Analysis of SLA Compliance in the Cloud: An Automated, Model-based Approach*

Frank S. de Boer
CWI Amsterdam
The Netherlands
F.S.de.Boer@cwi.nl

Elena Giachino
University of Bologna
Italy
elena.giachino@unibo.it

Stijn de Gouw
The Open University
The Netherlands
cdegouw@gmail.com

Reiner Hähnle
Technical University of Darmstadt
Germany
haehnle@cs.tu-darmstadt.de

Einar Broch Johnsen
University of Oslo
Norway
einarj@ifi.uio.no

Cosimo Laneve
University of Bologna
Italy
cosimo.laneve@unibo.it

Ka I Pun
Western Norway University of Applied Sciences
University of Oslo
Norway
Violet.Ka.I.Pun@hvl.no

Gianluigi Zavattaro
University of Bologna
Italy
zavattar@cs.unibo.it

Abstract. Service Level Agreements (SLA) are commonly used to specify the quality attributes between cloud service providers and the customers. A violation of SLAs can result in high penalties. To allow the analysis of SLA compliance before the services are deployed, we describe in this paper an approach for SLA-aware deployment of services on the cloud, and illustrate its workflow by means of a case study. The approach is based on formal models combined with static analysis tools and generated runtime monitors. As such, it fits well within a methodology combining software development with information technology operations (DevOps).

1 Model-Centric Analysis of SLA Compliance

Every customer wants to be sure about the quality of his purchases. In the cloud world, this quality assurance includes guarantees on service *performance*[1].

Service Level Agreements (SLAs) are legal documents, signed and agreed upon by cloud service providers and their customers, which specify the agreed quality of service. An SLA violation will result in penalties and possibly in a loss of money, clients, and credibility. Even though the stakes are high, there are only few tools with limited capabilities available to check the compliance of cloud services with SLAs. But why does it seem to be so difficult to provide tool support for SLA compliance checking and monitoring?

For a start, a number of complex and challenging questions arise: How to describe service performance? How many resources, for example, memory or virtual machines, should be assigned to a particular service and how should they be configured? How to react optimally at runtime to take advantage of the elasticity of the cloud? How to estimate the future behavior of a service and adjust the resource configuration accordingly?

*Partially funded by EU project FP7-610582 ENVISAGE: Engineering Virtualized Services www.envisage-project.eu.

[1] Other concerns include security, support, data management and data protection [15].

D. Ancona and G. Pace (Eds.): Verification
of Objects at RunTime EXecution 2018 (VORTEX 2018)
EPTCS 302, 2019, pp. 1–15, doi:10.4204/EPTCS.302.1

These are challenging issues! It is beyond current technology to address them in a general way for any given SLA and any given software. To develop effective tools for SLA compliance analysis, we believe it is essential to work at the level of *models*, and describe and analyze SLAs in a way that is independent of the concrete technology offered by the cloud service provider. Shifting to the modeling level increases the level of abstraction, reduces complexity, and removes dependency on a specific runtime environment.

The importance of models applies to SLAs as well as to software: a model-centric approach allows us to create a formal representation of the essential aspects of an SLA. At the same time, software services deployed on the cloud can be represented as an executable *service model*, annotated with parametric expressions for their use of resources. Combining the two models, i.e., of the SLA and of the cloud services, makes it now possible to use techniques with a formal basis, such as static software analysis or monitor generation. Such tools provide proven guarantees on service performance, thereby vastly raising the degree of automation.

Benefits of model-centric, tool-based SLA analysis

An effective solution to SLA design and compliance must coordinate all phases of service provisioning:

- Provide assistance in the configuration of SLA metric bounds and provisioning of virtual machines whose resources comply to the services' requirements
- Permit automatic monitoring of the service at runtime
- Enable speedy reaction to an SLA violation and assistance in its resolution
- Support deployment: significant simplification and increased automation

In this paper, we present an approach to facilitate SLA-aware deployment of services on the cloud by combining the formal executable model of the target system of deployed services and the formal representation of corresponding SLAs. We define a detailed workflow that takes advantage of the formal models by enabling automated tool support at various stages. With the help of a case study we domonstrate how our approach can be realized for a real-world cloud service provider.

The paper is organised as follows: Section 2 describes the cloud service performance metrics that our approach is supposed to measure and verify, Section 3 outlines the workflow of the model-centric SLA compliance analysis; Section 4 provides a short introduction of the modeling language used in this approach, Section 5 presents the case study, Section 6, and Section 7 concludes the paper.

2 What to be Measured? What to be Verified?

Service performance metrics measure and assess the performance level of a service, quantitatively and periodically. Typical metrics fall into one of the following categories:

Availability is the property of a service to be accessible and usable on demand. It includes (*i*) the *level of uptime*, namely the percentage of time a service is up within a defined period; (*ii*) the *percentage of successful requests*, namely the number of requests processed without an error over the total number of submitted requests; (*iii*) the *percentage of timely serviced responses to requests*, that is the number of service provisioning requests completed within a defined time period over the total number of service provisioning requests.

Response time is the time period between a client request event and a service response event. The service metrics used to constrain the response time may return either an *average time* or a *maximum time*, given a particular form of request.

Capacity is the maximum amount of a resource used by a service. It includes the *service throughput metric*, namely the minimum number of requests that will be processed by a service in a stated time period. If there is no extra resources provided, the more resources a service requires, the lower the service throughput will be.

Several factors contribute to the quality level of a service. They can be classified as *internal*, such as the available resources, code quality, or the computational complexity, and *external*. The latter ones are outside the direct control of the stakeholders and include, for example, network availability or the number of accesses/requests. The situation is metaphorically illustrated in Figure 1. Internal factors can principally be controlled (and, if so desired, be modified) at deployment time with techniques that either directly verify the code (static analysis) or with the help of an underlying mathematical model (model checking, simulation, etc.). Whenever the service implementation does not comply with the metric, the designer makes code modifications that eventually lead to compliance. A typical example of this is the analysis of the *resource capacity* of a service, which measures to which extent a critical resource is used. For instance, a static analysis technique can determine an upper bound on the resources needed by a service [2, 12]; if a service is deployed on an insufficient number of machines, then its response time increases, or it even becomes unavailable. Thus, internal factors can be expressed and analyzed inside a model, and integrated into the plans for the initial deployment of the service on the cloud.

