the current configuration and the current candidate solution in order to derive the respective difference (in terms of cost and/or performance) which could be positively or negatively needed for the considered application. In other words, a positive difference could map to a high utility making the current candidate solution as highly desired, while a negative difference could map to a very low utility, very close to zero. In the second case, an SLO constraint could be regarded as a specific restriction over deployment reasoning and could be specified by considering the current application configuration. For instance, we could specify that the total number of cores for the whole application should not be less than 10. Such a constraint would need the specification of a composite metric variable which would need to span over the sume of the respective metric variables (also covering the number of cores) mapping to the application components.

Both a metric or a metric variable can refer to the same metric template. The latter can be considered as a re-usability element encompassing all common elements between these two notions. Such a metric template, in particular, refers to the non-functional attribute being measured, its unit of measurement and optionally to its value type (which could be supplied for validation reasons to check whether the measurements of the respective metric, possibly computed from error-prone sources of information, belong to this type and are thus valid).

An example of a metric template can be seen in the following CAMEL textual fragment:

```
template CPUUtilisationTemplate{

        attribute CPUUtilisation  here we supply a reference to the non-functional attribute

        unit UnitTemplateCamelModel.UnitTemplateModel.Percentage  here we supply a reference to the unit of measurement

        value type TypeTemplateCamelModel.TypeTemplateModel.ZeroToHundredDouble  here we supply a reference to the
        template's value type

}
```

This fragment defines the CPUUtilisation template, i.e., a metric template which can be used for defining metrics (variables) that measure the CPU utilisation non-functional attribute. Obviously, the template refers to this attribute. It also refers to its unit of measurement which is a Percentage. Finally, it also refers to its domain of values which maps to the range [0.0, 100.0] of double values.

Such a metric template can be used for the definition of a raw metric measuring the raw CPU utilisation. This is shown in the following fragment:

```
raw metric RawCPUUtilisation{

        template CPUUtilisationTemplate  here we refer to the metric template re-used

}
```

As it can be seen from this fragment, the raw CPU utilisation encapsulates all those elements that are defined by the CPUUtilisationTemplate that is being referenced. Please note that we do not directly associate a raw metric with a sensor. As different sensors could be used to measure such a metric in different contexts (e.g., for different application components). We will come back to this point later on.

However, in order to define a meaningful constraint over CPU utilisation measurements, a raw metric is not usually handy. In this sense, we need to compute a composite metric on top of that metric. This can be seen in the following fragment that defines the composite metric of MeanCPUUtilisation:

```
composite metric AverageCPUUtilisation{
        template CPUUtilisationTemplate  here we refer to the template re-used
        formula: ('mean(RawCPUUtilisation)')  here we supply the computation formula for this metric
}
```

This fragment showcases that this composite metric includes a reference to the same template as in the case of the raw CPU utilisation metric. In addition, it depicts that this metric is computed by applying the mean operator / function over the raw CPU utilisation metric.

In order to also showcase the expression power in terms of metric variable modelling, we now refer to the CRM use case and supply all the metric variables that have been defined for it. We attempt to showcase here how the modeller could specify a utility function by specifying only metric variables that span the current application configuration and candidate solution. The following fragments depicts the respective metric variable definitions needed for this use case:

```
variable CandidateCost{
        template MetricTemplateCamelModel.MetricTemplateModel.CostTemplateEuros  here we refer to the template being re-used
        component CRMDepModel.SmartDesign  here we refer to the component on which the variable applies
        on candidates  here we refer whether we consider the node candidates / offerings of the component referenced
}
variable SmartDesignUnaryCost{
        template MetricTemplateCamelModel.MetricTemplateModel.CostTemplateEuros  here we refer to the template being re-used
        component CRMDepModel.SmartDesign  here we refer to the component mapping to this variable
}
variable SmartDesignCardinality{
        template MetricTemplateCamelModel.MetricTemplateModel.ComponentInstanceTemplate  here we refer to the template being
        re-used
        component CRMDepModel.SmartDesign  here we refer to the component mapping to this variable
}
variable CostUtility{
        template MetricTemplateCamelModel.MetricTemplateModel.UtilityTemplate  here we refer to the template being re-used
        component CRMDepModel.SmartDesign  here we refer to the component mapping to this variable
        formula: ('1 - ((SmartDesignUnaryCost * SmartDesignCardinality)- 2 * min(CandidateCost)) / (6 * max(CandidateCost) - 2 *
        min(CandidateCost))')  here we define the computation formula for this metric variable
}
```

Both the first and second metric variables are single and refer to the cost template which is a metric template for measuring the property of cost in the measurement unit of Euros. The first metric variable refers to the cost of all candidate offerings mapping to the SmartDesign component. Such a variable can then be considered as a set of costs over all these offerings. The second metric variable refers to the (unary) cost of the offering that has been selected for the same component for the current candidate solution.

The third variable is also single and refers to another template, responsible for specifying details for the measurements of the number of component instances. In this case, the component is also SmartDesign and we attempt to encapsulate via this variable the current number of instances of that component for the current candidate solution.

