SLA⋆: An Abstract Syntax for Service Level Agreements

Keven T. Kearney
Ricerca & Innovazione
Engineering Ingegneria Informatica
Rome, Italy
Email: keven.kearney@eng.it

Francesco Torelli
Ricerca & Innovazione
Engineering Ingegneria Informatica
Rome, Italy
Email: francesco.torelli@eng.it

Constantinos Kotsokalis
IT & Media Center
University of Dortmund
Dortmund, Germany
Email: constantinos.kotsokalis@udo.edu

***Abstract*—This paper describes SLA⋆, a domain-independent syntax for machine-readable Service Level Agreements (SLAs) and SLA templates. Historically, SLA⋆ was developed as a generalisation and refinement of the web-service specific XML standards: WS-Agreement, WSLA, and WSDL. Instead of web-services, however, SLA⋆ deals with services in general, and instead of XML, it is language independent. SLA⋆ provides a specification of SLA(T) content at a fine-grained level of detail, which is both richly expressive and inherently extensible: supporting controlled customisation to arbitrary domain-specific requirements. The model was developed as part of the FP7 ICT Integrated Project SLA@SOI, and has been applied to a range of industrial use-cases, including; ERP hosting, Enterprise IT, live-media streaming and health-care provision. At the time of writing, the abstract syntax has been realised in concrete form as a Java API, XML-Schema, and BNF Grammar.**

***Keywords*-SLA⋆; Service Level Agreement; SLA Template; Abstract Syntax;**

I. Introduction

This paper introduces SLA⋆, an abstract syntax for machine-readable Service Level Agreements (SLAs) and SLA templates - collectively SLA(T)s - developed as part of the FP7 ICT Integrated Project SLA@SOI[1].

The motivation for SLA⋆ is a project mandate for an SLA(T) model satisfying ostensibly conflicting requirements. On one side, the model must support domain-independent mechanisms for SLA negotiation, SLA conformance checking and quality-of-service (QoS) based service discovery: all problems which may be characterised as entailing multiple constraint satisfaction (MCS), and demanding a fine-grained specification of SLA(T) content. On the other hand, the model must be highly customisable: sufficient to meet the domain-specific requirements of very diverse SLA@SOI test-bed scenarios - including ERP hosting, Enterprise IT, live-media streaming and health-care provision.

To meet these requirements, SLA⋆ offers a domain-independent model of SLA(T) content grounded in an abstract constraint language, the concrete terms of which are formally specified by 'plug-in' domain-specific vocabularies. The constraint language ensures a consistent fine-grained structure for operations research (MCS), while the vocabularies provide for controlled extensibility.

SLA(T)s are inherently complex objects and the complete SLA⋆ specification reflects this complexity, such that a comprehensive account is not possible here. Rather than provide only a superficial overview, however, we will attempt instead to present the full picture 'in a nutshell', sacrificing lengthy explanation for curt, but substantive, content.

The paper is structured as follows. Section II first provides a brief review of related work. Sections III to VI present the condensed SLA⋆ syntax, describing, respectively, the *foundations* of the model, the *abstract constraint language*, *SLA(T) content* and the *domain-vocabularies*. Section VII then provides an example to illustrate the concrete application of SLA⋆, and Section VIII concludes with a brief summary of current status and future objectives.

II. Related Work

The most well-known work in the area of machine-readable SLA(T) models are the Open Grid Forum's *Web Services Agreement* (WS-Agreement) [1] and IBM's *Web Service Level Agreement* (WSLA) [2]. Other notable efforts include SLAng [3] and CC-Pi [4]. For various reasons, these approaches were deemed insufficient to meet SLA@SOI requirements. WS-Agreement, for example, gives a high-level account of SLA(T) content, but is missing a formal semantics and leaves the fine-grained content unspecified, while WSLA is overly constrained and lacks flexibility.

In broad terms, both WS-Agreement & WSLA, and indeed the W3C's *Web Service Description Language* (WSDL), suffer two basic restrictions. First, the SLA@SOI scenarios are not exclusively *web*-service scenarios. In particular, health-care provision entails human-based services. Second, all these standards are tightly-coupled to XML-Schema, whose standard semantics are unsuitable for the constraint-oriented reasoning & optimisation demands of operations research.

SLAng, instead, is specified in the OMG's *MetaObject Facility* (MOF) and thus has a degree of language-independence: with mappings to XML & *Human-Usable Textual Notation* (HUTN). SLAng also places greater emphasis on semantics, providing formal notions of SLA com-

[1]The research leading to these results is supported by the European Community's Seventh Framework Programme (FP7/2007-2013) and the SLA@SOI project (http://sla-at-soi.eu/) under grant agreement no.216556.

patibility, monitorability & constrained service behaviour. It is, however, targeted at electronic services and provides only a limited set of domain-specific QoS constraints.

