Efficient Online Monitoring of Web-Service SLAs

Franco Raimondi
Department of Computer Science, UCL
Gower Street
London, UK
f.raimondi@cs.ucl.ac.uk

James Skene
Department of Computer Science, UCL
Gower Street
London, UK
j.skene@cs.ucl.ac.uk

Wolfgang Emmerich
Department of Computer Science, UCL
Gower Street
London, UK
we@acm.org

ABSTRACT

If an organization depends on the service quality provided by another organization it often enters into a bilateral *service level agreement* (SLA), which mitigates outsourcing risks by associating penalty payments with poor service quality. Once these agreements are entered into, it becomes necessary to monitor their conditions, which will commonly relate to timeliness, reliability and request throughput, at runtime. We show how these conditions can be translated into timed automata. Acceptance of a timed word by a timed automaton can be decided in quadratic time and because the timed automata can operate while messages are exchanged at runtime there is effectively only a linear run-time overhead. We present an implementation to derive on-line monitors for web services automatically from SLAs using an Eclipse plugin. We evaluate the efficiency and scalability of this approach using a large-scale case study in a service-oriented computational grid.

Categories and Subject Descriptors

D.2.5 [**Testing and debugging**]: Monitors

General Terms

Services, Service Level Agreements, On-line monitoring

1. INTRODUCTION

There is a growing trend for IT systems to be integrated across organizational boundaries. Examples include supply-chain management using RosettaNet [9], or the Amazon Storefront web services that enable small retailers to establish and operate a web store-front on Amazon's infrastructure. These services are frequently implemented using web-service 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 will not abuse the service. These quality and usage constraints are often defined in *Service Level Agreements* (SLAs), which specify required Quality of Service (QoS) and associate penalty payments with violations. These penalty payments can be seen as an insurance policy against poor service provision and over-use.

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. 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. The service provider might also monitor whether the service quality drops below certain thresholds in order to determine whether to increase the resource provision in the hardware infrastructure or to reconfigure the software architecture that provides the service, as discussed by [19]. Service monitoring data will also inform the service provider whether there is sufficient capacity to increase service provision to new or existing clients.

In [29] 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 can be monitored in principle. By carefully considering how a service may affect a client, we have also determined what conditions relating to QoS should reasonably be included in an electronic-service SLA. We have implemented support for these conditions in our SLA language, SLAng [26], which is precisely defined using standard meta-modelling languages [28, 27]. We now describe a method for online monitoring of timeliness conditions that can be described in SLAng. We distinguish online from offline monitoring. Offline monitoring uses sensors that are woven into the service provisioning infrastructure to collect data about the service delivery. Those data may then be analyzed at certain intervals and mined for service quality violations to determine whether penalty payments are due. This is contrasted with online monitoring where the data about service delivery is analyzed while the service is being provided and alerts are generated as soon as a service-quality violation is detected.

SIGSOFT 2008/FSE-16, November 9–15, Atlanta, Georgia, USA

Off-line monitoring has a number of disadvantages. Firstly it requires the storage of possibly large volumes of data about the service provision. Secondly, for triggering software architecture reconfigurations or resource allocation decisions in data centres it is necessary to know about service quality violations as soon as they happen. We therefore focus on efficient techniques for online monitoring in this paper.

The contributions of this paper are twofold: firstly, we describe how timed automata [3] can be used as a formalism to support efficient online monitoring of timeliness, reliability and throughput constraints expressed in web-service SLAs. A review of related work indicates the novelty of this approach, and a theoretical analysis reveals its significance: in particular, we are able to reduce the question of whether an SLA is violated to the acceptance of a timed word by a timed automaton. We show that this is decidable in quadratic and because we can process each timed letter on the fly, the approach introduces only a linear run-time overhead for a web service invocation. Secondly, we present an implementation of this technique using an Eclipse plug-in and Apache AXIS handlers. We have evaluated the performance overhead and scalability of this implementation using a case study that monitors SLAs in a production grid system.

This paper is further structured as follows. Section 2 reviews the theoretical analysis of SLAs presented in [28, 26], and provides an overview of Timed Automata. Section 3 describes a set of SLA temporal patterns and their translation into Timed Automata. A methodology for monitoring these patterns and the analysis of its complexity are presented in Section 4; its implementation is described in Section 5. An example application and experimental evaluation results are reported in Section 6. Section 7 discusses related work, and concluding considerations are presented in Section 8.

2. BACKGROUND

We first review the analysis of SLAs monitorability presented in [29], and the discussion of requirements for electronic service SLAs given in [16]. Under a specified set of assumptions, these analyses identify both the kinds of conditions that are likely to be required in SLAs, and the locations in the service provisioning infrastructure in which monitoring will be required to determine compliance with the SLAs.

We then summarise the theory of timed automata, which in the remainder of the paper we show to be capable of efficiently checking compliance for the kinds of conditions identified in the foregoing analysis.

2.1 Service Level Agreements for Electronic Services

In [16] we have proposed that financial risks are a major disincentive to organisations wishing to outsource electronic services. The risks arise from the possibility that the QoS of a provided service will be lower than expected, resulting in inefficiencies that impact upon the ability of the client to capitalise on their investment in the service. Also, if the provider prematurely ceases to provide a service, then the client will lose the opportunity to recuperate initial integration costs. We argue that SLAs that associate financial penalties with poor QoS and early termination are a promising complementary technology to existing middleware technologies, having the potential to broaden the market in services by mitigating these risks.

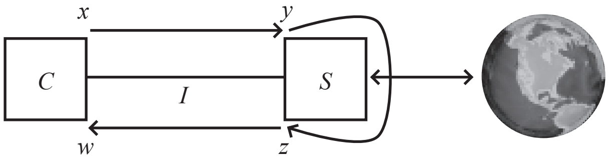


Figure 1: The three party scenario.

