Project N°:	**FP7-610582**
Project Acronym:	**ENVISAGE**
Project Title:	**Engineering Virtualized Services**
Instrument:	**Collaborative Project**
Scheme:	**Information & Communication Technologies**

Deliverable D2.3.2
Monitoring add-ons (Final Report)

Date of document: T36

Start date of the project: **1st October 2013** Duration: **36 months**

Organisation name of lead contractor for this deliverable: **FRH**

Final version

STREP Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:
Monitoring add-ons (Final Report)

This document summarises deliverable D2.3.2 of project FP7-610582 (Envisage), a Collaborative Project supported by the 7th Framework Programme of the EC. within the Information & Communication Technologies scheme. Full information on this project is available online at `http://www.envisage-project.eu`.

This deliverable is a prototype and description of the final outcome of Task T2.3: an event-based, service-oriented monitoring and visualization framework for observing the performance and detecting the failures of services.

List of Authors

Stijn de Gouw (FRH)
Behrooz Nobakht (FRH)
Richard Bubel (TUD)
Frank de Boer (CWI)
Antonio Flores-Montoya (TUD)
Rudolf Schlatte (UIO)

Contents

Chapter 1

Introduction

The starting point of this deliverable is the formalization of SLA as a property of a service metric function, as given in [13]. Chapter 2 provides a short summary of the technical details underlying [13] as described in the initial report D2.3.1. A service metric function is defined by a mapping of event traces to values which indicate the different levels of the provided quality of service. These events represent client interactions with an endpoint of an exposed service API. The main new material of this deliverable is reported in the chapters 3 and 4, which covers the main objective and goals of the corresponding Task T2.3.

In Chapter 3 we give the details of how, in ABS, we can define a layered declarative generic framework to capture various monitoring concepts – from QoS and SLAs to lower-level metrics, metric policies, and listenable and billable events. At the heart of this framework is the use of *attribute grammars* for the declarative specification of service metric functions. Verifying, at runtime, that a generated event trace satisfies a corresponding SLA amounts to parsing it according to the attribute grammar. For this purpose we have developed a tool for the automatic synthesis of parsers by means of executable ABS.

The monitoring framework allows the formal development, and analysis, of monitors as executable ABS. In Chapter 4 we present the implementation of typical use cases of the framework according to the description of Task T2.3: unified visualisation and querying framework (Section 4.1), SLA based formal auto-scaling (Section 4.2), correctness of monitors (Section 4.3.1), analysis of resource consumption of monitors (Section 4.3.2), and connecting external systems in real-time (Section 4.4). Of particular interest is the development and use of an HTTP API (described in Appendix A) which allows using ABS monitors as data sources for visualization, and the definition of data sinks in ABS (exposed as HTTP endpoints) to stream external data.

The material of this deliverable has led to the following articles:

1. Formal Verification of Service Level Agreements Through Distributed Monitoring [16].

2. Run-time Deadlock Detection [5] – this paper studies the use of attribute grammars in ABS for the specification and run-time verification of event trace properties.

3. Declarative Elasticity in ABS [7]. This is a joint paper with T1.3. The contribution from the paper for this deliverable is the integration of the deployment synthesis into the monitoring framework.

Chapter 2

Service Level Agreements

This chapter provides the formal definition of SLA metrics. An SLA is a contract for a service agreed between a customer and a service provider. The contract specifies a target quality of service (QoS) value for certain aspects – metrics – of the service, such as its performance (a typical example is response time), and determines the circumstances under which a customer is eligible for a compensation, and the corresponding amounts, in case the targeted value is not achieved.

As services are typically exposed to the customer as endpoints of Service APIs, formally, we consider the service metrics used in an SLA to be a mapping of an event trace to a QoS value. Events correspond to a timestamped interaction of a customer with an end-point of an exposed service API.