CC-Pi is more generic still, offering a theoretical framework for mapping SLAs to service constraints. The CC-Pi model is, however, tightly-coupled to the mechanics of negotiation, and does not address common constructs such as agreement party details or service interfaces.

SLA*, in contrast, is a complete abstract SLA(T) syntax which has been designed to be independent of underlying technologies. It is decoupled both from particular notions of service, and from particular modes of expression, and can be extended to diverse scenarios without sacrificing formality or semantics. The following sections introduce SLA*.

III. SLA*: Foundations

In purely formal terms, we define an SLA(T) as a document consisting of hierarchically structured symbolic content. The objective for SLA* is to specify the structure, while ignoring the symbols.

The primary abstraction mechanism employed to achieve this is a universal set, $L^\star$, whose members are the legal expressions (symbol structures) comprising an SLA(T). The SLA* model is constructed as a taxonomy of nominative *expression-types* (subsets of $L^\star$) distinguished by their information carrying properties. Any symbols may be used to construct expressions, so long as their interpretation is consistent with the model.

At the highest level we draw a distinction between a set of *values*, $V^\star \subset L^\star$, and a set of *entities*, $E^\star \subset L^\star$. The set $V^\star$ constitutes an abstract constraint language, and loosely encapsulates the terminal expressions in the SLA(T) content hierarchy, while the set $E^\star$ constitutes higher-level structures. Domain-specific detail is incorporated primarily through *domain vocabularies*, which specify legal value tokens (i.e. the symbols comprising $V^\star$), and custom expression-types. The following sections, IV, V & VI, address, in turn; values, entities and domain vocabularies.

IV. SLA*: Abstract Constraint Language

The fundamental purpose of an SLA(T) is to specify contractual obligations, which, in effect, entails placing limits on the actions of agreement parties. In SLA* these limits are primarily expressed as *constraints*: which may be *atomic* or *compound* (arbitrarily complex logical combinations of constraints). Atomic constraints include such expressions as:

- 'X < 4',
- 'X + Y >= Z',
- '*foo*(y) != *goo*(z)',
- 'Z is-one-of{ a, b, c }', or
- '*completion-time*(*O*) < 4 ms'.

In general, an atomic constraint could be defined as a relation between two values (e.g. 'X < Y'). SLA* takes a slightly more convoluted approach, however, and defines an atomic constraint as a value (e.g. 'X') bound to lie in some 'domain' (e.g. '< Y'): which approach allows the domain part of the expression to be employed independently of constraints. For full flexibility we also allow both atomic and compound domains.

Table I
Value Expressions

Type ⊂ *Super-type*	*Definition*
Constant ⊂ V*	Abstract super-type of constants.
Atom ⊂ *Constant*	An opaque constant.
CompOp ⊂ Atom	Comparison operators e.g. '=', '<', '>'.
LogicOp ⊂ Atom	Logical operators e.g. 'not', 'and', 'or'.
Regex ⊂ *Constant*	A regular expression (see text).
⇑ ⊂ Atom	A reference to an expression-type.
↑ ⊂ Atom	A reference to an expression token.
Function ⊂ V*	An ordered pair <*f*,*P*>, where: *f* ∈ Atom is the function's name, and *P* ⊂ V* is an ordered set of parameters.
Event ⊂ V*	An ordered pair <*e*,*P*>, where: *e* ∈ Atom is the name of a class of events, and *P* ⊂ V* is an ordered set of parameters.
Constraint ⊂ V*	Abstract super-type of constraints.
AtomicConstraint ⊂ *Constraint*	An ordered pair <*v*,*d*>, where: *v* ∈ V* is a value constrained to lie in the domain *d* ∈ *Domain*.
CompoundConstraint ⊂ *Constraint*	An ordered pair <*o*,*C*>, where: *o* ∈ LogicOp, and *C* ⊂ *Constraint* is an unordered set of *sub*-constraints.
Domain ⊂ *V**	Abstract super-type of domains.
AtomicDomain ⊂ *Domain*	An ordered pair <*o*,*v*>, where: *o* ∈ CompOp, and *v* ∈ V* specifies a domain boundary.
CompoundDomain ⊂ *Domain*	An ordered pair <*o*,*D*>, where: *o* ∈ LogicOp, and *D* ⊂ *Domain* is an unordered set of of *sub*-domains.

In essence, constraints represent *states*. To complete the picture, we also consider *state transitions*, or *events*, such as: *periodic time-signals*, *the detection of a guarantee violation*, *the throwing of runtime exceptions*. Events in SLA* have a parametric form similar to functions: the expression '*periodic*[*1 hour*]', for example, can denote a periodic time-signal repeating at intervals of 1 hour, while '*violation-of-C*' can denote the violation of some constraint, *C*. To ground event semantics, the complete specification also includes formal state-machine models of SLA & service life-cycles (not presented here).

