SLAC: A Formal Service-Level-Agreement Language for Cloud Computing

Rafael Brundo Uriarte, Francesco Tiezzi and Rocco De Nicola
IMT Institute for Advanced Studies Lucca, Italy
Emails: {rafael.uriarte, francesco.tiezzi, rocco.denicola}@imtlucca.it

***Abstract*—The need of mechanisms to automate and regulate the interaction amongst the parties involved in the offered cloud services is exacerbated by the increasing number of providers and solutions that enable the cloud paradigm. This regulation needs to be defined through a contract, the so-called Service Level Agreement (SLA). We argue that the current solutions for SLA specification cannot cope with the distinctive characteristics of clouds. Therefore, in this paper we define a language, named SLAC, devised for specifying SLA for the cloud computing domain. The main differences with respect to the existing specification languages are: SLAC is domain specific; its semantics are formally defined in order to avoid ambiguity; it supports the main cloud deployment models; and it enables the specification of multi-party agreements. Moreover, SLAC supports the business aspects of the domain, such as pricing schemes, business actions and metrics. Furthermore, SLAC comes with an open-source software framework which enables the specification, evaluation and enforcement of SLAs for clouds. We illustrate potentialities and effectiveness of the SLAC language and its management framework by experimenting with an OpenNebula cloud system.**

***Index Terms*—Cloud Computing; Service Level Agreement; Formal Languages; Constraint satisfaction problems.**

I. Introduction

Service Level Agreements (SLAs) are the formalization of the characteristics of a service. They can be used to regulate a service offered by one or more providers, and is composed of obligations, penalties and rights of the involved parties.

The majority of cloud providers do not define SLAs for their services, but provide only a textual description of their terms and conditions. This approach has many drawbacks, such as ambiguity and unfeasibility of the automation of the SLA's evaluation. Furthermore, the cloud users typically outsource their core business onto an entrusted cloud [1], which requires formal guarantees of compliance of the service with the SLA.

Several languages to specify SLAs and to automatize their evaluation and negotiation were proposed [2]. Yet, those languages are not able to cope with the set of characteristics of the clouds [2], [3], such as dynamism, deployment models and the growing importance of the broker role. In this paper, therefore, we present a language, named SLAC, for the definition of SLAs specifically devised for the cloud domain.

SLAs require non-ambiguous descriptions. Hence, SLAC focusses on: *(i)* formal aspects of SLAs; *(ii)* supporting multi-party agreements, *(iii)* business and utility aspects and; *(iv)* proactive management of the agreement as well as the cloud system. Intuitively, the formal semantics of SLAC associate a set of constraints to each term of the language, which are evaluated, at design-time, to identify inconsistencies in the specification and, at run-time, to verify the compliance with regards to monitoring data collected from the cloud system.

The formal ground of SLAC is at the basis of a software framework[1] developed for supporting the specification, evaluation and enforcement of SLAs in cloud systems.

To illustrate some of the concepts and to present a practical implementation of the SLAC language throughout the paper, we use an academic cloud case study. We have set up an experimental scenario in IMT Lucca using desktop computers to create a cloud system with the OpenNebula[2] tool. This cloud has been used to provide computational power for research purposes employing the IaaS model. Two versions of the service are available: *(i)* a single and powerful virtual machine (VM); and *(ii)* a cluster of smaller VMs. The first service aims at supporting centralized research applications, while the second one targets applications which require distributed environments.

The rest of the paper is organised as follows. Section II describes the core of SLAC, while Section III examines its business aspects. Section IV describes the architecture and implementation of the SLAC framework. Section V discusses the experiments with the SLAC framework. Section VI reviews related works and Section VII draws conclusions.

II. The SLAC Core Language

In this section we present the core ingredients of the SLAC language, which enable the description of basic SLAs for the cloud computing domain.

A. Syntax and Basic Concepts

The main elements in the SLA are: the description of the contract, the specification of contract terms and the definition of the guarantees for those terms.

Table I shows the syntax of the core language. It is formally defined in the Extended Backus Naur Form (EBNF), in which *italic* denotes non-terminal symbols, while `teletype` terminal ones. Commas may be omitted in sequences of objects (e.g., $Role^+$ stands for $role_1, \ldots, role_n$). Also, some objects are prefixed by a related keyword that is omitted whenever the object is missing. Thus, e.g. the SLA (`id:` Id ... `terms:` $Term$ `guarantees:`) would be

[1]The SLAC Management Framework is a free, open-source software; it can be downloaded from http://code.google.com/p/slac-language/.

[2]OpenNebula's web site: http://opennebula.org/.

written as (`id:` Id ... `terms:` $Term$), i.e. if no guarantee is specified then the keyword `guarantees:` is also omitted.

The non-terminal symbols Id, $PartyName$ and $GroupName$ are implementation specific, hence their details are intentionally left unspecified in this abstract syntax.

The *description* of a *SLA* comprises a unique identification code (*Id*) and at least two parties involved in the SLA. A party is constituted of an optional *PartyName* and one or more *Role*s. The definition of multiple roles for a single party enables the support of scenarios, such as community clouds, in which a provider can be also a consumer. Furthermore, the definition of only roles (no *PartyName*), enables the creation of templates, both for the definition of offers and of requests.

