SRL: A Scalability Rule Language for Multi-Cloud Environments

Kyriakos Kritikos*, Jörg Domaschka†, Alessandro Rossini‡
*ICS-FORTH, Heraklion, Greece
kritikos@ics.forth.gr
†Institute of Information Resource Management, University of Ulm, Germany
joerg.domaschka@uni-ulm.de
‡Department of Networked Systems and Services, SINTEF, Oslo, Norway
alessandro.rossini@sintef.no

Abstract—The benefits of cloud computing have led to a proliferation of infrastructures and platforms covering the provisioning and deployment requirements of many cloud-based applications. However, the requirements of an application may change during its life cycle. Therefore, its provisioning and deployment should be adapted so that the application can deliver its target quality of service throughout its entire life cycle. Existing solutions typically support only simple adaptation scenarios, whereby scalability rules map conditions on fixed metrics to a single scaling action targeting a single cloud environment (e.g., scale out an application component). However, these solutions fail to support complex adaptation scenarios, whereby scalability rules could map conditions on custom metrics to multiple scaling actions targeting multi-cloud environments. In this paper, we propose the Scalability Rule Language (SRL), a language for specifying scalability rules that support such complex adaptation scenarios of multi-cloud applications. SRL provides Eclipse-based tool support, thus allowing modellers not only to specify scalability rules but also to syntactically and semantically validate them. Moreover, SRL is well integrated with the Cloud Modelling Language (CloudML), thus allowing modellers to associate their scalability rules with the components and virtual machines of provisioning and deployment models.

I. INTRODUCTION

The advent of cloud computing has led to a proliferation of services that not only provide infrastructures and platforms to cloud-based applications, but also manage the life cycle of these applications. In fact, several solutions today (e.g., Amazon Web Services1) can dynamically adapt the provisioning and deployment of cloud-based applications. These solutions rely on scalability rule languages (e.g., Amazon CloudFormation2, and Cloudify’s Automatic Scaling Rules3), whereby scalability rules map conditions on fixed metrics to a single scaling action targeting a single cloud environment (e.g., scale out an application component). However, these languages fail to support complex adaptation scenarios, whereby a scalability rule could map conditions on custom metrics to multiple scaling actions targeting multi-cloud environments [1].

In this paper, we propose an extended version of the Scalability Rule Language (SRL) [2], a language for specifying rules that support complex adaptation scenarios of multi-cloud applications, i.e., applications that can be deployed across multiple private, public, or hybrid cloud infrastructures and platforms. SRL aims at compensating for the limitations of existing solutions by providing the necessary modelling concepts for specifying behavioural patterns of multi-cloud applications, as well as the scaling actions to change the provisioning and deployment in response to these patterns.

Basically, any scalability rule language has to encompass the definition of event patterns as well as scaling actions. The latter shall be executed whenever event patterns occur. Therefore, SRL provides mechanisms for (a) specifying event patterns, (b) specifying scaling actions, and (c) associating these scaling actions with the corresponding event patterns. Moreover, in order to identify event patterns, the components of multi-cloud applications must be monitored. Therefore, SRL provides mechanisms for (d) expressing which components must be monitored by which metrics, and (e) associating event patterns with monitoring data.

SRL is inspired by the OWL-Q language [3] with respect to the specification of quality of service (QoS) metrics, as well as the Esper Processing Language (EPL)4 with respect to the specification of event patterns with formulas including logic operators and timing. It was developed in the context of the Paasage project5, which provides an integrated platform for the modelling and execution of cloud-based applications on multiple clouds. In this respect, SRL complements other domain-specific languages (DSLs) adopted in the project, namely the Cloud Modelling Language (CloudML) [4], [5] for specifying provisioning and deployment models; Saloon [6], [7] for specifying capabilities of cloud providers; WS-Agreement [8] for specifying service-level objectives (SLOs); and the Common European Research Information Format (CERIF) [9] for specifying profiles of organisations.

Compared to our previous work [2], this paper provides three contributions that adapt and extend SRL: (a) new modelling concepts for restricting scalability rules through scaling policies, (b) additional modelling concepts for specifying more fine-grained event patterns, and (c) a set of Object Constraint Language (OCL) [10] constraints for capturing (part of) the semantics of the domain, which assist modellers in specifying valid scalability rules at design-time.

