1 Introduction

rSLA is a domain specific language (DSL) for expressing and managing service level agreements (SLAs) in a cloud environment. rSLA is coded in Ruby [?], a dynamic language that enables rapid prototyping and application development.

The rSLA DSL is described by an alphabet and by production rules that help to extend the language. The rSLA programming library provides a runtime engine for deploying and running an rSLA service in a cloud environment.

Although the scientific literature provides plentiful results on automated management of SLAs for distributed computing [?, ?, ?], cloud markets hesitate to adopt such solutions. Provisioning of cloud services is handled either manually or with software tools that do not embrace cloud service characteristics.

Cloud service management does not yet support automated and transparent solutions for the management of leased resources. Additionally, there is no established standard yet for the automatic expression and management of SLAs for cloud services.

rSLA provides a DSL library for the definition of rSLA objects and a runtime engine to create and process such objects. The DSL enables the automated generation of customized SLAs and the transparent management of cloud service compliance.

The rSLA is deployed on the IBM Bluemix platform [?] as a ruby web service using the sinatra gem[1]. A pilot version of the language is currently running for an IBM financial client. The monthly results from using the rSLA language to evaluate the service level compliance of resources leased by the client, showcase the rSLA DSL adequacy in managing cloud services.

How is the paper structured

-what is the problem that the language solves, motivation to solve this problem -language structure, alphabet, production rules -language runtime -current testing, future testing

2 Problem definition/ motivation

Cloud service management has not yet integrated automated and transparent solutions for controlling the provisioning of resources. Cloud customers pay for applications, however they do not have tools to verify their applications' service level values. Similarly, cloud providers do not have tools to evaluate on-demand their service level compliance and to optimally control their resource distribution.

There is no established standard for the automated expression and management of SLAs for cloud services. The research community has proposed solutions for defining SLA context [?, ?] and for processing such information [?, ?] in a distributed computing environment. The cloud industry, however, has not adopted any methods for the automatic management and evaluation of service level values.

[1] Sinatra, `http://www.sinatrarb.com/`

SLA terms enclose data values that are retrieved from monitoring. Monitoring in this context means the systematic observation of metric values that are used by one or more services. An SLA may include numerous such values, which in turn are processed for the evaluation of service level objectives (SLOs). In service level management, an SLO verifies if SLA conditions are violated or not. A cloud provider needs to control during service runtime the value levels of countless SLOs from multiple customers. Cloud markets do not yet provide such tools.

A scheduler is used to coordinate the execution of involved processes for the systematic control of service level values. A cloud scheduler executes tasks that define the monitoring of service metrics and that perform the evaluation of service values at defined points during a service runtime. In a cloud environment, a scheduler additionally takes into account the availability of cloud resources and their distribution among service customers.

In service level management, there are functional and operational dependencies between involved entities. For example, a composite metric, as its name implies, represents the composition result from one or more base, as well as composite metric values. The values of composite metrics in an SLA are used for the definition of conditions and objectives on service levels.

Hence, a schedule configuration for measuring base metric values, may impact the evaluation schedule for one or more composite metric values. Such dependencies raise research questions on how optimal scheduling configurations can apply in a cloud environment for service measurement and evaluation tasks.

Evaluation of service levels consists from multiple computing tasks. SLOs are evaluated in scheduled intervals to determine service level compliance. An evaluation process may define a set of SLO conditions. Such conditions take the form of logical statements that are configured in the SLO definition. Logical statements may require to compare a set of composite values against one or more threshold values.

rSLA provides an SLA programming library and a service runtime engine for creating and managing SLAs in a cloud environment. The language is not intended to be used only by engineers or ruby developers. An important goal of the rSLA design is to provide a high-level, easy to use and to extend tool that is suitable either for human or machine consumption.

IBM Bluemix is a cloud-based platform to build, run and manage applications.

3 rSLA DSL

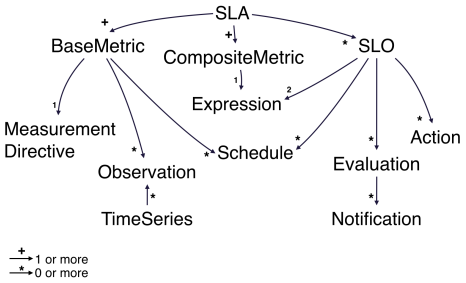
alphabet, vocabulary, language structure

production rules

3.1 rSLA language structure, alphabet

The rSLA language follows the semantic decomposition of the WSLA specification [**?**], where an SLA takes the form of a hierarchical tree with a single

root node and numerous uni-directional edges. In an rSLA tree, the root node represents an SLA object. Figure **??** illustrates diagrammatically the rSLA vocabulary as a tree of objects that the DSL implements. Listing 1 explains the rSLA tree using set notation to highlight the nesting between language objects. Such nesting is inherited from the WSLA specification [**?**].




```
SLA ⊃ { BaseMetric+,
    CompositeMetric∗, SLO+ }
BaseMetric ⊃ {
    MeasurementDirective,
    Observation∗, Schedule∗ }
CompositeMetric ⊃ { Expression
    ∗ }
SLO ⊃ { Expression∗, Action∗,
    Evaluation∗, Schedule∗ }
```

