Validation of Service Level Agreements using Probabilistic Model Checking

Maria Krotsiani, Christos Kloukinas and George Spanoudakis

Department of Computer Science
City, University of London
London EC1V 0HB, UK

{Maria.Krotsiani, C.Kloukinas & G.E.Spanoudakis}@ city.ac.uk

***Abstract* - With the fast growth of Information Technology (IT), organisations rely mostly on web services, cloud services and recently on Big Data Analytics services (BDA services), in order to support their business services. To securely use these services, service clients sign a Service Level Agreement (SLA) with service providers, regarding a particular service provision. Typically, SLAs define the properties that need to be preserved during the provision of a service (e.g., quality of service properties) and actions that will be applied if the service provision violates the defined properties (e.g., penalties or re-negotiation). Whilst significant research has focused on monitoring SLAs during the provision of services, the exploration and validation of the potential consequences of SLAs for the involved parties prior to putting them in operation is not addressed by existing research. In this paper, we present an approach to SLA validation that is based model checking. Our approach is based on the translation of SLAs expressed in WS-Agreement into models of the probabilistic model checker PRISM and the validation of SLA properties using the model checking capabilities of this tool.**

Index Terms – Service Level Agreement (SLA), Probabilistic Model Checking, Static Validation

I. INTRODUCTION

Service Level Agreements (SLAs) are widely used by service providers (consumers) as means of providing (getting) assurance regarding the quality and security of the provisioned services. SLAs constitute a kind of contract between service providers and service consumers that defines the level of service that should be guaranteed in service provision and the actions that will be undertaken if this level is not preserved.

The management of SLAs (i.e., the specification, monitoring and negotiation of SLAs) has been the subject extensive research. This work has generated languages for specifying SLAs (e.g., WSLA [6], WSLA+ [25], SLA* [13]), techniques for SLA negotiation (e.g., WS-Agreement Negotiation [23], PROSDIN [12]), and techniques for monitoring SLAs (e.g., Nagios [22], Ganglia [19], EVEREST [11][4]).

However, the work done for SLA specification does not support adequately the analysis of the consequences of the specifications that can be created for the different parties, so that they all know what they are committing to. For example, given an SLA, it would be useful for service consumers and providers, to be able to analyse and answer questions such as:

(i) (Service providers) How probable is it to pay more than $X in a time period Y?
(ii) (Service consumers and providers) How long will it take before the need to renegotiate an SLA?
(iii) (Service consumers and providers) What is the probability for renegotiation by the expiry date?
(iv) (Service providers) How long will it take before the need to modify the infrastructure of service provision in order to continue satisfying the SLA?

In this paper, we present an approach that enables the investigation and validation of properties as the above for SLAs. Our approach is based on formalising the semantics of the actions taken upon the violation of SLA guarantee terms and their conditions, so as to enable the formal analysis of the consequences of specific SLAs under different environment assumptions, i.e., different probabilities for the violation of specific guarantee terms. To achieve this, we have extended the WS Agreement language [10][21]. Our extension allows SLA parties to provide more details about the actions that should be taken upon the violation of specific guarantee terms, including the conditions that need to be satisfied for these actions to be taken, and extra parameters that complete the action specifications, e.g., the amount one has to pay in case of a penalty.

The SLAs that are specified in the extended form of WS Agreement that we propose are translated into probabilistic automata expressed in the language of PRISM [14][24], i.e., a probabilistic model checker. These automata are then used in order to investigate and validate properties, like (i)-(v) above. The translation of SLAs into probabilistic automata of the type supported by PRISM is based on a general scheme in which violations of SLA guarantee terms are translated into events that trigger transitions between automata states, and the actions associated with such violations are modelled by actions undertaken when the relevant transitions are triggered. To realise our approach, we have developed a translator from the extended version of WS Agreement to the language of PRISM. We have also performed various experiments demonstrating the feasibility of our approach.

To the best of our knowledge, our approach is the first one that focuses on the exploration and validation of SLA properties (consequences) prior to putting them in operation.

The rest of the paper is organized as follows. Section II presents the overall SLA management framework assumed by

our approach and shows how SLA validation fits within and may be used by it. Section III presents the extensions of WS-Agreement language, which we have introduced to enable the validation of SLA properties. Section IV discusses the translation from the extended version of WS Agreement to PRISM and gives examples of PRISM models generated from SLAs. Section V presents the outcomes of experiments that we have conducted to demonstrate the benefits and feasibility of our approach. Section VI reviews related work and, finally, Section VII provides concluding remarks and outlines directions for further research.

II. General SLA Management Framework

To better understand the purpose of this paper, it is important to consider the overall SLA management framework that is the context of our work. As shown in Fig. 1, this framework consists of tools (components) that support key traditional activities of SLA management, namely SLA specification, negotiation and runtime monitoring, and the new activity of SLA validation that we introduce in this paper.

SLA negotiation and specification are the activities that give rise to an SLA and are supported by SLA negotiation and specification tools. Once it is agreed and specified, an SLA needs to be monitored and enforced for the system that it is concerned with. This requires the translation of the SLA to an operational monitoring specification that will be possible to check given the event sensing and monitoring capabilities of the computational infrastructure where the service based system that is the subject of the SLA is deployed. Foster and Spanoudakis [5] have introduced a general-purpose monitoring architecture and algorithms supporting this process. This architecture includes: (a) event sensors having responsibility for capturing the primitive information needed for SLA monitoring, (b) monitors performing the actual checks of SLA guarantee terms against event streams produced by the sensors, and (c) an SLA2Monitor translator that can translate the SLA expressed in a high-level SLA language into the operational monitoring specification that can be checked by the EVEREST [4]. When the monitor identifies violations it informs an SLA Manager of the SLA terms that have been violated, to enable it to perform the actions specified in the SLA for the respective violations [5]. This automates guaranteeing an SLA at run-time but since SLAs and system behaviours can be complex, interacting parties can easily find themselves in situations they had not anticipated, e.g., having to pay a high penalty due to violated SLA terms. For this reason, we have extended the SLA management framework with tools supporting an SLA validation activity (see orange dashed rectangle in the left part of Fig. 1).