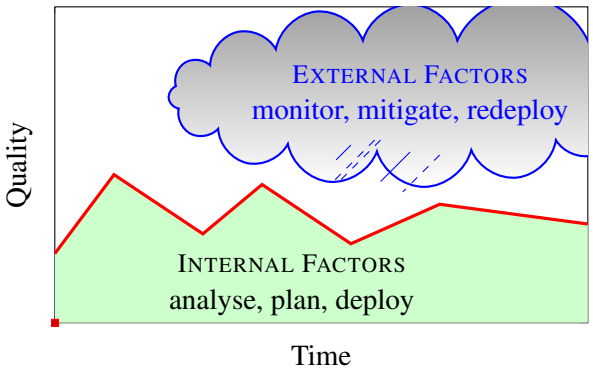

Figure 1: Service quality level over time and its influences

External factors can not be controlled or analyzed in advance, but they can be supervised by monitoring code that runs independently of the service implementation. Monitoring is always needed, as there are (performance) metrics that are affected by external factors, for example, hardware failures, which cannot be statically verified. In this case, neither the service implementation nor the resource configuration is at fault. However, a runtime monitor can still be helpful, for example, when it triggers a dynamic resource re-allocation that compensates for a faulty component. Thus, external factors cannot be expressed inside the model and must be monitored on the deployed service and then mitigated through (static or dynamic) redeployment of the service.

3 Model-Centric SLA Compliance: The Workflow

In Section 1, we argued that a model-centric representation of SLAs and services constitutes the basis for advanced tool support for cloud service configuration and deployment. In Section 2, we explained how the service quality is influenced by internal factors, to be addressed by compliance checking of service implementations against SLAs; and by external factors, to be mitigated with the help of runtime monitors. In Figure 2, we illustrate a workflow that realizes model-centric configuration of cloud services

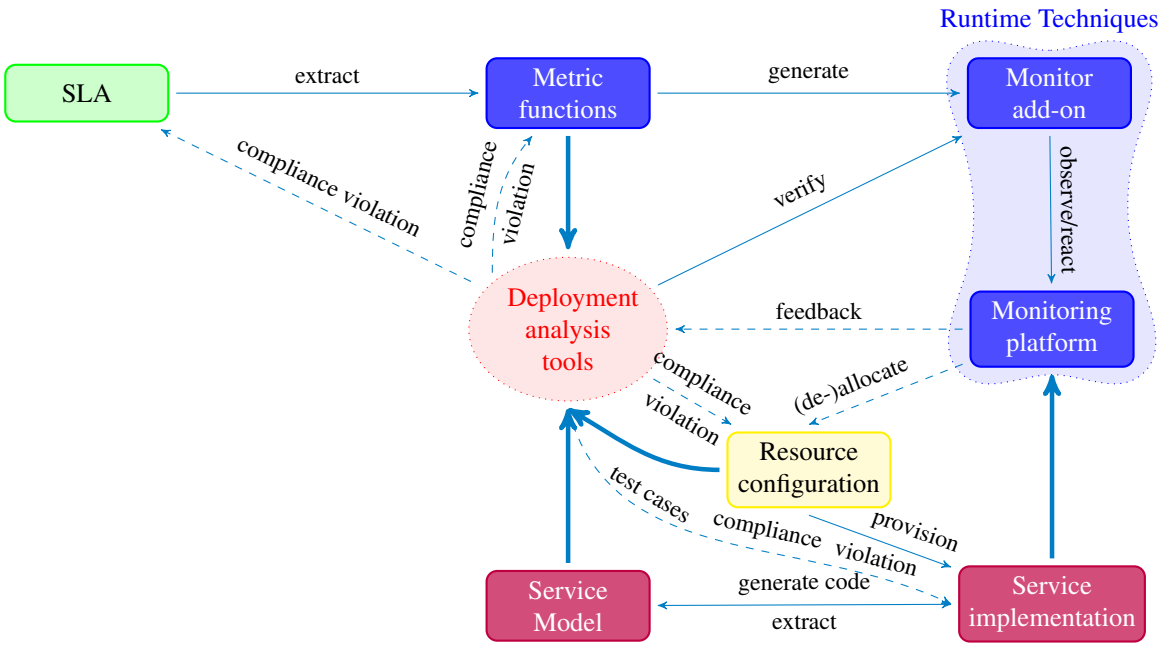

Figure 2: workflow of service configuration and deployment

to optimize SLA compliance under internal and external factors. The workflow is divided into *three* phases, namely, Negotiation, Observation and Reaction.

Static (deployment analysis) techniques play an important role in generating initial metrics and monitors that are used in run-time techniques. *Feedback loops*—represented by dashed arrows—to a previous phase of the analysis, represent modifications to the system that ensure, for example, continued compliance after external changes. Thick blue arrows indicate tool inputs.

Negotiation phase. This phase includes everything that might happen before signing a prospective SLA. At this stage the SLA metrics are set, so that the *service model*[2] can be verified against them. The *SLA* (top-left corner) is written in a machine-readable standard format (ISO 19086-2). Quality-of-service upper bounds, expressed in terms of *metric functions* over possible service measurements, are extracted from it. An initial *resource configuration* is defined over the types of resources that are allocated for the service (such as CPUs, memory, bandwidth, etc.). It can be specified manually, or it can be computed automatically by a solver that returns an optimal distribution of resources to service instances, given the knowledge of the initial instances to be deployed, their required computing resources and the resource costs [1]. At the same time, an executable *service model* is extracted from the components of the actual *service implementation*. The system is provisioned and deployed using the initial resource configuration.

A suite of *deployment analysis tools* now takes the three inputs (Metric functions, Service model, and Resource configuration) and produces responses as output to form a feasibility assessment. The tools can verify properties such as: upper bounds of the resource consumption (bandwidth, virtual machines, memory allocation, CPU processing cycles), liveness (deadlock-freedom) and safety (functionality). If the tools report that a service model violates an SLA constraint, then either the constraint can be relaxed or the resource allocation can be suitably enlarged during the negotiation phase (with a possible charge

[2]Phrases typeset in italics correspond to the artifacts in rectangle boxes in Figure 2.

for the client). The tools can also produce test cases that can be used to validate that the service model captures the implementation.