Listing 1: rSLA vocabulary

Fig. 1: rSLA DSL object diagram

Nodes that are close to the tree root in Figure 1, designate SLA branches like base, composite metrics and service level objectives. Edges between nodes are uni-directed to illustrate the rSLA tree hierarchy. The edge direction points to the nested element in the hierarchical relationship. Edges in Figure 1 are labeled with +, * or number symbols to indicate that the multiplicity of nested objects. Hence, an SLA can be defined by sets of base, composite metric and SLO objects.

In the rSLA alphabet nested relationships denote inclusive associations between objects. For example, an SLA includes base, composite metrics as well as SLOs. As shown in Figure 1, edges between rSLA objects do not share same multiplicity rules. The rSLA DSL follows the WSLA specification [**?**] with respect to the definition of rSLA objects and of their basic attributes.

rSLA notification and timeSeries objects are not initially required to build and run SLA instances in a cloud environment, but they may be required while one or more SLA management tasks are processed. Such objects are created by service level management operations like statistical analysis of data coming from monitoring or automated notification reports on scheduled events of service level evaluation.

The rSLA DSL exposes such objects as structured programming blocks that a user can edit and modify according to specific needs. The use and characteristics of rSLA programming blocks are analyzed in Section 3.3. Moreover, a user can introduce new elements in the rSLA vocabulary by integrating their definition in the rSLA programming library.

3.2 rSLA language production rules

The rSLA DSL uses production rules in Backus-Naur form (BNF) notation to describe the syntax of rSLA programming blocks. As we discuss in the next sec-

tion, ruby programming blocks represent the editing rules of the rSLA language. The description of rSLA blocks in BNF notation exemplifies how to use and extend such structures with the rSLA programming library.

In the BNF grammar, every rule is decomposed into another set of rules and literals. The symbol "::=" refers to *is defined* or *is produced by*. Literal or terminal symbols are denoted as "literal" and non-literal symbols are enclosed in brackets <>.

Grammar 2 illustrates an excerpt of the rSLA BNF production rules for the generation of an SLO object. Service level objectives (SLOs) describe promises from a provider to a customer with respect to service provisioning levels [**?**]. An SLA is defined by one or more SLOs. The right side of Line 1 in Grammar 2 summarizes the block of statements that the rSLA DSL requires as input for the creation of a new SLO object.

```
1 <SLO> ::= "slo" "do" <name> <precondition> <objective> <
      schedule> "end" ;
2 <name>::= "name" <string> ;
3 <precondition>::= <expression> ;
4 <objective> ::= <expression> ;
5 <schedule> ::= "schedule" "do" <frequency> <unit> <method> "
      end";
```

Grammar 2: Service level objective (SLO) production rules

An SLO object has a name that is represented by a string. An SLO is also specified by a precondition and by an objective that in the rSLA grammar are denoted with the symbol <expression>. The rSLA language supports the free formation of valid ruby statements. Free-formatted statements define expression objects. In the rSLA grammar an expression object is represented as a non-literal symbol that is further refined into statement combinations. The decomposition of a non-literal expression may produce various statements that associate both literal and non-literal symbols .

An rSLA expression may refer to other rSLA objects and may define numerical and logical expressions for their description. Section 3.3 provides an <expression> example for the creation of a new SLO object using an rSLA programming block.

An SLO object also embeds in its definition one or more schedules for its evaluation. In the rSLA grammar, a schedule represents a non-literal symbol that is syntactically and contextually decomposed into the non-literal symbols of frequency, unit and method. Frequency determines the schedule periodicity, thus how often a scheduler triggers execution tasks with respect to a schedule. The schedule unit determines in time units the intervals to fire schedule events. The non-literal method describes how to run tasks of the defined schedule. A method may consist of either literal or non-literal statements.

