A Methodology for On-line Monitoring Non-Functional Specifications of Web-Services

Franco Raimondi[1,2] James Skene[3] Wolfgang Emmerich[4]

Department of Computer Science
University College London
London, UK

Bożena Woźna[5,6]

Institute of Mathematics and Computer Science
Jan Długosz University
Al. Armii Krajowej 13/15, 42-200 Częstochowa, Poland

Abstract

Web services are increasingly used in complex settings, and it is therefore desirable to have methodologies and tools to verify at run-time the conformance of the services to their specifications. In this paper we present a methodology for monitoring non-functional specifications of web services (such as latency and reliability): we encode the specifications as timed automata and we present how violations can be detected by checking the acceptance of a word (representing the service execution) by a timed automaton. We present a preliminary implementation and we describe an installation example.

Keywords: Verification of non-functional properties, verification tools

1 Introduction

There is a growing trend for IT systems to be integrated across organizational boundaries, using network-provided services. Examples include supply-chain management using RosettaNet [5] 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; moreover, it is common that services belonging into different organizations become components of more complex services. However, when organizations do rely on the

[1] This work was partially supported by the European project PLASTIC.
[2] Email: f.raimondi@cs.ucl.ac.uk
[3] Email: j.skene@cs.ucl.ac.uk
[4] Email: w.emmerich@cs.ucl.ac.uk
[5] Email: b.wozna@ajd.czest.pl
[6] The author acknowledges support from the Polish research grant No. 3T11C01128.

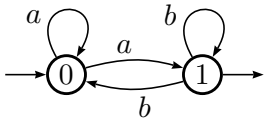

Fig. 1. A simple automaton.

web services provided by other organizations for the implementation of their business processes they usually want to monitor the conformance of these services to some required specifications.

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. 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 reasonable utilization levels. The service provider will also have to monitor its own services to protect itself against false claims of poor services.

The principal contribution of this paper is the presentation of how a number of requirements (or *specification patterns*, in the sense of [7]) for web services can be expressed using timed automata which, in turn, can be used as a formalism to support efficient monitoring of timeliness constraints. Specifically, we associate a timed automaton to each requirement, we timestamp SOAP messages that are exchanged as part of a web service and consider these are timed letters. We are then able to reduce the question of whether a specification is violated to the acceptance of a timed word by a timed automaton. This is decidable in polynomial time and because we can process each message on-line, we obtain a very limited overhead into the web service environment.

The rest of the paper is organised as follows: Section 2 reviews automata and timed automata; Section 3 analyses specification patterns for web service requirements and their translation into timed automata; Section 4 presents our methodology for on-line monitoring, and Section 5 describes a Java implementation and and installation example; comparison to related work is presented in Section 6. We conclude in Section 7.

2 Preliminaries: Automata and Timed Automata

An automaton is a tuple $A = (\Sigma, Q, Q^0, \delta, F)$. As an example, consider Figure 1 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.

Automata recognise languages: given an automaton A, $\mathcal{L}(A) \subseteq \Sigma^*$ is the lan-

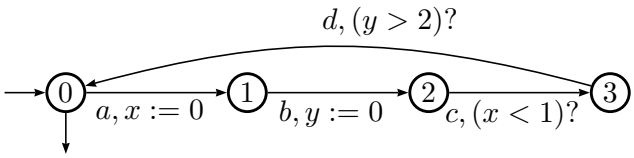

Fig. 2. A Timed Automaton.

guage accepted by the automaton. For the automaton of Figure 1, $\mathcal{L}(A)$ includes the words $\{a, ab, abb, abaaaaabaab, \ldots\}$. Intuitively, the language $\mathcal{L}(A)$ includes all the strings for which the automaton terminates in the final (accepting) state.

A *time sequence* is a sequence of real numbers $\tau = \tau_1\tau_2\ldots$ such that $\tau_i > \tau_{i-1}$ 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 as in the case of automata, and τ is a time sequence. For instance, a timed word looks like the following tuple: $\{(aab\ldots), (0.1, 0.3, 1.2, \ldots)\}$.

