Efficient Monitoring of Web Service SLAs*

Franco Raimondi, James Skene, Liang Chen, and Wolfgang Emmerich
Dept. of Computer Science
University College London
London WC1E 6BT, U.K
{f.raimondi|j.skene|b.chen|w.emmerich}@cs.ucl.ac.uk

ABSTRACT

Web services are increasingly used in inter-organizational settings. If an organization depends on the service quality provided by another organization it often enters into a bilateral *service level agreement* (SLA) to precisely determine service quality and permitted service use. SLAs then also determine penalty payments as risk mitigation against poor service quality and overuse of the service. Once these agreements are entered into, it becomes necessary to monitor for both poor service quality and also abuse of the provision beyond the agreed limits. We address the question of how service level agreements can be monitored efficiently and automatically. We show how timeliness constraints, such as latency, throughput, availability and reliability, in formal service level agreements can be translated into timed automata. We attach time stamps to SOAP messages and consider these messages as timed letters. We are then able to reduce the question of detecting SLA violations to acceptance of timed words by the timed automata that have been derived from the SLA. Acceptance of a timed word by a timed automaton can be decided in polynomial time and because the timed automata can operate while SOAP messages are exchanged at run-time there effectively is only a linear run-time overhead. We evaluate the efficiency and scalability of this approach using a large-scale case study in a service-oriented computational grid.

1. INTRODUCTION

There is a growing trend for IT systems to be integrated across organizational boundaries. Examples include supply-chain management using RosettaNet [10] and outsourcing of non-core business, for example the use of specialist providers for managing human resources and account payable services. These services are frequently implemented using web services technologies. When organizations do rely on the web services provided by other organizations for the implementation of their business processes they usually want contractual guarantees on service quality. Likewise, the providers want assurances that their clients are not abusing the service. These guarantees are often provided using *service level agreements* (SLAs), which determine provided and required quality of service and associate penalty payments with agreement violations. These penalty payments can be seen as an insurance policy against poor service provision and over-use.

In previous work, we have investigated how SLAs can be defined precisely. In [31], we have presented how the semantics of languages for service level agreements can be devised using a model-denotational approach. We have used this approach and published the SLAng language for defining SLAs for web services [29] as these web services are often used in inter-organizational settings. The relevant web services specifications include the Simple Object Access Protocol (SOAP), the Web Services Description Language (WSDL), the Universal Directory and Discovery Interfaces (UDDI) and the Business Process Execution Language (BPEL).

Testing the quality of provided web services using, for example, performance and reliability tests is necessary but insufficient. The service quality fundamentally depends on the provision of computational resources that the service provider maintains for the web service during the lifetime of the service, as well as the demand on those resources by other users of the same service. Service providers cannot make reliable quality of service guarantees without restricting the consumption of these services by concurrent users. In order to police a service level agreement, it is therefore necessary for the service user to monitor constantly, or at least in statistically significant intervals, the service quality that is provided at run-time. Likewise the service provider will have to monitor service quality at run-time in order to detect usages that exceed the utilization levels agreed in the SLA. The service provider will also have to monitor to protect itself against false claims of poor services.

In [32] we have delineated how systems of SLAs between a web service provider, an Internet Service Provider and a web service user should be arranged in a manner that they are principally monitorable. We have also sketched in [30] how model driven architecture techniques can be used to derive a monitoring implementation fully automatically from the semantics definition of the SLA language. However,

*The work described in this paper was partially funded by EU IST Grant 2005-026955 (PLASTIC) and EPSRC Grants GR/S90843/01 (OMII Managed Programme) and EP/C534891 (Divergent Grid).

the approach relies on OCL interpretation of the semantics definition and is too inefficient for production use.

The principal contribution of this paper is the presentation of how Time Continuous Temporal Logic (TCTL) and timed automata [2] can be used as a formalism to support efficient monitoring of timeliness constraints expressed in SLAs for web services. We derive a TCTL formula timed automaton for each timeliness constraint expressed in an SLA and compile the formula into a timed automaton. We timestamp SOAP messages that are exchanged as part of web service requests and responses and consider these as timed letters. We are then able to reduce the question whether the SLA is violated to acceptance of a timed word by the timed automata. This is decidable in polynomial time and because we can process each message on the fly the approach introduces only a linear run-time overhead into a web service environment. We have implemented the approach using Apache Axis handlers, the XML-based definition of Timed Automata and related parser of Uppaal [21], and an extension of the automata-based model checking procedure for LTL [7] for the verification of real-time requirements. We have evaluated the performance overhead and scalability of this implementation using a case study that monitors SLAs in a service-oriented grid.

This paper is further structured as follows. Section 2 reviews the theoretical analysis of SLAs presented in [31, 29], the formalism of timed automata and the syntax of the logic TCTL to reason about time. Section 3 extends TCTL with an additional operator to count events and investigates the problem of verifying traces against real-time requirements. Section 4 presents a translation from a subset of SLAng [29] to Timed Automata. Section 5 introduces a methodology for the on-line verification of Timed Automata against traces generated by web services. The results of the evaluation of this approach using a case study that monitors SLAs in a grid are presented in Section 6. Section 7 discusses related work, and concluding considerations are presented in Section 8.

2. BACKGROUND

We first review the theoretical analysis of SLAs as presented in [32]. In particular, primitives for parties, actions and events are introduced to define the scope of SLAs.

We then review the syntax of the temporal logic TCTL to reason about real-time, as defined in [2], and then we present the formalism of timed automata.

2.1 A formalism for Service Level Agreements

Consider the scenario from [32] depicted in Figure 1. In this scenario, a Client (C) communicates with a Service Provider (S) using a network provider (I) to request some operation, the result of which is returned from S to C using the same network provider I. The Service Provider S may additionally interact with other services (either external or internal to its structure) to provide the required result to C.

The scenario presented above can be formalised using:

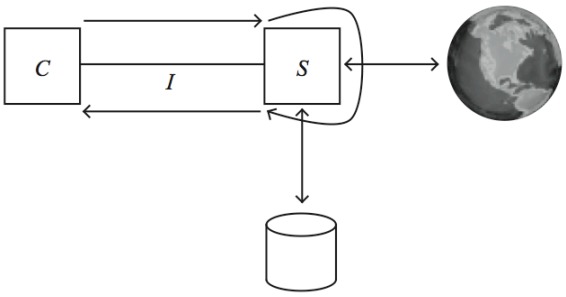

Figure 1: The three party scenario.