3.3 rSLA editing

The rSLA DSL exposes ruby programming blocks for the production of SLO objects. A DSL user can edit and extend such programming blocks according to

domain specific needs. An example of such a ruby programming block for the creation of an SLO object is illustrated by rSLA coding block 3.

rSLA coding block 3: SLO definition

```
slo do
     name "CpuUtil"
     precondition do CPUutilization.value<15 end
     objective do CPUmetric1.value<10 end
     schedule do
        frequency "30"
        unit "m"
        method "every"
    end
end
```

The SLO definition of 3 provides a configuration sample for the generation of SLO objects with the rSLA language. In the ruby block, precondition and objective expressions are defined using both non-literal and literal symbols. In this case, non-literal terms refer to other rSLA objects. For example in the precondition statement, an expression is defined by stating a condition for the numeric value of a CPUutilization rSLA object. Similarly, the objective sets a condition for the numeric value of a CPUmetric object.

The programming logic between a precondition and an objective expression is sequential and can be summarized by the following steps:

```
1 if eval(Precondition) → ¬Precondition then SLO_healthy = true
2  elsif ((¬∃Precondition) ∨ (Precondition = true)) ∧ eval(Objective) → true
3  then SLO_healthy=true
4  end
5 else SLO_healthy=false
6   end
```

Listing 4: rSLA SLO precondition-objective logic

On SLO evaluation, the rSLA engine will evaluate first the precondition statement block. If the logical outcome from the execution of the precondition block is false, the SLO is healthy. In case the precondition is true or if there is no precondition block, the rSLA runtime will proceed with the evaluation of the objective block. If the logical outcome from processing statements of the objective block is true, then the SLO is healthy. Otherwise the SLO evaluation indicates not-healthy. A non-healthy SLO may result into a violation.

4 rSLA runtime engine

The rSLA is implemented as a DSL for cloud SLAs. The rSLA programming library, currently, provides support for the following SLA management operations that take place while a provisioned service is running:

1. SLA creation and activation.

2. monitoring and measurement of service metrics as specified by the SLA context.
3. storage and processing of observed service metric values and of SLO evaluation results.
4. scheduling of rSLA objects.
5. service level evaluation.
6. notification events: reports.

The next paragraphs highlight rSLA implementation aspects for all supported operations.

4.1 SLA creation and activation

As discussed in Chapter 3, rSLA editing takes place using ruby programming blocks. The rSLA language takes advantage of this Ruby coding feature and exposes rSLA objects through multi-lined do..end coding blocks. When an rSLA runtime engine reads a new rSLA block, it generates a new rSLA object that belongs to the block related class. The attributes and function behavior of the generated object are mapped from the ruby block context.

Listing 5 describes an rSLA script for the creation of an SLA and of a base metric. The script can run in an rSLA service runtime environment to generate the two respective objects and to activate a schedule for the value measurement of the base metric.

```
1  sla do
2    tenant "Demo"
3    provider "IBM"
4  end
5
6  basemetric do
7      name "bareMetalProvisioning"
8      unit "provisioningtime"
9      measurementdirective do
10         entity "baremetal"
11         type "jsonArray"
12         source "http://provisioningxlet.stage1.mybluemix
               .net/server/baremetal/provisioningtime"
13         end
14   schedule do
15         frequency "1"
16     unit "m"
17     method "every"
18   end
19 end
```

Listing 5: rSLA SLA (lines 1-4) and basemetric (lines 6-19) creation script

The block for the creation of the SLA object simply describes two strings that designate the names of the SLA tenant and provider. The rSLA engine reads

such strings as the attribute values of the newly created SLA object. Similarly, the rSLA interpreter reads the base metric block and create a new base metric object that inherits the attributes of the rSLA BaseMetric class.

In the base metric block, a DSL user can define BaseMetric object attributes like the base metric name and measurement unit. Additionally, a DSL user can specify directives for the measurement of the created metric. A measurement directive represents a concept that is inherited from the WSLA specification [**?**].

The rSLA DSL exposes a measurement directive as a block of statements that a DSL user can specify. Such statements describe the configuration of a measurement directive object that guides the value measurement of the parent base metric. A measurement directive indicates the result type that is expected from a base metric measurement.

In the measurement directive block, the term $< entity >$ is used for the representation of restful [2] endpoints. Such may have a URL[3] form: `http://a.cloud.com/servers/s3456/metrics/cpuconsumption`. Here the restful endpoint specifies a metric with respect to CPU consumption.

A measurement directive object uses an attribute named *source* to denote the restful endpoint for fetching the relevant base metric value. The rSLA MeasurementDirective class provides an example on how to define measurement directive objects for rSLA base metrics. Such example, like the illustrated block of Listin 5 can be extended accordingly.

Last but not least, a DSL user can specify the details of a schedule for the measurement of the base metric. The schedule block details are explained in Section 4.4.

4.2 monitoring and measurement

rSLA needs a source for monitoring data and a tool for reporting.