The validation activity is realised through the translation of the SLA into the language of Prism [14][24]. More specifically, it produces a formal model of the actions that the SLA Manager will be taking each time a term violation of the specific SLA is reported. In order to be able to explore the consequences of these actions it also produces a formal model of the environment of the SLA Manager, which is shown inside the rectangle with the red dashed boundary in Fig. 1, i.e.,

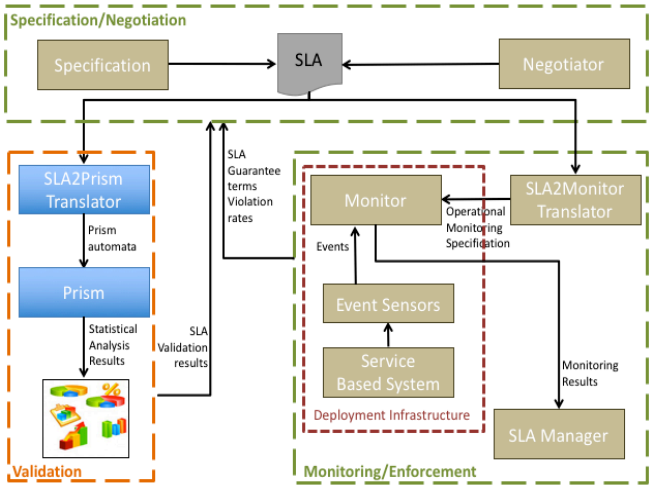

Fig. 1 General SLA Management Framework

the combination of the system itself and the monitor checking the SLA rules at run-time. Since that part is too complex to describe formally in detail, it is abstracted away and represented as a source of SLA term violations that occur stochastically and can be described with different exponential distributions that have some rate of arrival.

Using Prism, SLA parties can check the probability of and time to execution of actions that are associated with the different SLA guarantee terms. The outcomes of validation are passed back to the negotiation and specification activities in order to inform the tailoring of SLAs accordingly (e.g., to adjusting the terms in order to reduce the possibility of paying penalties; or to redefine more convenient terms for re-negotiations; or even to estimate possible configurations that might be required in order to avoid any defined modifications that can occur. Validation can be carried out before an SLA is agreed and instantiated, to explore its consequences under different operational scenarios. In such cases, validation is based on estimates of the expected violation rates of SLA guarantee terms. In addition, validation can be performed once an SLA has become operational to inform the negotiation processes that may need to be carried out due to violations of SLAs or changes of the needs of the relevant parties. In such cases, the validation can be based on real violation rates that have been detected by the monitor and passed back to the specification and negotiation activity.

III. WS-Agreement Language & Extensions

WS-Agreement [10] was introduced by the Open Grid Forum to address some key requirements for the specification of SLAs, such as supporting modularity, accommodating other external and domain specific standards, and allowing extensions.

In WS-Agreement, an SLA is composed of three main sections, i.e., the name, context and terms of as shown in Fig. 2. The first section provides an optional SLA name. The next section contains the SLA context, i.e., meta-data for the entire SLA (e.g., the SLA participants, its lifetime, etc.). The terms section specifies the terms of the SLA. these can be of two types: (a) Service Terms that describe the services regulated

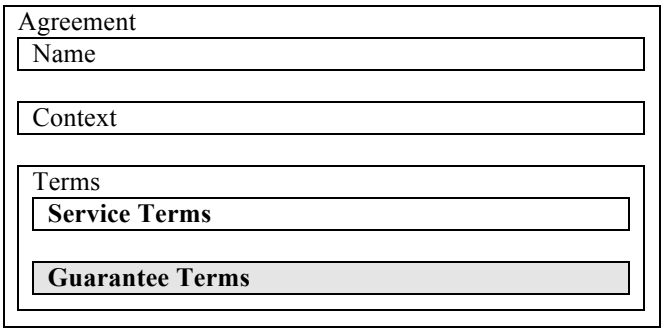

Fig. 2 Structure of the WS-Agreement SLA[10]

by the SLA and (b) the Guarantee Terms (GTs) that specify the service levels that should be satisfied during the provision of the service. More specifically, GTs define the expected quality of service, comprising (i) the obligated party, (ii) the list of services this guarantee applies to, (iii) an optional condition that must be met for a guarantee to be enforced, (iv) an assertion expressed over service descriptions (*ServiceLevelObjective*), and (v) one or more business values associated with this objective (*BusinessValueList*).

WS-Agreement has some limitations, since it does not support the specification of: (a) security and privacy Service Level Objectives (SLOs), (b) actions that need to be taken during the life cycle of an SLA (e.g., service provision platform modification actions and SLA negotiation actions), (c) multi-party SLAs, and (d) comprehensive models of complex services, including internal operations, service assets, and operation, data and other dependencies. This makes it harder to check whether an SLA can be monitored and to produce automatically a monitoring plan for it.

In our approach [27] we extended the *CustomServiceLevel* sub-element of the *ServiceLevelObjective* element, to support monitorable SLOs. To do so, we introduced *PreciseSLOType*, a new type for *CustomService-Level*. This allows specifying an SLO at a declarative level, i.e., as a property of a particular category that is to be applied to a service asset (e.g., internal or external operation and data elements of the service), and/or a procedural level, which provides the exact runtime assertion representing the SLO satisfaction. SLO assertions are expressed in the language developed in the CUMULUS project [2] – itself based on Event Calculus, a formal first order temporal logic language, also used by the EVEREST Framework [4].