- A set of Participants $P = \{C, I, S\}$.
- A set of actions A that can be performed by the participants. For instance, in the three party scenario, A may contain the actions $A = \{$request, send, process, respond$\}$. The party performing an action $a \in A$ is denoted by $\alpha(a)$. As an example, $\alpha(\text{send}) = I$.
- A set of events E; intuitively, events are the outcomes of actions that are observable by one or more participants. The set of events associated to an action is denoted by $\epsilon(a)$.
- The set of participants able to observe a given event e is denoted by $\rho(e)$; the set $\rho(e) \subseteq P$ is called the set of *respondents to e*. It is assumed that for all $e \in \epsilon(a)$, $\alpha(a) \in \rho(e)$ (i.e., at least those who perform an action a are responders).

Central to the definition of Service Level Agreements is the notion of *observations*:

Definition 1 (Observations). *Observations are logical predicates concerning the values of attributes of observed events [32].*

Let O be the set of all possible observations (i.e., the set of all possible logical predicates over events), and let $o \in O$ be an observation. We denote by $\pi(o) \subseteq E$ the set of events (with the required attributed) related to o.

In this paper we restrict ourselves to *time attributes*: specifically, we assume that the only attributes of events are time attributes. Under this assumption, we can write observations using TCTL, a temporal logic to reason about real time (see [1] and Section 2.2).

Following [32], we define an SLA to be a tuple (p, c, o), where $p, c \in P$ and $o \in O$. Intuitively, (p, c, o) denotes the fact that a provider p provides to a client c the guarantee that an observation $o \in O$ holds and *vice-versa*. The language defined in [29] enables the definition of SLAs and associated penalties when violations occur. Additionally, the language provides features to schedule different SLAs to hold at different times, possibly related to changes in the context.

We refer to Section 4 for a simple example of an SLA in SLAng, and to the files available at [29] for more details.

2.2 TCTL, a logic for real time

TCTL [1] is a temporal logic which allows to reason about durations and time intervals.

DEFINITION 2 (SYNTAX OF TCTL). *Let AP be a set of atomic propositions; let I be an interval in $\mathbb{R}$; let $\sim$ denote any of the binary relations $<, \leq, >, \geq, =$. TCTL formulae are defined inductively by:*

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \psi \mid \mathsf{E}\,[\varphi\ \mathsf{U}_{\sim c}\,\psi] \mid \mathsf{A}\,[\varphi\ \mathsf{U}_{\sim c}\,\psi]$$

where $p \in AP$ and $c \in \mathbb{R}$.

A formula of the form $\mathsf{E}\,[\varphi\ \mathsf{U}_{\leq 5}\,\psi]$ is read as "there exists an execution (of the system) such that within 5 time units ψ will hold, and until then φ holds". The A quantifier expresses the universality of the formula following it (i.e., "for all possible executions").

Traditionally, the syntax of TCTL includes unary operators to express that a formula will hold eventually, and that a formula will hold globally (even though these can be derived from the minimal set presented above). Specifically, the additional operators are $\mathsf{EF}_{\sim c}\varphi$ and $\mathsf{AF}_{\sim c}$ to express that "there exists an execution such that within a time bound c φ will eventually hold" (resp.: for all executions). These operators are equivalent to $\mathsf{EF}_{\sim c}\varphi \equiv \mathsf{E}\,[\top\ \mathsf{U}_{\sim c}\varphi]$ and $\mathsf{AF}_{\sim c}\varphi \equiv \mathsf{A}\,[\top\ \mathsf{U}_{\sim c}\varphi]$. The additional pair of operators $\mathsf{EG}_{\sim c}\varphi$ and $\mathsf{AG}_{\sim c}\varphi$ express that "there exists an execution such that φ globally holds withing the time bound c" (resp. for all executions). These two operators are equivalent to $\mathsf{EG}_{\sim c}\varphi \equiv \neg\mathsf{AF}_{\sim c}\neg\varphi$ and $\mathsf{AG}_{\sim c}\varphi \equiv \neg\mathsf{EF}_{\sim c}\neg\varphi$

The semantics of TCTL can be given using timed automata (see next Section); we refer to [1] for more details. Efficient algorithms and tools have been developed for model checking TCTL, using various techniques; see, for instance, [11, 12, 4, 26].

For the purpose of this paper, we are interested in TCTL as a formalism to express temporal requirements: those used in *observations* of SLAs. To this end, we use the idea of *specification patterns*. These are defined as *"the description of a commonly occurring requirement"* [14]; a classification scheme for real-time specification patterns has been defined in [20].

In Section 4 we present a translation from SLAng clauses to TCTL formulae using patterns. In turn, TCTL formulae can be encoded as timed automata. Intuitively, the timed automaton associated to a given TCTL formula encodes all the possible computations in which the formula TCTL holds. We refer to [1] for further details on the semantics of TCTL using timed automata.

2.3 Timed automata

A simple automaton is a tuple $A = (\Sigma, Q, Q^0, \delta, F)$. As an example, consider Figure 2 where $\Sigma = \{a, b\}$ is an alphabet, $Q = \{0, 1\}$ is a set of states, $Q_0 = \{0\}$ is the initial state, $F = \{1\}$ is the final state, and the transition relation δ enables the transitions depicted.

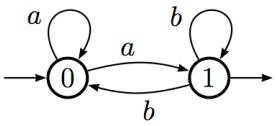

Figure 2: A simple automaton.

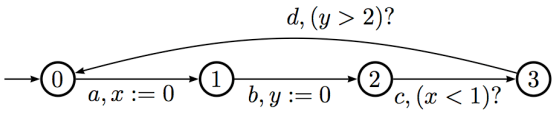

Figure 3: A timed automaton.

Automata recognise languages: given an automaton A, $\mathcal{L}(A) \subseteq \Sigma^*$ is the language accepted by the automaton. For the automaton of Figure 2, $\mathcal{L}(A)$ includes the words $\{a, ab, abb, abaaaaabaab, \dots\}$.