4.3 storage and processing

Processing, at the current implementation state, refers to compositions of service value sets for the creation of new rSLA objects.

timeseries: ready to use functions

4.4 scheduling

4.5 service level evaluation

4.6 notification reports

can also be alerts

[2] ReST: `http://en.wikipedia.org/wiki/Representational_state_transfer`

[3] Unique Resource Locator: `http://en.wikipedia.org/wiki/Uniform_resource_locator`

5 rSLA deployment

The language has literally been evolved on the IBM Bluemix cloud platform as a ruby sinatra service. the deployment section demonstrates the rSLA language evaluation by describing the deployment of a complete rSLA service running environment that interprets the rSLA language for the activation and management of SLAs.

5.1 rSLA service

5.2 rSLA Xlets

During its life-cycle, the SLA service requires different services to ensure the management of SLAs. These services are eventually offered as services through Bluemix PaaS. At the time being, rSLA uses one offered service for persistence and a list of services for monitoring and reporting. These latter, are provided as Xlets. An Xlet is a light weight application offered as a service through Bluemix PaaS. This application is designed to facilitate the integration of different offered services spanning over the different layers of the Cloud by providing a clearly defined REST API. An Xlet is customized according to its role in the overall system. As shown in Figure 2, each Xlet provides three interfaces:

- *CFBrokerInterface*: Since the Xlets are provided as services by Bluemix PaaS, they need to offer this generic interface that describes exactly how to provision the service, how to unprovision it, how to bind the service to a given application and how to unbind it.
- *ConfigurationInterface*: In order to ensure multi-tenancy and customization of an Xlet, it should offer an interface to configure its tenancy. This interface could offer other functionalities of customization. It allows in some cases to configure the access credentials for Cloud resources.
- *RuntimeInterface*: This interface describes the main business of the Xlet. It describes the specific functionalities to be offered by the application instance (e.g., monitoring services, reporting services).

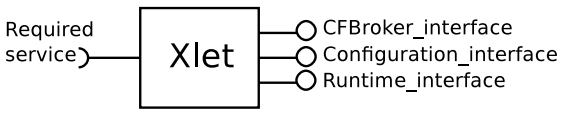


Fig. 2: Xlet generic design

All Xlets respect the same architecture but differs in their implementations from one use case to another. In our current work we defined different monitoring and reporting Xlets. Monitoring Xlets are in-line with the DMTF standard. They

allow collecting monitoring data for a specific type of resources with different granularities. For example

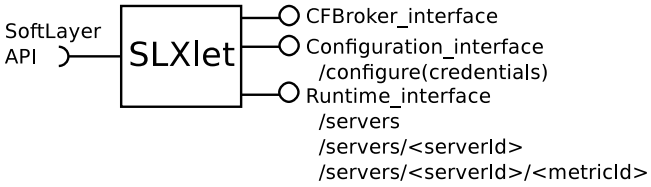


Fig. 3: SoftLayer Xlet design

Using Xlets within Bluemix allows us to benefit from the advantages of this PaaS. TODO-Mohamed ▶*advantages of using Xlets within Bluemix*◀

The different characteristics of an Xlet are described in the following: TODO-Mohamed ▶*characteristics of Xlets*◀ Scalable: this characteristic is inherited from the scalability of Bluemix environment. Reusable Manageable Flexible

As shown in Figure 4

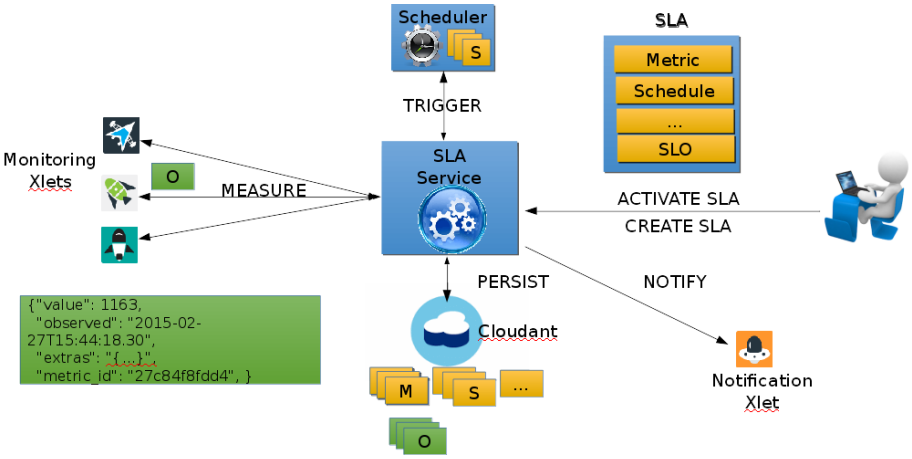


Fig. 4: rSLA runtime

5.3 rSLA persistence