The *terms* of the agreement express the characteristics of the service along with their respective expected values. Each SLA requires the definition of at least one term, which can be either a *Metric* or a *Group* of terms. In the SLAC language, for each term, the parties involved should be defined, i.e. the party responsible to fulfil the term (a single party) and the contractors of the service (one or more). This explicit definition contributes to support multi-party agreements, to reduce ambiguity and to leverage the role of the broker in the agreements.

A *metric* can be of three types: *(i) NumericMetric*, which are constrained by open or closed *Intervals* of values (that can be defined explicitly in the SLA or inferred from the evaluation of an expression) and a particular *Unit* (e.g. milliseconds, gigabytes); *(ii) BooleanMetric*, which can assume `true` or `false` values; and *(iii) ListMetric*, whose values are in a list.

The metrics and the way to measure them are pre-defined in the language in the light of the requirements of the cloud domain. As discussed in Section VII, they could be extended according to the needs of the involved parties. We have also pre-defined a set of valid items for each list metric available in the language; this helps to avoid ambiguity or spelling errors in their specification. We refer the interested reader to [4] for a complete account of metric definitions.

Expressions (*Expr*) have two main uses: the specification of interval values in numeric metric and of conditions in guarantees. An expression can be a *Literal*, a *NumericMetric* with a parameter indicating if its upper or lower interval should be used, infinity (`infty`) that does not set a lower or upper constraint in the metric, or the composition of sub-expressions by means of mathematical *Operator*s. For example, the numeric metric `RAM in [(2 + 4 * cCPU(min) ), 20]` sets the minimum amount of memory RAM in the context as 4 times the lower bound of the *cloud CPU unit* required in the SLA, plus 2. Notably, group instantiations cannot use the infinity value in the interval definitions.

Another feature of the language is the specification of granularities for terms using groups. A group of terms (*Group*) is identified by a name (unique in the contract) and is composed of one or more terms. Groups enable the re-use of the same term in different contexts. For example, in our use case let us suppose that that a consumer needs a centralized and two distributed VMs. In this case, the characteristics of each VM could be defined in a group, for example: `VM_Centralized` is a group defining 99% of availability, 4 cores and 16 GB of RAM for a machine, while `Small_VM` is a group specifying an availability of only 90%, 1 core and 1 GB of RAM.

Additionally, groups can include the instantiation of other groups. Continuing with the example above, to define a cluster constituted of the two less powerful VMs previously defined and specifying the maximum round-trip delay to the server as 0.6 ms, the SLA could have a third group, named `Cluster`, instantiating two `Small_VM` and the `RT_delay`. However, recursive definitions are not allowed, that is, a group cannot (directly or indirectly) refer to itself.

Notably, to use a group in a SLA definition, it is not sufficient to define it, as illustrated above, but it is also necessary to instantiate the group by specifying the number of instances. For example, previously we defined the group `Small_VM` that specified the characteristics of a type of VM; to actually deploy 2 instances of this group in the SLA, the term `[2, 2] of VM_Small` must be specified.

Guarantees are optional in the agreement. However, they can play a significant role as they ensure that the terms of the agreement will be enforced or, in case of violation, define the actions that will be taken. Specifically, a guarantee refers to a term defined in *Terms* section of the agreement, that is a single term (i.e. *NumericMetric, ListMetric, BooleanMetric*), an instantiation of a group (*GroupName*) or to any term (using the reserved keyword `any`).

When an event occurs (e.g., a violation), specified conditions are tested (defined by an *Expr*) and the execution of one or multiple *Action*s is requested (*ConditionAction*). Notably, actions may require the specification of the involved parties.

B. Formal Semantics

The semantics of a service level agreement are formulated as a Constraint Satisfaction Problem (CSP) that verifies: (i) at negotiation-time, whether the terms composing the agreement are consistent; and (ii) at enforcement-time, whether the characteristics of the service are within the specified values.

More specifically, the formal semantics of SLAC is given in a denotational style by the definitions in Table II. The semantics of a SLA is a function $[\![SLA]\!]$ that returns a pair composed of a set of group definitions and a constraint representing the semantics of SLA's terms. This pair constitutes the CSP associated to the agreement.

In a SLA, a group is defined by an identifier and a set of terms. The semantics of a $Group$ is defined by a function $[\![Group]\!]^D$ that is parameterised by the set D of all group definitions previously determined by the SLA translation. This parameter is used in order to enable the instantiation of other groups in the current group. Due to its simplicity and readability, this approach was chosen instead of pre-parsing techniques; however, it implies that only ordered instantiation is permitted, i.e. within each group only groups defined previously in the SLA (i.e. belonging to D) can be instantiated.