A *time sequence* is a sequence of real numbers $\tau = \tau_1\tau_2\dots$ s.t. $\tau_i > \tau_{i-1}$ and the sequence is *non-Zeno* (i.e., the sequence is not bounded). A *timed word* is a pair (σ, τ) where σ is a standard word and τ is a time sequence, e.g., $\{(aab\dots), (0.1, 0.3, 1.2, \dots)\}$.

Timed automata [2] extend simple automata by introducing a set of clocks $x, y, \dots$, a set of time constraints over transitions, and clock reset operations over transitions. As an example, consider the timed automaton in Figure 3: here two clocks appear (x and y). Clock x is reset to 0 with the operation $x := 0$ when a transition is performed from state 0 to state 1. Analogously, clock y is reset from state 1 to state 2. The label $(x < 1)$? from state 2 to state 3 imposes that the transition has to be performed when the value of clock x is less than 1; similarly, the transition from 3 back to 0 has to be performed when the value of clock y is greater than 2.

Timed automata accept timed words, i.e. *they recognise timed languages*. For instance, the timed automaton in Figure 3 recognises the language $\mathcal{L}(TA) = \{((abcd)^\omega, \tau) | (\tau_{4j+3} < \tau_{4j+1} + 1) \wedge (\tau_{4j+4} > \tau_{4j+2} + 2)\}$

Our idea is to encode specification patterns for Service Level Agreements as timed automata, so that the correctness (or the violation) of an execution can be verified by checking its inclusion in the language accepted by the automata. The details of this methodology are presented in Section 3.1.

3. COUNTING EVENTS AND VERIFICATION TECHNIQUES

A number of requirements involving time can be encoded as TCTL formulae. However, TCTL has no operators for "counting" events, and such requirements appear often in

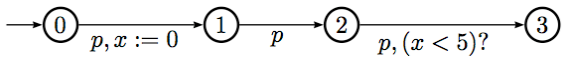

Figure 4: A timed automaton for counting events.

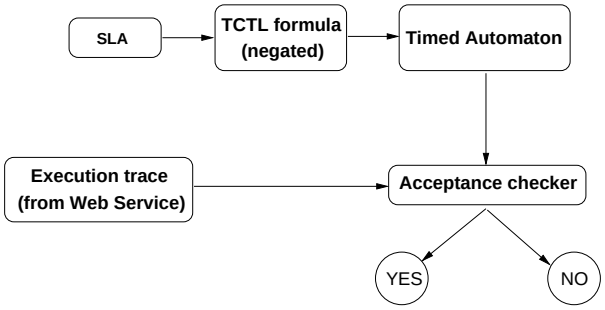

Figure 5: A timed automaton for counting events.

Service Level Agreements. For instance, a typical observation appearing in an SLA could be "there are never more than 10 requests in a given minute". There does not seem to be an easy way to express this kind of property in TCTL; moreover, counting events in an interval does not appear as a specification pattern, neither "qualitative" [14] nor "quantitative" [20].

In this paper we propose an extension of the syntax of TCTL by means of an additional operator EC to "count" propositions in a time interval: the operator $\mathsf{EC}\,(p, n, t)$ is read as *proposition p happens (at least) n times in the next t time units* (in the formula above, $n \in \mathbb{N}$ is a positive natural number, and $t \in \mathbb{R}$ is a real number).

Formally, given a timed automaton TA and a formula of the form $\mathsf{EC}\,(p, n, t)$, the formula is satisfied by the automaton (written $TA \models \mathsf{EC}\,(p, n, t)$ if there exists an execution of the automaton in which all the time constraints are satisfied and such that at least n states in which p holds are traversed consecutively, within a time bound of t.

As an example, Figure 4 depicts an automaton which accepts runs for $\mathsf{EC}(p, 3, 5)$, i.e., runs where p happens 3 times within 5 times unit.

Notice that, in general, an automaton accepting the formula $\mathsf{EC}\,(p, n, t)$ has $n + 2$ states.

Based on the observations above, we rewrite the definition of *observations* presented in Section 2.1 as follows

Definition 3 (Timed observations). *Observations are formulae of TCTL extended with the additional operator* $\mathsf{EC}\,(p, n, t)$.

3.1 Verification of traces

Figure 5 summarises the proposed methodology for verification of traces. Given an SLA between two participants over the execution of some service (in particular, over a Web Service):

1. The SLA is translated into an extended TCTL formula representing its *negation*. For instance, if the SLA requests that the client is not allowed to perform more than 3 requests in 5 seconds, the SLA is translated into the formula $\mathsf{EC}\,(\text{request}, 3, 5)$.
2. The formula is translated into the appropriate timed automaton accepting all the timed words in which the (negated) formula holds. The automaton accepting $\mathsf{EC}\,(\text{request}, 3, 5)$ is presented in Figure 4, where "request" is substituted by the proposition p.
3. The execution trace of a Web Service can be seen as a timed word: indeed, it is sufficient to timestamp events (in the sense of Section 2.1) with their time of occurrence to generate the timed word corresponding to a particular execution.
4. The key observation is the following: *an SLA is violated by an execution when the timed automaton corresponding to the SLA via a TCTL formula accepts the timed word encoding the execution trace.* In our methodology, an "acceptance checker" is responsible for this verification.

The problem of verifying a TCTL formula in a given model for TCTL has been investigated using model checkers. However, for our purposes we do not need the full machinery of model checking a generic temporal formula in a generic model for TCTL. Indeed, our aim is to check *acceptance*. We exploit this fact in two ways in the definition of the acceptance checker:

- Verification can be performed "on the fly" (i.e., while the trace is being built): this is a standard optimisation technique for model checking, for instance see [25]). See the next section for considerations about the length of traces.
- Instead of using traditional techniques for real time logics (e.g., clock regions and equivalence classes), we suggest an extension of the model checking technique used in the SPIN model checker [19]. In particular, to check acceptance, we check whether or not the timed word $\mathcal{W}$ corresponding to the actual execution is a member of the language $\mathcal{L}(TA_\varphi)$, where TA_φ is the timed automaton corresponding to an extended TCTL formula (see [7] and references therein for an introduction to automata-based model checking).

An implementation of this methodology is presented in Section 5 and an experimental evaluation can be found in Section 6.