WS-Agreement does not support the specification of actions what should be undertaken when guarantee terms are violated. To overcome this limitation, we extended the *CustomBusinessValue* sub-element of the *BusinessValueList* element, for expressing different types of actions that should be triggered by each GT violation [27]. Two actions have been predefined: (a) renegotiate *Pred*, which causes the SLA to be renegotiated when the guard *Pred* is satisfied, and (b) penalty *Pred Int*, which causes a penalty (or reward if negative) to be incurred. This makes more precise the specification of penalties in WS-Agreement. SLA modellers also free to use any other action name they wish. They can also guard all actions with a predicate. A guard predicate declare conditions that should be satisfied, in addition to the violation of the SLA guarantee term associated with the action, for the action to be executed.

The BNF grammar of the extended form of WS-Agreement that we have created is shown in Fig. 3. The grammar syntax is based on the syntax used by the K Framework [3]. More specifically, we use italics for non-terminals, bold for terminals, and *List{ X, "c" }* for a c-separated list of zero or more Xs. Notes inside square brackets ([]) on the right have to do with evaluation– *strict* means that all non-terminals must be evaluated first (in a non-deterministic order), *seqstrict* means that they must be evaluated left-to-right, and *strict(1)* means that only the first non-terminal needs to be evaluated. Both *Pred* and *NumExp* permit the ternary if-then-else operator *pred* ? *exp1* : *exp2*, which evaluates to *exp1* when *pred* is true, and to *exp2* otherwise. Apart from constants, one can use predefined functions. These are: **violations(***GT***)**, **penalty_ amount(***GT***)**, and **counter(***Action*, *GT***)**. These functions provide the number of times some *GT* has been violated so far, the amount accrued in penalties due to violations of some *GT*, and the number of times *Action* has been executed following a *GT* triggering.

IV. Running Example

To demonstrate the overall approach of this paper, we present an SLA with several GTs from the side of a service provider, who has different assets (A1–A4) to be assured, as shown in Table I, and their security properties (SLOs).

The first ones (A1, A2) are service data assets and the last two (A3, A4) operation assets. Data assets have GTs for Confidentiality (A2) and Integrity (A1). For example, in asset A2 the expected rate of a Confidentiality GT violation is λ=0.15 and if there is more than one violation (noted with [v>1]), there is a penalty of 10 units and a notification action.

For the Integrity security GT, we assume the same violation rate (λ=0.6) and the same actions, which cause a notification action for the first two violations of the term, but renegotiate the SLA if there are more violations.

Finally, operation asset A3 has an Availability GTs with a violation rate λ=0.2 and three actions. While there are fewer than *k* violations the service provider configuration should be modified, which will decrease the rate to λ=0.1. The second action requests a renegotiation if there are more than *2k* violations, and the third requests a notification always. The rates are shown in italics, as these do not form part of the SLA agreement – they are expectations (possibly commercially sensitive) of the estimated behaviour of the system.

Using the grammar of Fig. 3, the SLA for Table I would be specified as:

```
{ Running_Example ... { ... {
       { ConfA2 ... ...
         penalty (violations(ConfA2) > 1) 10
         notify  (violations(ConfA2) > 1)       }
       { IntA1 ... ...
         notify (violations(IntA1) < 3)
         renegotiate  (violations(IntA1) >= 3)  }
       { AvailA3 ... ...
         modify (violations(AvailA3) < k)
         renegotiate  (violations(AvailA3) > 3*k)
```

XWSAgreement ::= { *IdOpt* **...** *Terms* **...** }
Terms ::= { ... { *GuaranteeTerms* } }
GuaranteeTerms ::= List{*GuaranteeTerm*, “”}
GuaranteeTerm ::= { *Id* **...** *BusinessValueListType* }
BusinessValueListType ::= **...** *CustomBusinessValue*
CustomBusinessValue ::= List{*XAction*, “”}
XAction ::= **renegotiate** *Pred* | **penalty** *Pred Int* | *Id Pred*
Pred ::= **true** | **false** | **(** *Pred* **)** [bracket]
| *Pred* **&** *Pred* [strict(1)]
| *Pred* | *Pred* [strict(1)]
| **!** *Pred* [strict] | *Pred* **?** *Pred* **:** *Pred* [strict(1)]
| *NumExp* **=** *NumExp* [seqstrict] | *NumExp* **!=** *NumExp* [seqstrict]
| *NumExp* **>** *NumExp* [seqstrict] | *NumExp* **<=** *NumExp* [seqstrict]
| *NumExp* **<** *NumExp* [seqstrict] | *NumExp* **>=** *NumExp* [seqstrict]
NumExp ::= *Int* | **(** *NumExp* **)** [bracket]
| *NumExp* ***** *NumExp* [seqstrict, klabel(Mul)]
| *NumExp* **/** *NumExp* [seqstrict, klabel(Div)]
| *NumExp* **+** *NumExp* [seqstrict, klabel(Plus)]
| *NumExp* **-** *NumExp* [seqstrict, klabel(Minus)]
| *Pred* **?** *NumExp* **:** *NumExp* [strict(1)]
| **violations (** *Id* **)** | **penalty_amount (** *Id* **)** | **counter (** *Id* **,** *Id* **)**

Fig. 3 Extended WS-Agreement BNF grammar *(abstracted)*

TABLE I
RUNNING EXAMPLE ASSETS AND SLOS

Assets		Security Properties / GTs		
		Confidentiality	**Integrity**	**Availability**
Data Assets	A1	-	[v<3] NOTIFY [v>=3] RENEG $\lambda = 0.6$	-
	A2	[v>1] PENALTY(10) & NOTIFY $\lambda = 0.15$	-	-
Operation Assets	A3	-	-	$\lambda = 0.2$ [v<k] MOD➔λ=0.1 [v>3k] RENEG [true] NOTIFY
	A4	-	[v<3] NOTIFY [v>=3] RENEG $\lambda = 0.6$	-

```
      notify  true                                  }
    { IntA4 ... ...
      notify (violations(IntA4) < 3)
      renegotiate  (violations(IntA4) >= 3)    }
} } ... }
```