The semantics of a collection of terms $Term^+$ is defined by the logical conjunction of the constraints corresponding to each term. These constraints are generated by function $[\![Term]\!]_g^D$,

SLA	::=	id: *Id* parties: *PartyDef* *PartyDef*$^+$
		term groups: *Group** terms: *Term*$^+$ guarantees: *Guarantee**
PartyDef	::=	*PartyName*$^?$ roles: *Role*$^+$
Role	::=	consumer | provider | carrier | auditor | broker
Group	::=	*GroupName* : *Term*$^+$
Term	::=	*Party* -> *Party*$^+$: *Metric* | [*Expr*, *Expr*] of *GroupName*
Party	::=	*Role* | *PartyName*
Metric	::=	*NumericMetric* not$^?$ in *Interval* *Unit* | *BooleanMetric* is *Boolean*
	|	*ListMetric* has not$^?$ {*ListElement*$^+$} or {*ListElement*$^+$}*
NumericMetric	::=	cCPU | RT_delay | response_time | RAM | availability | jitter | ...
Interval	::=	]*Expr*, *Expr*[|]*Expr*, *Expr*] | [*Expr*, *Expr*[| [*Expr*, *Expr*]
Expr	::=	*Literal* | infty | *NumericMetric*(*Parameter*) | *GroupName* | *Expr* *Operator* *Expr*
Parameter	::=	min | max
Operator	::=	+ | - | * | / | > | >= | < | <= | and | or
Unit	::=	gb | mb/s | ms/min | minute | seconds | ms | month | ...
BooleanMetric	::=	back_up | replication | data_encryption | ...
ListMetric	::=	operating_systems | jurisdiction | hypervisor | ...
ListElement	::=	occi | ec2 | kvm | xen | ...
Guarantee	::=	on *Event* of (*Party* => *Party*$^+$:)$^?$ *GuaranteeMetric* : *ConditionAction*
GuaranteeMetric	::=	(*GroupName*:)* *NumericMetrics* | (*GroupName*:)* *ListMetrics*
	|	(*GroupName*:)* *BooleanMetric* | (*GroupName*:)* *GroupName* | any
Event	::=	violation
ConditionAction	::=	(if *Expr* then *Action*$^+$)$^+$ (else *Action*$^+$)$^?$ | *Action*$^+$
Action	::=	(*Party* => *Party*$^+$:)$^?$ *ManagementAction*
ManagementAction	::=	notify | renegotiate

TABLE I: Syntax of the SLAC language.

which takes as parameters the set D of group definitions and a further parameter g, which is a group name used to identify the context of the term and generate the appropriate constraint identifier in the term translation. Notably, value $\emptyset$ for parameter g is used for evaluating terms specified in the terms section of the agreement. The group definitions are only added to the set of definitions in the first field of the SLA pair, defining the constraint context. Therefore, a group is effectively considered in the SLA only if instantiated directly (in the terms section) or indirectly (in another group instantiated in the terms section).

Terms can be of two types: metric instantiations $Party -> Party^+ : Metric$ and group instantiations $[Expr, Expr]$ of $GroupName$. In the former case, the parties are used as a parameter for the evaluation of the metric. In the term translation we use a compact notation for metric and group names (e.g., NM stands for $NumericMetric$).

For the *numeric metrics* we have eight definitions, only differing for the type of interval and the presence of the not operator. We have reported in Table II just the two definitions for the closed interval; the others are similar and, hence, omitted. Essentially, the definition of a numeric metric is translated into a numeric constraint. This translation follows these steps: (i) the expressions are evaluated as numeric values; (ii) these values, together with the specified *unit*, are converted by the *con* function into values in the standard unit of that metric; (iii) the name of the constraint variable is composed by using the group name, the involved parties and the name

Boolean metrics are the simplest case, which the monitoring value of the defined metric has to be equal to the one specified in the SLA.

A *list metric* is translated into a constraint that checks whether all elements of at least one of the lists $\{LE^+\}$ are contained in the set of values of the specified metric.

Finally, the group instantiation corresponds to the conjunction of a constraint verifying the number of instances being run on the system and a collection of constraints for checking the behaviour of the instances. In particular, the constraint corresponding to each instance n (with $n \in \mathbb{N}$) is obtained from the constraint c of the group definition by applying the function $c \downarrow_{g,n}$; it replaces each constraint variable v occurring in c by $g : v : n$. Notably, we add a number of instance constraints corresponding to the maximum number of defined instances. For example, if two instances of a group are the upper bound of the defined interval, the constraint of that group is added twice (whose variables are properly renamed to avoid conflicts and ambiguity). A drawback in using the maximum number of instances is that the semantics related to groups might generate a large number of constraints, in particular if many groups refer to other groups. However, considering that SLAs are usually of limited size, the computational capacity and the efficiency of