1http://aws.amazon.com/
2http://aws.amazon.com/cloudformation/
3http://getcloudify.org/guide/2.7/developing/scaling_rules.html
4http://esper.codehaus.org/
5http://www.paasage.eu/
The remainder of the paper is structured as follows: Section II presents the modelling constructs of SRL. Section III presents the OCL constraints attached to SRL. Section IV describes a complex scalability rule specification. Section V reviews related work. Finally, Section VI concludes the paper and draws directions for future work.

II. Scalability Rules Language

In this section, we present the main modelling constructs of SRL. In practise, the models presented here have to be brought to execution by an SRL runtime system that is not considered here.

As the other DSLs adopted in PaaSage, SRL has been specified as an Ecore model [11] in the Eclipse Modeling Framework (EMF) [11]. Ecore is the de-facto reference implementation of OMG’s Essential Meta-Object Facility (EMOF) [12]. Moreover, this Ecore model has been complemented by OCL [10] constraints. By exploiting Ecore, OCL, and other model-driven techniques and methods, SRL provides Eclipse-based tool support, which allows modellers not only to specify scalability rules but also to syntactically and semantically validate them. This feature is missing from existing scalability rule languages (see Section V), which typically support syntactic validation only. Moreover, also by exploiting Ecore, SRL is well integrated with CloudML [4], [5], which allows modellers to associate their scalability rules with the components and virtual machines of provisioning and deployment models. This feature is also missing from existing scalability rules languages, which typically must be integrated manually.

A. Scalability Model and Rules

The specification of a scalability rule comprises a ScalabilityModel (see Figure 1), which contains all necessary building blocks to define scalability rules, such as the rules themselves, events, conditions, and actions. A ScalabilityRule is associated with an Event and a set of Actions. The Event represents either a single event or an event pattern that triggers the execution of the actions. The actions can either specify which components and virtual machines are changed by the scalability rule (i.e., case of scaling actions) and how or just remark that a global deployment decision has to be made (i.e., case of event creation actions) (see next section). The ScalabilityRule is associated with a set of ScalingPolicies that restrict how scaling actions are performed. Finally, the ScalabilityRule is associated with Entities such as the user or the organisation (specified in CERIF) owning this rule.

B. Actions

An Action (see Figure 1) can be classified as a ScalingAction or an EventCreationAction. The ScalingAction, in turn, can be classified as a HorizontalScalingAction or a VerticalScalingAction. The HorizontalScalingAction must be associated with a VM and can be associated with a InternalComponent (both specified in CloudML). When only the virtual machine is associated with the action, then all the components in this virtual machine are scaled; otherwise, only the specified component is scaled (out or in) along with a virtual machine of the same type. The attribute count defines the number of additional instances to create, or the number of existing instances to destroy. In contrast to horizontal scaling, the VerticalScalingAction must be associated with a concrete VMInstance. The attributes *Update define the amount of virtual resources (e.g., computing cores, RAM, etc.) to be added to or to be removed from this concrete virtual machine.

An EventCreationAction represents that the scaling actions are not sufficient to maintain the target QoS for a multi-cloud application. For instance, a multi-cloud application may still violate the target response time defined in an SLO despite the scale-out and scale-up actions performed. According to such a scenario, the provisioning and deployment model must be changed so the created event is consumed by the SRL platform, which either notifies the CloudML user to manually modify the deployment model or invokes the reasoning engine to automatically modify it.

C. Scaling Policies

A ScalingPolicy (see Figure 1) can be classified as a HorizontalScalingPolicy or a VerticalScalingPolicy. A HorizontalScalingPolicy defines the minimum and maximum amount of instances allowed for a component or virtual machine, so that scale-out and scale-in actions will not exceed these limits. Similarly, a VerticalScalingPolicy defines the minimum and maximum values allowed for attributes of a virtual machine (e.g., computer cores and RAM), so that scale-up and scale-down actions will not exceed these limits. What actually constitutes a virtual machine is specified in CloudML.

D. Events

An Event (see Figure 2) can be classified as a SimpleEvent or an EventPattern. The SimpleEvent, in turn, can be classified as a FunctionalEvent or a NonFunctionalEvent. The FunctionalEvent represents a functional error (e.g., a virtual machine or a component has failed). The NonFunctionalEvent represents the violation of a QoS metric (e.g., the response time.
of a component exceeds the target response time in an SLO). The NonFunctionalEvent is associated with a MetricCondition that defines the threshold for the metric.

