Mapping of Service Level Objectives to Performance Queries

Jürgen Walter
University of Würzburg
97074 Würzburg Germany

Dušan Okanović
University of Stuttgart
70569 Stuttgart Germany

Samuel Kounev
University of Würzburg
97074 Würzburg Germany

ABSTRACT

The concept of service level agreements (SLAs) defines the idea of a reliable contract between service providers and their users. SLAs provide information on the scope, the quality and the responsibilities of a service and its provider. Service level objectives (SLOs) define the detailed, measurable conditions of the SLAs. After service deployment, SLAs are monitored for situations, that lead to SLA violations.

However, the SLA monitoring infrastructure is usually specific to the underlying system infrastructure, lacks generalization, and is often limited to measurement-based approaches. This makes it hard to apply the results from SLA monitoring in other stages of the software life-cycle. In this paper we propose the mapping of concerns defined in SLAs to the performance metrics queries using the Descartes Query Language (DQL). The benefit of our approach is that the same performance query can then be reused for evaluation of performance concerns throughout the entire life-cycle, and regardless of which approach is used for evaluation.

CCS Concepts

•**Software and its engineering** → **Software performance;** *Software system models;*

Keywords

Service Level Agreements; Declarative Performance Engineering; Measurement-based Analysis; Model-based Analysis

1. INTRODUCTION

The importance of performance is common in many working environments. For example, a salesman is required to sell a specific amount of items over a certain period of time. If he has not reached the required number of sales, agreed on in his contract, he will earn less money. This concept, with only small changes, can be applied in different environments. The above principle is applied to the world of IT-services using specific contracts defining the requirements for service providers, to satisfy their clients.

A service level agreement (SLA) defines the idea of a reliable contract between service providers and their users. It contains the information about the responsibilities, the scope, and the expected quality of service (QoS). An SLA contains an aggregation of different Service level objectives (SLOs), where a single SLO defines a specific performance goal to be met. One common example of these goals is "service response time below a certain threshold." SLOs define the detailed, measurable conditions of the SLAs. Due to the various domains in which these contracts are needed, there are many different implementations, specific to their particular domain.

In the work by Woodside et al. [22], two distinct groups of approaches are identified in software performance engineering (SPE): *model-based* and *measurement-based* approaches. Model-based approaches are used early in the software development life-cycle. For example, in early stages of software development, when designing the software to meet the performance levels defined in SLA, one can employ model-based approaches. On the other hand, measurement-based approaches are used in later stages of the software development life-cycle. For example, if we want to monitor and evaluate the execution of SLAs in operation phase, we need to use *measurement-based approaches*. The issue here is that there is no (or almost no) connection between the two groups of approaches. What is required is a unified view on the software performance, regardless of the software life-cycle stage.

In our previous work we proposed the Declarative Performance Engineering (DPE) [21] and the Descartes Query Language (DQL) [3], as an approach to provide a declarative, unified language to trigger performance measurements and predictions. Using DQL, performance metric queries can be evaluated in the same way, regardless of the stage in the software development cycle. In this paper, we propose a mapping of SLAs to performance queries, to abstract from the evaluation mechanism (measurement or model-based) and provide a reusable SLA framework. The DQL language definition is extended to support the definition of SLAs and SLOs. Further, we implement a mapping to performance queries to automatically evaluate SLAs. This mapping allows for:

- An interface between an SLA language and performance evaluation mechanisms, for any kind of software.
- The reduction of the effort required for applying different performance evaluation methods in different SLA

ICPE '17 Companion, April 22 - 26, 2017, L'Aquila, Italy

ISBN 978-1-4503-4899-7/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3053600.3053646

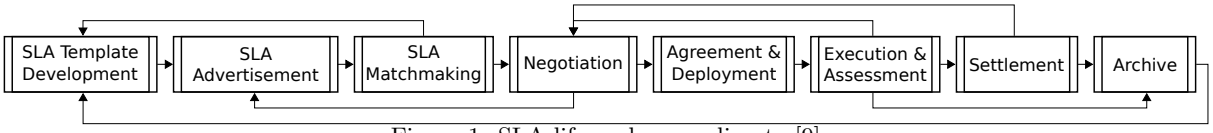

Figure 1: SLA life cycle according to [9]

life-cycle stages.