SLAs, in the role that we propose, are instruments by which one party may become liable to pay a penalty to another. This has two significant implications: parties will have an incentive to cheat; and parties may be able to force other parties (which are acting in good faith) to become liable to pay penalties. Clearly, for SLAs to be an attractive mechanism for mitigating risks, both of these possibilities must be avoided to the maximum extent possible, and to some degree this can be achieved in the design of the SLAs themselves. We refer to the contribution that an SLA makes to ensuring that any disagreement concerning the SLA will be resolved according to the original intent of the SLA as the *protectability* of the SLA. SLAs should also not be *exploitable*, meaning that one party cannot by their actions cause another party to have to pay a penalty unless the second party is in breach of their obligations with respect to the agreement.

The resolution of a dispute concerning an SLA will require the parties to first agree an account of their behaviour and the behaviour of the relevant service. Then, the intent of the SLA will need to be recovered from a concrete document of the agreement, and applied in the context of the SLA. We therefore argue that to be protectable, an SLA should be *monitorable*, so that the parties can obtain trustworthy information concerning events of relevance to the agreement, *understandable*, so that the intent of the SLA can be recovered, and *precise*, so that the intent is unambiguous.

In [27] we have shown how meta-modelling techniques using standards of the Object-Management Group can be used to specify an SLA language such that the SLAs are precise and understandable, as exemplified by our language SLAng (see `http://uclslang.sourceforge.net`). Ensuring such qualities is one contribution that a formal language can make to the statements it defines. Another is to render the statements machine processable, a common requirement for electronic-service SLAs, the parameters of which may be relevant to the configuration of a service. However, perhaps more importantly, the language definition can reduce the cost of preparation of statements by contributing semantics that anticipate what will need to be expressed. In the case of SLAng, this was informed by considering what electronic-service behaviours would be of concern to a client, what events are monitorable, and therefore what conditions may reasonably be included in an electronic-service SLA.

Consider the scenario 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 may also interact with the world externally to the network, for example, to cause some goods to be dispatched from a warehouse.

We assume that the client should only be concerned with the behaviour of the service and the network in so far as it affects the client, the details of the implementation of the service and the network being the exclusive concern of the providers. It is therefore possible to argue that the events that affect the client are the receipt of responses from the network (in response to requests that the client has previously dispatched), and any behaviour external to the network, instigated by the service provider, that may come to affect the client, for example the delivery of some goods to the client's premises.

Communications originating from a service have two main attributes with which the client may be concerned: what is returned; and when it arrives. Conditions related to the interval between a service request and the time of arrival of a correlated response are variously referred to as performance, latency or *timeliness* conditions. A number of well established techniques are available for testing *functional* requirements of services: in this paper we are only concerned with non-functional timeliness constraints.

Because the client has no access to the implementation of the service, its expectations concerning the behaviour of the service will depend on a description of the service given to them by, or negotiated with, the service provider. Before entering into an SLA the client will make the choice to integrate the service into its own operations on the basis of this description. If the service subsequently behaves in a manner other than that described, the client is likely to suffer. Hence, a condition that the client will want to protect in an SLA is that the service either behaves as described to a high degree or the client will be entitled to receive compensation. Such conditions are normally called *reliability* conditions.

Communications via electronic services have no other attributes, so we conclude that the client will be primarily concerned with timeliness and reliability conditions relating to these services, and with conditions relating to the behaviour of the service external to the system.

Our scenario has three participating parties. With which parties should C make an SLA in order to be compensated for violated timeliness and reliability conditions, as measured at their interface? Also, there are clearly two different types of service being provided: an electronic service, defined in terms of the exchange of meaningful messages across interfaces; and the network service, defined in terms of the movement of data packets between nodes. Do SLAs for these services need to be defined separately? We answered these questions for this scenario in [29], by considering the monitorability of various systems of SLAs for guaranteeing timeliness properties.

Four events are labelled in Figure 1, x, y, z and w. Assuming the client has a timeliness requirement of the form $w - z < t$ (i.e., the temporal distance between event w and event z is less than t), this can be insured by some combinations of several SLAs for the conditions $w - z < t$, $y - x < t_1$, $z - y < t_2$, $w - z < t_3$, $z - x < t_1 + t_2$, $w - y < t_3 + t_2$, such that $t_1 + t_2 + t_3 < t$. Assuming any party may offer an SLA to any other for any requirement, there are $2^{3\times2\times6} = 2^{36} \sim 6.9 \times 10^{10}$ possible systems of SLAs in this scenario, which is obviously a large number to consider.

It is clear that C can directly monitor x and w, S can directly monitor y and z, and I can monitor all events. Informally, in our model we allow one party to report on an event to another only if they do not enter into an SLA in respect of a requirement concerning that event. We say that an event is monitorable by a party if they can directly monitor it or have it reported to them. An SLA is monitorable by a party if all events that its requirements concern are monitorable by the party. We are interested in systems of SLAs in which all SLAs are mutually monitorable both by the provider and the client of the SLA.

We classify SLAs as *safe* or *unsafe*, according to whether the provider can through their actions guarantee that the requirement being offered will be matched, or if the provider receives SLAs from other parties to this effect. In this scenario I can guarantee $y - x < t_1$ and $w - z < t_3$. S can guarantee $z - y < t_2$. Clearly, safe systems of SLAs are desirable as parties will not risk paying penalties unless they substantially control the relevant events, or will receive penalty payments themselves in the event of a violation. We also wish to avoid systems of SLAs in which reciprocal or redundant guarantees are offered pointlessly.

We implemented a depth-first search algorithm to efficiently discover systems of SLAs meeting these criteria for our scenario. The result was significant. Only one system of SLAs is possible in which the client's requirement is satisfied, and all SLAs are both mutually-monitorable and safe: I insures $w - x < t$ for C, and S insures $z - y < t_2$ for I. We also found that reliability could be insured in the same system of SLAs.

This result tells us that we do not need SLAs at the network level in order to insure end-to-end performance for the service provider. Both conditions in the system relate requests to responses at a service interface, and the end-to-end behaviour of the network is implicit in the guarantee that I makes to C. The events pertinent to each SLA occur at a single interface with the network, so these locations are where monitoring must be performed.