Collectively, *constraints*, and the various *constants* (e.g. 'X', 'a', '4'), *functions* (e.g. 'X+Y', '*foo*(*y*)') and *events* over which the constraints operate, constitute V* (formalised in Table I), which effects an abstract constraint language.

The complete specification also includes an extended taxonomy of *Atom*-types. For present purposes, we need only mention a few:

- *Boolean* : boolean constants (e.g. 'true', 'false').
- *DateTime* : specific points-in-time (e.g. '4.30 pm Wednesday July 7^{th} 2010').
- *Duration* : periods of time (e.g. '20 ms', '4 days').
- *TxRate* : transaction-rates (e.g. '2 tx_per_day').
- *Currency* : monetary values (e.g. '5 euro').

In order to validate & compare value expressions, we need a generic means to determine the type of a constant - e.g. we will need to know that '4 days' is a Duration token, but that '5 euro' is not. To achieve this, we permit domain-vocabularies to specify constant tokens by means of *regular expressions*, which are Atoms containing a readily distinguishable wildcard token, e.g. '? euro' (examples are provided in Section VII).

Finally, V* also includes two *reference* types. The first, $\Uparrow$, denotes an *unambiguous* reference to an expression *type*, and is primarily used in domain-vocabularies (Section VI). The second, $\uparrow$, denotes an *unambiguous* reference to an expression *token*. To subtype $\Uparrow$ & $\uparrow$, we adopt the notation $\Uparrow T$ & $\uparrow T$ (*resp.*) where *T* is any type name. SLA* permits *all* value tokens to be incorporated by reference: i.e. *anywhere* a token of type $V \subset$V* is permitted, a token of type $\uparrow V$ may be used instead (see Section V-D).

In SLA* all these value-types have only an abstract form. The concrete form is determined by particular choices of domain-vocabulary and representational language (e.g. XML, HUTN).

V. SLA*: SLA(T) Content Hierarchy

As stated earlier, the high-level SLA(T) structures are defined in terms of *entities*. An entity (formalised in Table II) is an identifiable collection of *key/value attribute* pairs, where each attribute predicates something of the entity. Attribute values can be (sets of expressions) of any type, including other entities. SLA* specifies the content of an SLA(T) as an hierarchical composition of specialised entity *sub-types*, which are characterised by their required attributes and exhibit *attribute inheritance*.

As will be seen in subsequent sections, several entity attributes take entity references ($\uparrow$E* tokens) as values. To facilitate such references, we introduce the notion of a *named* entity: an entity with an explicit 'name' attribute (*cf.* the use of the 'id' attribute in XML). These names *must* be unique for entities at the same level within any given branch of the SLA(T) content hierarchy, but they need not be unique within the SLA(T) as a whole.

The following subsections describe the entities which make up the SLA(T) content hierarchy, starting at the top with SLA templates and SLAs.

Table II
ENTITIES

Type $\subset$ *Super-type* *Key* $\subset$ *Value-type*†	*Type Description* $\diamond$ *attribute description*
E$^\star \subset$ L*	An entity in the SLA(T) content hierarchy, comprising an unordered set of ordered pairs <*k*,*v*>, where; *k* ϵ Atom is the name of an attribute (the 'key'), and *v* $\subset$ L*gives the attribute's value.
NamedEntity $\subset$ E* *name* $\subset$ Atom $_1$	A named entity. $\diamond$ the name of an entity instance.

†We employ the notation T_k to denote that an attribute's value is an unordered set of expressions, where *T* is the expression-type, and *k* is a cardinality constraint.

A. SLA templates and SLAs

Historically, the high-level structure of an SLA(T), as given by SLA*, has its roots in, and still maintains much in common with, WS-Agreement. Briefly, an SLA(T) (formalised in Table III) comprises descriptions of:

- agreement *parties*,
- offered *functional* (*service*) *capabilities*, and
- *agreement terms*, which define *QoS guarantees*.

SLA* differs from WS-Agreement, however, in its use of *variables*, in the way it models *creation constraints*, and in the explicit inclusion of domain vocabularies and party *actions* (the latter being generalised from WSLA). Formally, an SLA has the same structure as an SLA template, but with additional attributes giving the time at which the SLA was agreed and it's effective lifespan.

B. Agreement Parties

SLA* specifies only minimal information about agreement parties (Table IV), namely; an *identifier* and an *agreement role*. If required, additional party details can be added as domain-specific entity attributes (see Section VI).

Conceptually, a signatory party in an SLA may sign an agreement *on behalf of others*. A company executive, for example, can sign a contract for a catering service *on behalf of* the company's employees, who are the *end-consumers* proper of the service. In SLA*, the individuals represented by a party are referred to as 'operatives', and a single SLA(T) may offer different QoS guarantees to different categories of operative. We use the neutral term 'operative' (as opposed to say, 'consumer'), since operatives may be defined for *any* party, regardless of their role.