- Bridging the gap between model- and measurement-based approaches [21, 22] for performance evaluation.
- Language support for service level improvement (according to Sturm et al. [16]) using reactive approaches, e.g., auto-scaling in cloud-based systems, to automatically mitigate the performance issues.

The subsequent sections provide an overview of the related work (Section 2), emphasize the addressed problem (Section 3), and outline our approach (Section 4). In Section 5 we describe our SLA language and the benefits of our user interface, in Section 6 the extensions to the DQL framework required to evaluate SLAs, and in Section 7 the mapping of SLOs to DQL elements. The evaluation of our approach is shown in Section 8. The last section provides conclusion and outlines the future work.

2. RELATED WORK

Works related to the one presented in this paper can be divided into three main categories: 1) works that deal with the SLA life cycle, 2) works dealing with languages for SLA specification, and 3) works on systems for SLA execution assessment.

SLA life cycle: There is a research area on SLA life cycle and management. Kritikos et al. [9] propose an SLA life cycle depicted in Figure 1. In Comuzzi et al. [2] different roles are assigned to organizational units within each involved party. When communicating among each other, units use internal SLAs. The idea is to manage the complete IT stack. In contrast to our approach, they do not focus on evaluation mechanisms throughout the software life cycle.

Languages: Survey by Kritikos et al. [9] shows that there are many languages available for SLA specification. Well known examples are WSLA [5], SLAng [15], GXLA [17], SLA* [4]. Early works on SLA formats usually focused on defining a machine readable format [17], e.g. XML, as they were supposed to be processed automatically. However, this increases the manual effort required for their maintenance, and calls for a format that is both machine- and human-readable [13, 18].

Some recent works focus on using SLAs in new environments, such as cloud computing, where the landscape, on which the service is running, does not consist of, e.g., an array of dependable servers, but fluctuates. For example, formats like CSLA [7] and SLA* [4] are designed to cope with this kind of environment.

Assessment: The most common approach for the evaluation of SLA execution, is to monitor the service during the production phase, and check for any violation of SLA. Although it would be possible, monitoring is usually performed by the provider, using its infrastructure, rather than by some third party [9]. Therefore, most monitoring solutions are proprietary and tied to a certain platform. For example, WSLA [5] framework also provides monitoring capabilities to accompany the SLA specification, but the prototype implementation is available only for IBM environments. In order to be independent of a specific SLA format, approaches by Okanović et al. [12] and van Hoorn [19] propose to use some intermediate form to represent the SLA and compare it to the measured values. In contrast to the previously presented approaches that use run-time data, Klatt et. al. [6] perform model-based SLA analysis. In their work, they integrate QoS prediction support in a large-scale SLA management framework and a service mashup platform.

To the best of our knowledge, no other approach aims to cover both the design- and production-time evaluation of SLAs, using the SLA definition that is both human- and machine-readable, while being general and applicable to any kind of system.

3. PROBLEM STATEMENT

The following user stories illustrate our motivation for this work.

- "*As a user, I want to evaluate my SLAs with different SPE methods in different stages of the software life cycle.*" SLAs are known to all stakeholders from the beginning of the life cycle, but this is not enough. Current practice evaluates software against SLAs only in latter stages, mainly in production. It would be beneficial to evaluate SLAs, e.g., using models, in early stages of software development. The evaluation against SLAs during the whole life cycle would allow for performance problems to be avoided in latter stages, when fixing becomes more expensive.
- "*As a user, I want the approach to require low effort.*" This can be achieved through an easily readable and understandable SLA definition language, with an editor that supports syntax highlighting and auto-completion. The language should also allow for the reuse of goals in different SLAs. Underlying mechanisms should automatically choose the right analysis approach, depending on the current development stage. Analysis results should be accessible using the same language.
- "*As a user, I want to use SLAs for different use cases.*" For example, qualitative analysis can detect that there was a violation, and can be useful for design space exploration. Quantitative analysis finds the number of violations, and is more detailed. Also, results can be used to automatically ensure the service levels, e.g., for auto-scaling.

In order to define non-functional properties for services, an SLO has to specify thresholds, metrics, and comparators. However, in modern environments there are new, previously unknown challenges. Due to inherent characteristics of modern environments, e.g., cloud, SLAs have to cope with the acceptable margins [14]. Also, sometimes it is not possible to meet all SLOs at the same time. SLA implementations should therefore support modeling of such conflicts.