An EventPattern can be classified as a BinaryEventPattern or a UnaryEventPattern. The BinaryEventPattern uses a binary operator to associate either two Events with each other or one event with a Timer. The attribute operator can be set to one of the common, logical operators such as AND or OR, the order operator PRECEDES, and the occurrence operator REPEAT_UNTIL. PRECEDES defines that an event has to occur prior to another one. REPEAT_UNTIL defines that an event has to occur multiple times until another event occurs.

A UnaryEventPattern is associated with just one event along with a unary operator. The attribute operator can be set to NOT, EVERY, REPEAT, and WHEN. NOT defines that the negation of an event must be considered. EVERY defines that every occurrence of an event must be considered (e.g., to define that a scalability rule must be triggered every time an event A occurs and not just once). REPEAT defines that an event has to occur multiple times. WHEN defines that an event has to occur according to a particular time constraint defined by a Timer.

A Timer represents a time constraint for an event pattern. The attribute type represents the kind of timer. WITHIN defines that an event has to occur within a particular time frame. WITHIN_MAX defines that an event has to occur within a particular time frame, but only up to a specific number of times. INTERVAL defines that an event has to occur after a particular amount of waiting time.

The following are two examples that illustrate the composition of events in event patterns. (a) The condition A AND (B OR C) can be expressed as a BinaryEventPattern \( X_1 \) that comprises the SimpleEvent A and another BinaryEventPattern \( X_2 \) both connected by the AND operator. \( X_2 \), in turn, comprises the two SimpleEvents B and C connected by the OR operator. (b) The condition that either A or three times B occurs within two minutes can be expressed as a UnaryEventPattern \( U_1 \) that comprises a BinaryEventPattern \( B_1 \), the WHEN operator, and a Timer defining a two minutes threshold. \( B_1 \), in turn, comprises the SimpleEvent A and a UnaryEventPattern \( U_2 \) connected by the OR operator. Finally, \( U_2 \) comprises just the SimpleEvent B, the REPEAT operator, and a value of 3 for the attribute occurrenceNum.

An EventInstance (see Figure 3) represents the actual (measurement) data associated with a particular event that occurred in the system (e.g., the actual value measured, the component producing the event, etc.) The attribute status represents the status of the event, i.e., if it is fatal, critical, warning, or success. This attribute can provide useful insight (e.g., for performing analysis on QoS) while also enabling the evaluation/assessment of the events. The attribute layer represents the layer in the cloud stack where the event has occurred. IaaS or PaaS denote that the event relates to IaaS or PaaS services used by an application, respectively, while SaaS denotes that the event relates to the application as a whole, or to a third-party SaaS (e.g., Amazon Simple Email Service9). Additional (sub-)layers further distinguish different SaaS service types: SCC (short for Service Composition) denotes that service composition is concerned, while BPM (short for Business Process Management) denotes that business processes are concerned. The difference between these two layers lies on the fact that business processes are usually realised through service compositions and are therefore at a higher abstraction level. For instance, a violation of a key performance indicator, which is critical for a business organisation may be caused by a violation of a SLO at the service composition realising this business process.

E. Metrics and Properties

In order to identify event patterns, the components and virtual machines must be monitored. A Property (see Figure 3) represents a non-functional property of a component or virtual machine. The attribute type represents the kind of property, where a value of MEASURABLE denotes that the property can be measured, such as response time or availability, while a value of ABSTRACT denotes that we deal with a non-measurable, abstract property realised by its more concrete sub-properties (which can be abstract or measurable themselves). For instance, in the security domain, the incident management quality is a property that is realised at least by the concrete and measurable reporting capability sub-property.

A Metric represents a generic metric encapsulating the details for measuring properties (e.g., an availability metric), which can be associated with either raw measurements or aggregations on them. A MetricTemplate represents common metric information that might have been repeated if encapsulated directly in Metric. Therefore, multiple Metric instances (i.e., multiple concrete metrics) can refer to the very same MetricTemplate instance. The MetricTemplate does not encompass dynamic details (which are modelled in Metric instead), such as how often the respective metric measurements are produced and which component instances on which virtual machines are concerned. The attribute type represents the kind of template (and hence the kind of Metrics associated with

9http://aws.amazon.com/ses/
this MetricTemplate). RAW denotes that the associated metric refers to raw measurements. COMPOSITE denotes that the associated metric is computed from other metrics. A RAW metric is associated with a Sensor, which produces the actual measurement data. The attribute configuration defines the configuration of the sensor, while the attribute isPush defines whether the measured data points will be pushed by the sensor or have to be pulled by the SRL runtime system instead.