C. Interface Declarations

An interface declaration (Table V) defines an obligation on one of the SLA parties to provide certain *functional* (*messaging*) *capabilities*, which are encapsulated in SLA* in the form of *interfaces*.

Table III
SLAs and SLA Templates

Entity	*Description*
SLATemplate $\subset$ E*	An SLA Template.
version $\subset$ Atom $_1$	$\diamond$ SLA* version serial no.
vocabularies $\subset$ (↑)Vocabulary $_{1+}$	$\diamond$ see Section VI†.
parties $\subset$ Party $_{1+}$	$\diamond$ see Table IV.
interfaceDeclrs $\subset$ InterfaceDeclr $_{1+}$	$\diamond$ see Table V.
variables $\subset$ VariableDeclr $_{0+}$	$\diamond$ see Table VIII.
terms $\subset$ AgreementTerm $_{1+}$	$\diamond$ see Table IX.
SLA $\subset$ SLATemplate	An SLA.
agreedAt $\subset$ DateTime $_1$	$\diamond$ time at which the SLA was agreed.
effectiveFrom $\subset$ DateTime $_1$	$\diamond$ time at which the SLA comes into effect.
effectiveUntil $\subset$ DateTime $_1$	$\diamond$ time at which the SLA ceases to be effective.
template $\subset$ ↑SLATemplate $_1$	$\diamond$ reference to the template on which the SLA is based.

†The notation '(↑)Vocabulary' indicates the option to include vocabularies either explicitly, or by reference.

Table IV
Agreement Parties

Entity	*Description*
Party $\subset$ *NamedEntity*	A party to the agreement.
role $\subset$ Atom $_1$	$\diamond$ the role of the party in the SLA(T), e.g. 'provider' or 'customer'.
operatives $\subset$ Operative $_{0+}$	
Operative $\subset$ *NamedEntity*	Distinguishes a category of party 'operatives' (see text)

An important feature of SLA* is that interfaces can be declared for *any* party, irrespective of their role. A *customer*-side interface for receiving reports, for example, can be used to define *provider*-side reporting obligations (see also 'guaranteed actions' in Section V-E). Each interface declaration states *which* party must provide *which* interface, together with a list of *endpoints* for interface access.

An endpoint is defined by a *communications protocol* (e.g. SOAP, HTTP, email, voice-telephony, etc.) and a *location/address* (whose form naturally depends on the protocol, e.g. URI, postal address, telephone-number, etc.). Endpoint locations must be treated as optional in both SLAs and SLA templates, since it is not necessarily the case that they can be fixed in advance. They must be added to the SLA and communicated to interface consumers, however, prior to the SLA coming into effect (failure to do so constitutes a violation of the SLA).

The interfaces themselves are expressed according to the formalism shown in Table VI. Interface specifications may be included explicitly as part of the content of an SLA(T), or by reference to an externally located source.

Table V
Interface Declarations

Entity	*Description*
InterfaceDeclr $\subset$ *NamedEntity*	An interface declaration.
provider $\subset$ ↑Party $_1$	$\diamond$ a reference to the party obligated to provide the interface†.
endpoints $\subset$ Endpoint $_{1+}$	
interface $\subset$ (↑)Interface $_1$	$\diamond$ see Table VI.
Endpoint $\subset$ *NamedEntity*	An interface endpoint.
location $\subset$ Atom $_{0..1}$	$\diamond$ the destination address for invocation messages.
protocol $\subset$ Atom $_1$	$\diamond$ a communication protocol.

†In any given SLA(T), if there is only one party for a given role, *r*, then it is safe to assume that $r \in$ ↑Party.

The interface model employed in SLA* is essentially a generalisation of WSDL 2.0, abstracting both from web-service to *service* and from the use of XML as concrete syntax. Accordingly, an interface comprises a set of named *operations*, each of which, in turn, comprises a set of named and typed, *input/output* properties. Since SLA* adopts a generic notion of service, however, the term 'operation' should not be interpreted as implying any particular *method of invocation*: semantically, an operation specifies a *communicative interaction*, and 'invoking an operation' is synonymous with 'sending a message'.

In addition to input/output properties, SLA* also introduces the notion of a *related* property: one whose value may be determined at 'execution' time, but which is not explicitly defined as either input or output (examples include; the *time at which an operation is invoked*, or the *identity of the invoker*).

As per WSDL 2.0, interfaces may also obtain specialisation hierarchies ('extensions') with operation inheritance. They may also incorporate (or refer to) vocabularies, for the specification of any interface-specific faults/datatypes[2]. Any protocol-specific binding information, however, must be extracted and encoded as Endpoint expressions.

Finally, to obtain a maximum degree of flexibility in defining QoS guarantees, SLA* also provides a mechanism for specifying *arbitrary* collections of interface operations (e.g. operations taken from different interfaces, or invoked at specific endpoints). This mechanism is just a dedicated value-type, S*, which is formalised in Table VII.