Software evolution can be hindered by the inability to have efficient communication between different stages of development. Being able to automatically map concepts from one

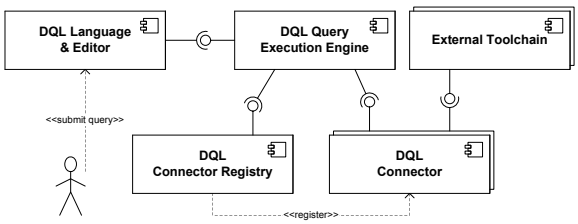

Figure 2: DQL Architecture [3]

stage to another would ensure that the quality of service is properly addressed throughout the software life cycle.

4. APPROACH

Software performance evaluation against SLAs should be performed at different life-cycle stages [9]. However, there is no unified approach nor tooling to achieve this. In our previous work [21], we proposed how to perform performance evaluation regardless of development stage. In this paper, we propose to reuse the performance metrics query interface, in order to query the system performance. The existing DQL architecture (Figure 2), with the supporting infrastructure, such as connectors for using external tool chains (measurement, simulation or analytical solvers/SPE techniques), can be reused to evaluate SLOs. This leads to an interchangeability of evaluation approaches for SLA evaluation.

We extend the DQL Framework [3] to evaluate SLA compliance by reusing existing connectors for different evaluation approaches, even though the core concepts are applicable to other SLA languages and frameworks as well.

DQL language extension The SLAs evaluation requires a language definition. We extend DQL language introducing an SLA syntax that builds upon established SLA definitions. Further, to proactively support the DPE vision we include an extension for SLA languages to support prioritizing and weighting of goals required for proactive assurance mechanisms (Section 5).

DQL framework extension What is needed is an extension to DQL Execution Engine to include an evaluation algorithm that supports comparators, penalties and fuzziness (Section 6).

Mapping Finally, we extend DQL framework with mapping of SLOs to DQL performance metrics queries, to provide a unified interface to the system under test (Section 7).

5. SLA LANGUAGE DEFINITION

We build our syntax upon the established CSLA [7] and WSLA [10] frameworks, and compose SLAs from SLOs. We extend their SLA definition to prioritize SLOs over others. In contrast to the mentioned frameworks, we provide a language implementation, instead of using XML files. We integrated our language implementation into the DQL framework (Figure 2). Extending the `DQL Language & Editor` component allows for language auto-completion and syntax highlighting. Providing a user-friendly interface decreases the initial effort and improves usability and readability in the long run.

Figure 3 shows an illustrative example to which we will refer in the following. Generalizing the example, Figure 4 presents the top-level integration of `GoalQuery` into DQL

```
EVALUATE QUANTITATIVELY
  AGREEMENTS
    sla1 CONTAINS (slo1, slo2*2, slo3)
  GOALS
    slo1: service1.availability > 90%
    slo2: service1.responseTime < 0.2ms,
        FUZZINESS 0.1
        PENALTY 6.0 EUR
                PER 3 VIOLATIONS
    slo3: (slo1 * 3, slo2) >= 97%
      PENALTY 7.0 EUR
FOR SERVICE "service1Identifier" AS service1
CONSTRAINED AS fast
USING connector@'domain_access';
```

Figure 3: Example Goal Definition

Syntax. `GoalQuery`s reuse `EntityReferenceClause`, `ConstraintClause` and `ContextAccess` from DQL. The underlying concepts are not common in other SLA languages. To be interpretable, all queries include a `ContextAccess`, located at the end. As can be seen in the example, it starts by a `USING` statement, followed by an identifier of the `connector` to evaluation mechanism, and finalized by domain access. The domain access can be a link or path to a performance model, log files or a measurement-script. The optional `ConstrainedClause` may influence processing within connector, if the connector supports constraints. For example, `CONSTRAINED AS fast` triggers approximative solutions for model-based analysis, or may be interpreted by a measurement-based connector to trigger shorter measurement periods and extrapolate afterwards. If the IDs of resources or services are too long or cryptic they may be mapped to human readable aliases in the `EntityReferenceClause`. The `EntityReferenceClause` enables to map identifiers from the context to query elements. The identifiers may vary for different evaluation approaches. For example, EMF models often use cryptic hash values as identifiers. Moreover it enables to create short names for complex identifiers. The `EntityReferenceClause` starts with the `FOR` terminal followed by one or more `EntityReference`s. An `EntityReference` starts with the type terminals `RESOURCE` or `SERVICE`, then `EntityID` (can be derived from the context) and is finalized by `AS` keyword and the alias.