For each COMPOSITE metric, the assigned MetricTemplate is associated with a MetricFormula. The MetricTemplate is also associated with a measurable Property, which represents the property measured by the Metrics realising this MetricTemplate. Furthermore, a MetricTemplate is associated with the Unit of measurement (e.g., the PERCENTAGE unit for all availability metric realising the respective metric template). To assist in checking the correctness of measurement values or their aggregations (especially if we consider that measurements can be produced by error-prone sources of information like sensors), a Metric is also associated with a ValueType, which represents the range of values the metric is allowed to take.

A MetricFormula is associated with one or more MetricFormulaParameters that represent the input for the formula, and a pre-defined function that defines the operation to be executed by the formula. There exist two types of parameters. In case of constants the parameter refers to a Value. Non-constant parameters are represented by MetricTemplates, in which case value (association) remains unset. The pre-defined functions are covered by MetricFunctionType and include amongst others AVERAGE, standard deviation (STD), MEAN, ADD, MINUS, and DIV. It is the function type that defines the semantics of the operation by also enabling the restriction on the number of required parameters, their kind, and the way they are combined (via OCL constraints, see Section III). For instance, MEAN should map only to one parameter, which should be of type MetricTemplate. To assist in the latter OCL-based restriction, a metric formula is associated with the arity of the function to allow checking that the given number of parameters is correct.

From the description of the formula, it follows that the formulas connect MetricTemplates, which have no measured values attached to them. Hence, for performing the actual computations, the metric templates associated with the formula have to be mapped to actual metrics that do have measured values attached. This holds for metric templates used as input parameters, but also for the result of the computation. This mapping happens for every COMPOSITE metric $M_c$ that has a formula attached. In order to perform the mapping, starting from $M_c$, the metric template $M_{ct}$ can be derived. The template leads to the formula, which in turn refers to the metric templates used as input parameters. For computing the values of $M_c$, the componentMetrics associated with Metric are matched to the metric templates associated with the formula. For better comprehension, consider the following example: The formula of an availability metric template $MT_1$ would divide the uptime metric template $MT_2$ with a specific constant value (representing the availability evaluation time space). Now, when a specific metric $M_t$ must be defined instantiating $MT_1$, then in $MT_1$’s formula $MT_2$ template will be replaced with the respective metric component $M_2$ of $M_t$ instantiating $MT_2$ template.

While a metric template refers to the Property being measured, there is no information about the entity to which this property belongs. However, such information is crucial for properly configuring the monitoring system as it enables for instance to define on which virtual machine (i.e., the entity or the virtual machine on which the entity is deployed) a sensor (i.e., the measurement mean) will be deployed. For that reason, a Metric is associated with the entity whose property is measured through a MetricObjectBinding, which can be classified as a MetricComponentBinding, a MetricVMBinding, and a MetricApplicationBinding. A MetricComponentBinding associates a particular component and respective application with a metric and leads to the deployment of one or more sensors reporting measurements for this component. The MetricVMBinding associates a virtual machine with a metric also indicating the need of deploying one or more sensors on this particular virtual machine to measure it. Finally, the MetricApplicationBinding associates the application as a whole with a metric. In this case, one or more sensors will have to be deployed to virtual machines on which one or more component instances of this application have been deployed to perform the respective measurements to be aggregated.

**F. Scheduling and Conditions**

Metrics may be associated with Windows (see Figure 4), which represent how multiple measurements will be temporarily stored and used to perform computations on that metric. The window size may be defined by a time frame, a fixed number of events, or both. In the last case, it may be sufficient to wait for either the first property to be fulfilled, or for both. The attribute sizeType represents the strategy to be used for this purpose. The attribute windowType, in turn, represents what happens when the window size is reached. SLIDING denotes that the window...
is slid by dropping superfluous elements, while FIXED denotes that the window is cleared.