In fact, reliability and timeliness conditions are not the only necessary conditions in this scenario. The ability of a client to create requests will tend to exceed the ability of the service provider to produce correct, timely results. Therefore, without the inclusion of a throughput constraint in these SLAs, it would be possible for the client to attempt to exploit these SLAs. We therefore argue that the conditions that will tend to be required in electronic-service SLAs, are timeliness, reliability and throughput conditions.

2.2 Timed automata

An 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.

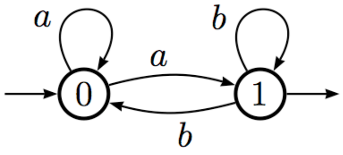


Figure 2: An automaton.

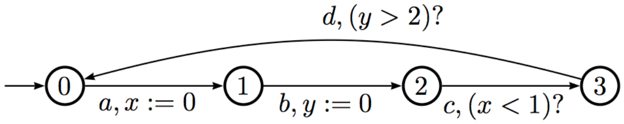


Figure 3: A timed automaton.

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 [3] extend 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*; the timed automaton in Figure 3 recognises the language $\mathcal{L}(TA) = \{((abcd)^\omega, \tau) | (\tau_{4j+3} < \tau_{4j+1} + 1) \wedge (\tau_{4j+4} > \tau_{4j+2} + 2)\}$

A key idea of this paper is to encode the specification patterns for Service Level Agreements defined in the next section 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 the following sections.

3. APPROACH

3.1 Patterns of SLA timeliness constraints

Specification patterns are defined in [11] as the "description of a commonly occurring requirement"; for instance, safety and liveness can be considered specification patterns. In this section we identify a set of specification patterns for timeliness constraints of web services and we show how timed automata characterising violations can be derived from these patterns. Generic-purpose specification patterns are analysed in [11], and an extension to patterns involving time intervals is presented in [17].

We have analysed the requirements for the services being developed within the European project PLASTIC (Providing Lightweight and Adaptable Service Technology for pervasive Information and Communication) [22]. These services are typically developed as web services and we were able to identify the following recurring patterns, which can be encoded in SLAng:

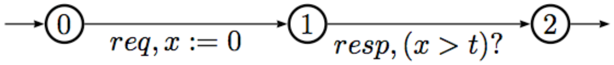


Figure 4: Timed automaton for latency violations.

1. **Latency requirements**. A number of requirements for PLASTIC web services have the form "the response of the service must follow the request within t seconds". Examples include temporal bounds on the notification services of the middleware and various constraints on the cooperation of services (such as discovery, composition, etc.). These latency requirements are instances of the *bounded response* pattern described in [11], with the addition of a temporal constraint as described in [17].

2. **Reliability requirements**. Another set of requirements for services includes constraints on the number of acceptable errors. Errors are defined as violations of the latency requirements or as other kinds of timeouts. Typically, it is required that the number of errors in a given time window does not exceed a fixed amount. This kind of pattern ("counting" the number of certain events in a time window) does not appear neither in [11] nor in [17]. We show below how this pattern can be encoded as a timed automaton.

3. **Throughput requirements**. A third kind of requirement appears often in the definition of the interaction between a service consumer and a service provider: the provider imposes restrictions on the number of requests that a client is allowed to submit in a given time window. This requirement is imposed to avoid the excessive usage of a service by a client, which might cause a degradation on the quality of service provided and, consequently, a violation of the latency and reliability requirements. Similarly to the previous point, this is a pattern involving "counting" events and we encode it as a timed automaton below.

3.2 Violations and Timed Automata

Each of the patterns presented above can be translated into a timed automaton, such that the language accepted by the automaton characterises exactly the *violations* of the specifications.

Figure 4 depicts in more detail an automaton accepting all the timed words in which a label *req* (encoding a request) is followed by a label *resp* after more than t time units. This automaton encodes violations for the latency requirements described in item (1) above. Notice that, if requests and responses are interleaved, it is necessary to match the appropriate pair to detect a violation. One possibility is the use of different labels; a more efficient approach using AXIS handlers is presented in Section 5.

Figure 5 represents an automaton accepting continuously words with three failure occurrences (labelled with $fail$) within t time units. As mentioned above, failures can be identified by latency violations or other kinds of timeouts.

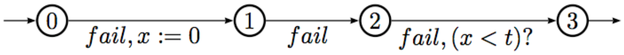


Figure 5: Timed automaton for reliability violations.

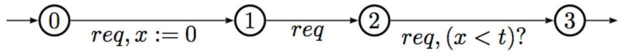


Figure 6: Timed automaton for throughput violations.

The automaton in this figure corresponds to an instance of the Reliability pattern described in item (2) above; notice that the number of states of the automaton is equal to the number of events to be counted + 1.

Figure 6 represents an automaton accepting all the time words in which three request (labelled with *req*) occur within t time units. The timed words accepted by this automaton correspond to violations of the throughput requirements presented in item (3) above. Notice that this automaton is structurally similar to the one encoding violations of reliability.

As mentioned in Section 2.1, timeliness, reliability, and throughput are the conditions that tend to be required in electronic-service SLAs. In this section we have presented how (the violation of) these conditions can be encoded as timed automata. Other patterns are possible and our approach does not preclude their representation in SLAng and their translation into timed automata.

4. DETECTING VIOLATIONS

The sequence of events occurring in a client-server interaction, with their time of occurrence, can be seen as a timed word in the sense of Section 2.2. This observation is key for the detection of violations of specifications. Indeed, if any timed sub-word of the timed word encoding the events is accepted by any of the automata encoding violations of the requirements, then a violation has occurred.

Reducing the problem of detecting violations to the problem of verifying that a word is accepted by an automaton has the additional advantage that verification can be performed on-line while events are occurring. Specifically, suppose that an automaton A encodes the violations of a requirement. Then, to identify a violation it is sufficient to pass all the occurring events (encoded by an increasing timed word (σ, τ)) to the automaton and verify if a final (accepting) state is reached. In this way, the automaton "evolves" in parallel with the execution of the services. If, in any given state of the automaton, an event is passed such that no transition exists, then the automaton rests to its initial state. In this case, however, it is not possible to discard all the events that led to the rejection.