$$
\begin{aligned}
\llbracket SLA \rrbracket &= \Big\langle\ \llbracket Group^{*} \rrbracket,\ \llbracket Term^{+} \rrbracket_{\emptyset}^{\llbracket Group^{*} \rrbracket}\ \Big\rangle \\
\llbracket Group_1 \ldots Group_n \rrbracket &= \Big\{\ \underbrace{\llbracket Group_1 \rrbracket^{\emptyset}}_{D_1}, \underbrace{\llbracket Group_2 \rrbracket^{\emptyset \cup D_1}}_{D_2}, \underbrace{\llbracket Group_3 \rrbracket^{D_1 \cup D_2}}_{D_3}, \ldots, \llbracket Group_n \rrbracket^{D_1 \cup \ldots \cup D_{n-1}} \Big\} \\
\llbracket GroupName\colon\ Term^{+}\ \rrbracket^{D} &= GroupName \overset{\text{def}}{=}\ \llbracket Term^{+} \rrbracket_{GroupName}^{D} \\
\llbracket Term_1\ \ldots\ Term_n \rrbracket_{g}^{D} &= \llbracket Term_1 \rrbracket_{g}^{D}\ \wedge\ \ldots\ \wedge\ \llbracket Term_n \rrbracket_{g}^{D} \\
\llbracket Party -> Party^{+} : Metric \rrbracket_{g} &= \llbracket Metric \rrbracket_{g,\ (Party \rightarrow \llbracket Party^{+} \rrbracket)} \\
\llbracket Party_1\ \ldots\ Party_n \rrbracket &= Party_1\ ,\ \ldots\ ,\ Party_n \\
\llbracket NM\ \texttt{in}\ \texttt{[}Expr_1, Expr_2\texttt{]}\ Unit \rrbracket_{g,p} &= con(\llbracket Expr_1 \rrbracket, \llbracket Unit \rrbracket) \leq \llbracket NM \rrbracket_{g,p} \leq con(\llbracket Expr_2 \rrbracket, \llbracket Unit \rrbracket) \\
\llbracket NM\ \texttt{not in}\ \texttt{[}Expr_1, Expr_2\texttt{]}\ Unit \rrbracket_{g,p} &= (con(\llbracket Expr_1 \rrbracket, \llbracket Unit \rrbracket) > \llbracket NM \rrbracket_{g,p}) \vee (\llbracket NM \rrbracket_{g,p} > con(\llbracket Expr_2 \rrbracket, \llbracket Unit \rrbracket)) \\
\llbracket BM\ \texttt{is}\ Boolean \rrbracket_{g,p} &= \llbracket BM \rrbracket_{g,p} == Boolean \\
\llbracket LM\ \texttt{has}\ \{LE^{+}\}\ (or\ \{LE^{+}\}_{i})_{i \in I} \rrbracket_{g,p} &= \{LE^{+}\} \subseteq \llbracket LM \rrbracket_{g,p} \vee \textstyle\bigvee_{i \in I} \{LE^{+}\}_{i} \subseteq \llbracket LM \rrbracket_{g,p} \\
\llbracket [Expr_1, Expr_2]\ of\ GN \rrbracket_{g}^{\{GN \overset{\text{def}}{=} c\}\ \cup\ D'} &= (\llbracket Expr_1 \rrbracket\ \leq\ \llbracket GN \rrbracket_{g}\ \leq\ \llbracket Expr_2 \rrbracket)\ \wedge\ \textstyle\bigwedge_{0 < n \leq \llbracket Expr_2 \rrbracket} c\downarrow_{g,n} \\
\llbracket NM \rrbracket_{g,p} &= g : p : NM \\
\llbracket BM \rrbracket_{g,p} &= g : p : BM \\
\llbracket LM \rrbracket_{g,p} &= g : p : LM \\
\llbracket GN \rrbracket_{g} &= g : GN
\end{aligned}
$$

TABLE II: Semantics of the SLAC language.

constraint solvers, this does not represent a problem in practice.

We have seen so far how a SLA is translated into a CSP. This can be used directly for checking the consistency of the terms within the SLA, at design time. Similarly, at run-time, data representing the status of the system is collected by the monitoring system of the cloud and, then, translated to a CSP. This translation gives a high-degree of flexibility; for example, in case the monitoring measures are not exact or consist of multiple values from a specific period, they can be specified as intervals. The SLA and monitoring CSPs are then combined for the evaluation at enforcement time.

After such evaluation of the SLA, the guarantees specified in the SLA are also evaluated. In particular, when a metric of the SLA is violated (*Event*), a *Condition* might be verified and the corresponding list of *Action*s is sent to the manager responsible for executing them. For instance, in case of a violation of a response time (*Event*), if this is higher than 10 ms (*Condition*) the provider has to notify the consumer (*Action*).

C. Example

In Table III we show an excerpt of a SLA, related to the case study presented in the Introduction of this paper, and the corresponding constraints generated by the semantics. In this example, all types of metrics and three groups are used. The generated constraints include the instantiated metrics, the duly renamed metrics in the $Cluster$ group and the ones in the $Small\_VM$ group added twice, as the $Cluster$ group makes use of maximum 2 of this group. Note that, as the $Centralized\_VM$ is not instantiated, no constraint is generated.

III. Business Aspects

Few works on SLAs deal with the business aspects of the provided services. In fact, the main existing SLA languages, at most, partially cover those aspects in the SLA [5]. This represents a significant barrier for the adoption of such languages, as they require non-standard and non-trivial, extensions to support important characteristics of the SLA.

To support the business aspects in SLAC we adopt the scheme of Karanke and Kirn [5] that divides the SLA into three phases: *(i) Information Phase*, in which the details about the services, consumers and providers are browsed and collected; *(ii) Agreement Phase*, in which the participants negotiate and define the terms and the pricing model; and *(iii)* the *Settlement Phase* which is related to the evaluation and enforcement of the SLA. Those requirements are fulfilled through an extension of the SLAC language presented in this Section.