3.2 Maximum counterexample diameter

Some devices may have limited storage capabilities (e.g., mobile phones). Storing the full log of timestamped events

of executions for a long period of time may be too space demanding, but at the same time it may be required to keep evidence of the possible violations in order to obtain the compensation prescribed by SLAs. The methodology presented in the previous section permits considerations on the size of the logs.

In particular, what is the amount of data needed to verify whether $\mathcal{W} \in \mathcal{L}(TA_\varphi)$? By estimating this, it is possible to discard or compress non-relevant data.

The problem of investigating the length of witness paths for temporal formulae has been investigated for Bounded Model Checking (BMC) techniques [5] for CTL and LTL. The idea is that, given a temporal formula φ, it is possible to compute a number giving the maximum number of steps needed in a model to verify that formula. There is no work in this direction for TCTL; moreover, there is a negative result for CTL: the required length of a path might be infinite for certain formulae [33], which clearly extends to TCTL.

However, if we consider only particular specification patterns, we are able to place a bound on the length of traces, called the *maximum diameter for a counter-example*. In particular, consider the formula $\varphi = \mathsf{EC}\,(p, n, t)$. Let's suppose this formula is monitored by a certain automaton: the amount of data required to evaluate φ is bounded in two ways. Firstly, if no events happen for t time units, then the process can "forget" the past events. Secondly, if less than n events happened in the last $t + \epsilon$ time units (with $\epsilon > 0$), then the first of the n events can be discarded. Thus, it is possible to place a bound on the length of traces to detect failures on the fly for EC formulae. Similarly, bounds can be computed when checking request-response patterns, and reliability patterns, depending on the actual parameters used.

Additional considerations. In fact, the management of traces and the administration of SLAs might need to take into account more parameters: it is possible that the violation of an SLA is detected by one party only, and the violation might be reported only after a certain period of time. In this case, if the other party has discarded the traces, there is no possibility of reconciliation. Thus, the evaluation of the data required to validate SLAs on-the-fly need to be integrated with additional parameters, such as the maximum time allowed for violation reporting. But, as mentioned above, historical data not required for on-the-fly validation may be compresses and stored more efficiently.

3.3 The complexity of the approach

The problem of checking whether a given (timed) word is recognised by a given timed automaton can be solved in polynomial time, similarly to the problem of model checking a TCTL formula [23]. This result implies that the acceptance checker presented above only needs at most a polynomial amount of time to check acceptance (polynomial in the size of the automaton), because the additional operator EC generates an automaton which is only polynomially larger than an automaton for a generic TCTL formula.

In fact, we can compute the worst case time complexity of the approach as follows. The acceptance checker does not have to explore the full automaton: when an event and its occurrence time are reported, the checker has to verify whether or not a successor state exists. This step requires at most time n (i.e., the size of the automaton). If sliding time windows are being used, then the verification has to be performed at most n times, recording a trace of at most n steps: this is due to the fact that a failure trace cannot contain more steps than the number of states of the automaton for th kind of properties corresponding to SLA patterns. Therefore, the verification process has to be repeated at most n times, taking a different state of the trace as the initial state of the execution. Consequently, the total time required for verification is at most n^2.

On a different note, the problem of satisfiability for TCTL is undecidable [15], i.e., given a generic TCTL formula φ, the problem of deciding the existence of a model for it undecidable. This implies that the satisfiability problem for the extended TCTL is undecidable as well. However, the problem of checking satisfiability does not arise in our approach, for which formulae and models (or execution traces) are given explicitly.

4. SLANG AND TIMED SPECIFICATIONS

SLAng [29] is a language for Service Level Agreements developed in the Department of Computer Science, University College London. Its design aims are *understandability, precision, practicality, and monitorability*: the language is intended to be understood by participants, to give unambiguous interpretation of the events, and to be useful in the definition of contracts in different contexts.

The syntax and semantics of SLAng are defined using EMOF models (a language similar to UML class diagrams), by associating a model of the language with a model of service usage. The precise semantics of SLAng clauses is described using the Object Constraint Language (OCL).

Previous attempts in monitoring SLAng employed OCL checkers to verify that SLAng clauses (expressed as OCL expressions) were satisfied. Effectively, this approach does not scale with the number of messages exchanged, because current OCL checkers do not recognise the temporal structure of events. Therefore, each new event added to the system is compared to all the previous events by an OCL checker.

In this section we present examples of the reduction of a Service Level Agreement defined in SLAng to TCTL formulae, and then to timed automata. This reduction enables the continuous monitoring of events, irrespective of the number of messages previously exchanged (see Section 6 for experimental results).

Notice that in this paper we consider timeliness properties only. While SLAng allows for the definition of agreements involving other measures (such as data size etc.), here we focus on temporal properties only. In particular, SLAng include the following clauses:

- `InputThroughputClause`: this clause defines the maximum number of requests that can be submitted in a given time window.

```
inputThroughputClause = {
   FailureModeDefinition[f1]() {
      // Max 10 requests in 1 min
      [...]
      inputWindow = ::types::Duration(1, min)
      inputConcurrency = 10;
      [...]
   }
}


failureModeDefinition = {
  // A failure if latency > 5s
  [...]
  maximumLatency = ::types::Duration(5, S)
  [...]
}

[...]

reliabilityClause = {
   // A failure is the thing defined above
   failureModes = { FailureModeDefinition[f1] }
   [...]
   // Reliability requested: 90% over 1 min intervals.
   reliabilityCount = 2
   window = ::types::Duration(1, min)
   [...]
}
```

Figure 6: HUTN code for SLAng (excerpts)

- `Latency`: this clause defines the maximum latency for a service.
- `ReliabilityClause`: this clause defines the maximum number of failures in a given time window.
- `FailureModeDefinition`: a failure is defined as the violation of one of the clauses described above.

In a typical instance, these clauses are included in a bigger specification involving definitions for participants, schedules, penalties, etc. We refer to [29] for more details. As an example, excerpts from an SLA written in SLAng are reported in Figure 6 using OMG's HUTN (Human-Usable Textual Notation) for an instance of the EMOF model defining the syntax of SLAng. Essentially, this is a reliability clause for a service provider, which is required to have less than 2 failures in any sliding time window of one minute. Additionally, the client is subject to an inputThroughputClause: no more than 10 requests per minute can be submitted to the service provider.