V. FORMAL SLA VALIDATION

As aforementioned, each GT violation action is guarded by a condition as is shown in the definition of *XAction* in Fig. 3 – expressed over a set of basic, implicitly updated variables: violations(GT), penalty_amount(GT), and counter(Act, GT).. Using the action part of the SLA specification we build a formal model of the system, expressed in the language of the Prism model-checker, which allows the analysis of temporal probabilistic models with its support for Markov chains, and Probabilistic-Timed Automata [14][24]. Our model is formulated as a Continuous Time Markov Chain (CTMC), which allows us to model the rates with which different SLA GTs are violated. The derived Prism model is split into two

```
ctmc // A Continuous-Time Markov Chain model
const double sec = 1;
const double minute = 60;
const double hour = 60*minute;
const double day = 1;//24*hour;//set day as time unit
const double week = 7*day;
const double year = 365*day;
const double month = year/12;//an approximation

const int T; // represents time in properties
const int Xm; // represents amount in properties
const int MxInt = max(Xm+1,3,3*k+1);
const int k; // SLA variable

formula ConfA2ViolationRate=0.15/day;
formula IntA1ViolationRate=0.6/day;
formula IntA4ViolationRate=0.6/day;
formula AvailA3ViolationRate= (cntr_modify_AvailA3=0)
 (0.2/day): (0.1/day);
// Environment - Monitoring module - auto-derived
module IntA1
 [IntA1Violated] true -> IntA1ViolationRate: true;
endmodule
module ConfA2
 [ConfA2Violated] true -> ConfA2ViolationRate: true;
endmodule
module IntA4
 [IntA4Violated] true -> IntA4ViolationRate: true;
endmodule
module AvailA3
 [AvailA3Violated] true -> AvailA3ViolationRate:
true;
endmodule
```

Fig. 4 Prism model representing the run-time monitor part

main parts. The first one simulates the SLA Manager environment – see Fig. 4. It consists of a Prism module for each GT, that fires a violation for that GT at a rate specified by the designer. Rates are defined (by the user, separately from the SLA) as Prism formulas, i.e., non-parameterized macros, which allow us to support non-constant rates, as with AvailA3ViolationRate in Fig. 4. Each time a violation fires for a GT it synchronises, i.e., executes at the same time, with the transition that has the same name in the SLA Manager module, which is shown in Fig. 5 and Fig. 6. Each transition in CTMC Prism models has the form “[Name] Guard -> Rate: Assignments;”.

The SLA Manager module also has one transition per GT, which is enabled when the SLA is active and becomes disabled when renegotiation occurs, since SLAs follow a general lifecycle, where a renegotiate action terminates the SLA. Thus these transitions block the respective transitions of the environment modules in Fig. 4 from producing further violations when an SLA becomes inactive. Unlike the transitions in the GT_i environment modules, the transition rate here is 1. Thus, when they synchronise with the respective transitions from the GT_i modules, the rate of the synchronised transition is that of the GT_i module. This is important because in this way the only event that causes time to advance is the firing of some GT violation, whereby the time between successive firings is computed stochastically using an exponential distribution with rate $\lambda=\sum\lambda_i$, the sum of all the rates of violations [16]. *This is why there are no other transitions in any module* (even though they would have made it much easier to express variable updates) – *because they would have caused the time to advance even though there was no violation*. Finally, unlike the transition in the GT_i module

```
formula guard_penaltyConfA2 = (INCvltns_ConfA2) > 1;
formula guard_notifyConfA2  = (INCvltns_ConfA2) > 1;
formula guard_notifyIntA1      = (INCvltns_IntA1) <3;
formula guard_renegotiateIntA1 = (INCvltns_IntA1)>=3;
formula guard_notifyIntA4      = (INCvltns_IntA4)<3;
formula guard_renegotiateIntA4 = (INCvltns_IntA4)>=3;
formula guard_modifyAvailA3 = (INCvltns_AvailA3)<k;
formula guard_renegotiateAvailA3 =
(INCvltns_AvailA3)>(3*k);
formula guard_notifyAvailA3 = true;

formula INCvltns_ConfA2 =
((vltns_ConfA2+1>MxInt)?MxInt:(vltns_ConfA2+1));
// ...
formula INCcntr_notify_ConfA2 =
((cntr_notify_ConfA2+(guard_notifyConfA2?1:0)>MxInt)?
MxInt:(cntr_notify_ConfA2+(guard_notifyConfA2?1:0)));
// ...
formula INCcntr_penalty_ConfA2 =
((cntr_penalty_ConfA2+(guard_penaltyConfA2?1:0)>MxInt
)?MxInt:(cntr_penalty_ConfA2+(guard_penaltyConfA2?1:0
)));
formula INCpenalty_amount_ConfA2 =
((penalty_amount_ConfA2+(guard_penaltyConfA2?10:0)>Mx
Int)?MxInt:(penalty_amount_ConfA2+(guard_penaltyConfA
2?10:0)));
formula INCcntr_modify_AvailA3 =
((cntr_modify_AvailA3+(guard_modifyAvailA3?1:0)>MxInt
)?MxInt:(cntr_modify_AvailA3+(guard_modifyAvailA3?1:0
)));
```

Fig. 5 Prism SLA manager module – I

that has no actions associated with it, the transitions of the SLA Manager module have a long set of actions.