D. Variable Declarations

In basic form, variable declarations in SLA* are a convenience mechanism allowing any value expression to be denoted by a shorthand label - or *variable* - which may then

[2]The XML-based realisation of SLA* supports translation from WSDL 2.0 to interface entities, and from XML-Schema to vocabulary entities.

Table VI
FUNCTIONAL INTERFACES

Entity	*Description*
Interface $\subset$ E* *vocabularies* $\subset$ (↑)Vocabulary $_{0+}$ *extended* $\subset$ ↑Interface $_{0+}$ *operations* $\subset$ Operation $_{0+}$	An interface specification. ◇ see Section VI. ◇ list of super-interfaces. See note†
Operation $\subset$ *NamedEntity* *faults* $\subset$ ⇑ $_{0+}$ *inputs* $\subset$ Property $_{0+}$ *outputs* $\subset$ Property $_{0+}$ *related* $\subset$ Property $_{0+}$	An invocable interface operation. ◇ potential run-time faults. ◇ input parameters. ◇ result properties. ◇ related properties (see text).
Property $\subset$ *NamedEntity* *datatype* $\subset$ ⇑ $_{1}$ *domain* $\subset$ *Domain* $_{0..1}$ *auxiliary* $\subset$ Boolean $_{1}$	An interface operation property. ◇ the property's datatype. ◇ an *optional* domain restriction on property values. ◇ auxiliary properties can have 'null' values.

†It is a requirement in SLA@SOI that operations need not be *explicitly* declared, in which case the interface is effectively defined just by its endpoint location & protocol (see Table V).

Table VII
ARBITRARY SETS OF OPERATIONS

Type $\subset$ *Super-type*	*Description*
S$^\star \subset$ V*	Comprises a triple <*I*,*O*,*E*>, where: *I*,*O* & *E* are unordered sets of ↑InterfaceDeclr, ↑Operation or ↑Endpoint (*resp.*), which represents any operations referenced in *O*, or defined by the interfaces in *I*, that are accessible through the endpoints in *E*.

be used in place of the original expression. By definition, therefore, variable names are ↑V* tokens (i.e. references to V* expressions).

As an extension of this, however, SLA* also defines a specialised *customisable* variable, which provides a means to express 'options' (*cf.* 'creation constraints' in WS-Agreement). In this latter case, the variable declaration is augmented by a *Domain* expression specifying a set of alternatives (e.g. '>4 and <10', 'one-of{ a, b, c }', etc.), and the expression assigned to the variable must be a particular value taken from this domain. In SLA templates, this value is interpreted as the 'default' option, while in SLAs it is interpreted as a 'chosen' option. These expressions are formalised in Table VIII (see Section VII for examples).

E. Agreement Terms

The obligations that parties commit to on signing an SLA are encoded in two forms. Interface declarations (above) are one form: defining obligations on parties to provide *functional* capabilities. Additional obligations, in particular QoS,

Table VIII
VARIABLE DECLARATIONS

Entity	*Description*
VariableDeclr $\subset$ *NamedEntity* [*name* $\subset$ ↑V* $_{1}$] *expr* $\subset$ V* $_{1}$	A variable declaration. ◇ see text. ◇ the expression 'assigned' to the variable.
Customisable $\subset$ VariableDeclr [*expr* $\subset$ Atom $_{1}$] *domain* $\subset$ *Domain* $_{1}$	A 'customisable' variable (see text). ◇ the assigned expression must be an Atom. ◇ specifies a 'set of alternatives'.

Table IX
AGREEMENT TERMS

Entity	*Description*
AgreementTerm $\subset$ *NamedEntity* *variables* $\subset$ VariableDeclr $_{0+}$ *pre* $\subset$ Constraint $_{0..1}$ *guarantees* $\subset$ Guarantee $_{1+}$	An agreement term. ◇ locally scoped variables. ◇ the conditions under which the agreement term applies. ◇ one or more guaranteed *states* and/or *actions*.
Guarantee $\subset$ *NamedEntity*	Abstract super-type of guarantees.
State $\subset$ *Guarantee* *priority* $\subset$ Atom $_{0..1}$ *pre* $\subset$ *Constraint* $_{0..1}$ *post* $\subset$ *Constraint* $_{1}$	A guaranteed state. ◇ a domain-specific priority value. ◇ the conditions under which the guarantee holds. ◇ the state which is guaranteed.
Action $\subset$ *Guarantee* *actor* $\subset$ ↑Party $_{1}$ *policy* $\subset$ Atom $_{1}$ *pre* $\subset$ Event $_{1}$ *limit* $\subset$ Duration $_{1}$ *post* $\subset$ *ActionDefn* $_{1}$	A guaranteed action. ◇ the party/operative obliged to perform the action†. ◇ e.g. 'mandatory' or 'forbidden'. ◇ the (class of) events which 'trigger' the action. ◇ a time-window for performing the action.
ActionDefn $\subset$ E*	Abstract super-type of action *post*conditions (see Table X).