The information phase requires the support of offers and requests of services. The core language of SLAC natively supports these features through the specification of parties by means of their roles, the use of groups without instantiation (to specify in the offers the services available) and intervals for numeric metrics (for example, a VM with memory between 4 and 8 GB). Such features enable the definition of *templates* of services that, when accepted, fulfil the missing data generating a SLA.

The agreement phase encompasses the negotiation and the support for different pricing models in the agreement. Those models are classified as flat and variable pricing. *Flat* pricing indicates that the price of a service is the same for the whole duration of the agreement. A *variable* pricing model instead, allows fluctuations in the price during the agreement.

To illustrate the differences between the flat and variable models we exemplify with a solution that employs both: the

SLA	**Constraints:**

term groups:
 Small_VM:
  Imt → Rafael:cCpu in [1,2] #
  Imt → Rafael:RAM in [1,1] #
 Centralized_VM:
  Imt → Rafael:cCpu in [2,4] #
  Imt → Rafael:RAM in [8,16] #
 Cluster:
  Imt → Rafael:RT_delay in [0.0,0.6] ms
  [2,2] of Small_VM
terms:
 [1,1] of Cluster
 Imt → Rafael:interface has {OCCI, UCI} or {EC2}
 Imt → Rafael:replication is True
	#SLA Terms
	1 ≤ Cluster ≤ 1 ∧
	imt,rafael:replication == $True$ ∧
	($\{OCCI, UCI\}$ ⊆ imt,rafael:interface ∨ $\{EC2\}$) ⊆ imt,rafael:interface) ∧
	
	#Constraints of the Cluster group
	0.0 ≤ Cluster:imt,rafael:RT_delay:0 ≤ 0.6 ∧
	2 ≤ Cluster:Small_VM:0 ≤ 2 ∧
	
	#Constraints of Small_VM, instantiated in Cluster Group
	1 ≤ Cluster:Small_VM:imt,rafael:cCpu:0:0 ≤ 2 ∧
	1 ≤ Cluster:Small_VM:imt,rafael:RAM:0:0 ≤ 1 ∧
	1 ≤ Cluster:Small_VM:imt,rafael:cCpu:1:0 ≤ 2 ∧
	1 ≤ Cluster:Small_VM:imt,rafael:RAM:1:0 ≤ 1

TABLE III: Example of the semantics at work on the academic cloud case study.

Amazon Cloud Service[3], providing on-demand VM instance. In this case the price is the same for the duration of the service, i.e. the pricing model is flat. However, Amazon also provides a service named Spot Instances that employs an auction scheme using the variable price model to offer multiple resources. The price of the services fluctuates according to the customers offers. Hence, a consumer makes an offer and, when his bid exceeds the current price of the requested service, the service is provided to this consumer. Then, if the current service price becomes higher than the consumer's bid the service is interrupted and resumed when it becomes lower again.

These two models, flat and variable, can use different pricing schemes. Table IV lists these schemes (based on [6]), ticking the columns *Flat* and *Variable* if the model is available in that scheme and setting the requirements to support them in the SLA language.

The modifications in the SLAC language to support these schemes include the SLA expiration date, which is used to define a time window in which the service is valid and, depending on the model, other mechanisms. To enable the flat model, the language should allow the specification of standard service offers and requests. The support of the variable model requires the specification of a dynamic functions in the SLA to retrieve information (the current price) and, in some cases, to specify a price expiration date, which enables the modification of the price after its expiration. After the expiration, this price is updated according to the chosen scheme.

Finally, the settlement phase is related to the enforcement, accounting and billing of the SLA. It encompass the definition of the billing period (that, in turn, depends on the scheme), new metrics and business related actions. The metrics for the business aspects also include Key Performance Indicators (KPI), such as the response time of the provider's support service (`support_RT` or support response time) and the Mean Time To Repair (`MTTR`) failures on the system. Business related actions enable, for example, to `reserve` resources for future instantiation, to offer financial `credits` for service usage (with the same party), to allow the employment of complementary services by a partner with the provision of `bonus` resources, and to define payments (`pay`) to the involved partners. For the complete list of them, again we refer the reader to [4].

[3]http://aws.amazon.com/

Table V summarizes the modifications on the syntax of the core language to support the business aspects discussed above. In this Table, the symbol ::= stands for a new definition, while + = adds the defined elements to the core language's definition.

All business aspects are optional and can be used both, at top level on the SLA specification (in this case, they are valid for the whole agreement) and within the definition of a $Group$. The possibility of specifying the pricing model and the billing for groups gives the flexibility for defining independent business aspects for multi-party agreements.

To support the pricing schemes, the SLA needs to retrieve external information, in particular the current price of the service. This is achieved through the function `from`, which takes as parameter the $Address$ (e.g. an URL) from which the information is retrieved.

The pricing model and schemes are particularly useful for searching compatible services and for negotiation. Concerning their evaluation, the variable model requires the retrieval of the current price and time, and evaluates them according to the pricing scheme. These steps are pre-defined for each scheme and are performed separately from the CSP evaluation.