Thus, the feedback provided by the deployment analysis in terms of compliance violations guides the negotiation phase by discarding resource configurations that hinder the ability of the service to meet the SLA. Feedback may also be used to select a better metric bound, given the available resources. A third feedback loop may connect back to the program and allow changes in the code to be applied, so as to better adapt to the given SLA and available resources.

Once the configuration is approved by the deployment analysis tools—guaranteeing that, *in the absence of external factors*, the service implementation and the resource configuration comply to the SLA—the next phase can start.

Observation phase. The SLA is now signed and the service implementation is up and running. Factors under external control, such as the network infrastructure, may affect the behavior of the service in ways that could not be predicted statically. To supervise the service metrics we use a monitoring system, namely code external to the service that continuously monitors its execution and uses self-healing to repair resource failures or mitigate SLA violations.

The code of the *monitor add-ons* is automatically generated (or configured), starting from the specific metric functions they are intended to monitor. Static techniques may be used at this stage for proving the correctness of the generated code, i.e., that the monitors are observing the right property. Moreover, static techniques may be performed again at runtime, periodically, on the service model, to estimate the future behavior of the service in a next time window. Feedback from the monitoring system can significantly augment the precision of the analysis.

Reaction phase. System monitoring lets the service provider report violations of the agreed SLA via a *monitoring platform*. However, the ultimate goal for a provider is to *dynamically adapt* the resource configuration so that SLA violations remain under a penalty threshold while minimizing the cost of the running system. This can be achieved by adding appropriate resources to the service (e.g., scaling up the number and/or size of the virtual machines).

The observation phase takes measurements on services. If an SLA mismatch is observed by the monitoring platform, in the reaction phase, the number of allocated resources is increased or decreased accordingly. As was done for the initial configuration, also in this phase the modification of the resources assigned to objects can be done either manually or automatically. A solver computes which new resources are required and how new service instances should be distributed on these resources, or how old objects and resources that are no longer necessary should be un-deployed, given the knowledge of the current resource configuration and the new requirements indicated by the monitoring framework. Fully automatic dynamic elasticity can be obtained thanks to the combined use of the monitoring framework and the external deployment solver [14].

4 ABS: A Modeling Language and Tool Suite for Systems Deployed on the Cloud

ABS [23] is a modeling language which can be used to realize model-centric analysis of SLA compliance according to the workflow outlined in Section 3. We briefly summarize the main relevant features of ABS (see `www.abs-models.org`).

The ABS tool suite (abs-models.org/abs-tools)

Simulation tool for rapid model exploration and visualization

Deadlock analysis tool automatically checks that the model is deadlock free, focusing on the communication protocols in the model

Systematic testing tool provides a technique to eliminate redundant test cases for the concurrent execution of ABS models.

Test case generation tool for the automatic generation of test cases for concurrent objects in ABS

Termination and resource consumption tool automatically infers cost bounds for selected parts of the model for, e.g., execution cost or transmission data size

ABS Smart Deployer finds the optimal deployment of components on virtual machines, given a user specification of how components should be connected and of their resource requirements

Code generation tools enable rapid prototyping on real machines and integration with other programs, using Haskell or a Java library

Formal verification tool supports deductive analysis of behavioral properties, including communication traces

Monitoring framework for SLA metrics is used to automatically configure correct monitors for the deployed system and monitor the system at a high level, according to the SLA

Figure 3: ABS tool suite

ABS is a language for **A**bstract **B**ehavioral **S**pecification, which was designed for analyzability. It combines implementation-level specifications with verifiability, high-level design with executability, and formal semantics with practical usability. ABS is a concurrent, object-oriented modeling language built around a simple functional language with user-defined algebraic datatypes. Models are easy to understand and written in a familiar, Java-like syntax. In addition, ABS enables replaying a real-world log in the corresponding executable model through a so-called Model API [31]. It also explicitly supports the modeling of resource consumption on virtual machine instances [24]. Thus, the language allows analysis of *deployment decisions*, including a configurable model of cloud provisioning [17], and has been used for industrial case studies [3]. Both the *resource requirements* and *timing properties* of models can be expressed and analyzed, which makes it easy to compare deployment decisions at the level of models [33] by means of a large portfolio of analysis and deployment tools (see Figure 3).

5 Case Study

The company Fredhopper provided the Fredhopper Cloud Services[3] to offer search on a large product database to e-Commerce companies as services (SaaS) over the cloud computing infrastructure (IaaS). At the time of the case study, Fredhopper Cloud Services powered over 350 global retailers with more than 16 billion $ in online sales per year. A customer (service consumer) of Fredhopper is a web shop, and an end user is a visitor to the web shop.

Software services offered by Fredhopper are RESTful and deployed as *service instances* that accept

[3]Fredhopper was recently acquired and integrated into the ATTRAQT Group plc, see www.fredhopper.com