†See note for Table V

or *non-functional*, guarantees, are specified as *agreement terms*, which are formalised in Table IX.

In SLA*, a single SLA(T) may contain several agreement terms with varying *pre*conditions: where *pre*conditions are expressed as *Constraints*, and serve to define the circumstances under which the term comes into force (*cf.* 'qualifying conditions' in WS-Agreement). Each agreement term can contain several *guarantees*, of which there are two kinds: guaranteed *states* and guaranteed *actions*.

A guaranteed *state* is some 'state of affairs' (expressed as a *Constraint*) which a party is obliged to maintain. A 'service

level objective', such as '*completion-time*(*O*)<4 ms', is a typical example. If required, guaranteed states can also have localised *pre*conditions.

A guaranteed *action*, instead, is an obligation on a party to perform (or refrain from performing) some specific action under specific conditions. Each action is described by:

- a reference to the actor,
- the (class of) events which 'trigger' (or *pre*condition) the action,
- a time-window specifying a deadline for performing the action (i.e. the maximum amount of time which is allowed to pass following the action's trigger event),
- a policy (e.g. 'mandatory' or 'forbidden'), and
- a description of the action itself, which is referred to as the action's *post*condition.

The complete SLA* specification defines several kinds of action *post*condition, the most significant of which are the following:

- *invocation* : denoting the action of invoking a specific interface operation (e.g. posting a violation report),
- *payment* : denoting a transfer of economic goods from one party/operative to another[3],
- *termination* : denoting the preemptive cancellation of the SLA,
- *renegotiation* : denoting the renegotiation of the terms of the SLA, and
- *workflow* : for constructing composite actions.

By nature, action postconditions have idiosyncratic form, which for present purposes we illustrate with just a couple of examples: *invocations* & *payments* (Table X). The other *post*conditions (in particular, workflows) have more complex formalisms, an account of which is beyond the present scope.

VI. SLA*: Domain Vocabularies

To accommodate domain-specific requirements, SLA* employs *domain vocabularies* (Table XI), which are formal specifications (*alt.* 'schemas' or 'ontologies') defining the *terms* which are valid for some application domain. As with interface specifications (above), vocabularies may be included explicitly as part of the content of an SLA(T), or by reference to an external source.

As noted earlier, V* includes a reference-type, $\Uparrow$, for referencing expression-types. For present purposes, we define $\Uparrow$ as the set of expression-type names (e.g. 'Atom', 'Constraint') and qualified entity attribute keys (e.g. 'Party.role', 'Endpoint.protocol').

Vocabulary terms come in various flavours, of which there are three basic kinds:

- *Nominal* : a standardised name,
- *Parametric* : a standardised function or event *signature*,

[3]Payments could, in principle, be treated just as information transfer (*cf.* invocations), but we have opted instead to maintain economic transfer as a distinct notion.

Table X
Action Postconditions

Entity	*Description*
Invocation $\subset$ *ActionDefn*	The invocation of a specific interface operation.
operation $\subset \uparrow$Operation $_1$	$\diamond$ a reference to the operation to be invoked.
parameters $\subset$ PARAM $_{0+}$	$\diamond$ specifies required input values. Each PARAM is a pair $<p,v>$, where; $p \in \uparrow$Property references an *input* property, and $v \in$ V* specifies its value.
Payment $\subset$ *ActionDefn*	Transfer of economic goods.
recipient $\subset \uparrow$Party $_1$	$\diamond$ the recipient party/operative†.
value $\subset$ V* $_1$	$\diamond$ an expression which quantifies the payment.

†See note for Table V

Table XI
Domain Vocabularies

Entity	*Description*
Vocabulary $\subset$ E*	A domain-vocabulary.
terms $\subset$ Term $_{1+}$	
Term $\subset$ E*	Abstract super-type of 'terms'.
name $\subset$ *Constant* $_{0..1}$	$\diamond$ an Atom or Regex.
defn $\subset$ Atom $_1$	$\diamond$ semantic definition of the term.
Nominal $\subset$ *Term*	A standardised name.
roles $\subset \Uparrow$*Constant* $_{1+}$	$\diamond$ defines the 'roles' of the term in an SLA(T) (see text).
Parametric $\subset$ Nominal	A *function* or *event* 'signature'†.
parameters $\subset \Uparrow$V* $_{0+}$	$\diamond$ an ordered list of parameter types.
DataType $\subset$ *Term*	A constant-type (datatype) definition†.
super-type $\subset \Uparrow$*Constant* $_1$	$\diamond$ the datatype's super-type.
ArrayType $\subset$ *Term*	Array type-definitions†.
element-type $\subset \Uparrow$V* $_1$	$\diamond$ the element type.
EntityType $\subset$ Term	An entity definition†.
super-type $\subset \Uparrow$E* $_1$	$\diamond$ the entity's super-type.
attributes $\subset$ AttributeType $_{0+}$	$\diamond$ attribute definitions.
AttributeType $\subset$ *NamedEntity*	An entity attribute definition.
defn $\subset$ Atom $_1$	$\diamond$ semantic definition of the attribute.
value-type $\subset \Uparrow$ $_1$	$\diamond$ the attribute value's type.
cardinality $\subset$ Domain $_1$	$\diamond$ the attribute's cardinality.