Two billing models are available in SLAC. An agreement can be post-paid, by defining the billing frequency (e.g., `monthly`), or pre-paid, by specifying an $ExpirationDate$ (for instance, `pre-paid till: 10/05/2015`).

IV. SLAC Management Framework

The SLAC management framework has two main components: the Service Scheduler and the SLA Evaluator. These components are integrated with a monitoring system which retrieves the data for the evaluation of the SLA from the cloud and with a cloud management platform. In the following subsections we detail the framework and its implementation.

Pricing scheme	**Flat**	**Variable**	**Flat Support**	**Variable Support**
Fixed pricing	✓	✓	Offer, Request	Price Expiration, Current Price
Bilateral Agreement	✓	✓	Offer, Request	Price Expiration
Exchange	✓	✓	Offer, Request	Service Expiration
Auction	✓	✓	Offer, Request	Service Expiration, Current Price
Posted Price	✓		Offer, Request	
Tender	✓		Offer, Request	

TABLE IV: Pricing schemes.

```
           SLA  +=   PricingBilling
        Groups  +=   PricingBilling
PricingBilling  +=   pricing model: Model pricing scheme: scheme
                     Pricing expiration date: ExpirationDate? billing: Billing?
         Model  ::=  flat | variable
        scheme  ::=  fixed_pricing | bilateral_agreement | exchange | auction
                |    posted_price | tender
       Pricing  ::=  price expiration date: ExpirationDate? service current price: from(Address)?
       Billing  ::=  pre-paid till: ExpirationDate | monthly | yearly | ...
       Actions  +=   pay Interval Unit | reserve: Expr Unit of (Term+)
                |    credit Interval Unit | bonus: Expr Unit of (Term+)
 NumericMetric  +=   offer | support_RT | MTTR | ...
    ListMetric  +=   currency | support_type | ...
```

TABLE V: Syntax of the business aspects for the SLAC language.

A. Scheduler

The scheduler receives and processes requests from the customers after the negotiation phase. In the first step the scheduler sends the SLA to the the parser, which converts the SLA into constraints and sends them to a consistency checker.

In the deployment of the services, the scheduler makes a requests for the deployment of the new service to the cloud management tool. After that, the scheduler sets up the monitoring system to retrieve the data concerning the metrics related to the SLA. From this point forward, it repeats the following phases till the end of the agreement: it receives the monitoring data, sends it to the SLA evaluator and reports the results to the interested parties.

B. SLA Evaluator

The SLA evaluator receives the SLA written in SLAC, parses it and generates a set of constraints corresponding to the specification along with the service definition, which are sent to the Scheduler for the deployment.

When the monitoring data is received, it is transformed into a set of constraints and, together with the constraints generated from the SLA, are passed to a constraint solver that verifies their satisfiability. In case of non-satisfiability, the SLA guarantees are evaluated and the due actions will be activated.

It is worth noticing that not all monitoring data is required for the evaluation of the constraints, which enables the evaluation even with partial observations of the system. For instance, in case one of the monitoring components fails, even if the data of a metric is not collected, the satisfiability of the SLA can be tested with the available information.

C. Implementation

The current prototypical version of the framework works in a centralized fashion and implements only the core part of SLAC; we leave the integration of business aspects for future developments. Its components were developed using the Python programming language, following the described architecture.

To retrieve the necessary data to evaluate the SLA, the scheduler is integrated with the Panoptes [7] monitoring system. The deployment of the service is performed by the scheduler integrated with the cloud management tool OpenNebula[4].

The SLA Evaluator parses the SLA with the Simpleparse library[5], by relying on the EBNF grammar reported in Table I. The constraints, in turn, are handled by the Evaluator using the Z3 solver [8].

Since several agreements in the cloud domain are still manually defined by the parties, a great advantage of the adoption of SLAC is its ease of use, mainly in scenarios where non-experts are involved. Therefore, we developed a plug-in for the Eclipse platform that provides features, such as syntax highlighting and completion, for the definition of SLAC SLAs.

Figure 1 shows the technologies used in the implementation of the SLAC Management Framework and summarizes parsing, consistency checking and evaluation phases.

[4]http://opennebula.org
[5]http://simpleparse.sourceforge.net/

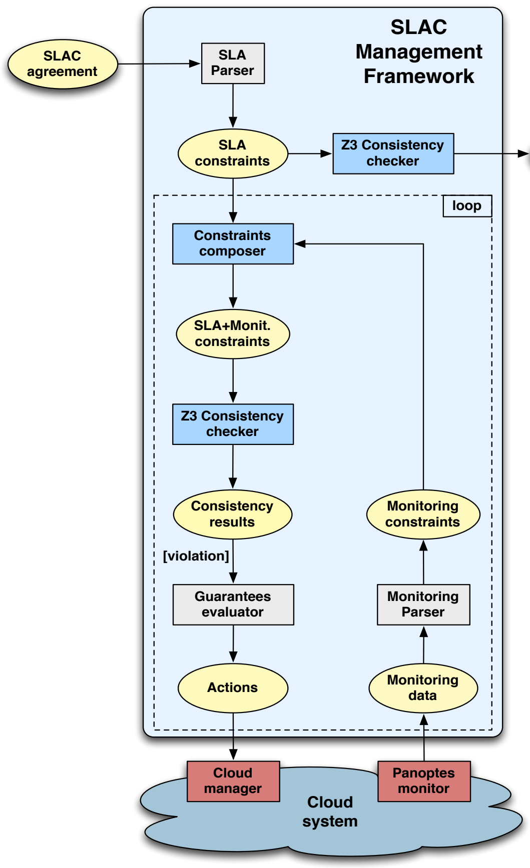

Fig. 1: SLAC Management Framework: evaluation process.