As an example, consider a requirement prescribing that no more than 2 requests can be submitted in a given minute (this is a throughput requirement encoded by an automaton similar to the one in Figure 6). Suppose that the following sequence of events is observed: request at $t = 0$, request at $t = 0.9$, request at $t = 1.1$, request at $= 1.2$. This sequence of events is a violation of the requirement, because three requests occur between $t = 0.9$ and $t = 1.2$. However, if these events were passed to the automaton in Figure 6, the automaton would reset to its initial state at $t = 1.1$, reset its clock to $x = 0$, and it would not detect the violation at $t = 1.2$. Therefore, when a state without successor is reached, it is not possible to discard all the previous history. Instead, the automaton should be run again by discarding the first state (at $t = 0$): in this case the new event would fire a violation by reaching the final state.

The maximum number of events that need to be observed to detect a violation can be estimated easily for the patterns presented in Section 3.1. Essentially, the number of events required to detect a violation at any given time is equal to the number of states in the automaton minus 1. Following [5], we call this number the *diameter* of the witnesses for a violation. The diameter of witnesses is of interest for two main reasons.

Firstly, 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 useful to keep evidence of the possible violations. However, notice that in doing so, a client can present evidence for the violations made by a server, but the client could not present evidence if the server was to make claims about over-usage of a service: indeed, if this was the case, then the client should have kept the whole log of events, both for violations and for correct behaviour. Nevertheless, knowing the diameter of witnesses allows an efficient management of logs for failures, by keeping only the relevant data for violations.

Secondly, the diameter of counterexamples is used in the evaluation of the worst case complexity of the approach:

THEOREM 1. *On-line monitoring for the patterns of Section 3.1 has a worst case complexity $O(n^2)$, where n is the number of states of the automaton.*

PROOF. Let A be an automaton and (σ, τ) a finite timed word, where $\sigma = \sigma_1 \dots \sigma_k$ and $\tau = \tau_1 \dots \tau_k$. As mentioned above, the pair (σ_i, τ_i), consisting of a letter and a time, identifies an event and its time of occurrence. The problem of on-line monitoring is establishing whether or not a new event $(\sigma_{k+1}, \tau_{k+1})$ causes a transition of A (possibly to the final state of A), assuming that (σ, τ) was a valid sequence of events for A. The complexity of establishing whether or not $(\sigma_{k+1}, \tau_{k+1})$ causes a valid transition is linear in the size of A: indeed, it is sufficient to check the transition relation of A and verify that $(\sigma_k, \tau_k) \rightarrow (\sigma_{k+1}, \tau_{k+1})$ is permitted (in fact, checking that a word is accepted by a timed automaton is linear, see [3]). As mentioned above, the maximum diameter of a witness for a violation is equal to the number of states of an automaton, which is bounded by n. Therefore, the verification that $(\sigma_k, \tau_k) \rightarrow (\sigma_{k+1}, \tau_{k+1})$ is a valid transition has to be repeated at most n times. □

Thus, our approach has a quadratic complexity. Moreover, the results obtained in Section 6 show that it is often not necessary to perform n^2 operations for the patterns considered here because the number of transitions that lead from a state is very small compared to the total number of states.

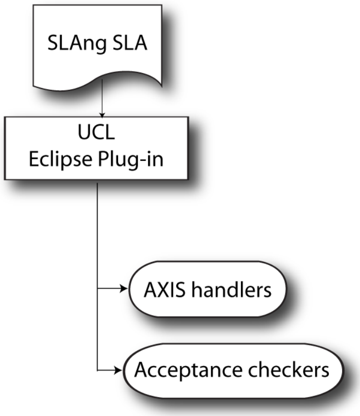


Figure 7: Eclipse plug-in for SLAng and monitors.

5. IMPLEMENTATION

We have implemented an Eclipse plug-in to create, edit, and check the correctness of SLAs written in SLAng. This plug-in is available from `http://uclslang.sourceforge.net/`

For the purposes of this work (see Figure 7), we have created a new plug-in which works in parallel with the editor for SLAng. This plug-in can generate *automatically* the automata encoding violations of SLAng terms for latency, reliability, and throughput. The automata are stand-alone Java checkers for the patterns presented in Section 3.1, implementing the ideas of Section 4. Additionally, our tool produces a Java handler which is used to analyse and dispatch messages to the checkers: in our framework we use the open source AXIS engine from Apache to process SOAP messages and use its handler mechanism to invoke the automatically generated checkers. A sample screenshot of the plug-in is shown in Figure 8. A preliminary version of the plug-in is available from
`http://www.cs.ucl.ac.uk/staff/f.raimondi/uk.ac.ucl.cs.slangta_0.0.1.zip`

The automatically generated Java code can be compiled and installed both at the client and at the server side. In particular:

- Handlers are probes in the SOAP message chain. In our implementation these handlers are AXIS handlers; on the server side, they are installed in a standard way as handlers in the message chains for requests and responses for the service to be monitored (notice that this approach does not require any interaction with the existing deployment of the service). Handlers can also be installed outside of an AXIS server, by using the system property `axis.ClientConfigFile` for AXIS clients.

- Checkers are added at the client or server side. On the server side, a checker is installed for every client that should be monitored. A checker is a simple Java class invoked by the handler; the checker implements the verification methodology presented in Section 4.

Figure 9 depicts the structure of the deployed monitors.

As mentioned above, the Java code for handlers and checkers is generated automatically. Theorem 1 gives a quadratic upper bound for the complexity of the verification step: in our implementation (see the pseudo-code in Figure 10, listing the transition step for the automaton) we are able to achieve in most of the cases a linear-only overhead by observing that, if a violation occurs, then the violation is very likely to be caused by the *last* observed event, and so the automaton's evolution should start from the event that occurred $n-1$ steps before the current one, where n is the number of states of the automaton. In the code of Figure 10, `states` is a list of events occurred (each event correspond to a state of the automaton) and `MakeTrans` is a function to generate transition steps in the checker. If the value returned by this function is 1, then a final (accepting) state has been reached, corresponding to a violation of the constraint to be checked.

