Scalability Rule Modelling

A multi-cloud management platform like Melodic should support two types of application reconfiguration (as Melodic actually does):

1. *local reconfiguration*: in this case of application adaptation, the number of instances of an application component is modified (e.g., increased) in order to handle a specific non-functional performance issue (e.g., the lack of resources to handle the current workload that leads to a performance drop in the respective component),
2. *global reconfiguration*: in this case, the whole application is reconfigured and not just a single component of it. This is performed to handle a certain issue that can affect the whole application performance. The reconfiguration is usually performed by performing deployment reasoning for it and then applying the new deployment solution produced for this application.

Global reconfiguration is usually launched when a specific (global) SLO (e.g., mean application response time less than 2 minutes) is violated. Similarly, local reconfiguration should be launched when a certain metric condition is violated and is usually assorted with the execution of a certain scaling action. Such an action is in most of the cases a horizontal scaling one, thus leading to the modification of the number of instances of the affected component. The mapping of the condition violation to the scaling action to be performed is usually denoted as a *scalability rule*.

CAMEL enables the modelling of both metric conditions / SLOs as well as scalability rules. In this page, our focus will be mainly on how the latter can be specified in CAMEL 2.0.

In CAMEL, a scalability rule is indeed a mapping between an event and a scaling action. This can be also depicted by the following example which represents a small extension of the CRM use case. Each example in this page will be illustrated through the supply of a corresponding CAMEL textual fragment:

```
rule ScaleSmartDesign{

        event CPUUtilViol  the event referenced

        actions [HScaleSmartDesign]  the action that needs to be performed

}
```

In this fragment, we specify a scalability rule for scaling the SmartDesign component. This rule indicates that upon the occurrence of the CPUUtilViol event, the HScaleSmartDesign horizontal scaling action will be performed to increase the number of instances of the SmartDesign component by 1. As we will see later on, the event referenced is launched when the mean CPU utilisation of the resource hosting that component is above a certain threshold.

An event that triggers a scalability tule can be either single or composite. Single events are usually non-functional ones. Such events usually map to the violation of a certain metric condition. Let's check this by relying on the current example / use case:

```
non-functional CPUUtilViol{

        constraint CRMMetricModel.CPUUtilConstr  the metric constraint referenced

        violation  indicates that the violation of the constraint triggers the event

}
```

As it can be seen, the above fragment illustrates that the non-functional event CPUUtilViol is triggered when the constraint (metric condition) CPUUtilConstr is violated. This constraint has been specified as follows:

```
metric constraint CPUUtilConstr: [CRMMetricModel.AvgCPUUtilisationContext] < 80.0
```

The constraint indicates that the metric identified by the AvgCPUUtilisationContext, i.e., the mean CPU utilisation measured over the SmartDesign component, should take only measurement values that are less than 80%. More details about how (metric) constraints should be specified can be found in the following Confluence page: Modelling of Utility Function and Constraints in Camel Model 2.0

Composite events are agglomeration of events on which a certain time-based or logical operator is applied. They can be classified as either unary or binary event patterns depending on the arity of the operator applied on them. Such composite events actually define a certain hierarchy of events in the form of a tree which has on its leaves single (non-functional) events.

Unary event patterns use either the NOT logical operator or the EVERY, REPEAT or WHEN time-based operators:

- NOT indicates a logical negation of the event on which it is applied.
- EVERY indicates that the event is triggered each time that the respective event referenced occurs
- REPEAT signifies that the referenced event should occur a certain amount of times (see occurrence num attribute in CAMEL meta-model) in order to consider that the event pattern should be triggered
- WHEN indicates that the referenced event should occur within a specific time period in order to consider that the event pattern should be triggered. This period is defined through the use of a Timer (to be analysed below)

In the following, we show an imaginary unary event pattern specification fragment built on top of the CPUUtilViol non-functional event:

```
unary event pattern CPUUtilViolTwoTimes REPEAT (CPUUtilViol^2)
```

Here we indicate that the unary event pattern, i.e., an event composition, will occur when the CPUUtilViol event happens two times. To be more particular, such an event pattern would occur when we see two times the mean CPU utilisation of the resource hosting the SmartDesign component within the period of 2 minutes (as each measurement of this metric is produced every 1 minute) to be more that 80%. For example, the first violation could be considered as coincidental while the second one would make us more certain that there is a need to scale the SmartDesign component.

Binary event patterns apply a time-based or logical operator on either two events or one event and a timer. In the latter case, the timer can be considered as a special event denoting the passing of a certain time period. The following logical operators can be used in such a pattern:

- AND: this denotes a logical conjunction of the two events referenced in the pattern. This indicates that both events need to occur to consider the binary event pattern triggered. Please note that we do not care about the order of the two events which can be arbitrary.
- OR: this denotes a logical disjunction of the two events. As such, we denote that at least one of the two events should occur in order to have the binary event pattern triggered.
- XOR: this denotes a logical XOR of the two events referenced. The respective semantics is that one of the two events should exclusively occur in order to have the binary event pattern triggered.