Metrics may also be associated with a Schedule, which represents any aspect of the operations/measurements that must be executed on a regular, timely basis (e.g., when an operation must run and when the scheduling must end). For a COMPOSITE metric, a schedule defines when and how often the metric will be evaluated by applying the associated MetricFormula. For a RAW metric, a schedule defines how often its value is measured by the respective metric sensor. The attribute type represents whether successive runs happen at a fixed rate or with a fixed delay. Finally, the attribute intervalUnit represents the time unit used for the schedule interval.

A Condition represents an abstract condition that has a threshold value. The attribute operator represents a comparison operator i.e., greater than or less than (including and excluding equality) as well as (in)equality. The attribute validity denotes for how long the condition will be valid. A Condition can be classified as a MetricCondition or a MetricTemplateCondition. A MetricConditionn represents a constraint imposed on a metric. A constraint is violated when the comparison of the values of the metric with the threshold of the condition using the metric’s operator is false. The violation of a metric condition will lead to the triggering of a simple, non-functional event (see Section II-D). Such constraints may be derived from SLOs as defined by WS-Agreement. A MetricTemplateCondition represents a condition on a metric template. This concept enables expressing general requirements that might hold for a use irrespective of the applications. Such requirements can then be assessed in the respective system via the generation of system-defined metrics, realising the respective templates, whose values will be compared against the corresponding thresholds.

III. OCL CONSTRAINTS

Ecore provides useful features for modelling and metamodelling, such as checking the cardinality of attributes and associations, and validating models according to the corresponding metamodels. However, it lacks the expressive power required for capturing the semantics of all modelling constructs in a language. OCL, in contrast, provides useful features for capturing (parts of) the semantics of the domain. OCL is a declarative language for specifying expressions such as constraints and queries on Meta-Object Facility (MOF) [12] models and metamodels. OCL is also an integral part of the Queries/Views/Transformations (QVT) [13] specification, where it is adopted in the context of model transformation.

The Eclipse Model Development Tools (MDT) project includes the Eclipse OCL component, which provides: (a) an API for parsing and evaluating OCL expressions on Ecore models, (b) an Ecore implementation of the OCL abstract syntax model, and (c) an API for customising the parsing and evaluation environments. This component also includes: (d) an editor for specifying OCL expressions embedded in Ecore models, (e) a console for interactively evaluating OCL expressions on Ecore models, and (f) a model validation capability, based on the attached OCL constraints, incorporated in Ecore model editors.

We annotated the Ecore model of SRL with OCL constraints. Thanks to the combination of Ecore and Eclipse OCL, SRL provides Eclipse-based tool support, which allows modellers not only to specify scalability rules but also to syntactically and semantically validate them. In the following, we summarise the main OCL constraint types we have developed, which enable (a) the validation of scalability rules based on the semantics of the domain, and (b) the automatic derivation of values for particular attributes. The interested reader may refer to the appendix for an excerpt of the OCL code corresponding to these constraints.

Derivation of values: An OCL expression is used to derive values for particular attributes. A representative case concerns deriving the attribute type of a Property, which depends on the association subProperties. A if then else construct is used to express that if the association subProperties is empty, the attribute type should be MEASURABLE; otherwise, it should be ABSTRACT.

Checking of value correctness: OCL invariants check that the values provided for particular attributes are correct. Representative cases are: (a) The threshold of a metric condition should be included in the value type of this condition. (b) The maximum number of instances in a horizontal scaling policy should be greater than or equal to the corresponding minimum number.

Checking of value combination correctness: OCL invariants check that the combination of values provided for attributes or associations of instances is correct based on the semantics of the domain. Representative cases are: (a) In a binary event pattern, either two events or one event and a timer are specified. (b) If a metric is raw, it must be measured by a sensor; otherwise, it must be associated with a non-empty list of composite metrics. (c) If a metric template maps to a composite metric, it must be associated with a formula.

Checking of reflexive associations: OCL invariants avoid that a class instance refers to itself through a so-called reflexive association. This is enforced by calling particular class methods annotated with Java code. Representative cases

---

1http://www.eclipse.org/modeling/mdt/
2http://wiki.eclipse.org/OCL
are: (a) A metric cannot be derived by itself. (b) A metric formula associated with a metric template cannot refer to this template recursively.