If we compare the actions of the different transitions inside the SLA Manager module we can see that they follow the same pattern – actually they are the same modulo certain renamings. All such transitions are responsible for incrementing the value of the different counters. The differences among the actions of these transitions lie on renamings to capture the fact that a particular GT_i has been violated. This allows us to produce GT-specific versions of the different guards and variable updates in the model. For example, in transition **`ConfA2Violated`** variable `SLAactive` is set to true, since there is no renegotiate action for that GT, while in transition **`IntA2Violated`** it is set to the negation of `guard_renegotiateIntA2`. The latter guard is the GT-specific guard of action renegotiate and is defined elsewhere in the model as a slightly modified version of the guard appearing in TABLE I (v>=3):

```
formula guard_renegotiateIntA2 = (INCvltns_IntA2) >= 3;
```

So we see that instead of directly using the variable `vltns_IntA2`, we use instead the expression `INCvltns_IntA2`. This is done because when this transition fires, the variables have not yet been updated to reflect the new violation – it is exactly this transition that is responsible for updating them. So all references to variables vltns_X have to be replaced with the expressions INC_vltns_X, which are defined as:

```
formula INCvltns_X = ((vltns_X+1>MxInt)
? MxInt :(vltns_X+1));
```

This formula simply expands to the value of the variable plus one when that is not greater than the maximum value that the variable can take or to the maximum value itself otherwise, in order to avoid a variable overflow that would corrupt the model. Since Prism does not support parameterised formulas, we produce a different such formula for each variable we need

```
module SLA_Manager
SLAactive : bool init true;
vltns_ConfA2:[0.. MxInt] init 0;
vltns_IntA1 : [0 .. MxInt] init 0;
vltns_IntA4 : [0 .. MxInt] init 0;
vltns_AvailA3 :[0 .. MxInt] init 0;
cntr_notify_ConfA2:[0..MxInt]init 0;
cntr_notify_IntA1 : [0 .. MxInt]init 0;
cntr_notify_IntA4:[0..MxInt]init 0;
cntr_notify_AvailA3:[0..MxInt]init 0;
cntr_penalty_ConfA2:[0..MxInt]init 0;
penalty_amount_ConfA2:[0..MxInt]init 0;
cntr_modify_AvailA3:[0..MxInt]init 0;

 [ConfA2Violated]  SLAactive -> 1:
  (SLAactive'= ! (false))
  & (vltns_ConfA2'=INCvltns_ConfA2)
  & (cntr_penalty_ConfA2'=INCcntr_penalty_ConfA2)
  & (penalty_amount_ConfA2'=INCpenalty_amount_ConfA2)
  & (cntr_notify_ConfA2'=INCcntr_notify_ConfA2);
[IntA1Violated]  SLAactive -> 1:
  (SLAactive'= ! (guard_renegotiateIntA1))
  & (vltns_IntA1'=INCvltns_IntA1)
  & (cntr_notify_IntA1'=INCcntr_notify_IntA1);
 [IntA4Violated]  SLAactive -> 1:
  (SLAactive'= ! (guard_renegotiateIntA4))
  & (vltns_IntA4'=INCvltns_IntA4)
  & (cntr_notify_IntA4'=INCcntr_notify_IntA4);
 [AvailA3Violated]  SLAactive -> 1:
  (SLAactive'= ! (guard_renegotiateAvailA3))
  & (vltns_AvailA3'=INCvltns_AvailA3)
  & (cntr_modify_AvailA3'=INCcntr_modify_AvailA3)
  & (cntr_notify_AvailA3'=INCcntr_notify_AvailA3);
endmodule
```

Fig. 6 Prism SLA manager module – II

to increment, as we cannot define a bounded increment function. However, the rest of the variables remain as they were in the guards, as these need to refer to the variable values *before* the transition fired. Updating the values of the variables follows a similar technique, whereby we set the new value of a variable X' to be the result of the formula INCX. This formula is defined to take into account the respective guards of the relevant actions, e.g.:

```
formula INCpenalty_amount_ConfA2 =
((penalty_amount_ConfA2 + (guard_penaltyConfA2 ? 10 : 0
) >MxInt) ? MxInt : (penalty_amount_ConfA2 +
(guard_penaltyConfA2 ? 10 : 0)));
```

This long expression is simply trying to increment the penalty amount paid due to violations of ConfA2 by the amount 10 that was requested in the respective penalty action but only do so if the guard of that action (shown in bold here) is satisfied, leaving it as it was otherwise (the ": 0" part). At the same time it makes sure that the new value will not become greater than the maximum possible value. This demonstrates the general rule for translating guards and variable updates in the model – for each action type X of a GT Y we produce a formula guard_XY and a formula to update variable cntr_X_Y. The guard needs to be the disjunction of all type X action instances of a GT (there might be more than one), and variable cntr_X_Y needs to be incremented by an amount that is in accordance with the guards of each action instance of type X. So for two X action instances (X1, X2) we would have:

```
formula INCcntr_X_Y =
  (cntr_X_Y + (guard_X1Y ? 1 : 0 )
              + (guard_X2Y ? 1 : 0 ) > MxInt)
  ? MxInt
  : (cntr_X_Y + (guard_X1Y ? 1 : 0 )
               + (guard_X2Y ? 1 : 0 ));
```

1) A number of n modules representing the environment of the SLA Manager, each of which emulates a violation of a $GT_{k <= n}$ with the respective rate GT_k`ViolationRate` (which is provided by the user):
```
module GTk
 [GTkViolated] true -> GTkViolationRate : true;
endmodule
```
2) At most $n \times m$ guards for the actions of each GT_k:
`formula guard_actnTp`GT_k `=` Or_{act} `guard(a)`
Where `actnTp` is in actionTypes(GT_k) and `act` is in actionInstances(GT_k).
3) At most $n \times m$ incrementation formulas for GT-specific variables cntr_X_Y_$INC_{var}GT_k$.

Fig. 7 Prism elements produced for an SLA

So given an SLA with n guarantee terms (GTs) and m actions (As), we produce a model with the elements shown in Fig. 7. Once we have the model, we can use Prism to check a number of properties so as to get a better understanding of the SLA – for example:

(i) What is the expected time for eventually (F) having a renegotiation?

- `R{"time"}=? [ F !SLAactive ]`

Prism reports that it would take 3.4195 days until an SLA renegotiation – and does so in 2.59 seconds.