†For all terms except Nominal: *name* $\subset$ Atom $_1$ (i.e. Regex names are not permitted).

- $\sim$*Type* : used to specify extensions to the SLA* expression-type hierarchy.

A Nominal associates a name token with one or more 'roles', enumerating the expression-types for which the

name token is a valid instance. For example (using the notation: *name* {*roles*}):

- 'false' {Boolean} : declares the token 'false' to be a valid instance of Boolean.
- 'http' {Endpoint.protocol} : declares the token 'http' to be a valid *endpoint protocol*.
- '? euro' {Currency} : declares the regular expression token '? euro' as a *pattern* for valid Currency constants ('?' is a wildcard denoting any numeric value).

Parametrics, which define formal function/event *signatures*, are Nominals augmented with an ordered list of parameter-types. A Parametric with *name*='is_morning', *parameters*={DateTime} and *roles*={Boolean}, for example, defines the function *is_morning*:DateTime→Boolean, which function can then be used anywhere that a Boolean value is permitted. Events are distinguished just as those parametrics whose roles include the expression-type Event (or any of it's sub-types).

Type terms, instead, provide a means to introduce new expression-types (subsets of L⋆), and come in three kinds:

- *DataType* : used to define domain-specific datatypes: e.g. a DataType entry with *name*='Weight' & *supertype*=*Atom* would define a category of weight values.
- *ArrayType* : specifies an *array*-type value expression: e.g. an ArrayType entry with *name*='array_of_Weight' & *element-type*=Weight would define an array of weight values.
- *EntityType* : specifies a domain-specific entity sub-type.

In addition to the formal details, every vocabulary term must also provide a definition of the term's semantics. We do not prescribe a formalism for describing semantics, but the definition must be sufficient to inform both application developers and prospective SLA parties of the term's intended significance.

To accompany the SLA⋆ specification, we have also developed an extensive *core* vocabulary covering the most common:

- *nominals* - e.g. standard comparison/logical-operators, agreement roles, endpoint protocols, datatypes and metric units.
- *functions* - e.g. standard arithmetic/set-operations, time-series and QoS metrics (including: *completion-time*, *availability*, *throughput*, *mean-time-to-failure*, and others).
- *classes of event* - e.g. violations, warnings, periodic time-signals and runtime faults.

The next section illustrates the use of these core vocabulary terms in an SLA template.

VII. Example

To illustrate the concrete application of SLA⋆, this section presents an 'hello world' example of an SLA template: describing a simple email-based service offered by a provider 'Fred'. For brevity, we assume an external interface specification available at the URL 'http://xyz.com/service', of which all we need know is that it defines an operation called 'request'. Briefly, the SLA template offers customers a choice of either *basic* or *premium* 'service levels' - each offering different guarantees on the completion-time of 'requests' - and enforces a penalty in case of violations.

The example does not include an explicit vocabulary specification, but relies on the external core SLA⋆ vocabulary (http://sla-at-soi.eu/core), and in particular on the following core terms:

- *nominals*:
  - 'provider', 'customer' : party roles.
  - 'email' : an endpoint protocol.
  - 'mandatory' : an action policy.
  - 'one-of', '==', '<', '<=' : comparison-operators.
  - 'or' : logical-operator.
  - '? tx_per_day' : a transaction rate constant.
  - '? min', '? hr', '? week' : duration constant.
  - '? euro' : a currency constant.
- *parametrics*:
  - *arrival-rate*:S⋆→TxRate : giving the rate of invocations of an operation.
  - *completion-time*:S⋆→Duration : giving the time taken to complete the execution of an operation.
  - *violated*:*Constraint*→Event : denoting the detection of a violation of a constraint.