Figure 5 depicts the coarse grained syntax of our DQL `GoalQuery` language extension. A language statement starts by a processing identifier, followed by an sla, and slo clause. The processing identifier specifies the kind of actions to perform based the SLAs. We include the keywords: `EVALUATE QUANTITATIVELY`, `EVALUATE QUALITATIVELY` and `ENSURE`. After the processing identifier, the `SlaClause`, started by the `AGREEMENTS` identifier, consists of one or more SLA definitions. An SLA (Figure 6) references a set of SLOs using the `CONTAINS` identifier. Priorization of SLOs happens by multiplying weights. In our running example, `slo2` is twice as important as `slo1` and `slo3` since it is multiplied by 2 while the others have no weight assigned (matches a weight of 1). After the `SlaClause`, the `GOALS` identifier starts the `SloClause`, containing one or more SLO definitions. An SLO, (see Figure 7), consists of an unique `id`, a `MetricReference`, a `Comparator` ($<, \leq, >, \geq, \ldots$) and a `threshold`. Referring to the example, `slo1`'s `MetricReference` is

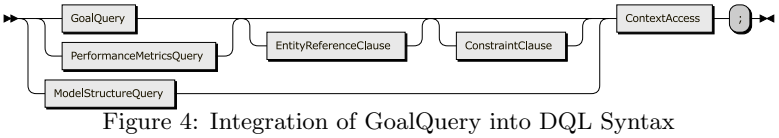

Figure 4: Integration of GoalQuery into DQL Syntax

Figure 5: DQL GoalQuery Syntax

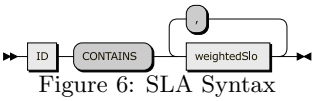

Figure 6: SLA Syntax

defined as `service1.availability`. Further, SLOs can be extended employing two optional penalty and fuzziness definition. `FUZZINESS` describes an allowed deviation from the desired threshold of an SLO which blurs the metric's threshold. This is useful to cope with uncertainties and fluctuations which can occur in cloud environments. Similar to other SLA languages (e.g. WSLA and CSLA), it is possible to associate a penalty with an SLO using the `PENATLY` keyword. For example, it is possible to restrict the penalty to apply for multiple violations. The penalty in `slo2` has to be multiplied times the violations divided by three.

As stated previously, we allow for composition of SLOs. This allows for extra penalties when a specific combination of SLOs is present. In the example (Figure 3), `slo3` specifies that `slo1` and `slo2` have to be compliant in 97 % of all requests. Similar to SLAs, SLOs in a composition can be weighted in order to prioritize their importance.

6. FRAMEWORK EXTENSION

In order to evaluate SLAs, we extended the `DQL Query Execution Engine` component (cf. Figure 2). This component abstracts from concrete connectors and model specific external tool chains. It only triggers methodology provided by the connector interface. Through this we get a reusability for every connector. Our extension supports qualitative and quantitative SLA evaluation. The qualitative evaluation breaks on the detection of the first violation which is more effective (in case of an early violation detection) than evaluating all SLOs. This is sufficient, for example, when exploring the design space for non-SLA-violating configurations. Quantitative evaluation is required when the performance engineer is interested, e.g., in the number of violations and for penalty calculation.

Algorithm 1 shows pseudo code for our SLA evaluation. It iterates over all referenced SLOs. Our implementation caches SLO evaluations that are referenced multiple times to be analyzed (measured, simulated, analyzed analytically) only once (lines 4-6). Line 4 checks if the SLOs have been evaluated before, as SLOs may be referenced in multiple SLAs. If not, the mapping to performance metrics queries happens in line 5. This line also includes the execution of the newly created query resulting in metric value(s) required for SLO evaluation. Lines 7 to 16 evaluate the SLO, based on the derived metric value. Line 7 takes comparator and threshold of the SLO to compare with derived metric.