**Other domain constraints:** Other OCL invariants are used to capture cases not covered by the above constraint types according to the semantics of the domain. A representative example concerns ensuring that a metric’s component metrics must map in a one-to-one manner to the input metric templates of the formula of the metric’s realised template.

### IV. Scalability Rule Example

In this section, we present how a non-trivial, common scalability rule is represented in SRL. This example operates at the level of a single component, namely a Couchbase\(^9\) distributed database, and embodies empirical knowledge about this database gained via experiments into a scalability rule. For a Couchbase cluster of three instances, the following pattern proved to be a good rule of thumb for triggering a scale out: (a) all three instances have been having an average CPU load beyond 50% for at least 5min, and (b) concurrently at least one instance has been having an average CPU load beyond 85% for at least 1min. Furthermore, it is sufficient to gather sensor values for the current CPU load every second. Suppose \(cpu_1\) represents the average CPU load for instance \(i\). Thus, in order to trigger the scalability rule, the following composite condition must be assessed:

\[
((cpu_1 \geq 50) \land (cpu_2 \geq 50) \land (cpu_3 \geq 50)) \land \\
((cpu_1 \geq 85) \lor (cpu_2 \geq 85) \lor (cpu_3 \geq 85))
\]

Based on the above analysis, we created a SRL model, which is presented in two parts: (a) the scalability rule along with the events used to trigger it, and (b) the metrics and conditions that, when evaluated, trigger the action of the scalability rule. In order to enable the deployment of the cluster, we complemented the SRL model with a provisioning and deployment model in CloudML. In this model, the Couchbase component is identified by `Couchbase`, while the virtual machine on which its instances are deployed is identified by `LL` (i.e., Large Linux).

Figure 5 shows part (a) of the SRL model. This model encompasses one scalability rule that associates one binary event pattern with a scale-out action. The event pattern structure is also shown to illustrate the way the event conditions are specified. In this structure, the six simple non-functional events map to the six conditions to be evaluated, while the five binary event patterns (with one as the root) impose the logical relation hierarchy between these conditions.

Figure 6 shows part (b) of the SRL model. The metrics mapping to the composite event structure in Figure 5 are shown along with their scheduling information. The two metric templates map to common information for three families of metrics: (a) raw (sensor) metrics measuring CPU load, (b) average CPU load metrics over five minutes, and (c) average CPU load metrics over one minute. To keep the figure readable, we omitted one metric from each metric family as well as all binding information. Regarding the bindings, one metric from each family maps to one metric component binding.

\(^9\)http://www.couchbase.com/

![Figure 5. Excerpt of SRL model showing a scalability rule and a composite event](image)

Figure 5. Excerpt of SRL model showing a scalability rule and a composite event

which is associated with a particular Couchbase instance to be measured.

Please note that the described rules show reasonable results only when applied to a Couchbase cluster of up to five nodes. Different rules are required for larger clusters. Hence, reaching a size of five nodes with a heavy load should trigger an `EventCreationAction` that would lead to changing the provisioning and deployment model of the cluster.

It is true that the respective scalability rule requires a non-negligible modelling effort. Yet, SRL encourages re-usability of existing patterns and in particular, existing metrics. In fact, SRL is already accompanied with a particular model defining common units, properties, and metric templates. Metrics can then be instantiated from these metric templates based on the requirements of the particular application at hand. In this respect, the model will only have to define the metrics, the conditions on them, the non-functional events, and compose them along with the corresponding scaling actions into a scalability rule. Thus, in the end, the modelling effort between the various scalability rule languages and SRL will be equivalent or similar.

### V. Related Work

SRL shares characteristics with complex event processing (CEP) languages as well as languages for specifying QoS requirements and constraints. SRL is also closely related to other scalability languages proposed in the context of cloud computing. Therefore, the related work analysis is organised according to these three main language categories. Table I provides a comparison of SRL and several other languages or language categories, whereby we consider expressiveness with respect to defining event patterns, metrics, and scaling actions. Either one or a set of languages is evaluated against the criteria. In the latter case, the languages are grouped as they share the same characteristics. Each criterion evaluation...
Table 1. COMPARISON OF RELATED WORK

<table>
<thead>
<tr>
<th>Language</th>
<th>Event Pattern Expressiveness</th>
<th>Metric Expressiveness</th>
<th>Scalability Rule Expressiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEP Languages</td>
<td>high</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>OWL-Q [3]</td>
<td>low</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Drools Rule Language (DRL)</td>
<td>low</td>
<td>medium</td>
<td>low</td>
</tr>
<tr>
<td>WSO2 Siddhi [12]</td>
<td>low</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>SRL</td>
<td>high</td>
<td>high</td>
<td>high</td>
</tr>
</tbody>
</table>

The first three values denote the level of expressiveness while the last one denotes that the particular aspect is not applicable or relevant for the considered language or set of languages.