Timed automata [1] extend automata by introducing a set of clocks $x, y, \ldots$ (these are variables ranging over positive real numbers), a set of time constraints defined over clocks and associated with transitions, and clock reset operations over transitions. As an example, consider the timed automaton in Figure 2: 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 when a transition occurs 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. State 0 is both the initial and the final state.

Timed automata accept timed words, i.e. *they recognise timed languages*. For instance, the timed automaton in Figure 2 recognises the language $\mathcal{L}(TA) = \{((abcd)^*, \tau) | (\tau_{4j+3} < \tau_{4j+1} + 1) \wedge (\tau_{4j+4} > \tau_{4j+2} + 2)\}$, i.e., the language consisting of repeated *abcd* strings, where in each string the temporal distance between a and c is less than 1, and the temporal distance between b and d is greater than 2.

Notice that for the purposes of this paper we consider *finite* automata only, in the sense that the acceptance condition is characterised by a final state and not by a set of states occurring infinitely often, as in the case of Büchi automata [4]. Thus, in this paper all automata executions have a finite length.

3 From Specifications to Timed Automata

Specification patterns are defined in [7] 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 web services and we show how timed automata characterising violations can be derived from these patterns. Generic-purpose specification patterns are analysed in [7], and an extension to patterns involving time intervals is presented in [8].

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) [12]. These services are typ-

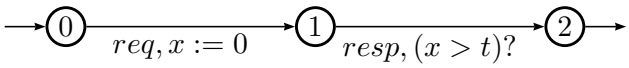

Fig. 3. A Timed Automaton for Latency Violations.

ically developed as web services and we were able to identify the following recurring patterns:

(i) **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 [7], with the addition of a temporal constraint as described in [8].

(ii) **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 [7]. 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 [7] nor in [8]. We show below how this pattern can be encoded as a timed automaton.

(iii) **Throughput requirements**. A third kind of requirements appear often in the definition of the interaction between a service consumer and a service provider: the provider imposes restrictions on the number of request 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.

3.1 Timed Automata to Encode Violations

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.

More in detail, Figure 3 depicts 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 (i) 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

[7] Notice that we do not consider here functional errors such as, for example, a web service returning a wrong data type.

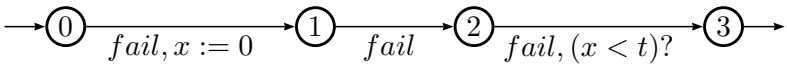

Fig. 4. A Timed Automaton for Reliability Violations.

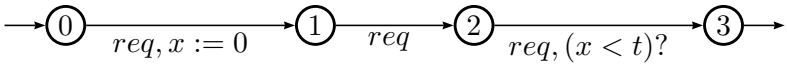

Fig. 5. A Timed Automaton for Reliability Violations.

presented in Section 5.

Figure 4 represents an automaton accepting all the time words in which three failures (labelled with $fail$) occur within t time units. As mentioned above, failures can be identified by latency violations or other kinds of timeouts. The automaton in this figure corresponds to an instance of the Reliability pattern described in item (ii) above; notice that the number of states of the automaton is equal to the number of events to be counted + 1.

Figure 5 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 (iii) above. Notice that this automaton is structurally similar to the one encoding violations of reliability.

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. This observation is key to detect 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 5). 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 5, 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.

In the next subsection we present an analysis of the maximum number of events to be stored and of the complexity of this methodology.