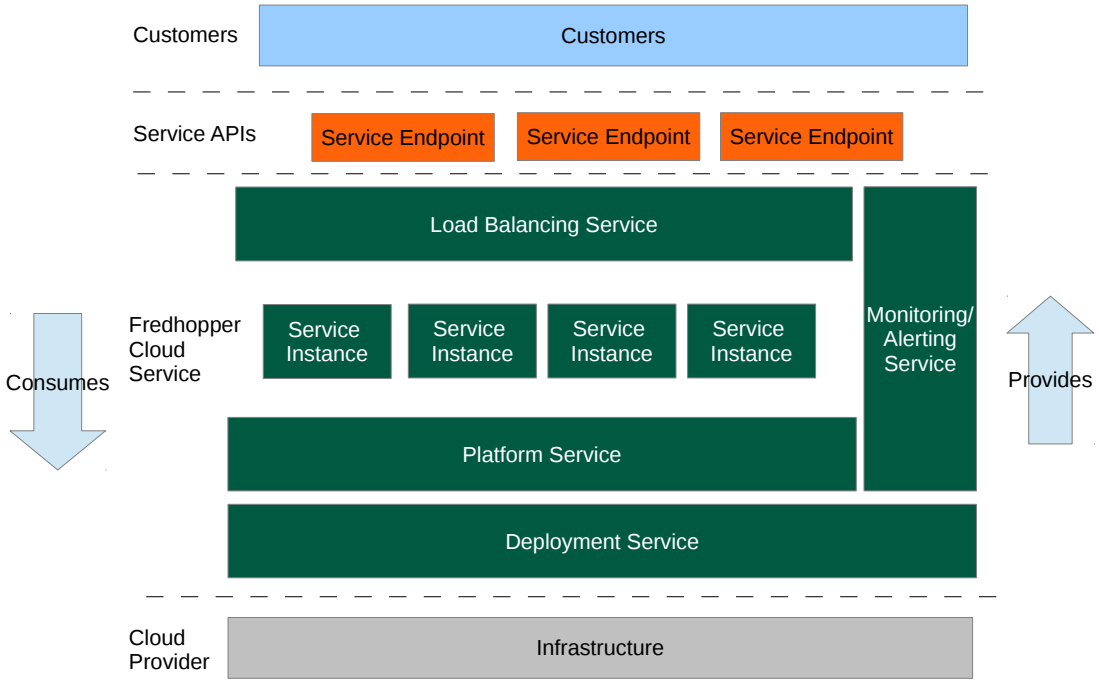

Figure 4: The architecture of the Fredhopper Cloud Services

connections over HTTP. Each instance offers the same service and is exposed via Load Balancer endpoints that distribute requests using a round-robin strategy over the service instances. Figure 4 shows a block diagram of the Fredhopper Cloud Services.

The number of requests can vary greatly over time, and typically depend on several factors. For instance, the time of the day where most of the end users are located plays an important role. Typical lows in demand are observed between 2 am and 5 am. Figure 5 shows a visualization of monitored data in Grafana, the visualization framework used by ABS. The top graph shows the number of query's completed per second (qps), the middle graph (current requests) shows the number of concurrently served requests averaged over all service instances of the customer, and the downmost graph visualizes the average CPU usage over time.

SLA. Peaks in demand of Fredhopper Cloud Services typically occur during promotions of the webshop or around Christmas. To ensure a high quality of service, web shops negotiate an aggressive Service Level Agreement (SLA) with Fredhopper. QoS attributes of interest include query latency (*response time*) and throughput (*queries per second*). The SLA negotiated with a customer could express, e.g., *service degradation* requirements as follows:

"Services must respond to queries in less than 200 milliseconds over 99.5% of the service up-time." (a)

An SLA specifies properties of service metric functions. For the example SLA, the service metric function is defined as the percentage of client requests which are processed in a "slow" manner, i.e., the percentage of queries slower than 200 milliseconds.