A. CEP Languages

EPL, is a CEP language that enables defining complex event patterns in a highly expressive way. With respect to modelling event patterns, SRL has been inspired by EPL and thus shares with it almost the same degree of expressiveness. Other CEP languages (e.g., Drools Rule Language (DRL) and WSO2 Siddhi) have a similar level of expressiveness. Some CEP languages provide additional modelling constructs. For instance, DRL captures all possible timing relations between events. On the other hand, all CEP languages do not offer the right level of expressiveness for other scalability aspects, such as metrics, scaling actions, and scalability rules. While SRL is sufficiently open to be extended towards capturing additional CEP aspects, we currently do not see the need for that. Actually, we consider that the event pattern expressiveness of SRL is high enough to cover most of the cases in cloud computing. In the context of PaaSage and its respective industrial business use cases, the need for different scalability rules with different complexity levels arises. To this end, we will evaluate whether the power, abstraction, and expressiveness provided by SRL match both common and highly specialised use cases, thus validating our claims. Finally, the synergy between CEP languages and SRL should not be underestimated, by considering the fact that the definition of event patterns and metrics may map to transforming SRL models to CEP language expressions. This would allow exploiting a CEP engine to realise the underlying monitoring and assessment system.

B. QoS Languages

Several QoS languages have been proposed, mainly in the context of service computing. These languages differ in their expressiveness, abstraction, and semantics [16]. OWL-Q [3] can be distinguished as the most expressive among these QoS languages, especially in terms of modelling all aspects of metrics. To this end, with respect to modelling QoS metrics, SRL has been inspired by OWL-Q and thus shares with it almost the same degree of expressiveness.

C. Scalability Rule Languages

SRL is more expressive than most existing scalability rule languages, as these languages are quite simplistic and map conditions on fixed metrics to a single scaling action targeting a single cloud environment. Some of these languages have been proposed in the context of European projects, such as Reservoir [17] and Optimis [18].

Moore [14] proposes a reactive and proactive approach to cloud elasticity, involving a language that enables defining simple scalability rules. This language provides several of the features also found in SRL, such as metric condition (with a metric aggregation function), metric sliding window, scalability limit (scaling policy), scaling action details (e.g., scaling type), and scope of a rule (the affected components). The language reaches a medium level of metric expressiveness by complementing limited metric definition (i.e., just the metric name) with additional metric details in the scalability rule. Yet, it relies on bad design choices, such as mixing scaling policies

12https://docs.wso2.com/display/CEP310/WSO2+Complex+Event+Processor+Documentation/
and rules. On the contrary, SRL has been better designed in a modular way encouraging re-use of (parts of) specified rules: all metric information is specified through the metric and metric template concepts, while the limits posed on scaling actions are specified through the scaling policies concept. This separation enables these policies to be associated with different scalability rules for various application deployments. SRL also allows defining complex metric conditions and multiple types of scaling actions, as well as associating more than one scaling action with a scalability rule. This last feature serves the purpose to support complex adaptation strategies.

The SYBL scalability rule language [15] attains a good level of expressiveness and considers three main layers: (a) application, (b) component, and (c) programming (e.g., level of infrastructure including rules on CPU usage). This language can express logical combinations of constraints on metric values, which can lead to triggering specific strategies encompassing particular scalability actions, such as scale-in and scale-out. Despite its good level of expressiveness, SYBL is not as expressive as SRL in terms of specifying complex conditions and complete metric definitions. Moreover, SYBL provides limited information on which units are provided by the language and whether custom units can be defined. Finally, SYBL refers to the components and virtual machines that are changed by scaling actions through a simple identifier. In contrast, SRL refers to the elements of a provisioning and deployment model in CloudML.