(ii) What is the probability that a renegotiation will occur within the first 4 days?

- `P=? [ F<=(4*day) !SLAactive ]`

Prism reports that the probability is 0.6784 and does so in 2.345 seconds.

(iii) What is the probability to pay more than Xm currency units in the first month?

- `P=? [F<=month (penalty_amount_ConfA2>Xm)]`

Prism reports that P= 0.0046 in 0.083 sec, for k=1 and Xm=20.

(iv) What is the probability to have a violation on confidentiality or integrity of any data asset within a month?

- `P=? [F<=month (vltns_IntA1+vltns_ConfA2>=1)]`

Prism reports that it is 0.9114 (in 0.037 sec).

(v) What is the probability to reach double the infrastructure resources (i.e., to have $2k$ number of modifications for the operation assets) within the first month?

- `P=? [F<=month (cntr_notify_AvailA3 >(2*k))]`

Prism reports that it is 0.0459 (in 1.874 sec).

VI. Experimental Results

One can use Prism to evaluate a number of different properties as we have already done – the more properties one evaluates, the better they understand the ramifications of the SLA at hand. Prism allows one to see also how properties change along with certain model parameters. For example, what would the previous properties be if we considered time as a variable instead of some specific constant? The results for k=1, with T ranging within [1,15] are shown in Fig. 7 for properties (ii) – (v) and for values of Xm that are in the [20,50] interval. As the time now ranges in days instead of a month, the time to produce each of these graphs with Prism is relatively short – even for Xm=50 (which causes the upper range of all integer variables to be equal to 51) the time to complete the analysis is less than 17 sec. As the results show the SLA will not remain active for long – in around 10 days the probability of initiating a renegotiation (property ii) is almost 1 and that is independent of the value of Xm, our budget for penalties. Indeed, Xm is irrelevant for properties (ii), (iv) and (v) as we can see from the graphs, since all curves for different values of Xm coincide. The graphs of properties (iii)–(v) show that renegotiation will occur due to a high number of violations but that we will not have to pay more than our penalty budget within that time interval (property iii), which is a positive result.

Had we obtained an unexpected result during the analysis, it would be an indication that there is something wrong. The problem could be with either (or both):

1) The property itself – maybe we are not actually specifying what we should be specifying.
2) The SLA itself and/or our understanding of it – maybe it is not doing what we think it is doing.

Developing reasonable SLAs that satisfy a number of different properties is quite difficult – it is easy to slip up and introduce errors when programming (which is what we are doing essentially). Since with SLAs our mistakes can be translated into monetary penalties directly (or even indirectly by being forced to allocate too many resources to a client), it is imperative that we have as good an understanding of what exactly we are committing to. What may sound good initially is not necessarily so. The same is true when specifying properties – these can be quite tricky to express correctly and sometimes it helps immensely to refer to existing patterns of properties [15]. Another issue with properties expressed in Prism's language is that they are evaluated at the *initial* state by default. If one wishes to evaluate them at a different state then they need to use what Prism calls a *filter* that take the form *filter(op, prop, states)*, e.g., what is the maximum probability of paying a penalty from all states where there has been one violation of Confidentiality for asset A2?

- `filter(max, P=? [F (cntr_penalty_ConfA2>1)], (vltns_ConfA2=1))`

Prism reports that P= 0.1031 for k=1 and Xm=15 and that there is only one state that satisfies the state filter.

In Fig. 9 we can see the SLA analysis time for different values of the Xm and T parameters when checking properties (ii)–(v). We can see that the time increases along with Xm, which is reasonable because Xm causes an increase in the size of all the model integer variables thus increasing the required memory space for analysis. We can also see that properties (ii) and (v) require substantially more time than the other two – indeed they are ranked as (iv), (iii), (v), (ii). This is because property (iv) covers only the relatively few states where the sum of violations of A1 & A2 are less than 2, while property (ii) on the other extreme, needs to cover all possible states.

VII. Related Work

SLAs have been widely used to specify terms and conditions for a service provision between service consumers and service providers [1]. In order to automate the SLA specification process, several specification languages have been defined over the years, with the aim to simplify the SLA

specification process for the involved parties and to minimise the time and cost required for this. Despite though the extended research on SLA languages and SLA management, not much work has been done on validation of the SLA process.

Zseby in [26] presented three different types of sampling techniques of SLA validation, with non-intrusive two-point measurements. However, these validation techniques rely on the actual network traffic in order to capture packets with the main focus of validating the delay guarantees of an SLA.

Haq et al. in [8] introduced the need for proactive and reactive SLA validation and in [9] the need of three different SLA validation models (from resource, infrastructure and business points of view), and presented an SLA holistic validation framework for service value chains and Cloud based service value chains. Nevertheless, both approaches refer to the validation of an SLA after the establishment of the agreement, and not prior to it.

Extensive work has also been conducted with regards to verification of SLAs. Ishakian et al. in [29] present a formal verification for SLA transformations. Their framework enables users to propose specific transformation rules, which are fed to an SLA transformation engine, in order to explore any possible safe co-location configurations in a cloud setting. The work done in [7] presents a framework for SLA verification by involving a third-party auditor. In this work the authors provide two testing algorithms to detect an SLA violation on a VM memory size and to defend various attacks from an untrusted cloud.

The authors in [20] introduce a formal model to optimise and reason about service availability and budget compliance through monitoring. The monitoring model used in this work is designed to observe in real-time these two service characteristics and react to them based on the defined SLA of the services.

In [17] the authors propose a proactive approach for evaluating and testing the logical composition of guarantee terms of SLAs, by defining a four-valued logic that allows evaluating both the individual guarantee terms and their logical relationships. They have created a test criterion based on the modified condition decision coverage (MCDC), in order to obtain a cost-effective set of test requirements from the structure of the SLA. Furthermore, by analysing the syntax and semantics of the agreement, they define specific rules to avoid non-feasible test requirements.