2.1 Deployment Architecture

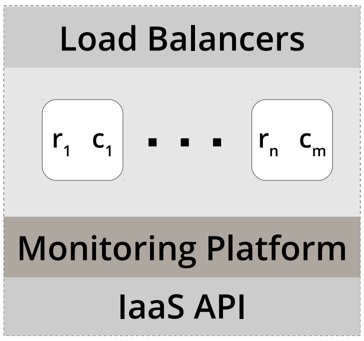


Figure 2.1: High-level Deployment Architecture: The diagram presents a logical presentation of a deployment environment. At the bottom, the deployment environment uses an external IaaS provider to provision physical resources. Potentially, every customer c_m might be allocated *multiple* resources r_i for their services. The services are delivered to customers through a layer of load balancers. The monitoring platform is a layer *orthogonal* to the environment that performs monitoring on resources and services.

Figure 2.1 depicts a high-level deployment architecture. In a deployment environment (e.g., "the cloud"), every server (e.g., r_n) from the IaaS provider is used for one or more service instances of a customer (e.g., c_m). For example, the Query Service API for a customer of SDL Fredhopper. Typically, multiple servers are allocated to a single customer. The number of servers allocated to a customer is not visible to the customer. Customer use a single endpoint – in the load balancer layer – to access their services.

The ultimate goal is to maintain the environment in such a way that customers, and their end users, experience the delivered services up to their expectations while minimizing the cost of the system. The first goal can be achieved by adding resources; however, this conflicts with the second goal since it increases the cost of the environment for the customer. In the rest if this chapter, we formalize the above intuitive notions as *service availability* and *service budget compliance*.

We present a distributed monitoring platform that aims at optimizing these service characteristics in a deployment environment. The monitoring platform works in two cyclic phases: *observation* and *reaction*. The observation phase measures different aspects of the services in the deployment environment, which is then used to calculate the corresponding levels of the service characteristics. In the reaction phase, if needed, a platform API is used to make the necessary changes to the deployment environment (e.g. adjust the number of allocated resources) to optimize the service characteristics.

2.2 Definitions and Assumptions

In this section we introduce the necessary notions, definitions and assumptions for the monitoring model.

Time T. In our framework time T is a universally shared clock, based on NTP[1], that is used by all elements of the system in the same way. T is discrete. We fix the unit of time to be *milliseconds*. This level of granularity means that between two consecutive milliseconds, the system is not observable. For example, we use the UTC time for all services, monitors and platform API. We refer to the current time by t_c.

Resource r. We denote by r a resource which provides computational power or storage.

Service s. A general abstraction of a service in the deployment environment is denoted by s. A service exposes an API that is accessible through a delivery layer, such as HTTP. In our example, a service is the Query API that is accessible through a single HTTP endpoint.

Monitoring Platform P. In our framework, a *monitoring platform P* is responsible on (de-)allocation of resources for computation or storage. We abstract from a specific implementation of the monitoring platform through the following API:

```
interface Platform {
  void allocate(Service s);
  void deallocate(Service s);
  Number getState(Service s);
  boolean verifyα(Service s);
  boolean verifyβ(Service s);
}
```

There is only *one* instance of P available. P internally uses an external infrastructure provisioning API to provide resources (e.g. AWS EC2). The platform provides a method `getState(Service s)` which returns the number of resources allocated to the given service s at the current time t_c.

Basic measurement. We let $\mu(s, r, t)$ be a function that produces a real number corresponding to a *single* monitoring check on a resource r allocated to service s at time t. For example, for SDL Fredhopper cloud services, a basic measurement is the number of completed queries up to the current time.

Service Metric. We let f_s be a function that aggregates a sequence of basic non-negative measurements to a single non-negative real value: $f_s : (\bigcup_n \mathbb{R}^n) \to \mathbb{R}$. For example, for SDL Fredhopper cloud services, the service metric function f_s calculates the average number of queries per second (qps) given a list of basic measurements.

[1] `https://tools.ietf.org/html/rfc1305`

Monitoring Window. It is a duration of time τ throughout which basic measurements for a service are taken.