Amazon’s CloudFormation language allows creating load balancers that scale in and out virtual machine instances based on horizontal scaling policies. These policies specify conditions on particular metrics that mainly concern the resources used and not the component or application level. In addition, the scaling actions concern a pre-configured virtual machine image, which must be guaranteed to include the appropriate components by the user. Compared to SRL, CloudFormation has a low level of expressiveness in defining scalability rules.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have presented an extended version of the Scalability Rule Language (SRL), a language for specifying scalability rules that support complex adaptation scenarios of multi-cloud applications, i.e., applications that can be deployed across multiple private, public, or hybrid cloud infrastructures and platforms. SRL aims at compensating for the limitations of existing solutions by providing the necessary modelling concepts for specifying behavioural patterns of multi-cloud applications, as well as the scaling actions to change the provisioning and deployment in response to these patterns.

Compared to our previous work [2], this paper provides three contributions that adapt and extend SRL: (a) new modelling concepts for restricting scalability rules through scaling policies, (b) additional modelling concepts for specifying more fine-grained event patterns, and (c) a set of OCL constraints for capturing (part of) the semantics of the domain, which assist modellers in specifying valid scalability rules at design-time.

Future work will focus on implementing a prototype runtime and monitoring system for evaluating the performance and scalability of the language processing, as well as the usability of the language. In addition, we will further extend the set of modelling concepts to improve the sensor description and to enable the specification of more complex metric expressions. Moreover, we will design a concrete textual syntax for the language, which will lower the learning curve and increase the usability. Finally, we are planning to specify the formal semantics of the language.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Commission’s Seventh Framework Programme (FP7/2007-2013) under grant agreement number 317715 (PaaSage).

REFERENCES


In this section, we provide OCL code for some of the constraints of the types identified in Section III. The presentation of each constraint consists of its description followed by the respective OCL code.

**Value Derivation:** The derivation of value for attribute type of `Property` class depends on the `subProperties` association content. If the association is empty, the attribute value should be `MEASURABLE`; otherwise, it should be `ABSTRACT`.

```ocl
class Property{
  attribute name: String;
  attribute description: String[?];
  property subProperties: Property[*]{ordered};
  attribute type: PropertyType {derived}{
    derivation:
      if (subProperties <> null)
        then PropertyType::ABSTRACT
      else PropertyType::MEASURABLE
    endif;
  }
}
```

**Value Correctness Constraint:** The threshold of a metric condition should be included in the value type of the respective metric.

```ocl
invariant CorrectThresholdInMetricCondition:
  if (metric.valueType.oclIsTypeOf(camel::type::Range))
    then metric.valueType.oclAsType(camel::type::Range).
      includesValue(self.threshold)
  else if (metric.valueType.oclIsTypeOf(camel::type::RangeUnion))
    then metric.valueType.oclAsType(camel::type::RangeUnion).
      includesValue(self.threshold)
  else true endif
endif;
```

**Value Combination Correctness Constraint:** If a metric is `RAW`, it should be measured by a sensor and should not be associated with a metric formula. Otherwise, it should be associated with a non-empty list of composite metrics and indirectly associated with a metric formula.

```ocl
invariant RAW_Metric_To_Sensor:
  (self.template.type = MetricType::RAW implies sensor <> null);
```

```ocl
invariant Composite_Metric_To_Components:
  (self.template.type = MetricType::COMPOSITE implies componentMetrics->notEmpty());
```

**Reflexive Association Checking Constraint:** A metric formula associated with a metric template cannot contain this template recursively.

```ocl
invariant MetricTemplate_No_Cycle_In_Formula:
  (self.template.type = MetricType::RAW or
   (self.type = MetricType::COMPOSITE and not(self.checkRecursion(self,self)));
```

**Other Domain Constraint:** A metric’s component metrics must map in a one-to-one manner to the input metric templates of the formula of the metric template.

```ocl
invariant ComponentMetricsMapFormulaTemplates:
  (self.template.type = MetricType::RAW) or
  (self.template.type = MetricType::COMPOSITE and
   self.template.formula.parameters->forAll(
     p | p.oclIsTypeOf(MetricTemplate)
     implies
     self.componentMetrics->exists(
       template = p)
   )
   and self.template.formula.parameters
     ->select(p | p.oclIsTypeOf(MetricTemplate)
     )->size() =
     self.componentMetrics->size()
  );
```