On the other hand, the following time-based binary operators are supported:

- PRECEDES: this denotes the sequential ordering of the two events referenced in order to have the binary event pattern triggered.
- REPEAT_UNTIL: the operator enforces that the first event references should occur as many times as indicated in the respective range and to be followed with the second event referenced in order to have the binary event triggered. The range is denoted through the use of the lower and upper bounds.

In the following, we supply a certain complex example of how a whole event tree can be specified.This event tree is depicted in the following figure:

This event tree can be denoted by the following expression that utilises the respective operators needed: every(repeat_until(precedes(repeat(A,3),B),1:2,C)). Please note that here A, B, C are single, non-functional events for which we have omitted their definition for brevity reasons. The following fragment denotes how such an event tree could be specified in CAMEL:

```
unary event pattern repeat_A REPEAT (A^3)  repeat(A,3)

binary event pattern precedes_A_B{  precedes(repeat(A,3),B)

        operator PRECEDES

        left event repeat_A

        right event B

}

binary event pattern repeat_until_ABC{  repeat_until(precedes(repeat(A,3),B),1:2,C)

        operator REPEAT_UNTIL

        [1,2]

        left event precedes_A_B

        right event C

}

unary event pattern every_ABC EVERY (repeat_until_ABC)  every(repeat_until(precedes(repeat(A,3),B),1:2,C))
```

As it can be seen, we attempt to construct in a bottom-up manner the respective event tree starting from the first intermediate nodes and going up the tree levels. The first event, repeat_A maps to the "repeat(A,3)" part of the expression. The second event, precedes_A_B, builds on the first to produce the expression part "precedes(repeat(A,3),B)". The third event, repeat_until_ABC, builds on the second to apply the REPEAT_UNTIL operator on that event and C and cover the expression part: "repeat_until(precedes(repeat(A,3),B),1:2,C)". Finally, the last event (every_ABC), the highest in the hierarchy, builds on the third to cover the whole expression. Please note that the event expression was splitted in this way as we are restrained to deal with binary and unary event patterns only. This is in contrast to metric formulas where now we can specify arbitrary expressions that can refer to other metrics and certain mathematical operators. Possibly, we could also move to this direction for the specification of events in the near future.

Timers can be considered as specialised events that might also play an assistive role in the formulation of event patterns. Three kinds of timers can be specified. The first two of them are those which play the assistive role, especially when unary event patterns are specified with the WHEN operator. The last one can be actually used to denote the special event that can replace a normal one in a binary event specification. Let's have a closer look to the kinds of these timers and their respective semantics.

- WITHIN: it imposes that the respective event should occur with a certain time period in order to consider the respective event pattern as satisfied.
- WITHIN_MAX: has similar semantics with respect to WITHIN with the sole exception that the event pattern can be triggered only when the occurrence number of the respective event referenced does not go over a certain limit / bound.
- INTERVAL: it indicates a special time-based event which occurs when the respective time period encapsulated has been passed.

We will attempt to highlight the assistive role of a WITHIN Timer in the CRM use case. In this respect, we continue and build on the example of the CPUUtilViolTwoTimes unary event pattern. In this case, we need to be re-assured that two out of the three mean CPU utlisation measurements (each produced every minute) should violate the respective metric condition in order to have the respective event pattern triggered. This can be modelled based on the following fragment:

```
timer ThreeMin{
        type WITHIN  here we denote the type of the time
        time 3  here we denote the number of time points in the interval
        unit UnitTemplateCamelModel.UnitTemplateModel.Minutes  here we denote the unit of the time points
}
unary event pattern ThreeMinCPUUtilViol WHEN(CPUUtilViolTwoTimes : ThreeMin)
```

As it can be seen, we first specify a WITHIN timer that maps to a period of 3 minutes. Then, we utilise this timer to specify a unary event pattern which indicates that it can be triggered when the CPUUtilViolTwoTimes occurs within the 3 minute period denoted by this timer.

Finally, we focus our analysis on scaling actions. Such scaling actions can be of two different types: (a) horizontal scaling ones where, as already indicated, we attempt to modify the instances of a certain application component; (b) vertical scaling ones, where we attempt to modify the amount of resources given to an application component. Currently, Melodic only supports the execution of horizontal scaling actions. Thus, we concentrate on explaining only the way this kind of action can be modelled.

To specify an horizontal scaling action, we just need to refer to the component that needs to be scaled as well as supply the number of instances of that component that should be modified. This number can be either positive or negative. In the first case, we denote the addition of this number of component instances. In the second case, we denote the removal of this number of component instances. Going back to our current example and the scalability rule that has been defined for it, we now supply the respective CAMEL fragment that showcases the horizontal scaling action that needs to be modelled for that rule:

```
horizontal scaling action HscaleSmartDesign{
        software CRMDepModel.SmartDesign  reference to the component to be scaled
        count 1  the number of instances of that component to be created
}
```

The modelled horizontal scaling action denotes that the SmartDesign component needs to be horizontally scaled by increasing by one its number of instances.