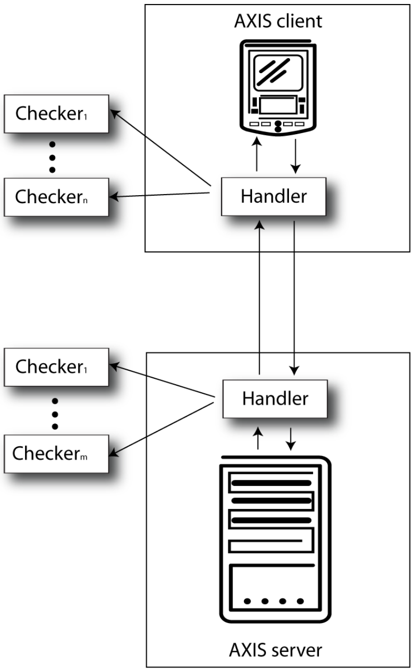


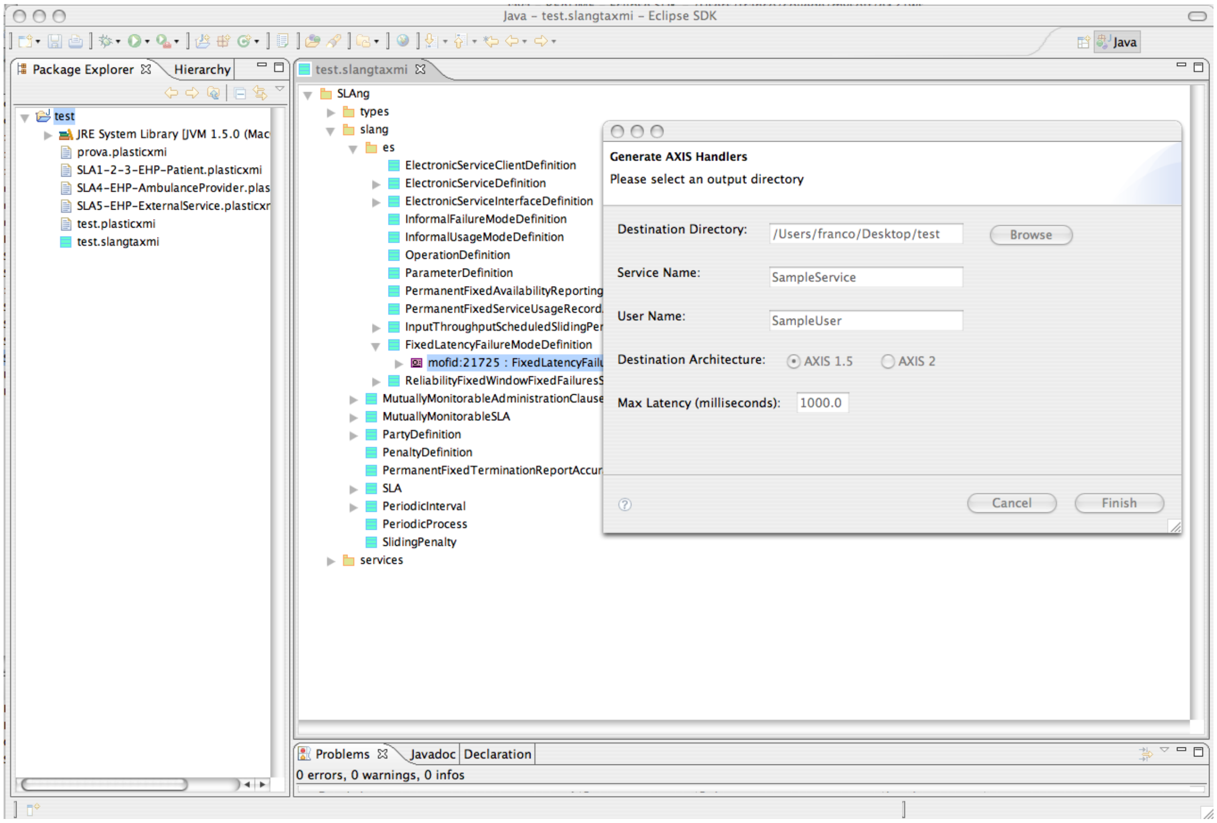
Figure 9: Deployment of the monitors

A further improvement in performance can be achieved when verifying latency: instead of storing a table of message IDs with the time of request, we use SOAP attachments to add a timestamp to the messages which should be checked against latency. This is done by the automatically generated handler, using a code similar to the one reported in Figure 11, where "msg" is the actual SOAP message obtained from the handler.

The timestamp can be read from the message context returned by AXIS with the response message. Thus, the check for latency can be performed by using the information found in the response message context only.

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


Java – test.slangtaxmi – Eclipse SDK
Package Explorer
Hierarchy
test
JRE System Library [JVM 1.5.0 (Mac
prova.plasticxmi
SLA1-2-3-EHP-Patient.plasticxmi
SLA4-EHP-AmbulanceProvider.plas
SLA5-EHP-ExternalService.plasticxr
test.plasticxmi
test.slangtaxmi
test.slangtaxmi
SLAng
types
slang
es
ElectronicServiceClientDefinition
ElectronicServiceDefinition
ElectronicServiceInterfaceDefinition
InformalFailureModeDefinition
InformalUsageModeDefinition
OperationDefinition
ParameterDefinition
PermanentFixedAvailabilityReporting
PermanentFixedServiceUsageRecord
InputThroughputScheduledSlidingPer
FixedLatencyFailureModeDefinition
mofid:21725 : FixedLatencyFailu
ReliabilityFixedWindowFixedFailuresS
MutuallyMonitorableAdministrationClause
MutuallyMonitorableSLA
PartyDefinition
PenaltyDefinition
PermanentFixedTerminationReportAccur
SLA
PeriodicInterval
PeriodicProcess
SlidingPenalty
services
Generate AXIS Handlers
Please select an output directory
Browse
Cancel
Finish
Problems
Javadoc
Declaration
Java