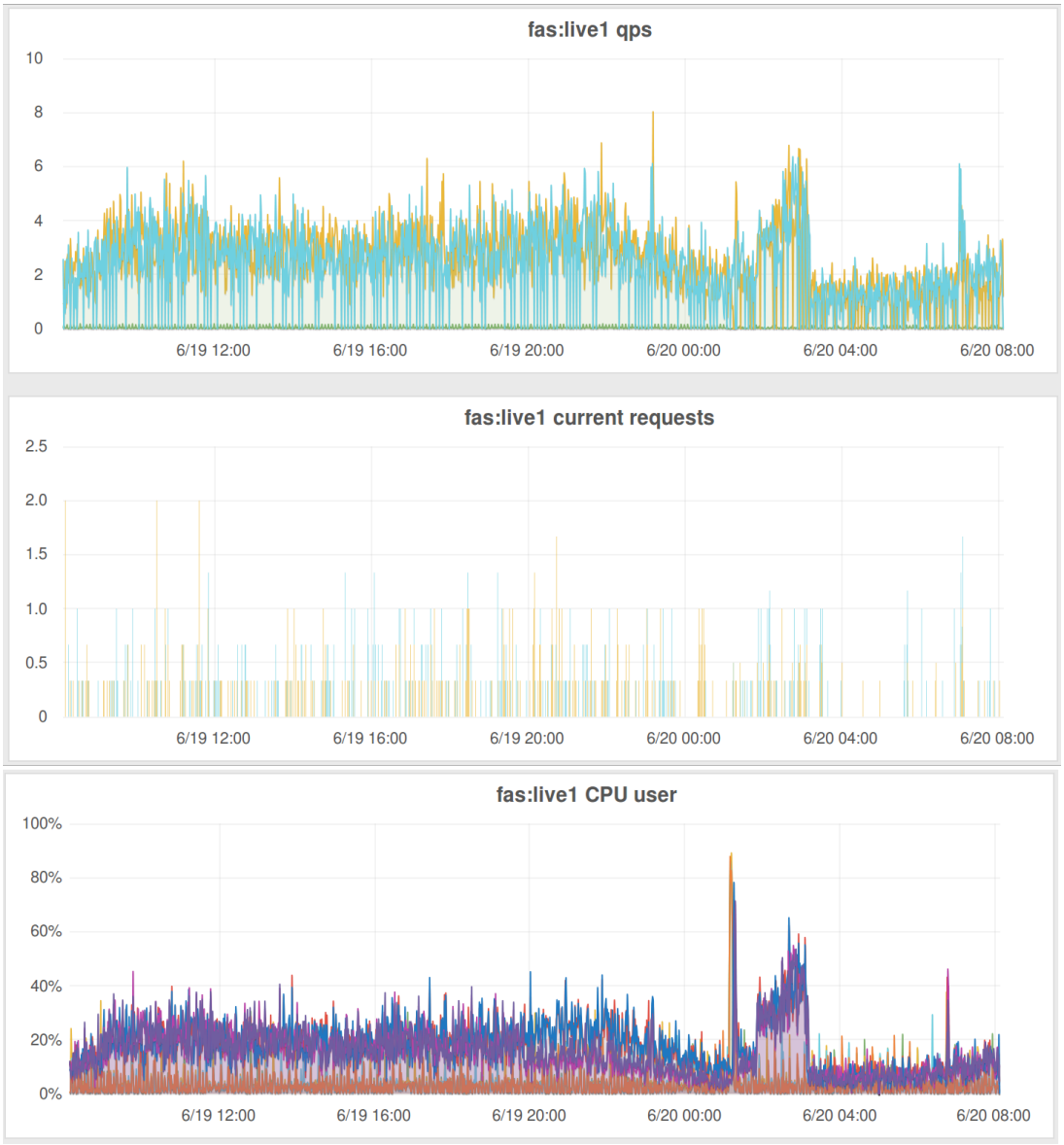

Figure 5: Visualization of metrics

Formalizing the service metric function. In ABS we formalize a service metric function using an attribute grammar as a partial mapping of *traces of events* to values. The events represent client interactions with an endpoint of an exposed service API. The mapping can be partial to detect and exclude illegal orderings of service invocations. The values correspond to different levels of the provided quality of service (QoS). The definition of the attributes is given for each production in the form of ABS code. This ensures that the attribute values are computable and sufficiently expressive (ABS is Turing complete) to capture general metrics. To establish whether a trace of service events is legal, and if so, what QoS level it should give rise to, the event trace is parsed according to the grammar. As such, grammars are a user-friendly formalism and are particularly well-suited for the specification of both data- and protocol-oriented properties of event traces. All regular grammars (with attributes) are currently sup-

ported. In such grammars, each production contains at most one non-terminal which must occur as the very last symbol in the production. Regular grammars can be parsed efficiently and incrementally, i.e. there is no need to re-parse the entire event trace.[4]

To formalize our service degradation metric, we identify the processing of a client request sent to an endpoint of the exposed query service API by an event

```
invoke(Time t, Rat procTime)
```

This event indicates that the request has been issued at time `t` and that it has processing time `procTime`. In our formalization, a *service view* identifies all the events that are relevant for a particular service metric and associates a name to each such event. These names will be used as grammar terminals. Since there can be many SLAs in a managed cloud service, and each SLA may concern a different subset of events from the service API(s), service views allow users to select only those events relevant for that SLA. For simplicity, we assume that we treat all requests in the same way. A view that simply identifies the invoke event as the only relevant event and associates the name "query" with this event, is expressed as follows:

```
view Degradation { invoke(Time t, Rat procTime) query }
```

Figure 6 contains a grammar[5] that computes as the main metric the percentage of slow queries "degradation". The string "fas.200" gives the name of the metric. The parameters of the invoke event, e.g., "procTime", are directly referred to in the grammar by their name and are used to compute the "degradation" percentage. The grammar further makes use of the auxiliary concepts "cnt", the total number of queries, and "slowCnt", the total number of "slow queries".

```
Pair<String, Rat> degradation = Pair("fas.200", 0);
Int cnt = 0;
Int slowCnt = 0;

S ::= query
      { cnt = cnt + 1;
        slowCnt = slowCnt +
                  case (procTime > 200) {True => 1;
                                         False => 0;};
        degradation = Pair("fas.200", 100 * slowCnt / cnt);
      }
      S
```

Figure 6: Grammar for Service Degradation

The resource-aware service. We create an abstract *service model* in ABS of the various services shown in Figure 4. A detailed model of the services is not necessary to exploit the ABS tool-set, it suffices to create a course-grained model that captures the Service APIs with stub implementations, as

[4]No such method is known for general context-free grammars, and it is unlikely to exist as this would give a procedure to parse them in linear time (in the size of the trace/sequence).

[5]With the usual semantics of CFG's, this grammar generates the empty language: no words (sequence of query's) of finite length are derivable. An epsilon production would have to be included. As a convenience to the user, we allow to omit epsilon productions by using the *prefix-closure* of the given grammar. For the given grammar, this is all finite sequences of query events.

we shall see. The service model can be refined with more detailed implementations whenever necessary to allow more detailed analyses. By way of example we show the model of a Query Service (Figure 7) and the Load Balancing Service (Figure 8). The load balancer distributes requests by means of a round robin policy and forwards them to query service instances. (Here, `current` is the number of service instances available in the current round and `services` the instances available in the next round, which may change dynamically depending on the scaling policy.) The actual service instances process the requests and return a response, e.g., a list of products that match the query in the case of Fredhopper Cloud Services. The given ABS model abstracts from a detailed implementation and focuses on execution cost by means of the statement `[Cost: cost] log = log+1`. The annotation `[Cost: cost]` is a measure of the estimated number of instructions. An initial value for it can be obtained by using the SACO tool [2] for cost analysis of models in the ABS tool suite [33], or by averaging execution times from real-world client logs produced from existing code.

```
class QueryServiceImpl (...) implements QueryService {
  ...
  Response invoke (Request request) {
    assert state == RUNNING;
    Int cost = cost(request);
    Int time = currentms();
    [Cost: cost] log = log + 1;
    time = currentms() - time;
    latency = max(latency, time);
    return success();
  }
}
```

Figure 7: Query Service

```
class LoadBalancerEndPointImpl
implements LoadBalancerEndPoint {
  Int log = 0;
  State state = STOP;
  List<QueryService> services = Nil;
  List<QueryService> current = Nil;
  ...
  Response invoke (Request request) {
    log = log + 1;
    assert state == RUNNING;
    if (current == Nil) { current = services; }
    EndPoint p = head(current);
    current = tail(current);
    return await p!invoke(request);
  }
  ...
}
```

Figure 8: Load Balancing Service Endpoint

Negotiation phase. Before we can accept a proposed SLA, we need to determine whether we can meet it with appropriate expense by deploying a number of `QueryServiceImpl` instances. We assume a setting where `QueryServiceImpl` instances run on virtual machines with an allocated *capacity* of `K` execution resources (CPU execution capacity, also called ECU).

Static analysis with SACO [2] yields `cost/K` as the total time required by the `invoke` method to reply to a single query. Therefore, we obtain $\texttt{(cost/K)} \leq \texttt{0.2}$ as a first bound from the SLA (a) on page 5. In order to meet the *service degradation* requirement expressed in the SLA (a), we need to determine the minimum number of resources in a configuration that complies with the SLA. For simplicity, we here assume a uniform arrival time for the requests, ignore the overhead of load balancing and distribution, and let `n` be the number of machines with `k` execution resources that we need. In this case, we know that $\texttt{(cost/(n} \times \texttt{k))} \leq \texttt{0.2}$, and we obtain $\texttt{(5} \times \texttt{cost/k)} \leq \texttt{n}$. For more complex scenarios (especially involving sub-services and synchronization), the ABS tool suite [33] comes in handy to help calculating the required number of machines.