Monitoring Measurement. It is a function that aggregates the basic measurements for a service over its resources in the last monitoring window. The last monitoring window is defined as $[t_c - \tau, t_c]$. To produce the monitoring measurement, f_s is applied. Formally:

$$\mu(s, r, \tau) = f_s(\langle \mu_i(s, r, t) \rangle_{i=0}^{\infty}) \text{ where } t \in [t_c - \tau, t_c]$$

where $\mu_i(s, r, t)$ is the i-th basic measurement of services s on resource r at time $t \in [t_c - \tau, t_c]$.

Unlimited Capacity. We assume that the external infrastructure provider is capable to provision an *unlimited* number of resources. In reality, every service provider, such as SDL Fredhopper, has a separate contract with an IaaS provider such as AWS. In these contracts, there cannot be a guarantee for *unlimited* capacity. We simplify this limitation by assuming that the platform API (IaaS layer abstraction) is capable to provision as many resources as requested.

Single Resource Type. To simplify reasoning, we assume that all resources are of the *same type*; i.e., they have the same computing power, memory, and I/O capacity. For example, if we are using AWS, we could choose the instance type `m1.large` for all the services. If a business delivers different types of services, it cannot avoid the fact that different services may require different *capabilities* from their underlying resources; i.e., one service might require high I/O throughput for its process whereas another might demand high parallelism support from the hardware. Such capability profiles are provisioned through different types of resources from a IaaS provider. For example, AWS offers[2] families of EC2 instance types each of which exposes different sets of capabilities in terms of computation power, I/O, and network. We simplify our analysis by assuming that there is a *single resource type* that is able to provide the necessary capabilities for all services.

Resource Initialization Time t_i. We assume that every resource r that is initialized is ready for use in at most t_i amount of time. The time t_i is bounded by a finite constant value. Initialization time can vary between different resource types. The initialization time is also part of the contract with the IaaS provider. Based on the resource type, when a resource is launched and initialized, it might take a different amount of time to be in a state that is *operational* and ready to be used. We simplify the property of initialization time of a resource to be a fixed constant for the single resource type.

2.3 Service Availability $\alpha(s, \tau, t_c)$

In this section we first define a few auxiliary definitions, and then define the notion of "service availability".

Service Capacity. We use $\kappa_\sigma(s, \tau) = \sum_{r \in \sigma(s)} \mu(s, r, \tau)$ to denote the capability of service s, which is the aggregated monitoring measurements of its resources $\sigma(s)$ over the monitoring window τ.

Agreement Expectation. We let $E(s, \tau, t_c)$ be the minimum number of requests that a customer expects to complete in a monitoring window τ. The agreement expectation depends on the current time t_c since it may change over time. For example, SDL Fredhopper customers expect a different qps during Christmas.

We define the **Availability of a Service** $\alpha(s, \tau, t_c)$ in every monitoring window τ as:

$$\alpha(s, \tau, t_c) = \frac{\kappa_\sigma(s, \tau)}{E(s, \tau, t_c)}$$

[2] `http://aws.amazon.com/ec2/instance-types/`

Capacity Tolerance. $\varepsilon_\alpha(s,\tau) \in [0,1]$ defines how much the service capacity $\kappa_\sigma(s,\tau)$ can deviate from $E(s,\tau,t_c)$ in every time span of duration τ.

Service Guarantee Time. We let t_G be the duration within which a customer expects the service availability to reach an acceptable value after a violation. Typically, t_G is an input parameter from the customer's contract.

Example Intuitively, $\alpha(s,\tau,t_c)$ represents the actual capability of a service s, over a time period τ, compared to the expectation $E(s,\tau,t_c)$. For values $\alpha(s,\tau,t_c) \ll 1-\varepsilon_\alpha(s,\tau)$ the resource for service s are at "under-capacity", while for values $\alpha(s,\tau,t_c) \gg 1+\varepsilon_\alpha(s,\tau)$ there is "over-capacity". The goal is to optimize $\alpha(s,\tau,t_c)$ towards 1.