However, all work done regarding the SLA verification is not conducted prior to sign the agreement but it is rather occurring during the SLA period, in order to detect possible breaches of the terms or reconfigure at run-time services to continue being compliant with the agreed terms.

VIII. Conclusion and Future Work

Specifying SLAs that can be used to govern automatically a service-based system is a difficult activity. We have extended WS-Agreement to allow more precise definition of the SLA Guarantee Terms (GT) that need to be monitored during run-time and the actions that need to be taken

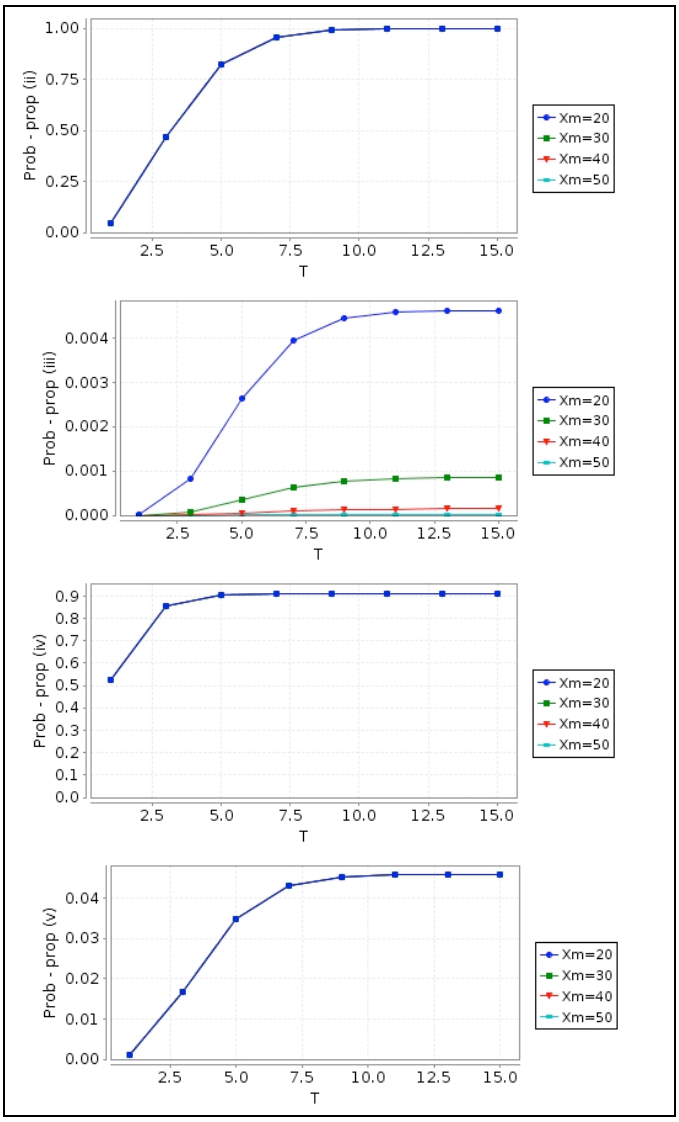

Fig. 8 Properties (ii) – (v) across time (Xm=[20,50], ,k=1, T=[1,15])

whenever a GT is violated. As the actions can lead to monetary penalties and/or to an increase in the resources allocated to clients it is imperative that SLA specifiers have a way to validate the SLAs they offer and understand their consequences – a task that is not at all easy to perform. For this reason, we have developed a method to translate an SLA into a formal language for temporal stochastic models, that allows us to analyse it and validate it by examining different types of properties that one may be interested in – both to validate that the SLA behaves as expected and to identify issues that could potentially lead to exposure to high penalties. The derived models can be used to explore different environmental assumptions, such as different rates of GT violations (which depend on the actual rates of client requests and the ability of our services to respond to these while respecting the SLA GTs), different penalty amounts, different time-intervals, etc.

This approach opens the way for more precise SLAs on which we have a higher degree of trust. It also enables the possibility of re-validating an SLA at run-time, once real

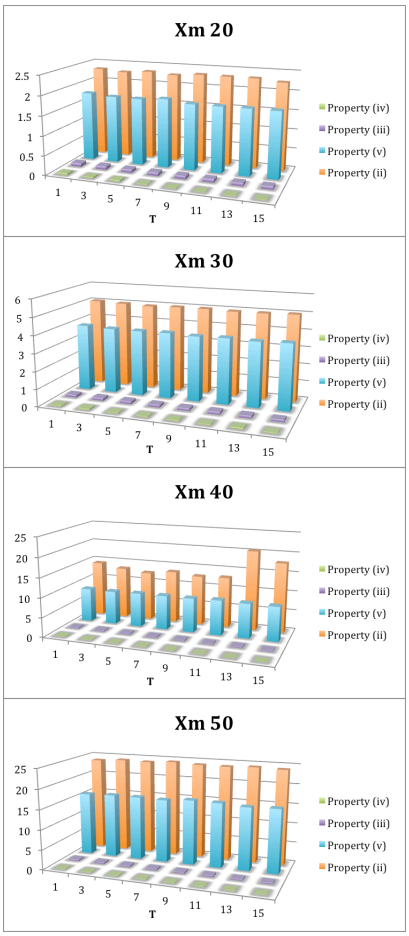

Fig. 9 Execution Times

violation rates have been observed, instead of depending on initial estimations of these at SLA pre-deployment time, as well as to evaluate proposed SLAs during re-negotiation.

ACKNOWLEDGMENT

This work was partly supported by the EU-funded project TOREADOR [28] (grant no H2020-688797).

REFERENCES

[1] A. Maarouf, A. Marzouk, and A. Haqiq, "A Review of SLA Specification Languages in the Cloud Computing". In Proc. of the 10th International Conference on Intelligent Systems: Theories and Applications (SITA), pp. 1–6, 2015.