This ignores the actual arrival time of requests as well as any *external* factors (see Figure 1) which may disrupt service execution. To ensure compliance to the service metrics under non-ideal conditions, we use a *monitoring platform*, external to the service, that continuously observes it.

The observation phase. The observation phase in our framework [7] consists of computing the value of the service metric function as specified by the grammar in Figure 6 from a given event trace. This involves parsing the event trace according to the grammar. From the grammar we automatically synthesize an ABS implementation of the corresponding parser. The use of grammars allows us to build on well-established and widely known parsing technology with optimal performance. Observations can also come from external systems which publish events to the model using an API over HTTP.

Given our *service model* in ABS, we can now replay a *real-world log* using this API, which generates corresponding invoke events for the model according to the specified timings in the logfile (see Figure 9). The resulting trace of invoke events is then parsed according to the grammar in order to compute the "degradation" service metric.

Reaction phase. Figure 10 shows a monitor corresponding to the grammar in Figure 6 for service degradation. Here `metricHist` contains the time-stamped history of metric values which is provided by the general ABS monitoring framework. The monitoring framework further integrates a powerful tool (the ABS Smart Deployer [14]) for the automated deployment of new service instances, based on high-level requirements of deployment configurations. A solver synthesizes a provisioning script executable in ABS that implements `DeployerIF` with appropriate scaling actions, such as allocating new virtual machines, and configuring and deploying additional service instances on these machines. This approach guarantees that the scaling actions preserve the deployment requirements.

The above ABS monitor reacts to the service degradation metrics (cf. Figure 6) by asking the deployer to scale up or down the service instances. For instance, if degradation is larger than `5/1000` (cf. SLA (a)), the method `scaleup` is invoked to get more service instances; and if degradation is less than `1/1000`, the method `scaleDown` is invoked to reduce service instances. Monitoring can be expensive; we must ensure that the monitoring does not degrade performance below the level stipulated in the SLA. Static analysis and simulation of the ABS model *together* with the monitor allows to analyze how the monitor effects the SLA *before* the system is deployed. ABS allows monitors to be deployed asynchronously and decoupled.

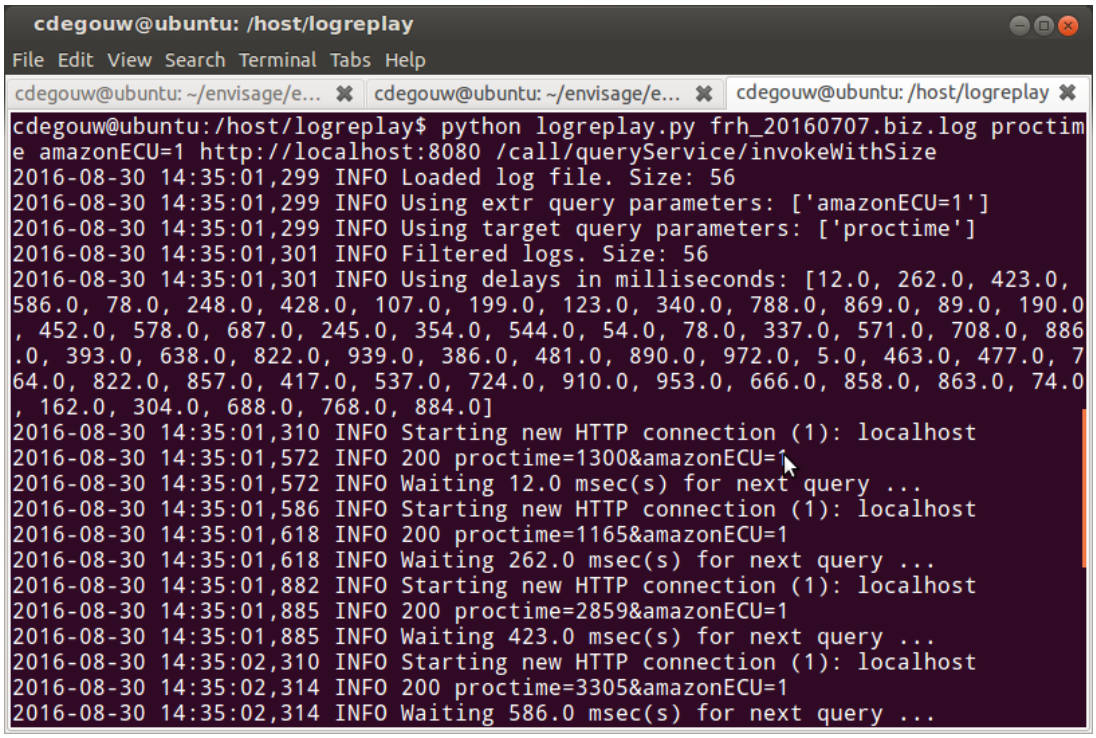

Figure 9: Log replay

```
Unit monitor (DeployerIF deployer) {
  Rat degradation = head(metricHist);
  if (degradation > 5/1000) {
      deployer.scaleUp();
  } else if (degradation < 1/1000) {
      deployer.scaleDown();
  }
}
```

Figure 10: Monitor for Service Degradation

6 Related Work

The methodology presented in this paper has been devised in the context of the EU project Envisage to provide efficient development of SLA-aware and scalable services, supported by highly automated analysis tools using formal methods.

While there are several proposals for formalizing SLAs [27,28], there is no study on how such SLAs can be used to both verify and monitor the service and upgrade it as necessary. In this respect, to the best of our knowledge, our technique that uses both static analysis and run-time analysis is original. Below we report the main related work on analysis, deployment and runtime monitoring of systems, focussing on work which is relevant in the context of cloud systems.

Static analysis estimates computational complexity (e.g. time) or resource usage of a given program and provides guarantees that the program will not exceed the inferred amount of resources [16,20]. Typically, such analyses apply to traditional sequential applications and, in order to use the above techniques

in our context, we had to study the non-trivial extension to concurrent active object systems [2,4,12,13]. The reader is pointed to the related work sections of these papers for a thorough comparison with the literature.