Destination Directory: /Users/franco/Desktop/test

Service Name: SampleService

User Name: SampleUser

Destination Architecture: ● AXIS 1.5 ○ AXIS 2

Max Latency (milliseconds): 1000.0

0 errors, 0 warnings, 0 infos

Figure 8: Screenshot of the Eclipse plug-in for the generation of monitors.

```
// Every time the handler passes a new event to
// the checker, the following code is executed (if
// more than n events have occurred).

// The initial place is the current index minus n,
// modulo n (n = num. of states)

initplace = (cur_index − n ) % n;

// Loops over the past recorded n states
for (int i=0; i < n−1; i++ ) {

    // The current i−th place
    int cur_p = (myplace+i) % (numStates−1);

    // make a transition with the state at place
    // cur_p
    int res = MakeTrans(states.elementAt(cur_p));
    if ( res == 1 ) {
        // a violation occurred, do something
    }
}
// If no violation occurred, remove the oldest
// state and update appropriately.
```

Figure 10: Pseudo-code for the verification step.

```
// message is obtained from the message Contex
AttachmentPart tStamp=msg.createAttachmentPart();
String contentString1= SomeDateString";
tStamp.setContent(contentString1,␣"text/plain");
msg.addAttachmentPart(tStamp);
```

Figure 11: Attaching timestamp to SOAP message.

in a service-oriented grid is often 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 [12]. 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 different Fortran executables to compute resources, to visualize results and to upload the consolidated search result to a data portal. Figure 12 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 throughput clause 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 throughput clause of no more than two submissions per second for the service `GridSAM` [18] and for the service `Plotting`.
- Moreover, a client is interested in latency of job submission and plotting services. For instance a client

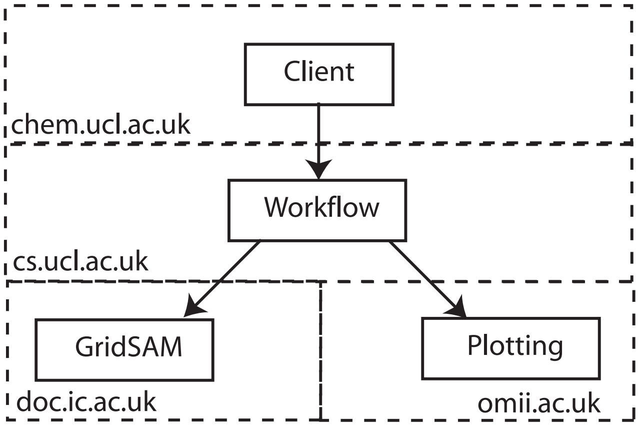


Figure 12: Web services in the Grid Case Study

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 no more than 1 failure in 10 invocations of the job submission service (because of GridSAM fault tolerance mechanisms), and no more than one failure in 1000 invocations of the plotting service.

We have expressed these SLAs using the editor for SLAng, and we have generated Java checkers using our tool presented in the previous section.

To evaluate the run-time overhead of our approach, we have modified the handlers in Axis to print a time 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 12). At the first boundary we check a throughput clause for the Plotting service, at the second boundary we check the throughput clause and latency clause 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 a throughput clause of no more than two 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 were correctly identified and reported. 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 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

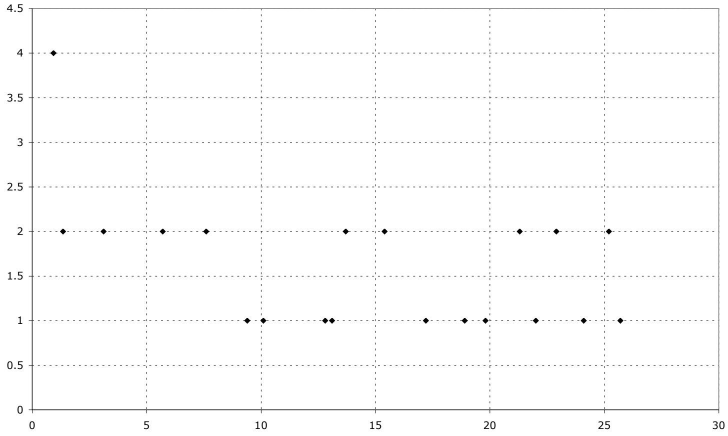


Figure 13: Time (in mS) for monitoring the throughput clause of the Plotting service.

from a client's point of view is less than 1000 milliseconds and the second monitor validates with a throughput clause that the load a client puts on the job submission provider does not exceed two requests per second. For this experiment, we monitored the BPEL workflows on cs.ucl.ac.uk (see Figure 12) generated by clients during 4 hours of activity. There were approximately 16 batches of jobs (i.e., 16 workflows were invoked), generating a total of nearly 230,000 SOAP messages: this can be seen from Figure 14, which reports the traffic in terms of the number of messages analysed by the handler per second from the beginning of the observation (T = 0) till the end (T=14,500, in seconds.)

We have measured the time elapsed for each handler invocation (in milliseconds). In the data set observed in this experiment there are 166,900 data points (of the total 229,000) with a value of 0. This means that the actual time that was used for the validation of the latency and throughput constraint was less than the measurement precision of 1 millisecond for 72% of the total messages considered. The average time for the validation per message is 0.4 milliseconds. The standard deviation is 4 milliseconds. Around 300 data points were above 5 milliseconds with the largest one being under 500 milliseconds. The explanation for this distortion (and thus the large standard deviation) 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. Figure 15 reports the distribution of the execution times for the handler (notice the logarithmic scale). 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 87.7 seconds and with a total experiment duration of 14,525 seconds, this gives an overhead of under 0.6%: indeed, the presence of the monitors was not noticed by the users of these services. Thus, we have been able to effectively monitor SLAs on a production environment, with an average of under 1 millisecond per service invocation 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 automatically generated handlers and monitors.