4.1 The Diameter of Witnesses and the Complexity of On-line Monitoring

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. 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 [3], 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 4.1 *On-line monitoring for the patterns of Section 3 using automata has a worst case complexity $O(n^2)$, where n is the size 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 [1]). 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

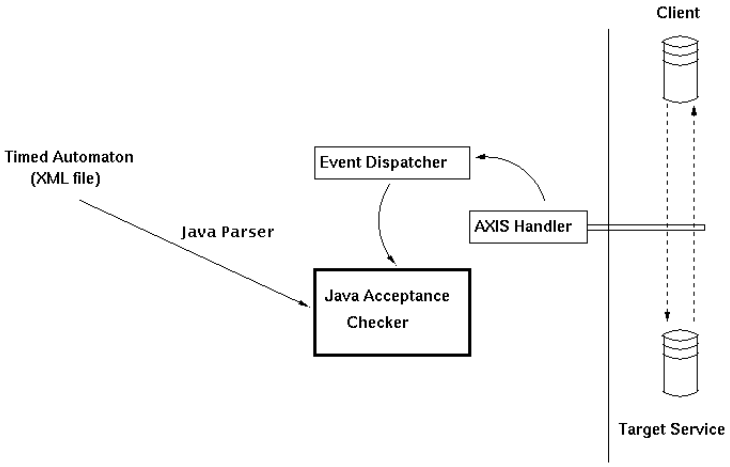

Fig. 6. Implementation of the automata-based monitor

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 polynomial complexity; moreover, the results obtained in Section 5.1 show that it is often not necessary to perform n^2 operations for the patterns considered here.

5 Implementation

Our implementation of the automata-based on-line monitor for web services is presented in Figure 6. We use Tomcat 5 and the open source AXIS engine version 1.2 from Apache to process SOAP messages and use its handler mechanism for monitoring, as detailed below.

We use a simplified version of the XML syntax for describing timed automata used in the model checker UPPAAL [11]. We have developed a parser to translate an XML definition of a timed automaton into an acceptance checker, implementing the verification method presented in Section 4. Essentially, the acceptance checker starts from the initial state of an 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 action is taken. Currently, the checker writes violations to a file, but it could be easily modified to send emails or to perform other kinds of actions. The Java Acceptance Checker is a stand-alone application which receives events from an event dispatcher.

In our implementation (shown in Figure 6) we defined a Java class for the event dispatcher. This class is invoked by AXIS handlers in the message chain of a 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, or at both sides. The event dispatcher performs a first selection of the messages to be processed, based on the names of the participants and the kind of events to be monitored. This allows to operate the

checkers even with interleaving actions, possibly from different clients.

Notice that, for certain requirements such as latency, more than two actors can be involved. For instance, from a client's point of view, the latency of a service is the result of the latencies of the service provider and the underlying network. We refer to [17] for a discussion on the location of the monitors in these cases.

As mentioned above, when more clients are present (and when more request can be submitted concurrently) it is necessary to keep track of the pairs request-response. In our implementation, we use AXIS handlers to add a SOAP attachment to clients' requests, containing the timestamp of the originating request (this timestamp could be also digitally signed for un-trusted servers). When a response is generated by the server, the attachment is copied *verbatim* into the response. Therefore, the client can extract the timestamp for the request from the response message itself and there is no need to keep data structures to relate message IDs for requests and responses, thereby improving the efficiency of this approach.

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

5.1 Installation Example

We set up a simple example to evaluate the feasibility of monitoring a web service. To this end, we have implemented a simple web service to generate random numbers following a given input distribution. Clients can invoke the service by submitting the parameters of the distribution. The web service returns a random number according to the input parameters. We imposed the throughput requirement that a client cannot submit more than 3 requests in a given minute, and that the service latency should be less than 20 milliseconds (notice that, as mentioned above, a monitor for reliability would be similar to the one for throughput).

We have deployed the client and the web service on two separate machines on a wired local network (two Apple MacBook Pro with dual 2GHz processor and 2Gb of RAM). We installed a monitor on the server side by generating a .class events dispatcher and a .class acceptance checkers. These class files have been placed in the directory structure of the AXIS installations and the service has been deployed with the appropriate handlers for request and reply flows. Violations of the throughput clause were correctly reported to a text file without significant overheads.

We then installed a monitor on the client side. This is done by creating an appropriate client configuration file (in our example, this is called client-config.wsdd) and by including the relevant AXIS libraries in the client's classpath. The JVM is then notified of the handler using the command line option `-Daxis.ClientConfigFile=client-config.wsdd`.

As in the case of the server, the client was able to detect all latency violations; in this case, too, we did not notice significant changes in the performance of the service.