The problem of translating customer expectations into metrics to be measured at runtime is addressed, e.g., by WSLA [27] which introduces a framework to define and break down customer agreements into a technical description of SLAs and terms to be monitored. In [25], a method is proposed to translate the specification of SLA into a technical domain directed in SLA@SOI EU project. In the same project [9] defines terms such as availability, accessibility and throughput as notions of SLA. In [8] the authors describe how they introduce a function to decompose SLA terms into measurable factors and how to profile them. The problem of actually monitoring metrics and react to such observations, has been addressed e.g. in MONINA [22], which is a DSL with a monitoring architecture which supports certain mathematical optimization techniques, and offers two pre-defined parameters that can be used in monitoring to adapt the system: cost and capacity. Also a prototype implementation is available. Hogben and Pannetrat [21] examine the challenges of defining and measuring availability to support real-world service comparison and dispute resolution through SLAs.

In the context of cloud computing, the problem of automating application deployment has attracted a lot of attention and many system management tools exist [18,26,29,30]. These support the specification of deployment plans but not automatic distribution of the available computing resources to the services to be deployed. For these reasons, these tools can be seen as deployment engines to concretely execute deployment plans. Our approach to automatic optimal deployment is inspired by the Aeolus component model [10, 11] and the Zephyrus configuration optimizer [1]. The Aeolus model paved the way to formally reason about deployment and reconfiguration while Zephyrus is a configuration tool grounded on the Aeolus model. The use of Aeolus/Zephyrus guarantees that the synthesized deployment plans are optimal, in the sense that the total cost of the acquired resources is minimal. Optimality is guaranteed by taking into account the entire application architecture. This contrasts with the current auto-scaling technologies [5, 6, 19, 32], supporting the automatic increase or decrease of the number of instances of a service, when some conditions (e.g., CPU average load greater than 80%) are met. As auto-scaling monitors and scales single specific services, only local properties can be guaranteed.

7 Conclusion

This paper describes the analysis SLA compliance for services deployed on the cloud, by combing the formal models of the SLA and of the cloud service. Based on these two formal models, a detailed model-centric, tool-supported workflow is defined to obtain a configuration of cloud services in a semi-automated manner. The basis for our approach is the modeling language ABS that supports the modeling of deployment decisions on elastic infrastructure, and is the basis for a scalable monitoring framework for deployed services based on service metric functions. Using an industrial case study from Fredhopper Cloud Services, we show that our model-based approach can help to address the challenging questions posed in Section 1. Our specific combination of model-based tools is a good match for a DevOps methodology.

References

[1] Erika Ábrahám, Florian Corzilius, Einar Broch Johnsen, Gereon Kremer & Jacopo Mauro (2016): *Zephyrus2: On the Fly Deployment Optimization Using SMT and CP Technologies*. In Martin Fränzle,

Deepak Kapur & Naijun Zhan, editors: *SETTA*, *LNCS* 9984, pp. 229–245, doi:10.1007/978-3-319-47677-3.

[2] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, German Puebla & Guillermo Román-Díez (2014): *SACO: Static Analyzer for Concurrent Objects*. In Erika Ábrahám & Klaus Havelund, editors: *Tools and Algorithms for the Construction and Analysis of Systems, 20th Intl. Conf.*, *LNCS* 8413, Springer, Grenoble, France, pp. 562–567, doi:10.1007/978-3-642-54862-8.

[3] Elvira Albert, Frank S. de Boer, Reiner Hähnle, Einar Broch Johnsen, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa & Peter Y. H. Wong (2014): *Formal Modeling of Resource Management for Cloud Architectures: An Industrial Case Study using Real-Time ABS*. *Journal of Service-Oriented Computing and Applications* 8(4), pp. 323–339, doi:10.1007/s11761-013-0148-0.

[4] Elvira Albert, Jesús Correas & Guillermo Román-Díez (2016): *Resource Analysis of Distributed Systems*. In Erika Ábrahám, Marcello M. Bonsangue & Einar Broch Johnsen, editors: *Theory and Practice of Formal Methods*, *LNCS* 9660, Springer, pp. 33–46, doi:10.1007/978-3-319-30734-3.

[5] Apache: *Apache Mesos*. `http://mesos.apache.org/`.

[6] Amazon AWS: *Amazon CloudWatch*. `https://aws.amazon.com/cloudwatch/`.

[7] Frank S. de Boer & Stijn de Gouw (2014): *Combining Monitoring with Run-Time Assertion Checking*. In Marco Bernardo, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen & Ina Schaefer, editors: *Formal Methods for Executable Software Models, 14th Intl. School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM, Advanced Lectures*, *LNCS* 8483, Springer, Bertinoro, Italy, pp. 217–262, doi:10.1007/978-3-319-07317-0.

[8] Yuan Chen, Subu Iyer, Xue Liu, Dejan S. Milojicic & Akhil Sahai (2007): *SLA Decomposition: Translating Service Level Objectives to System Level Thresholds*. In: *Fourth International Conference on Autonomic Computing (ICAC), Jacksonville, Florida, USA*, IEEE Computer Society, p. 3, doi:10.1109/ICAC.2007.36.

[9] Marco Comuzzi, Constantinos Kotsokalis, George Spanoudakis & Ramin Yahyapour (2009): *Establishing and Monitoring SLAs in Complex Service Based Systems*. In: *IEEE International Conference on Web Services, ICWS, Los Angeles, CA, USA*, IEEE Computer Society, pp. 783–790, doi:10.1109/ICWS.2009.47.

[10] Roberto Di Cosmo, Michael Lienhardt, Jacopo Mauro, Stefano Zacchiroli, Gianluigi Zavattaro & Jakub Zwolakowski (2015): *Automatic Application Deployment in the Cloud: from Practice to Theory and Back*. In Luca Aceto & David de Frutos-Escrig, editors: *26th International Conference on Concurrency Theory, CONCUR, Madrid, Spain*, *LIPIcs* 42, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 1–16, doi:10.4230/LIPIcs.CONCUR.2015.1.