V. Experimentation

To illustrate the expressiveness of SLAC and its practical benefits, we carried out a series of experiments on the academic cloud case study described in the Introduction.

Since negotiation protocols are not part of this work, the researchers have selected the offered IaaS cloud services according to pre-defined SLAs. These SLAs are specified in SLAC and enforced with its framework. The cloud system, resulting from the integration of our framework with OpenNebula, when prompted by a request, automatically deploys the virtual machines, configures the monitoring system and periodically evaluates the SLAs.

To compare how different SLAs affect the system functioning, we collected information of five hours of system use, creating a *dataset* which was used in all experiments.

The first experiment is related to the SLA presented in Table III, which instantiates a service without guarantees. Due to their absence, the users as well as the system administrator receive no information concerning the service provision, i.e. they are not even notified when a SLA is violated.

In the second experiment we extend the SLA to include guarantees for the quality of service and business aspects. In the considered case study if the SLA is violated the user is guaranteed to receive a compensation. Indeed, in this scenario, in which the service is not paid, the provider offers extra credits to the customer for future use of the service (i.e., one hour of bonus in case of cluster service). With the same dataset employed for the first experiments, the SLAs were violated in average six times. To depict the behaviour of the services, Figure 2 shows the `RT_delay` (with 0.6 ms as upper bound) of a cluster group running during the experiments. Analysing the notifications and the enforcement information of the framework, we found out that the main reasons for the violations in the SLAs were the slow network and the lack of proactive management by the cloud system. The insights from the SLA enhanced with guarantees allows both users and the cloud administrator to be notified of the violations and, thus, to take the appropriate actions in order to deal with the issue. Actually, guarantees pave the way for the development of self-managing cloud systems, as well as for the definition of dynamic SLAs (i.e. contracts that enforce different conditions according to the evolution of the system).

VI. Related Works and Discussion

Most cloud computing providers offer SLAs as text description, and shift the burden of monitoring and enforcing the SLA to the customers [9]. However, several solutions were proposed in the literature to define and enforce a machine-readable SLA. The most well-known are: SLAng [10], a domain specific language for IT services; and WSOL [11], WSLA [12], WS-Agreement [13] and SLA*[14], for general purpose services. Table VI shows the comparison between SLAC and those languages for the definition of SLAs. The selected features are related to the scope of our solution, and in no way conclusive.

These proposals provide a high-level specification to support multiple domains and none of them are specifically devised for cloud computing. The work closest to ours is SLAng due to its formalism and domain specific nature (IT services domain). SLAng only considers two parties in the SLA, i.e. the consumer and the provider, which limits its use in the cloud domain (e.g., SLAng cannot be applied to scenarios involving the community cloud model or brokers). Moreover, the specification of SLAs is rather complex, requiring the full comprehension of the model and technologies used in the specification of the language.

Our work differs from those mentioned in the comparison table in many aspects: it emphasizes the formal aspects of SLA; considers the particularities of the cloud computing domain; specifies the semantics of the SLA conformance verification; supports some of the most important business aspects of the SLA; and provides the base for dynamism in SLAs.

An important decision while defining a SLA specification language is the appropriate level of granularity and trade-off between expressiveness and specificity [1]. In the case of cloud computing, a wide range of services is available and the domain has an uncommon set of characteristics, which are difficult to grasp in a SLA specification language. Moreover, contracts related to cloud services mostly involve only aspects from the cloud domain, i.e. they are not multi-domain. Therefore, SLAC was defined as a domain specific language for clouds.

The metrics available in the language are pre-defined in the light of the requirements of the cloud domain. However, they

	Features	WSOL	WSLA	SLAng	WSA	SLA*	SLAC
General	Cloud Domain	-	-	-	-	□	■
	Multi-Party	-	-	-	-	-	■
	Broker Support	-	-	-	-	-	■
Business	Business Metrics	□	□	□	□	□	■
	Price schemes	□	□	-	□	□	■
Formal	Syntax	■	■	■	■	■	■
	Semantics	-	-	□	-	-	■
	Verification	-	-	□	-	-	■
Tools	Evaluation	■	■	■	■	■	■
	Open-Source	■	-	■	■	■	■

TABLE VI: Comparison of the SLA languages. ■ represents a feature covered in the language, □ partially covered feature and - when no support is provided.