We start by analysing `inputThroughputClause`: the requirement in the Figure demands that no more than 10 requests can be performed in 1 minute. The requirement corresponds to (the negation of) the formula $\mathsf{EC}\,(\text{request}, 10, 60)$. This formula, in turn, can be represented using an automaton similar to the one in Figure 4, against which execution traces can be validated.

`failureModeDefinition` is a standard bounded response pattern [20]: a response should follow a request within a certain time. This is translated into the TCTL formula $\mathsf{AG}\,(\text{request} \rightarrow \mathsf{AF}_{<5s}\text{response})$. The negation of this formula, in turn, is translated into a simple automaton where the transition to the accepting state is labelled with the guard "$(x > 5)$?".

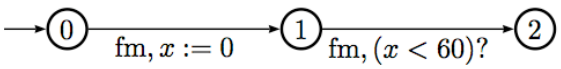

Figure 7: Timed automaton for the reliability clause.

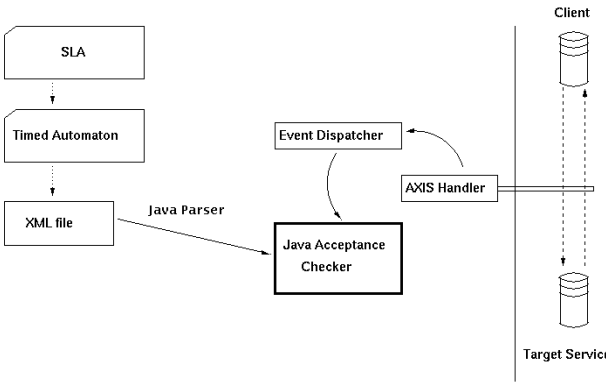

Figure 8: A possible implementation of an automata-based monitor.

The reliability requirement imposes that, in any given minute, a failure mode (denoted by the proposition `fm`) cannot occur more than 2 times. Thus, reliability (in fact, its negation) can be expressed using the automaton in Figure 7. Notice that this automaton accepts all the sequences of events in which two failure modes (`fm`) occur within 60 seconds: an execution trace satisfying this automaton would be a violation of the reliability requirement.

Overall, all the temporal properties that can be expressed in SLAng can also be encoded as TCTL formulae and, consequently, as timed automata.

Location of the monitors: the issue of mutual monitorability of SLAs has been addressed in [32], investigating, among other things, where monitors should be installed. In our example, the acceptance checker takes a timed automaton as input: it seems natural to install the monitors for failure modes and for reliability on the client, and on the server for `inputThroughputClause`.

The details of the implementation of this methodology are presented in the next section.

5. ON-LINE MONITORING METHOD

Our implementation of an automata-based online monitor for Web Services is presented in Figure 8. We use the Open Source Axis engine from Apache to process SOAP messages and use its handler mechanism for monitoring, as detailed below.

```
<?xml version='1.0' encoding='UTF-8'?>
<!ELEMENT ta (name,clocks,location+,init,end,transition+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT clocks (clock*)>
<!ELEMENT clock EMPTY>
<!ATTLIST clock id ID #REQUIRED>

<!ELEMENT location (comment?)>
<!ELEMENT comment (#PCDATA)>
<!ATTLIST location id ID #REQUIRED>
<!ELEMENT init EMPTY>
<!ATTLIST init ref IDREF #IMPLIED>
<!ELEMENT end EMPTY>
<!ATTLIST end ref IDREF #IMPLIED>

<!ELEMENT transition(source,target,label?,reset*,guard*)>

<!ELEMENT source EMPTY>
<!ATTLIST source ref IDREF #REQUIRED>
<!ELEMENT target EMPTY>
<!ATTLIST target ref IDREF #REQUIRED>
<!ELEMENT label (#PCDATA)>
<!ELEMENT reset EMPTY>
<!ATTLIST reset ref IDREF #REQUIRED>
<!ELEMENT guard (#PCDATA)>
```

Figure 9: DTD for timed automata in XML.

An SLA between two participants (top left) is first translated into a timed automaton following the procedure presented in Section 4. The automaton is represented as an XML file. We use a slightly modified (and simplified) version of the XML syntax for timed automata used in the model checker UPPAAL[1]. The DTD for our definition of timed automata is presented in Figure 9.

We have developed a parser to translate an XML definition of a timed automaton into an acceptance checker, implementing the method presented in Section 3. Essentially, the acceptance checker starts from the initial state of the automaton and accepts an action and a timestamp as input; based on the input, a transition may be performed to a "next" state, and clocks are updated accordingly. If a final (accepting) state is reached, then a violation has occurred and some form of action is taken. Currently, the checker writes violations to a file, but it could be easily modify to send emails to participants or to perform other kinds of actions. An example trace representing the evidence for the violation of an InputThroughputClause is shown in Figure 10 (time is in milliseconds since Jan 1st 1970). The violated InputThroughputClause prescribes that no more than two calls can be performed in a given second. This is translated into an automaton with four states, with a clock condition on the last transition prescribing that the time elapsed from the first transition has to be less than 1 second. The execution reported in Figure 10 is a violation because the service was called for the first time at T=*626, then a second time at T=828 (this was a valid call), and then a third time at T=*910, i.e. less than 0.3 seconds from the first call, and this violates the SLA.

The Java Acceptance Checker is a stand-alone application which receives events from an event dispatcher we have implemented. The event dispatcher is a Java class invoked

[1]See `http://www.cs.auc.dk/~behrmann/utap/libutap-0.90.tar.gz`.

```
ITC1: FIRST CALL, SETTING INIT. STATE = l0
        at time=1170166745626
ITC1: TRANSITION FOUND, GOING TO l1
        at time=1170166745626
ITC1: TRANSITION FOUND, GOING TO l2
        at time=1170166746828
ITC1: TRANSITION FOUND, GOING TO l3
        at time=1170166746910
ITC1: EXECUTION ACCEPTED: *** VIOLATION ***
        at time 1170166746910
```

Figure 10: Evidence trace for SLA violation.

by Axis handlers in the message chain of communication between a client and a target service. The handlers (and the event dispatcher with the acceptance checker) can be installed at the client side, at the server side (see previous section), or both. The handlers for the events dispatcher are injected both in the request and in the response flow. The event dispatcher performs a first selection of the messages to be processed, based on the name of the participants and the kind of events to be monitored (e.g., InputThroughputClause, Latency, etc.)