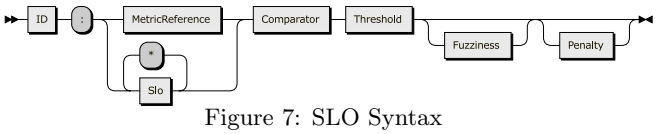

Figure 7: SLO Syntax

The comparison also considers the optional fuzziness setting. Note that the comparator has to handle single values as well as sets. Line 8 sets SLO to violated, line 15 to compliant. Lines 9 and 10 cause a break of evaluation on first SLO violation for qualitative evaluation, while line 12 sums penalty costs for quantitative analysis.

Algorithm 1 SLAs Evaluation

```
1: function EvaluateSLAs(List<SLA> slas, boolean isQualitativeEvaluation)
2:     for all sla in slas do
3:         for all slo in sla do
4:             if slo.isCompliant == null then
5:                 slo.metric ← run query (slo)
6:             end if
7:             if evaluateSLO(slo, metric) then
8:                 slo.isCompliant ← false
9:                 if isQualitativeEvaluation then
10:                    return false
11:                else
12:                    sla.penalty + = slo.penalty
13:                end if
14:            else
15:                slo.isCompliant ← true
16:            end if
17:        end for
18:    end for
19:    return true
20: end function
```

To summarize, our framework extension automates the detection of violations by interpreting query results and calculates penalty costs.

7. MAPPING

The mapping of SLOs to performance metrics queries happens in the line 5 of Algorithm 1. Our implementation creates one metrics query per SLO. Another option for processing would to merge queries so that one query includes all `MetricReference`s in one `MetricReferenceClause`. In figure 8, performance metrics query specific syntax is shown (`EntityReferenceClause` and `ModelAccess`, are already depicted in Figure 4, and here are omitted).

In the following we explain the mapping. We start the `PerformanceMetricsQuery` with the `SELECT` and the `MetricReference` extracted from the SLO (Figure 7). Then we add `EntityReferenceClause`, `ConstraintClause` (if set), and `ContextAccess` from the `GoalQuery`. The optional Degree of Freedom (`DoFClause`) of `PerformanceMetricsQuery` can be ignored for the mapping.

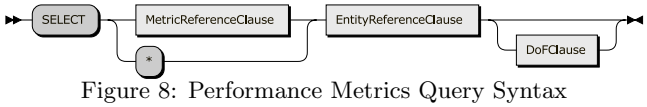

Figure 8: Performance Metrics Query Syntax

```
SELECT service1.availability
FOR SERVICE "service1Identifier" AS service1
CONSTRAINED AS fast
USING connector@'domain_access';

SELECT service1.responseTime.max
FOR SERVICE "service1Identifier" AS service1
CONSTRAINED AS fast
USING connector@'domain_access';
```

Figure 9: Performance Queries Resulting from SLA Example in Figure 3

The example SLA from Figure 3 can be mapped to the queries depicted in Figure 9. The query at the top is required to answer `slo1`, the query for `slo2` is at the bottom, `slo3` can be deduced from previous two.

8. EXAMPLE OF USE

The following example demonstrates how to evaluate SLAs measurement and model-based using our approach.

Sample Application To evaluate our approach, we selected the Pet Clinic application, that represents a portal for vet appointments.[1] We deployed it on a Dell Power Edge R815 with 48 cores, each core equipped with an Opteron 6174 CPU 2.6 GHz. The application was running on an Ubuntu 14.04.5 VM, with 16 GB RAM. We modified the "browse and edit" workload shipped with the application removing the "edit" operations to avoid database contention (due to locking). The resulting workload is open, with an exponentially distributed arrival rate. Each vet customer calls the following sequence of interfaces: The vet visits the frontpage (`welcomeGET`), looks at the list of all vets (`showVetListGet`), searches pet owners (`initFindFormGet` and `processFindFormGet`) and displays all pet owners (`showOwnerGET`). Then the vet triggers two times a specific pet owner page (`processFindFormGet`). Besides workload, we modified the application to cache the vet catalog once at startup, to improve performance.