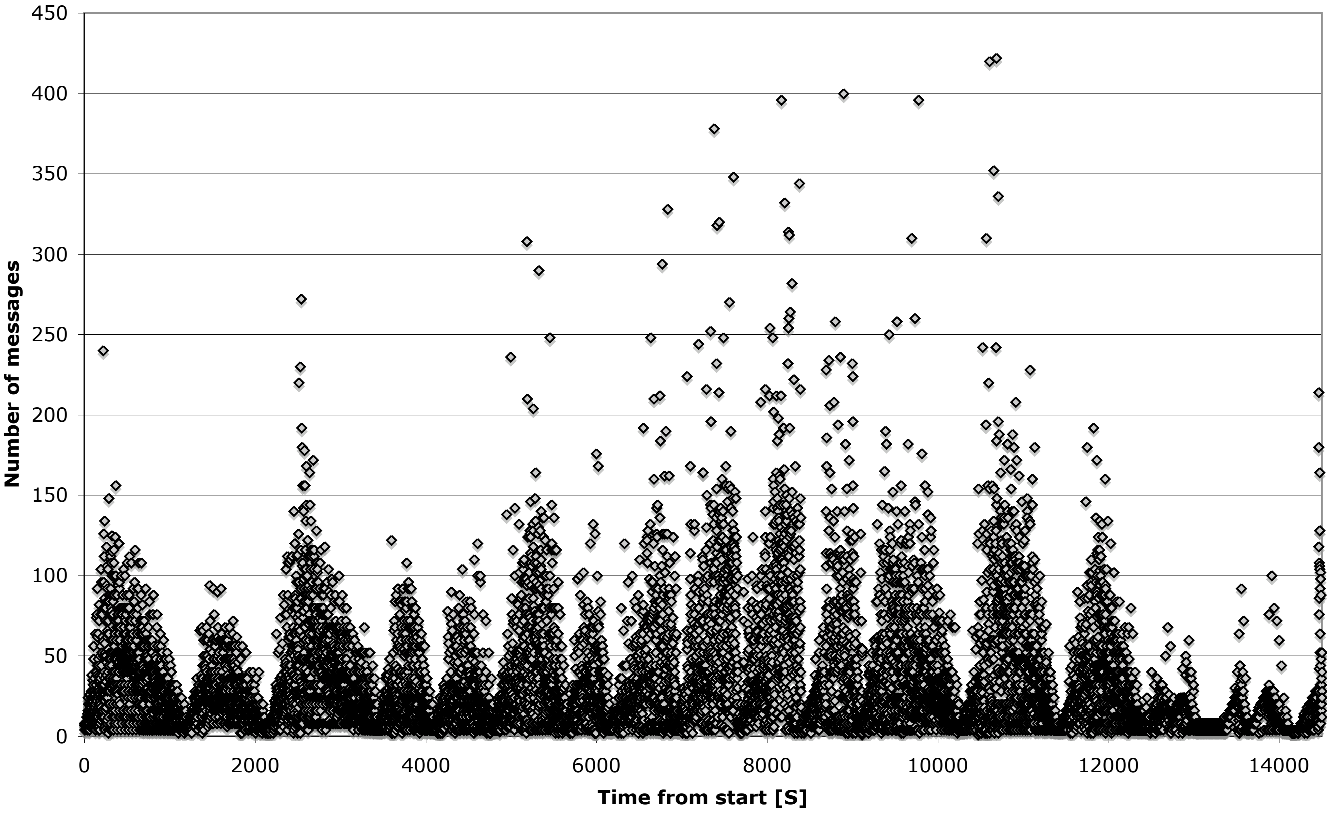


Figure 14: Number of messages per second

7. DISCUSSION OF RELATED WORK

We build upon the work of Alur et al on timed automata [3], which have been used for verifying timeliness constraints in specifications in, for example [6, 10, 2]. The use of timed automata for monitoring and diagnosis has been suggested in [1]: in this work the authors propose a methodology to monitor a certain system (e.g. a plant) and diagnose its state. In our work, instead, we are interested in monitoring of *service quality* according to an SLA between two parties, and in the *automatic* generation of monitors: to the best of our knowledge, timed automata have not been used before for this purpose and no implementation has been made available. In previous work [23] we have suggested the use of automata for monitoring web services; In addition to that work, here we derive automata directly from SLA specifications, present a full implementation, and validate our work against a production case study.

The automatic generation of monitors from requirements has been investigated by Havelund and Rosu in [15] using a technique based on states and transitions. Their aim was to monitor LTL safety properties in program executions. Our work differs from this approach in that we are interested in *timed* properties (which cannot be expressed in LTL), and we monitor SLAs between different and separated parties instead of a single piece of software.

SLAs are formal and precise statements of non-functional requirements. Thus our work is related to requirements monitoring that was proposed by Fickas and Feather [14]. The implementation of this approach was described in [8], 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 [7], which is written in LISP. We have used FLEA and the AP5 implementation for monitoring purposes in [13] 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 [24], 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 discuss, however, how these temporal logic formulae can be monitored efficiently.