[11] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli & Gianluigi Zavattaro (2014): *Aeolus: A component model for the cloud*. *Inf. Comput.* 239, pp. 100–121, doi:10.1016/j.ic.2014.11.002.

[12] Abel Garcia, Cosimo Laneve & Michael Lienhardt (2017): *Static analysis of cloud elasticity*. *Sci. Comput. Program.* 147, pp. 27–53, doi:10.1016/j.scico.2017.03.008.

[13] Elena Giachino, Einar Broch Johnsen, Cosimo Laneve & Ka I Pun (2015): *Time Complexity of Concurrent Programs—A Technique Based on Behavioural Types*. In Christiano Braga & Peter Csaba Ölveczky, editors: *Formal Aspects of Component Software, 12th Intl. Conference, FACS, Niterói, Brazil, Revised Selected Papers*, *LNCS* 9539, Springer, pp. 199–216, doi:10.1007/978-3-319-28934-2_11.

[14] Stijn de Gouw, Jacopo Mauro, Behrooz Nobakht & Gianluigi Zavattaro (2016): *Declarative Elasticity in ABS*. In Marco Aiello, Einar Broch Johnsen, Schahram Dustdar & Ilche Georgievski, editors: *Proc. 5th IFIP WG 2.14 European Conference on Service-Oriented and Cloud Computing (ESOCC)*, *LNCS* 9846, Springer, pp. 118–134, doi:10.1007/978-3-319-44482-6_8.

[15] Cloud Select Industry Group (2014): *Cloud Service Level Agreement Standardisation Guidelines*. Developed as part of the Commission's European Cloud Strategy. Available at `http://ec.europa.eu/information_society/newsroom/cf/dae/document.cfm?action=display&doc_id=6138`.

[16] Sumit Gulwani, Krishna K. Mehra & Trishul M. Chilimbi (2009): *SPEED: precise and efficient static estimation of program computational complexity*. In Zhong Shao & Benjamin C. Pierce, editors: *Proc. 36th*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, Savannah, GA, USA, ACM, pp. 127–139, doi:10.1145/1480881.1480898.

[17] Reiner Hähnle & Einar Broch Johnsen (2015): *Designing Resource-Aware Cloud Applications*. *IEEE Computer* 48(6), pp. 72–75, doi:10.1109/MC.2015.172.

[18] Red Hat: *Ansible*. `https://www.ansible.com/`.

[19] Kelsey Hightower, Brendan Burns & Joe Beda (2017): *Kubernetes: Up and Running Dive into the Future of Infrastructure*, 1st edition. O'Reilly Media, Inc.

[20] Jan Hoffmann & Martin Hofmann (2010): *Amortized Resource Analysis with Polynomial Potential*. In Andrew D. Gordon, editor: *Programming Languages and Systems, 19th European Symposium on Programming, ESOP, Paphos, Cyprus*, *LNCS* 6012, Springer, pp. 287–306, doi:10.1007/978-3-642-11957-6_16.

[21] Giles Hogben & Alain Pannetrat (2013): *Mutant Apples: A Critical Examination of Cloud SLA Availability Definitions*. In: *IEEE 5th International Conference on Cloud Computing Technology and Science, CloudCom, Bristol, United Kingdom, Volume 1*, IEEE Computer Society, pp. 379–386, doi:10.1109/CloudCom.2013.56.

[22] Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner & Schahram Dustdar (2014): *Generic event-based monitoring and adaptation methodology for heterogeneous distributed systems*. *Softw., Pract. Exper.* 44(7), pp. 805–822, doi:10.1002/spe.2254.

[23] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte & Martin Steffen (2011): *ABS: A Core Language for Abstract Behavioral Specification*. In Bernhard K. Aichernig, Frank de Boer & Marcello M. Bonsangue, editors: *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, *LNCS* 6957, Springer, pp. 142–164, doi:10.1007/978-3-642-25271-6_8.

[24] Einar Broch Johnsen, Rudolf Schlatte & S. Lizeth Tapia Tarifa (2015): *Integrating deployment architectures and resource consumption in timed object-oriented models*. *Journal of Logical and Algebraic Methods in Programming* 84(1), pp. 67–91, doi:10.1016/j.jlamp.2014.07.001.

[25] Mahbub K., Spanoudakis G. & Tsigkritis T. (2011): *Translation of SLAs into monitoring specifications*. In P. Wieder, J. Butler, W. Teilmann & R. Yahyapour, editors: *Service Level Agreements for Cloud Computing*, Springer, pp. 79–101.

[26] Luke Kanies (2006): *Puppet: Next-generation configuration management*. *;login: the USENIX magazine* 31(1).

[27] Alexander Keller & Heiko Ludwig (2003): *The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services*. *J. Network Syst. Mgmt.* 11(1), pp. 57–81, doi:10.1023/A:1022445108617.

[28] D. Davide Lamanna, James Skene & Wolfgang Emmerich (2003): *SLAng: A Language for Defining Service Level Agreements*. In: *Proc. 9th IEEE Intl. Workshop on Future Trends of Distributed Computing Systems (FTDCS), San Juan, Puerto Rico*, IEEE Computer Society, pp. 100–106, doi:10.1109/FTDCS.2003.1204317.

[29] Opscode: *Chef*. `https://www.chef.io/chef/`.

[30] Puppet Labs: *Marionette Collective*. `http://docs.puppetlabs.com/mcollective/`.

[31] Rudolf Schlatte, Einar Broch Johnsen, Jacopo Mauro, Silvia Lizeth Tapia Tarifa & Ingrid Chieh Yu (2018): *Release the Beasts: When Formal Methods Meet Real World Data*. In Frank S. de Boer, Marcello M. Bonsangue & Jan Rutten, editors: *It's All About Coordination: Essays to Celebrate the Lifelong Scientific Achievements of Farhad Arbab*, *LNCS* 10865, Springer, pp. 107–121, doi:10.1007/978-3-319-90089-6_8.

[32] Docker Team: *Docker Swarm*. `https://docs.docker.com/engine/swarm/`.

[33] Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer & Rudolf Schlatte (2012): *The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems*. *STTT* 14(5), pp. 567–588, doi:10.1007/s10009-012-0250-1.