SLA Evaluation Settings For the described setting, we measured a calibration workload at low utilization, used to trigger performance model extraction employing the Performance Model Extractor (PMX) [20] tool. The extracted model enables to evaluate SLAs using model-based analyses in addition to measurements. We evaluated the SLA depicted in Figure 10 triggering two existing DQL connector implementations. We performed:

- measurement-based analysis using the connector for Kieker [1], that is able to trigger analysis runs and analyze log files using the Kieker Monitoring framework [19]. In our experiment we triggered the measurements separately using an Apache JMeter load script and performed DQL analyses only at the monitoring log files, excluding ramp up time.
- model-based Analysis: using Descartes Modeling Language (DML) connector [8], that triggers the analysis of an architectural performance model using the analysis approach based on a transformation to Queueing Petri Net (QPN) [11] and simulation using SimQPN.

[1]https://github.com/spring-projects/spring-petclinic

```
EVALUATE QUANTITATIVELY
 AGREEMENTS
  sla CONTAINS (slo1,slo2,slo3,slo4,slo5)
 GOALS
  slo1:welcomeGET.responseTime.mean<1.3ms,
    PENALTY 1.0 EUR
  slo2:showVetListGET.responseTime.mean<1.7ms
    PENALTY 1.0 EUR
  slo3:initFindFormGET.responseTime.
     mean<1.3ms, PENALTY 1.0 EUR
  slo4:processFindFormGET.responseTime.
     mean<4ms, PENALTY 1.0 EUR
  slo5:showOwnerGET.responseTime.mean<2.6ms
    PENALTY 1.0 EUR
USING dml@'model.properties';
```

Figure 10: PetClinic SLA Definition

Entity	Evaluation	Analysis Method			
		measurements		model-based	
customers		732	1300	732	1300
slo1	Violating Operations	false	true	false	true
slo2		false	true	false	true
slo3		false	true	false	true
slo4		false	true	false	true
slo5		false	true	false	true
sla1	Conformance	true	false	true	false
	Penalty	0	5	0	5

Table 1: SLA Evaluation Results.

The additional SLA evaluation required no modifications to connectors. One can switch from DML-based analysis to Kieker-based analysis changing the final line of the query by replacing `USING dml@'model.properties';` with `USING kieker@'logfile';`.

Results We triggered SLA evaluation for two different workload settings. The first setting has an arrival rate of 732 customers per second, which causes negligible resource contention. The second setting has 1300 customers per second, causing a higher resource contention resulting in SLA violations. Table 1 depicts results for both settings. Considering the presented two settings, both evaluation methods provided similar results. However, the model does not reflect all of the peaks that appeared in measurements. The measurements of response times have been fragile due to inaccuracies in workload provisioning of JMeter and side effects caused by garbage collection or log storage operations. In general, mean values of response times are much more stable than peaks. Future work could focus on improving performance models to reflect indeterministic peak behavior not caused by application load.

9. CONCLUSIONS

In this work we integrated the definition and the evaluation of SLOs and SLAs into the Descartes Query Language (DQL), with the goal to use DPE to reduce the effort of the evaluation of SLAs in different stages of software life cycle. We propose the extension to DQL to support the mapping of SLOs to DQL performance queries. Extensions were made also to the DQL Execution Engine, to support the evaluation of SLAs using the mapping to performance metric queries.

Future work We identified several possible improvements to the approach. Some DQL connectors allow to derive metrics using optimizations, e.g., simulation breaks at a certain accuracy before the configured run length has been reached. Therefore, future work should discuss different run lengths. Moreover, SLA evaluation results in as many performance queries as SLOs, even if multiple SLOs target the same service and metric. An optimization would be to detect overlaps and execute less queries. We also plan to implement processing strategies for the `ENSURE` keyword in goal queries. We already included the extensions to SLA languages to include prioritization and weighting of goals. We envision to connect to auto-scaling mechanisms that also build upon query results.

Acknowledgments

This work is supported by the German Research Foundation (DFG) in the Priority Programme "DFG-SPP 1593: Design For Future—Managed Software Evolution" (HO 5721/1-1 and KO 3445/15-1) and by the Research Group of the Standard Performance Evaluation Corporation (SPEC). Further, we want to thank Lukas Harzenetter and Niko Stadelmaier contributing to language implementation.

References

[1] M. Blohm, M. Pahlberg, S. Vogel, J. Walter, and D. Okanovic. Kieker4DQL: Declarative Performance Measurement. In *Proceedings of the 2016 Symposium on Software Performance (SSP)*, November 2016.