Finally, the last variable is composite. It refers to a metric template that is suitable for the measurement of the overall utility (if we consider it as another non-functional property). It also refers again to the same component, the SmartDesign one. This variable actually represents the utility function to be maximised and which has been mapped to a certain optimisation requirement. This utility function is computed from a formula which involves the rest of the metric variables defined. To summarise its content, this is a linear utility function that attempts to assign a utility to the current solution (mapping to a cost which is equal to the unary cost of the current solution's selected offering multiplied by the current solution's number of instances of the respective component) which is proportional to the difference that this solution has from the worst solution over the difference between the worst (mapping to 6 instances of the SmartDesign component multiplied by the maximum candidate cost) and best possible solution (mapping to 2 instances of the same component multiplied by the minimum candidate cost).

In order to actually measure a metric, we need to define its actual context of application. This is in contrast to a metric variable where the respective context is specified within its definition. A metric context provides some additional contextual details that span the following information: (a) how often the metric is being measured – i.e., the metric schedule; (b) over which window the metric measurement takes place; (c) what is the object being measured; (d) metric-type-specific details. The fourth piece of information depends on the type of the metric. In case of raw metrics, it

maps to the sensor that will be used to produce the measurements for this metric. In this case, we are actually dealing with a RawMetricContext. In case of composite metrics, there is a need to refer to the contexts of all the metrics from which this composite metric is computed as well as the way the respective component measurements are grouped during the computation of this metric. Such a context is then named as CompositeMetricContext.

Six types of measurement groupings can be alternatively specified whose semantics are explained as follows:

1. PER_INSTANCE: here the aggregation is performed for each instance of the respective object (e.g., application component) being measured. For example, the mean CPU utilisation will be computed for each component instance through the raw CPU measurements computed for that instance.
2. PER_HOST: here the aggregation is performed at the host level. This means that the aggregation is performed for each host of the measured object. For example, if we have 2 instances of a component in a VM, each mapping to a different container, then the mean CPU utilisation will be computed by considering the raw CPU utilisation measurements only from these two instances.
3. PER_ZONE: here the aggregation is performed at the zone level. For instance, if we have 4 instances of the measured component over 2 VMs that are situated in the same zone, then the aggregation will be performed over the measurements of these 4 instances.
4. PER_REGION: as a region comprises multiple zones, the aggregation is performed through combining the measurements over all these zones. Continuing our example and considering that we have 8 instances of the measured component, 4 in each two zones of a specific region, then the aggregation will rely on the measurements of all these instances.
5. PER_CLOUD: as a cloud can comprise multiple regions, the aggregation is performed through combining the measurements coming from all these regions. In the current example, if we consider that we have 2 regions with 8 instances each for the measured component, then the aggregation will rely on the measurements from the 16 instances of that component.
6. GLOBAL: here the aggregation is performed over all the measurements, irrespectively from the place that are produced. So, if there are 32 instances of the measured component, 16 in each cloud from the two in which it has been deployed, this means that the aggregation will be performed over the measurements from all these 32 instances.

Please note that the ordering of the groupings is quite significant and needs to be respected. In other words, it is wrong to model two metrics that are related in a metric derivation tree where the higher metric in the hierarchy has a grouping which is lower than that of a metric which is situated lower in the hierarchy. The correct ordering of groupings from the lowest level to the highest is from the PER_INSTANCE (enumeration) value to the GLOBAL one. This is enforced via the use of an OCL rule in CAMEL to enable the validation of the metric specifications based on this aspect and thus the production of correct metric models.

The combination of groupings and composite metric specifications can enable the creation of an hierarchy of metric (derivation) trees and respective contexts. For instance, if we consider the current example, we could have 6 levels in a metric tree mapping to 6 composite metrics measuring CPU utilisation. Normally, in most of the cases, we could expect that a metric tree hierarchy of 3-4 levels will suffice to cover most of the use cases. Please note, though, that the grouping to be applied in this hierarchy will differ with respect to the levels that it can actually encompass.

We, now, move the analysis to the rest of the information pieces comprising a metric context. A sensor can be either system-specific and user-specific. A system-specific sensor usually covers the resource level. For such a sensor, we need to define the name its Java class in the form of a configuration string. A user-specific sensor can be either installed within the platform and become an element of control for it or outside of it. In the first case, the sensor will be specified as a normal software component for which a respective script configuration would need to be supplied for its life-cycle management (see also Deployment Modelling sub-page). In the second case, no additional detail needs to be specified for the sensor mapping to an empty config string for it. Any sensor can be either push- or pull-based. In the first case, this sensor pushes its measurements to the platform. In the second case, the platform should retrieve its measurements through the exploitation of a certain interface that this sensor should exhibit.

A metric schedule defines how often a metric is being measured and is encapsulated by the Schedule class. To provide the definition for an instance of such a class, we need to specify mainly two pieces of information: (a) the time interval via an integer; (b) the time unit of measurement (e.g., seconds). In this respect, we could then define a schedule which is triggered, e.g., every 10 seconds, logically speaking, mapping to the measurement frequency of a raw metric.