In summary, our SLA monitors is deployed as follows:

1. The SLA is expressed as a timed automaton using XML.
2. The timed automaton is parsed and a Java Acceptance Checker is generated.
3. The class file for the Acceptance Checker is deployed on the web server in conjunction with the Event Dispatcher.
4. Axis handlers are injected in the message chain of the service to be monitored

Notice that it is also possible to install an event dispatcher and an acceptance checker outside a server, using the system property `axis.ClientConfigFile` for Axis clients.

The code for the parser and for the event dispatcher, together with example files are available from the authors upon request.

6. EVALUATION

We evaluate the SLA monitoring method described in this paper using a grid computing case study. This appears appropriate as there are several grids that use web service technologies, and computational grids typically span across different organizations, which might have service level agreements with each other. Moreover, the computational load in a service-oriented grid may be very significant.

The particular grid application that we use to evaluate our approach is in the area of computational chemistry and described in detail in [16]. In this application, a client component is used to submit searches to a web service that is implemented as a BPEL workflow. The BPEL workflow then eventually calls a number of web services to submit

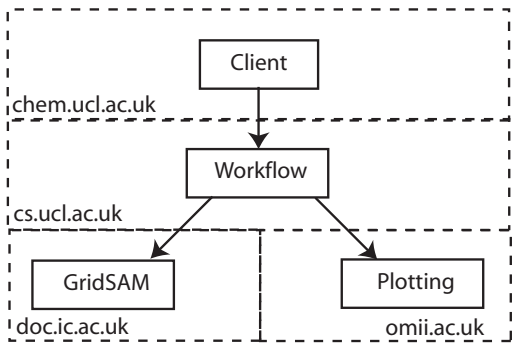

Figure 11: Web services in the Grid Case Study

different Fortran executables to compute resources, to visualize results and to upload the consolidated search result to a data portal. Figure 11 shows an overview of the different services involved and their deployment across different organizational domains.

The reason why SLAs need to be defined and monitored is because the service providers do not wish to be subjected to load that they could not bear, and clients require a certain level of service quality. To this end, the following Service Level Agreements can be put in place:

- Because a full search in this application takes about 8 hours an SLA between chem.ucl.ac.uk and cs.ucl.ac.uk would have a InputThroughputClause that limits the total number of searches for a given client to 3 per 24 hour period.
- Likewise the job submission service at doc.ic.ac.uk can be brought down by a client if jobs are constantly submitted and we effectively need to demand a InputThroughputClause of no more than two submissions per second for the service `GridSAM` and for the service `Plotting`.
- Moreover, a client is interested in latency of job submission and plotting services. For instance a client may require the latency for job submission is less than 1000 milliseconds.
- Finally, we are also interested reliability constraints and would not wish to see more than 1 failure in 10 invocations of either the job submission or the plotting web service.

Timed automata for these agreements can be defined using the XML syntax presented in the previous Section and similarly to the patterns presented in Section 4. Figure 12 presents the timed automaton corresponding to the InputThroughput clause of no more than two requests in a second (i.e., a third request would reach the accepting state; notice that time is given in milliseconds). The XML files for the remaining agreements are defined in a similar way and are available from the authors upon request.

To evaluate the run-time overhead of our approach, we have deployed two additional handlers in Axis to print a time