[2] M. Comuzzi, C. Kotsokalis, C. Rathfelder, W. Theilmann, U. Winkler, and G. Zacco. A framework for multi-level SLA management. In *Int. Conf. on Service-oriented Computing (ICSOC/ServiceWave '09)*, pages 187–196, 2009.

[3] F. Gorsler, F. Brosig, and S. Kounev. Performance queries for architecture-level performance models. In *5th ACM/SPEC Int. Conf. on Perf. Eng. (ICPE 2014)*, pages 99–110, 2014.

[4] K. T. Kearney, F. Torelli, and C. Kotsokalis. SLA*: An abstract syntax for service level agreements. In *11th IEEE/ACM Int. Conf. on Grid Computing*, pages 217–224, 2010.

[5] A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1):57–81, 2003.

[6] B. Klatt, F. Brosch, Z. Durdik, and C. Rathfelder. Quality Prediction in Service Composition Frameworks. In *5th Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing (NFPSLAM-SOC 2011)*, December 2011.

[7] Y. Kouki and T. Ledoux. CSLA: A Language for improving Cloud SLA Management. In *Int. Conf. on Cloud Computing and Services Science, CLOSER 2012*, pages 586–591, 2012.

[8] S. Kounev, N. Huber, F. Brosig, and X. Zhu. A Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures. *IEEE Computer*, 49(7):53–61, July 2016.

[9] K. Kritikos, B. Pernici, P. Plebani, C. Cappiello, M. Comuzzi, S. Benrernou, I. Brandic, A. Kertész, M. Parkin, and M. Carro. A survey on service quality description. *ACM Comput. Surv.*, 46(1):1:1–1:58, 2013.

[10] H. Ludwig, A. Keller, A. Dan, R. P. King, and R. Franck. Web Service Level Agreement (WSLA) Language Specification, v1.0, Jan. 2003.

[11] P. Meier, S. Kounev, and H. Koziolek. Automated transformation of component-based software architecture models to Queueing Petri Nets. In *19th IEEE/ACM Int. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2011.

[12] D. Okanović, A. van Hoorn, Z. Konjović, and M. Vidaković. SLA-driven adaptive monitoring of distributed applications for performance problem localization. *Computer Science and Information Systems*, 10(1):25–50, 2013.

[13] T. Parr. Humans should not have to grok XML. http://www.ibm.com/developerworks/library/x-sbxml/x-sbxml-pdf.pdf, 2001.

[14] D. Serrano, S. Bouchenak, Y. Kouki, T. Ledoux, J. Lejeune, J. Sopena, L. Arantes, and P. Sens. Towards QoS-oriented SLA guarantees for online cloud services. In *13th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid)*, pages 50–57, 2013.

[15] J. Skene, F. Raimondi, and W. Emmerich. Service-level agreements for electronic services. *IEEE Transactions on Software Engineering (TSE)*, 36(2):288–304, 2010.

[16] R. Sturm, W. Morris, and M. Jander. *Foundations of Service Level Management*. SAMS, 2000.

[17] B. Tebbani and I. Aib. GXLA a language for the specification of service level agreements. In *First International IFIP TC6 Conference on Autonomic Networking*, pages 201–214, 2006.

[18] R. Vaderna, Ž. Vuković, D. Okanović, and I. Dejanović. A domain-specific language for service level agreement specification. In *7th Int. Conf. on Inf. Tech.*, pages 693–697, 2015.

[19] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *3rd ACM/SPEC Int. Conf. on Perf. Eng. (ICPE '12)*, pages 247–248, 2012.

[20] J. Walter, C. Stier, H. Koziolek, and S. Kounev. An Expandable Extraction Framework for Architectural Performance Models. In *Proceedings of the 2017 International Workshop on Quality-Aware DevOps (QUDOS'17) co-located with 8th ACM/SPEC International Conference on Performance Engineering (ICPE 2017)*. ACM, April 2017.

[21] J. Walter, A. van Hoorn, H. Koziolek, D. Okanovic, and S. Kounev. Asking "What?", Automating the "How?": The Vision of Declarative Performance Engineering. In *7th ACM/SPEC Int. Conf. on Perf. Eng. (ICPE '16)*, March 2016.

[22] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 171–187, 2007.