We are currently working with various partners to apply this methodology to more realistic scenarios. Preliminary results on a distributed grid application show that the approach is feasible even in presence of a high number of messages (in the order of 10^4 in 10 minutes).

6 Related Work

The approach presented in this paper has the same aims as that of Robinson [13], 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 [2]. 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. Their second approach uses our xlinkit rule engine [10]. 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 [9]. They use Event Calculus [14] 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 constraints (such as the Throughput clauses we discussed above) that require knowledge about events that are not observable by a BPEL engine.

Song Dong et al report on their use of timed automata and the Uppaal libraries for the verification of web service orchestrations in [6]. 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.

On a previous work [16,15], we employed an OCL evaluator to check requirements but this approach did not scale up with the number of messages exchanged, due to the nature of the OCL evaluation procedures.

7 Conclusion

We have presented a methodology for on-line monitoring non-functional specifications of web services based on automata. 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 timestamps) to look for violations of the specifications.

Differently from previous approaches, the performance of our acceptance checker does not depend on the total number of messages in the system. Moreover, each checker is very small (the .class file is typically less than 10Kb). Our aim here was to provide an efficient methodology and to prove its feasibility, and thus various extensions remain to be investigated. In particular, we are currently evaluating in detail the overheads of this methodology in collaboration with the industrial and academic

partners of the European project PLASTIC (http://www.ist-plastic.org) and of the UK EPSRC project Divergent Grid.

References

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

[2] Baresi, L., C. Ghezzi and S. Guinea, *Smart monitors for composed services*, in: *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing* (2004), pp. 193–202.

[3] Biere, A., A. Cimatti, E. Clarke and Y. Zhu, *Symbolic model checking without BDDs*, in: *Proc. of TACAS'99*, LNCS **1579** (1999), pp. 193–207.

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

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

[6] Dong, J. S., 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*, Lecture Notes in Computer Science **4260**, Springer Verlag, 2006 pp. 226–245.

[7] Dwyer, M. B., 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)* (1998), pp. 7–15.

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

[9] Mahbub, K. 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* (2004), pp. 84–93.

[10] Nentwich, C., L. Capra, W. Emmerich and A. Finkelstein, *xlinkit: A Consistency Checking and Smart Link Generation Service*, ACM Transactions on Internet Technology **2** (2002), pp. 151–185.
URL http://dx.doi.org/10.1145/514183.514186

[11] Pettersson, P. and K. G. Larsen., Uppaal*2k*, Bulletin of the European Association for Theoretical Computer Science **70** (2000), pp. 40–44.

[12] PLASTIC, *Providing Lightweight and Adaptable Service Technology for pervasive Information and Communicatio*, http://www.ist-plastic.org (2007).

[13] Robinson, W. N., *Monitoring Web Service Requirements*, in: *RE '03: Proceedings of the 11th IEEE International Conference on Requirements Engineering* (2003), p. 65.

[14] Shanahan, M., *The Event Calculus explained*, in: *Artificial Intelligence Today*, Lecture Notes in Computer Science **1600**, Springer Verlag, 1999 pp. 409–430.

[15] Skene, J. and W. Emmerich, *Generating a Contract Checker for an SLA Language*, in: *Proc. of the EDOC 2004 Workshop on Contract Architectures and Languages, Monterey, California* (2004).
URL http://www.cs.ucl.ac.uk/staff/w.emmerich/publications/COALA04/coala.pdf

[16] Skene, J. and W. Emmerich, *Engineering Runtime Requirements-Monitoring Systems using MDA Technologies*, in: *IFIP Symposium on Trustworthy Global Computing*, Lecture Notes in Computer Science **3705**, Springer, 2005 pp. 319–333.
URL http://www.cs.ucl.ac.uk/staff/w.emmerich/publications/TGC05/tgc.pdf

[17] Skene, J., 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* (2007), to appear.
URL http://www.cs.ucl.ac.uk/staff/w.emmerich/publications/WOSP2007/wosp.pdf