For example, if we expect a query service to be able to complete 10 queries per second, and let the monitoring window τ be 5 minutes, then $E(s,\tau,t_c) = 10 \times 60 \times 5 = 3000$. Now suppose we allocate only one resource to the service, and measure the service during a single monitoring window τ and find out that $\mu(s,r,\tau) = 2900$. Then $\alpha(s,\tau,t_c) = \frac{2900}{3000} = 0.966$. If we have $\varepsilon_\alpha(s,\tau) = 0.03$, this means that service s is under-capacity because $\alpha(s,\tau,t_c) < 1-\varepsilon_\alpha(s,\tau)$.

2.4 Budget Compliance $\beta(s,\tau)$

In this section we first provide a few auxiliary definitions, and then present a formal definition for "service budget compliance".

Resource Cost. We let $€(r,\tau) \in \mathbb{R}$ be the cost of resource r in a monitoring window τ, which is determined by a fixed resource cost per time unit.

Service Cost. We let $€_\sigma(s,\tau) \in \mathbb{R}^+$ be the cost of a service s in a monitoring window τ, and defined as $€_\sigma(s,\tau) = \sum_{r\in\sigma(s)} €(r,\tau)$.

Service Budget. we let $B(s,\tau)$ be an upper bound on the expected cost of a service in the time span τ. Intuitively, $B(s,\tau)$ is the allowed budget that can be spent for service s over the time span τ. The service budget is typically chosen to be fixed over any time span τ.

We define the **Service Budget Compliance** $\beta(s,\tau)$ that, intuitively, represents how a service complies with its allocated budget as:

$$\beta(s,\tau) = \frac{€_\sigma(s,\tau)}{B(s,\tau)}$$

Budget Tolerance. $\varepsilon_\beta(s,\tau) \in [0,1]$ defines how much the service cost $€(s,\tau)$ can deviate from $B(s,\tau)$ in every time span of duration τ.

Service Guarantee Time. t_G is similar to that defined for service availability.

Example Assume every resource on the environment costs 1 (e.g., in €) per hour. Suppose we set a budget to 1.5 per hour for every service, allocate *one* resource to the service and define a monitoring window τ to be 5 minutes. Every hour has 12 monitoring windows. This means that each resource costs $€(r,\tau) = \frac{1}{12} \approx 0.08$ per monitoring window. Since there is only one resource, the service cost is $€(s,\tau) = \sum_{r\in\sigma(s)} €(s,\tau) \approx 0.08$ per monitoring window. On the other hand, if we calculate the budget for one monitoring window, we have $B(s,\tau) = \frac{1.5}{12} = 0.125$ per monitoring window. This yields budget compliance as $\beta(s,\tau) = \frac{0.08}{0.125} = 0.64$.

Chapter 3

General Monitoring Framework in ABS

In Chapter 2 we described how an SLA can be formalized as properties of service metric functions. In this chapter we describe how we monitor such an SLA.

In Chapter 2 we used the notion of service metric functions in an abstract way, i.e., without addressing how we can define them. In Section 3.1 we show how grammars are a convenient formalism to define service metric functions, and illustrate our method with a simple running example. Our approach has several desirable properties:

- It provides a systematic way to define service metric functions at the abstraction level of the Service APIs, at which SLAs are formulated;
- It ensures that metric functions can be efficiently evaluated;
- The generated monitors are amenable to further analyses, such as correctness or resource analysis.

In Section 3.2 we describe the general architecture of the monitoring framework. In Section 3.3 we discuss the implementation of the SAGA tool, which automatically generates executable, event-based ABS monitors from the declarative grammars.

3.1 Attribute Grammars

In [6], we enhanced run-time assertion checking with general attribute grammars [15] for the specification and verification of properties of *histories*, e.g., sequences of method calls and returns. Attribute grammars allow the specification in a declarative manner of both data- and protocol-oriented properties. Checking whether these properties hold amounts to parsing the given history.