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE ta SYSTEM "ta-dtd.dtd">
<!-- violation if 3 or more request in 1s -->
<ta>
  <name>doc.ic.ac.uk</name>
  <clocks>
    <clock id="x" />
  </clocks>
  <location id="l0" />
  <location id="l1" />
  <location id="l2" />
  <location id="l3" />
  <init ref="l0" />
  <end ref="l3" />
  <transition>
    <source ref="l0" />
    <target ref="l1" />
    <label>request</label>
    <reset ref="x" />
  </transition>
  <transition>
    <source ref="l1" />
    <target ref="l2" />
    <label>request</label>
  </transition>
  <transition>
    <source ref="l2" />
    <target ref="l3" />
    <label>request</label>
    <guard>x &lt; 1000</guard>
  </transition>
</ta>
```

Figure 12: XML timed automaton for InputThroughputClause.

stamp to a log file before and after the invocation of the handler for monitoring SLAs: by taking the difference of the two timestamps we can evaluate the time overhead for the SLA monitor. The experiment was conducted on a grid of commodity Linux servers with hyper-threaded CPUs, 2GB of Memory and Tomcat 5.0 that hosts Axis 1.2.

We have installed monitors at the boundaries between cs.ucl.ac.uk and omii.ac.uk, and between cs.ucl.ac.uk and doc.ic.ac.uk (see Figure 11). At the first boundary we check an InputThroughputClause for the Plotting service, at the second boundary we check InputThroughputClause and Latency for the GridSAM service. The pattern of the remaining SLA for reliability do not differ from these and could be analysed in the same way.

We set an InputThroughputClause of no more than 2 requests per second for the Plotting Service and we created a simple Java client to invoke the service at random intervals between 0.3 and 3 seconds, in order to obtain some violations. All the violations where correctly identified and reported to the log file (see Figure 10 for an example of a report). Figure 13 reports the experimental results for 1 run: the elapsed processing time (in millisecond) is reported on the vertical axis, while the total duration of the experiment (30 seconds) is reported on the horizontal axis. As shown in the graph, the first call to the service took the longest time (4 millisecond), while the remaining calls took between 1 and 2 milliseconds for an average of 1.6 milliseconds. The violations occurring between seconds 10 and 15

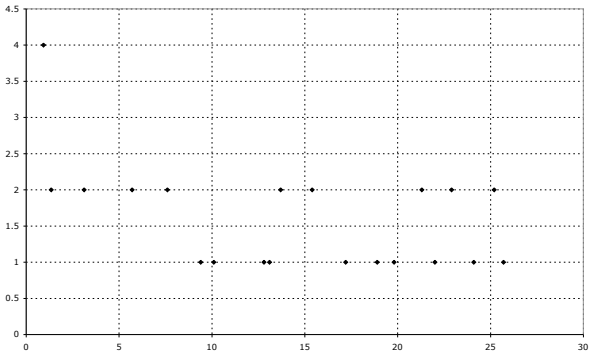

Figure 13: Time (in mS) for monitoring InputThroughputClause of the Plotting service.

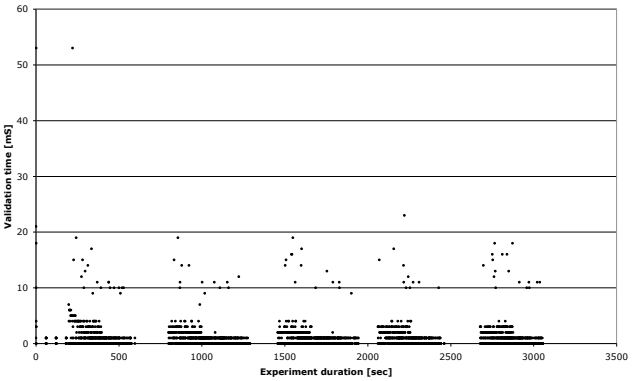

Figure 14: Overhead of validating latency and throughput

did not modify the overall time required.

We installed two similar monitors for the GridSAM service. The first monitor validates that the latency of messages from a client's point of view is less than 1000 milliseconds and the second monitor validates with an inputThroughput clause that the load a client puts on the job submission provider does not exceed two requests per second. For this experiment, we used the BPEL workflow on cs.ucl.ac.uk (see Figure 11) to generate around 2,220 invocations, each consisting of a request and a reply over a duration of 50 minutes. The invocations are divided into five batches of 100 job submissions. This can be seen from Figure 14 in that towards the end of each batch the measurement density is reduced.

Java can measure time in increments of one millisecond. In the data set observed in this experiment there are 2,718 data points (of the total 4,440) with a value of 0. This means that the actual time that was used for the validation of the latency and inputThroughput constraint was less than the measurement precision of Java for 61% of the total messages considered. The average time for the validation per message is 0.8 milliseconds. The standard deviation is 2.18. 93 data points were above 4 milliseconds with the largest one being under 60 milliseconds. The explanation for this distortion is that the time measurement of the validation overhead is not the only load on the machines, as they also perform the computational services of the experiment.

We consider these results extremely encouraging. We can measure the overhead of our monitoring approach in the percentage of time used for validation over the total duration of the experiment. The total time spent validating SLAs in this experiment was 3.514 seconds and with a total experiment duration of 3055 seconds, this gives an overhead of only 0.1%. We have been able to effectively monitor SLAs on a production environment, with an average of under 1 millisecond on commodity machines that were under significant computational load. In order to deploy the validation we did not have to modify any of the web services and could just configure the SOAP engine to add our handler.

7. DISCUSSION OF RELATED WORK

We build upon the work of Alur et al on TCTL and timed automata [2], as well as the implementation of Timed Automata in the Uppaal library [21]. TCTL and timed automata have been used for verifying timeliness constraints in specifications in, for example [6, 13, 1]. To the best of our knowledge, timed automata have not been used before at run-time for the monitoring of service quality.

SLAs are formal and precise statements of non-functional requirements. Thus our work is related to requirements monitoring that was first proposed by Fickas and Feather [18]. The implementation of this approach was demonstrated in [9], where Cohen et al use a "Formal Language for Expressing Assumptions", which in essence is a temporal logic. However, their implementation relies on triggers in the AP5 active database [8], which is written in LISP. We have used FLEA and the AP5 implementation for monitoring purposes in [17] and based on this experience are able to assert that the monitoring approach presented in this paper is both more lightweight and significantly more efficient.

Our approach presented in this paper has the same aims as that of Robinson [27], who argues on the importance of monitoring web service quality. Robinson proposes the use of temporal logic and KAOS to define timeliness constraints. Robinson does not indicate, however, how these temporal logic formulae can be monitored efficiently.

Baresi et al have proposed various techniques for monitoring BPEL web service compositions [3]. This work is related as they are able to monitor for timeouts, for external failures and for functional contracts. They propose two different approaches. Their first approach uses hand-coded monitors written in C# to process intercepted messages. Our approach simplifies the monitor construction considerably by deriving timed automata implementations that perform the monitoring automatically from a TCTL formula that is derived from a SLAng timeliness constraint. Their second approach uses our xlinkit rule engine [24]. It is aimed at monitoring functional contracts. Xlinkit executes first order logic rules but does not support temporal operators that

would be required to express timeliness constraints.

Mahbub and Spanoudakis proposed a framework for monitoring web service compositions in [22]. They use Event Calculus [28] to express temporal constraints for service executions. The approach relies on an interpretation of events from an event database that is fed from a BPEL engine. However, there are SLA constraints (such as the inputThroughput clauses we discussed above) that require knowledge about events that are not observable by a BPEL engine. Moreover, the paper makes no statements as to how efficient these monitors are. Our evaluation on the other hand has demonstrated by way of a scalable experiment that we can decide whether a constraint is violated within a few milliseconds.

Song Dong et al report on their use of timed automata and the Uppaal libraries for the verification of web service orchestrations in [13]. The main difference between their work and the approach we have presented in this paper is that they intend to analyze properties of orchestrations prior to deployment, while we are monitoring the timeliness constraints of web services and their orchestrations at run-time.