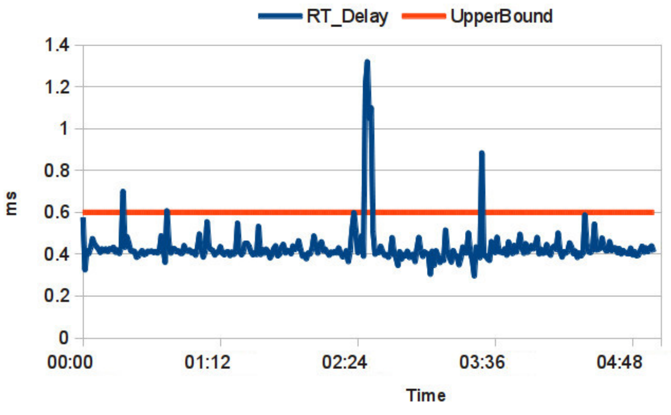

Fig. 2: Delay of the communication of two VMs of a Cluster group.

are easily extensible, i.e. the language and the architecture of the framework permit both, the addition and redefinition of metrics. The drawback of these changes is the incompatibility between SLAs using non-standard metrics and SLAs written in the standard language. To decrease the impact of such changes, the user could provide a translation function of the new metric using one or more built-in metrics.

The set of characteristics of the SLAC language, such as multi-party, group definition and specification of the involved parties on each term, enables and leverages the definition of SLAs including a broker.

Concerning the performance and scalability of the SLA management framework implementation, it evaluates in average 35 SLAs per second, each one specifying 20 metrics, using a 2.8 GHz i7 Processor. The same test with 100 metrics per SLA evaluates 12 SLAs per second. It should be stressed that this framework is a proof-of-concept; more efficient distributed implementations can be envisaged.

Finally, it is worth noticing that SLAC focusses on Infrastructure-as-a-Service (IaaS). Since most requirements of the other service models are also taken into account, we do not envisage any major issue in extending SLAC to cover them, especially the case of Platform-as-a-Service (PaaS). We plan to specifically address the other models in future works.

VII. Conclusions

In this paper we propose SLAC, a language for SLAs based on the specific requirements of cloud computing. The syntax of the core language is described as well as the semantics of the conformance check of SLAs, defined in terms of constraint satisfaction problems. It has also presented a software framework implementing the language and an extension of the language to support business aspects has been defined. Finally, an academic cloud case study has been used to illustrate the approach and to experiment with the software framework.

We intend to provide support to other typical aspects of SLAs, such as schedule times for metrics, negotiation and extend SLAC to other service models. Also, we plan to develop a specific case study involving business aspects, to demonstrate the applicability of the language in real-world solutions.

Acknowledgements

This work has been partially sponsored by the EU project ASCENS (257414) and by the MIUR PRIN project CINA (2010LHT4KM).

References

[1] T. Dillon, C. Wu, and E. Chang, "Cloud Computing: Issues and Challenges," *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, pp. 27–33, 2010.
[2] L. Wu and R. Buyya, "Service Level Agreement (SLA) in Utility Computing Systems," *arXiv preprint arXiv:1010.2881*, 2010.
[3] R. Buyya, C. S. Yeo, and S. Venugopal, "Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities," in *HPCC*. IEEE, 2008, pp. 5–13.
[4] R. B. Uriarte, F. Tiezzi, and R. D. Nicola, "Definition of the Metrics and Elements of the SLAC Language," IMT Institute for Advanced Studies Lucca, Lucca, Italy, Tech. Rep., 2014. [Online]. Available: http://sysma.imtlucca.it/tools/slac/
[5] P. Karaenke and S. Kirn, "Service level agreements: An evaluation from a business application perspective," *Proceedings of eChallenges*, 2007.
[6] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger, "Economic models for resource management and scheduling in grid computing," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 13-15, pp. 1507–1542, 2002.
[7] R. B. Uriarte and C. B. Westphall, "Panoptes: A monitoring architecture and framework for supporting autonomic clouds," in *NOMS*. IEEE, 2014, pp. 1–5.
[8] L. M. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.
[9] P. Patel, A. Ranabahu, and A. Sheth, "Service Level Agreement in Cloud Computing," in *Cloud Workshops at OOPSLA09*, 2009.
[10] D. D. Lamanna, J. Skene, and W. Emmerich, "SLAng: A Language for Defining Service Level Agreements," in *FTDCS*. IEEE, 2003, pp. 100–106.
[11] V. Tosic, B. Pagurek, and K. Patel, "WSOL - A language for the formal specification of various constraints and classes of service for web services," in *ICWS*, vol. 3. IEEE, 2002, pp. 375–381.
[12] H. Ludwig, A. Keller, A. Dan, R. King, and R. Franck, "Web service level agreement (WSLA) language specification," *IBM Corporation*, 2003. [Online]. Available: http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf
[13] A. Andrieux, K. Czajkowski, A. Dan, and K. Keahey, "Web services agreement specification (WS-Agreement)," Global Grid Forum, Tech. Rep., 2004. [Online]. Available: https://www.ggf.org/Public_Comment_Docs/Documents/Oct-2006/WS-AgreementSpecificationDraftFinal_sp_tn_jpver_v2.pdf
[14] K. T. Kearney, F. Torelli, and C. Kotsokalis, "SLA*: An abstract syntax for Service Level Agreements," in *GRID*. IEEE, 2010, pp. 217–224.