[2] CUMULUS Project, Deliverable D3.2 Core Certification Mechanisms vol. 2, 2013. Available from: http://www.cumulus-project.eu

[3] G. Roşu, and T. Şerbănuţă, An overview of the K semantic framework, The Journal of Logic and Algebraic Programming, Volume 79, Issue 6, 2010, pp. 397-434, ISSN 1567-8326, DOI: 10.1016/j.jlap.2010.03.012

[4] G. Spanoudakis, C. Kloukinas, and K. Mahbub, "The SERENITY Runtime Monitoring Framework," in Security and Dependability for Ambient Intelligence, vol. 45, Eds. S. Kokolakis, A. M. Gómez, and G. Spanoudakis, Springer US, pp. 213–237, 2009.

[5] H. Foster and G. Spanoudakis, "Advanced service monitoring configurations with SLA decomposition and selection", In Proceedings of the 2011 ACM Symposium on Applied Computing, pp. 1582-1589, 2011.

[6] H. Ludwig, A. Keller, A. Dan, R.P. King and R. Franck, "Web Service Level Agreement (WSLA) Language Specification", 2002.

[7] H. Zhang, L. Ye, J. Shi, X. Du, and M. Guizani, "Verifying cloud service-level agreement by a third-party auditor", Security and Communication Networks, *vol.* 7, no. 3, pp. 492-502, March 2014.

[8] I. U. Haq, E. Schikuta, I. Brandic, A. Paschke, H. Boley, "SLA Validation of Service Value Chains", In Proc. of the 2010 Ninth International Conference on Grid and Cloud Computing, pp. 308-313, November 2010.

[9] I. U. Haq, I. Brandic, and E. Schikuta, "SLA validation in layered cloud infrastructures", In Proc. of the 7th international conference on Economics of grids, clouds, systems, and services (GECON'10), Eds. J. Altmann and Omer F. Rana. Springer-Verlag, Berlin, Heidelberg, pp. 153-164. 2010.

[10] K. Andrieux, K. Czajkowski, K. Keahey, A. Dan, K. Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, "Web Services Agreement Specification (WS-Agreement)", Global Grid Forum GRAAP-WG, vol. 192, pp. 1–80, 2004.

[11] K. Mahbub and G. Spanoudakis, "Monitoring WS-Agreements: An Event Calculus–Based Approach", In Test and Analysis of Web Services pp. 265-306, Springer Berlin Heidelberg, 2007.

[12] K. Mahbub and G. Spanoudakis, "Proactive SLA negotiation for service based systems: Initial implementation and evaluation experience", In Proceedings - 2011 IEEE International Conference on Services Computing (SCC 2011), pp. 16–23, 2011.

[13] K.T. Kearney, F. Torelli and C. Kotsokalis, "SLA*: An Abstract Syntax for Service Level Agreements", In 2010 11th IEEE/ACM International Conference on Grid Computing, pp. 217–224, 2010.

[14] Kwiatkowska M., Norman G. and Parker D. "PRISM 4.0: Verification of Probabilistic Real-time Systems". In Proc. of the 23rd International Conference on Computer Aided Verification (CAV'11), vol. 6806, LNCS, pp. 585-591, 2011.

[15] M. B. Dwyer, G. S. Avrunin and J. C. Corbett, "Patterns in property specifications for finite-state verification," In Proc. of the 1999 Intl Conf. on Soft. Eng., pp. 411-420, doi:10.1145/302405.302672 (patterns.projects.cs.ksu.edu/), 1999.

[16] M. Kwiatkowska, G. Norman and D. Parker, "Stochastic Model Checking", In Proc. SFM'07, vol. 4486, LNCS, pp. 220-270, 2007.

[17] M. Palacios, J. García-Fanjul, J. Tuya and G. Spanoudakis, "Coverage-based testing for service level agreements", In IEEE Transactions on Services Computing, vol. 8, no. 2, pp. 299-313, 2015.

[18] M. Shanahan, "The Event Calculus Explained", pp. 1–22, 1999.

[19] M.L. Massie, B.N. Chun and D.E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience", Parallel Computing, vol. 30, no. 7, pp. 817–840, 2004.

[20] N. Behrooz, S. de Gouw, and F. S. de Boer. "Formal Verification of Service Level Agreements Through Distributed Monitoring." In ESOCC, pp. 125-140. 2015.

[21] N. Oldham, K. Verma, A. Sheth, and F. Hakimpour, "Semantic WS-agreement partner selection". In Proc. 15th Int. Conf. World Wide Web, pp. 697–706, 2006.

[22] Nagios. "Nagios - The Industry Standard in IT Infrastructure Monitoring", 2016. Available from: http://www.nagios.org/

[23] O. Waeldrich, D. Battré, F. Brazier, K. Clark, M. Oey, A. Papaspyrou, P. Wieder and W. Ziegler, "WS-Agreement Negotiation Version 1.0", GRAAP-WG, Open Grid Forum, 2011, Available from: http://www.ogf.org/documents/GFD.193.

[24] Prism Model Checker, http://www.prismmodelchecker.org/

[25] S. Nepal, J. Zic and S. Chen, "WSLA+: Web service level agreement language for collaborations", In Proceedings - 2008 IEEE International Conference on Services Computing (SCC 2008), pp. 485–488, 2008.

[26] T. Zseby, "Deployment of Sampling Methods for SLA Validation with Non-Intrusive Measurements", In Proc. of Passive and Active Measurement Workshop Fort Collins, Colorado, April 2002.

[27] TOREADOR Project, Deliverable D4.1 MBDA-as-a-service SLA and Assurance, Available from: http://toreador-project.eu/

[28] TOREADOR project, http://www.toreador-project.eu/

[29] V. Ishakian, A. Lapets, A. Bestavros, and A. Kfoury, "Formal Verification of SLA Transformations". In Proceedings of the 2011 IEEE World Congress on Services (SERVICES '11). IEEE Computer Society, pp. 540-547, 2011.