Baresi et al have proposed various techniques for monitoring BPEL web service compositions [4]. 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 SLAng timeliness constraints. Their second approach uses our xlinkit rule engine [21]. 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 [20]. They use Event Calculus [25] to express temporal constraints for service ex-

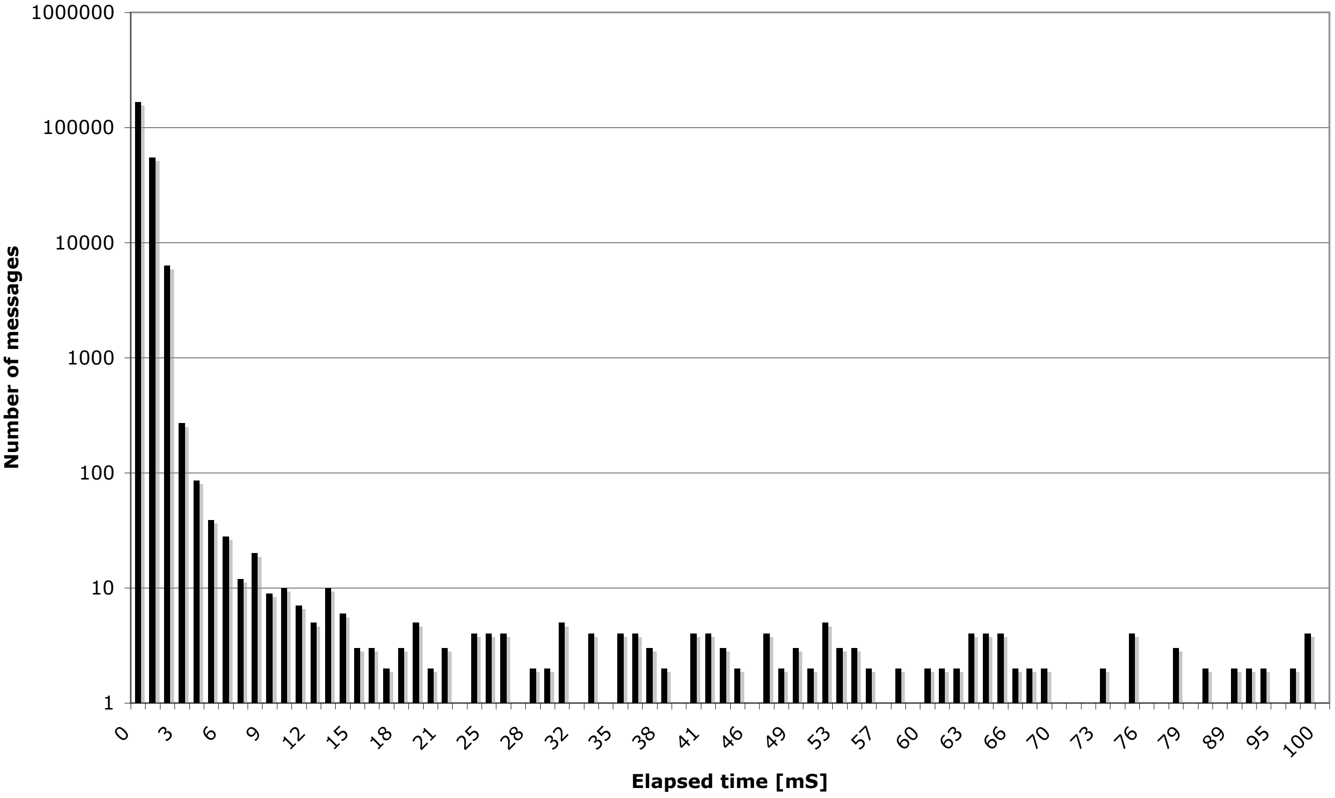


Figure 15: Distribution of the execution times for the handler

ecutions. 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 throughput 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 [10]. 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 Web Service SLAs based on timed automata. We have presented a Java implementation using Eclipse, 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 and obtain average verification times of under a millisecond.

Our aim here was to provide an efficient methodology and to prove its feasibility, and thus various extensions remain to be investigated. For instance, 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 and of the UK EPSRC project Divergent Grid to extend our methodology.

Acknowledgements

The work described in this paper was partially funded by EU IST grant 2005-026955 (PLASTIC) and EPSRC Grant EP/C534891 (Divergent Grid).

9. REFERENCES

[1] K. Altisen, F. Cassez, and S. Tripakis. Monitoring and Fault-Diagnosis with Digital Clocks. In *6th Int. Conf. on Application of Concurrency to System Design (ACSD'06)*, pages 101–110. IEEE CS Press, 2006.

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

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

[4] L. Baresi, C. Ghezzi, and S. 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.

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

[8] D. Cohen, M. S. Feather, K. Narayanaswamy, and S. 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.

[9] 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.

[10] 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 *LNCS*, pages 226–245. Springer Verlag, 2006.

[11] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In M. Ardis, editor, *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, pages 7–15, New York, 1998. ACM Press.

[12] 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.

[13] 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.

[14] 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.

[15] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 342–356, 2002.

[16] W. E. J. Skene, F. Raimondi. Service-Level Agreements for Electronic Services. Technical report, 2008. Submitted for Publication.

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

[18] W. Lee, S. McGough, S. Newhouse, and J. Darlington. A Standard Based Approach to Job Submission through Web Servi ces. In S. Cox, editor, *Proc. of the UK e-Science All Hands Meeting, Nottingham*, pages 901–905. UK EPSRC, 2004. ISBN 1-904425-21-6.

[19] G. Lodi, F. Panzieri, D. Rossi, and E. Turrini. SLA-Driven Clustering of QoS-Aware Application Servers. *IEEE Transactions on Software Engineering*, 33(3):186–197, 2007.

[20] 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.

[21] 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.

[22] PLASTIC. `http://www.ist-plastic.org`.

[23] F. Raimondi, J. Skene, W. Emmerich, and B. Woźna. A methodology for online monitoring non-functional specification of web-services. In D. K. C. Attiogbé, editor, *Proceedings of the First International Workshop on Property Verification for Software Components and Services (PROVECS'07)*, number 567 in ETH Technical Report, pages 50–59. COLOSS Team - University of Nantes, 2007.

[24] 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.

[25] M. Shanahan. The Event Calculus explained. In *Artificial Intelligence Today*, volume 1600 of *LNCS*, pages 409–430. Springer Verlag, 1999.

[26] J. Skene. *The SLAng SLA Language*. UCL, http://uclslang.sourceforge.net, 2006.

[27] J. Skene and W. Emmerich. Specifications, not Meta-Models. In *Proc. of the ICSE 2006 Workshop on Global integrated Model Mannagement (GaMMa 2006)*, pages 47–54. ACM Press, 2006.

[28] 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 CS Press, May 2004.

[29] 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*, pages 3–14. ACM Press, Feb. 2007.