Here we formalize a service metric function, which is a mapping of event traces to values indicating the different levels of the provided QoS, by (right) regular grammars with inherited attributes. These events indicate client interactions with an endpoint of an exposed service API. Right regular grammars consist of production rules of the form $N ::= a\ M$, where N and M are non-terminals and a is a terminal. In general, the event trace is updated and parsed again after each new event. Restricting to (right) regular grammars with inherited attributes optimizes the parsing of the event trace both with respect to time and memory. This is because we do not need to store the entire event trace, which in turn allows the automated derivation of a finite state automaton that can be used for the parsing process.

In what follows, we explain the main concepts of this use of attribute grammars, and the general monitoring framework in ABS, by means of the following example taken from the Fredhopper case study. The services offered by FRH are exposed at endpoints. In practice, these services are implemented to be RESTful and accept connections over HTTP. Software services are deployed as *service instances.* Each instance offers the same service, and is exposed via Load Balancer endpoints that distribute requests over the service instances.

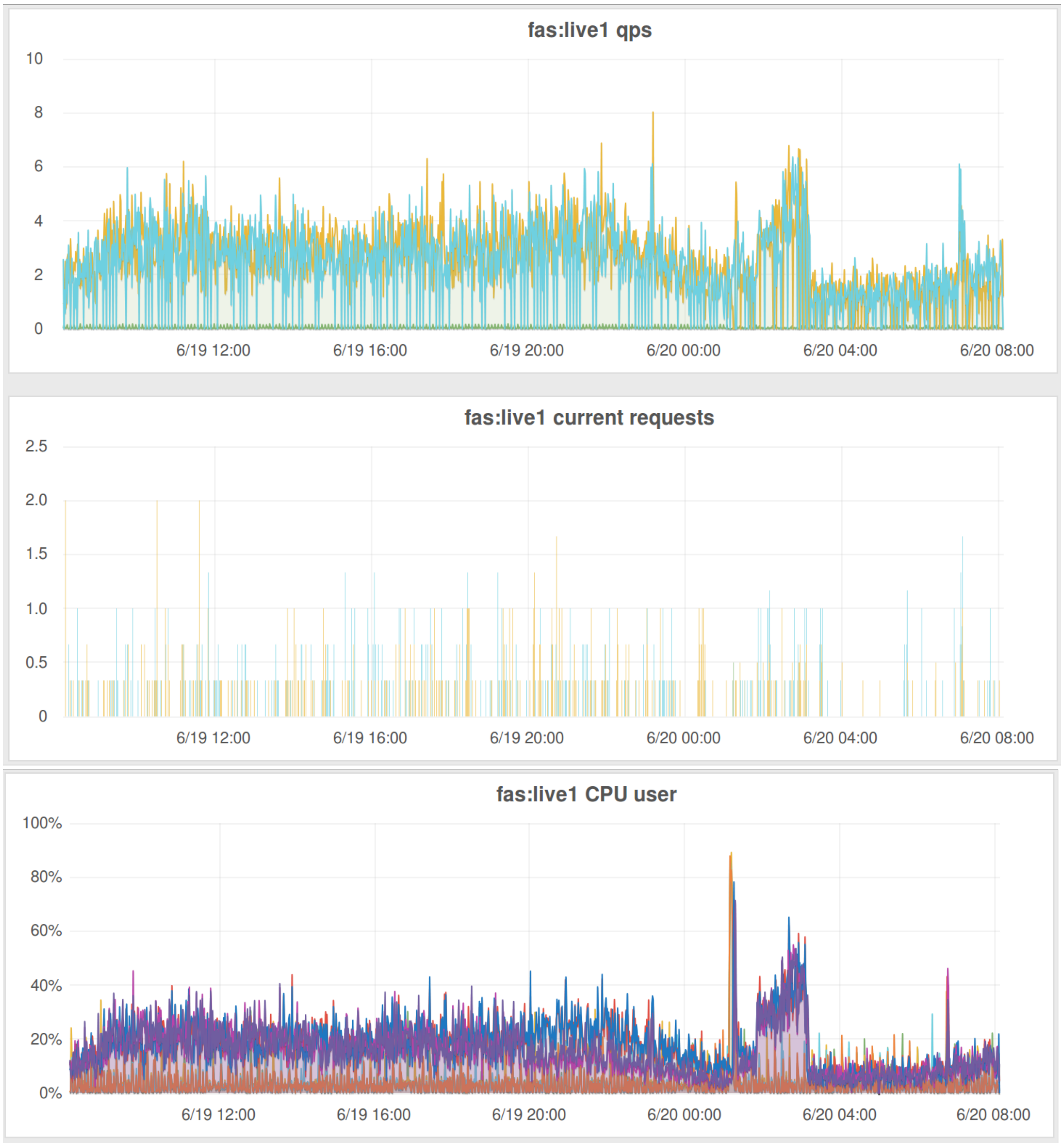


Figure 3.1: Visualization of metrics

The number of requests can vary greatly over time, and typically depends on several factors. For instance, the time of the day in the time zone where most of the end users are located plays an important role. Typical lows in demand are observed between 2am and 5am. A visualization of monitored data in Grafana (the visualization framework used by ABS) is depicted in Figure 3.1.

Peaks in Fredhopper Cloud Services typically occur during promotions of the shop or around Christmas. To ensure a high QoS, web shops negotiate an aggressive SLA with FRH. 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 maintain 100 queries per second with less than 200 milliseconds of response time over 99.5% of the service uptime, and 99.9% with less than 500 milliseconds."

An SLA specifies properties of service metric functions. In this case, the service metric function is defined in terms of the percentage of client requests which are processed in a "slow" manner. For the example SLA, the service degradation is concerned with the percentage of queries slower than 200 (and 500) milliseconds.