At the time of writing, the abstract SLA⋆ syntax has been realised in concrete form as a Java API, an XML Schema and a BNF Grammar. Here we use (a simplified form of) the BNF serialisation, augmented with line numbers for ease of reference. The template is as follows, with explanations below:

```
01: slatemplate{
02:   version : sla-star-v1
03:   vocabularies :
04:     http://sla-at-soi.eu/core
05:   parties :
06:     Fred : party{
07:       role : provider
08:     }
09:   interfaceDeclrs :
10:     IF1 : interfaceDeclr{
11:       provider : Fred
12:       endpoints :
13:         E1 : endpoint{
14:           location : fred@xyz.com
15:           protocol : email
16:         }
17:       interface : http://xyz.com/service
18:     }
19:   variables :
20:     S : var{
21:       expr : IF1/request
22:     },
23:     X : var{
24:       expr : basic
25:       domain : one-of { premium, basic }
26:     }
```

```
27:   terms :
28:     AT1 : agreementTerm{
29:       pre : arrival-rate(S) <= 2 tx_per_day
30:       G1 : state{
31:         pre : X == basic
32:         post : completion-time(S) < 1 hr
33:       },
34:       G2 : state{
35:         pre : X == premium
36:         post : completion-time(S) < 10 min
37:       },
38:       G3 : action{
39:         actor : Fred
40:         policy : mandatory
41:         pre : violated[ G1.post and G2.post ]
42:         limit : 1 week
43:         post : payment{
44:           recipient : customer
45:           value : 1 euro
46:         }
47:       }
48:     }
49: }
```

The opening lines (1..4) state that the example is an SLA template, give the SLA⋆ version serial number, and declare the use of the core SLA⋆ vocabulary. Lines 6..8 introduce Fred as the service provider, while the interface declaration, IF1 (lines 10..17), obligates Fred to provide the email-based 'request' service at the (endpoint) address 'fred@xyz.com'.

For convenience, lines 20..22 declare the variable, 'S', as stand-in for the expression 'IF1/request': an ↑Operation token pointing to the operation 'request' in the interface declared by IF1. Lines 23..26, instead, declare a *customisable* variable, 'X', giving the customer options 'premium' or 'basic', with 'basic' as the default.

Lines 28..48 encapsulate all the template's guarantees as a single agreement term, AT1, with the *pre*condition (line 29) that the term is only in force when the customer does not exceed two invocations per day.

Lines 30..37 define two guaranteed states, G1 and G2. The first of these applies when the customer selects 'basic' as the value of variable 'X' (line 31), and restricts the 'completion-time' of the request operation to the domain '$<$ 1 hr' (line 32). The second applies when the customer selects 'premium' (line 35), and constrains the 'completion-time' to the domain '$<$ 10 min' (line 36).

Finally, lines 38..47, describe a guaranteed action, G3. The party obligated to perform the action is Fred (line 39), and the action is 'mandatory' (line 40): triggered whenever there is a violation of one of the *post*conditions of G1 or G2 (line 41: 'G1.post' & 'G2.post' are both ↑*Constraint* tokens). Line 42 states that Fred has 1 week in which to complete the action, which is defined in lines 43..46 as a payment to the customer of 1 euro.

This example highlights the expressivity of SLA⋆: it is possible to say a lot with relatively few statements. The engine driving this expressivity is the abstract constraint language (V⋆), and its arbitrarily detailed realisation by domain-vocabularies.

VIII. Conclusion

The SLA⋆ model is a stable result of the SLA@SOI project, which meets the project objectives and has been tested in practical application. The model offers a language-independent specification of SLA(T) content at a fine-grained level of detail, which is both highly expressive and inherently extensible. The model is not without problems, however, and we continue to seek improvements.

At the domain-independent level we are currently investigating two key issues. The first is how best to handle dependencies between different SLA(T)s: as may be the case, for example, in scenarios involving third party monitors. The second is to explicate a robust semantics for action postconditions in respect of their relations to constraint assertion languages (for automatic verification & monitoring), and to programming languages (for automated execution).

At the domain-specific level, we continue to improve and refine the core SLA⋆ vocabulary. In particular, we are currently investigating additional terms supporting formalised negotiation protocols and certificate-based mechanisms for 'signing' SLAs.

Finally, in terms of practical application, the main focus of future work lies in extending SLA⋆'s concrete realisation: both in the development of robust mappings from SLA⋆ to other SLA(T) formalisms (e.g. WS-Agreement), and in the provision of support tools (e.g. for editing & validation) and guides for SLA(T) developers.

Acknowledgment

SLA⋆ is the result of an iterative process of knowledge acquisition & modelling. We would like to thank *all* partners in the SLA@SOI project for the very considerable time and effort they dedicated to this process.

References

[1] A. Andrieux, K. Czajkowski, A. Dan, *et al*, *Web Services Agreement Specification (WS-Agreement)*, March 14 2007, available at: http://www.ogf.org/documents/GFD.107.pdf

[2] H. Ludwig, A. Keller, A. Dan, *et al*, *Web Service Level Agreement (WSLA) Language Specification*, January 28 2003, available at: http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf

[3] D.D. Lamanna, J. Skene & W. Emmerich, *SLAng: A language for defining service level agreements*, in Proc. of the 9th IEEE Workshop on Future Trends in Distributed Computing Systems-FTDCS, 2003, pages 100-106

[4] M. Buscemi & U. Montanari, *CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements*, in Programming Languages and Systems, 2007, pages 18-32.