8. CONCLUSIONS AND FUTURE WORK

We have presented a methodology for online monitoring Service Level Agreements based on timed automata and Web Services. We have presented a Java implementation using Axis and Axis handlers.

Our methodology is non intrusive: there is no need to instrument existing services with new code. Instead, we inject handlers in the message chain and we reason on the kinds of messages exchanged (and their timestamp) to look for violations of the agreements.

The approach presented in this paper can be deployed quickly even without knowledge of the underlying application: the case study presented in Section 6 involves services operating on complex scientific data and workflows. Nevertheless, monitoring SLAs between participants only required the knowledge of the level of service required and the location of the services. We have been able to implement our solution in a production environment in less than a day.

Differently from previous approaches, the performance of our acceptance checker does not depend on the total number of messages in the system. Moreover, each SLA checker is very small (the .class file is typically less than 10Kb). By using an on-the-fly verification technique we have been able to handle a few hundreds of events per second nad obtain average verification times of a few milliseconds.

Our aim here was to provide an efficient methodology and to prove its feasibility, and thus various extensions remain to be investigated. For instance, as mentioned in Section 3, in the scope of this paper we considered only time attributes of events. Therefore, we considered only SLAs dealing with timeliness issues of services, but SLAs may also impose requirements on non-temporal properties. Additionally, in our implementation we did not consider scheduled SLAs, i.e., SLAs varying over time. For instance, one could think of SLAs changing with the day of the week, or with other contextual parameters. To this end, we are currently working with the industrial and academic partners of the European project PLASTIC (`http://www.ist-plastic.org`) and of the UK EPSRC project Divergent Grid to extend our methodology.

9. REFERENCES

[1] R. Alur, C. Courcoubetis, and D. Dill. Model checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[2] R. Alur and D. Dill. A theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] Luciano Baresi, Carlo Ghezzi, and Sam Guinea. Smart monitors for composed services. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 193–202, New York, NY, USA, 2004. ACM Press.

[4] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, W. Yi, and C. Weise. New generation of Uppaal. In *Proceedings of the International Workshop on Software Tools for Technology Transfer*, 1998.

[5] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of TACAS'99*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, 1999.

[6] V. Braberman, A. Olivero, and F. Schapachnik. Issues in Distributed Timed Model Checking. *Int. Journal on Software Tools for Technology Transfer*, 7(1):4–18, 2005.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[8] D. Cohen. Compiling complex database transition triggers. *SIGMOD Rec.*, 18(2):225–234, 1989.

[9] Don Cohen, Martin S. Feather, K. Narayanaswamy, and Stephen S. Fickas. Automatic monitoring of software requirements. In *Proceedings of the 19th international conference on Software engineering*, pages 602–603, New York, NY, USA, 1997. ACM Press.

[10] S. Damodaran. B2B Integration over the Internet with XML – RosettaNet Successes and Challenges. In *Proc. of the World-Wide-Web Conference, 2004*, pages 188–195, 2004.

[11] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III*, volume 1066 of *LNCS*, pages 208–219. Springer-Verlag, 1995.

[12] P. Dembiński, A. Janowska, P. Janowski, W. Penczek, A. Pólrola, M. Szreter, B. Woźna, and A. Zbrzezny. VerICS: A tool for verifying Timed Automata and Estelle specifications. In *Proc. of the 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 278–283. Springer-Verlag, 2003.

[13] J. S. Dong, Y. Liu, J. Sun, and X. Zhang. Verification of Computation Orchestration via Timed Automata. In Z. Liu and J. He, editors, *Proc. of the* 8^{th} *Int. Conference on Formal Engineering Methods*, volume 4260 of *Lecture Notes in Computer Science*, pages 226–245. Springer Verlag, 2006.

[14] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state

verification. In Mark Ardis, editor, *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, pages 7–15, New York, 1998. ACM Press.

[15] E. Allen Emerson, Aloysius K. Mok, A. Prasad Sistla, and Jai Srinivasan. Quantitative temporal reasoning. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 136–145, London, UK, 1991. Springer-Verlag.

[16] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. L. Price. Grid Service Orchestration using the Business Process Execution Language (BPEL). *Journal of Grid Computing*, 3(3-4):283–304, 2005.

[17] W. Emmerich, A. Finkelstein, C. Montangero, S. Antonelli, S. Armitage, and R. Stevens. Managing Standards Compliance. *IEEE Transactions on Software Engineering*, 25(6):836–851, 1999.

[18] S. Fickas and M. Feather. Requirements Monitoring in Dynamic Environments. In *Proc. of the* 2^{nd} *IEEE Int. Symposium on Requirements Engineering, York*, pages 140–147. IEEE Computer Society Press, 1995.

[19] G. J. Holzmann. The model checker SPIN. *IEEE transaction on software engineering*, 23(5):279–295, 1997.

[20] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 372–381, 2005.

[21] K. G. Larson, P. Pettersson, and W. Yi. Uppaal in a nutshell. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 1(1-2):134–152, 1997.

[22] K. Mahbub and G. Spanoudakis. A framework for requirents monitoring of service based systems. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 84–93, New York, NY, USA, 2004. ACM Press.

[23] Nicolas Markey and Jean-François Raskin. Model checking restricted sets of timed paths. *Theor. Comput. Sci.*, 358(2):273–292, 2006.

[24] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151–185, 2002.

[25] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 377–390. Springer-Verlag, 1994.

[26] P. Pettersson and K. G. Larsen. Uppaal2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, February 2000.

[27] W. N. Robinson. Monitoring Web Service Requirements. In *RE '03: Proceedings of the 11th IEEE International Conference on Requirements Engineering*, page 65, Washington, DC, USA, 2003. IEEE Computer Society.

[28] M. Shanahan. The Event Calculus explained. In *Artificial Intelligence Today*, volume 1600 of *Lecture Notes in Computer Science*, pages 409–430. Springer Verlag, 1999.

[29] J. Skene. *The SLAng SLA Language*. UCL, http://uclslang.sourceforge.net, version 1.1 edition, 2006.

[30] J. Skene and W. Emmerich. Engineering Runtime Requirements-Monitoring Systems using MDA Technologies. In *IFIP Symposium on Trustworthy Global Computing*, volume 3705 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2005.

[31] J. Skene, D. Lamanna, and W. Emmerich. Precise Service Level Agreements. In *Proc. of the* 26^{th} *Int. Conference on Software Engineering, Edinburgh, UK*, pages 179–188. IEEE Computer Society Press, May 2004.

[32] J. Skene, A. Skene, J. Crampton, and W. Emmerich. The Monitorability of Service-Level Agreements for Application-Service Provision. In *Proc. of the* 6^{th} *Int. Workshop on Software and Performance (WOSP), Buenos Aires, Argentina*. ACM Press, February 2007. To appear.

[33] M. W. Whalen, A. Rajan, M. P. E. Heimdahl, and S. P. Miller. Coverage metrics for requirements-based testing. In *ISSTA'06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 25–36, New York, NY, USA, 2006. ACM Press.