A Window refers to the window of measurement according to which a composite metric computation can be performed. There are two types of metric windows: fixed and sliding. When a fixed window becomes full, then usually we perform the respective composite metric measurement and the window becomes empty again. On the other hand, a sliding window slides the measurements via which a composite metric can be computed. This provides more flexibility when combined with a metric schedule. For instance, we could define a sliding window which has a size of 10 measurements and which is used for the composite metric computation when every 5 new component measurements are produced and stored in this window (e.g., each measurement could be produced every 10 seconds and we could perform the metric computation every 5 * 10 = 50 seconds based on this sliding window which would have 5 new and 5 old measurements). A metric window can also have different size types. Two main size types can be discerned: (a) time-based: here the window becomes full when a certain time period has passed; (b) measurement-based: here the window becomes full when a certain number of measurements is produced and stored in it. The other types attempt to mix the basic size types. For instance, the first-match maps to the case where the window becomes full either when the respective time period passes or the respective amount of measurements is reached.

An ObjectContext represents the actual object that is being measured. Such an object context can then refer to either two different kinds of elements: (a) an application / software component; (b) a data element. If one from these two element kinds is referred, this means that only this element is being measured. For instance, we can measure the raw execution time of a software component or the actual current size of a data set. Otherwise, this means that we have the measurement of the actual relation or binding between these two element kinds. For instance, we could measure the raw number of bytes that have been accessed by a data processing component over a respective data set.

In order to now exemplify the modelling of all the above elements, we continue our metric specification example and we assume that we attempt to specify contexts for the CPU utilisation metrics for the SmartDesign component of the CRM use case. The following fragment showcases the respective modelling involved for this continuation:

```
object context SmartDesignObjectContext{
        component CRMDepModel.SmartDesign  here we refer to the component being measured, i.e., the SmartDesign one
}
sensor CPUUtilisationSensor{
        isPush  here we specify that the sensor pushes its measurements to the platform
        config 'tbd'  here we specify the name of the Java class mapping to the sensor code as we are dealing with a system sensor
}
schedule RawCPUUtilSchedule{
        interval 5  here we specify the amount of time points in the schedule period
        time unit UnitTemplateCamelModel.UnitTemplateModel.Seconds  here we specify the time unit for each time point in the
        period
}
schedule AvgCPUUtilSchedule{
        interval 1  here we specify the amount of time points in the schedule period
        time unit UnitTemplateCamelModel.UnitTemplateModel.Minutes  here we specify the time unit for each time point in the period
}
window AvgCPUUtilWindow{
        type sliding  here we specify that the window is sliding
        size type measurements-only  here we specify that the window size type is measurements-only
        measurement size 18  here we specify the size of the window (in terms of measurements)
}
raw metric context RawCPUUtilisationContext{
        metric MetricTemplateCamelModel.MetricTemplateModel.RawCPUUtilisation  here we specify the metric mapping to this
        context
        sensor CRMMetricModel.CPUUtilisationSensor  here we refer to the sensor used to produce the respective metric's
        measurements
        schedule RawCPUUtilSchedule  here we define the schedule of the raw metric
        object context SmartDesignObjectContext  here we refer to the object context that defines the object being measured
}
composite metric context AvgCPUUtilisationContext{
        metric MetricTemplateCamelModel.MetricTemplateModel.AverageCPUUtilisation  here we specify the metric mapping to this
        context
        grouping per-instance  here we refer to how the component metric measurements are grouped during the composite metric's
        computation
        window AvgCPUUtilWindow  here we refer to the window of this metric
        schedule AvgCPUUtilSchedule  here we define the schedule of the raw metric
        object context SmartDesignObjectContext  here we refer to the object context that defines the object being measured
        composing contexts [RawCPUUtilisationContext]  here we refer to the contexts of the component metrics of this composite
        metric
}
```

As it can be seen from the above fragment, we define 5 auxiliary elements and 2 metric contexts. The 5 auxiliary elements include one object context, mapping to the SmartDesign object/component being measured, the sensor for the RawCPUUtilisation metric, the 2 schedules for the two metrics and 1 window for the AverageCPUUtilisation metric. The sensor is push-based, i.e., it pushes its values to the platform, and is system-supplied which then maps to specifying its respective Java class name in form of a configuration string. The RawCPUUtilisation metric is

measured every 5 seconds while the AverageCPUUtilisation metric is measured every 1 minute based on a measurement-based sliding window having a size of 18 measurements. This means that in this window we will have 12 new measurements (12 * 5 seconds makes 1 minute) and 6 old ones.

For the context of the raw metric, we just specify all the needed details spanning the metric itself, its sensor, schedule and object context. In the case of the composite metric context, we need to specify 3 additional information pieces: (a) the window of the metric; (b) the composing contextts which map to the raw metric context of the RawCPUUtilisation metric; (c) the way the grouping of the component metric measurements is performed. In the latter case, we define that this is conducted at the instance level. This means that we calculate the mean value over the raw measurements for each instance of the SmartDesign component.