Suppose we want to formalize our service degradation metric. We identify the processing of a client request, that interacts with an endpoint of an exposed service API, by an event of the form

`invoke(Time t, Rat procTime)`

This event indicates that the request has been issued at time `t`, and that it has a processing time `procTime`. In our formalization, a service view identifies all events that are relevant for a particular service metric and associates a name to each such event. A view, which simply identifies the invoke event as the only relevant event and associates it with the name "query", is expressed as follows:

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

Figure 3.2 includes the grammar which computes, as the main metric, the percentage of slow queries `degradation`. The string `fas.live.200ms` gives a name to 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 `degradation`. The grammar makes use of the auxiliary variables `cnt` and `slowCnt` as well.

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

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

Figure 3.2: Grammar for Service Degradation

Note that since we restrict to regular grammars with inherited attributes, we do not need to explicitly state the association between the (non-terminal) attributes and the non-terminals, on the one hand, and the association between the built-in attributes and the terminal, on the other hand.

A monitor corresponding to the above grammar for service degradation is depicted in Figure 3.3. Here `metricHist` contains the time-stamped history of metric values, which is provided by the general monitoring framework. The monitoring framework further integrates a powerful tool (the ABS Smart Deployer) for the automated deployment of new service instances, that is based on high-level requirements of deployment configurations. A solver synthesizes an ABS class implementing `DeployerIF` with appropriate scaling actions.

The ABS monitor of Figure 3.3 reacts to the metric values (available in `metricHist`) by asking the deployer to scale up or down the service instances. Note that running a monitor can be expensive, thus, great care must be taken so it does not itself degrade performance below the level stipulated in the SLA. Static analysis and simulation of the ABS model, *together* with the monitor, allows to analyze the effect of the monitor on the SLA *before* the system is deployed. ABS allows monitors to be deployed asynchronously and decoupled.

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

Figure 3.3: Monitor for Service Degradation

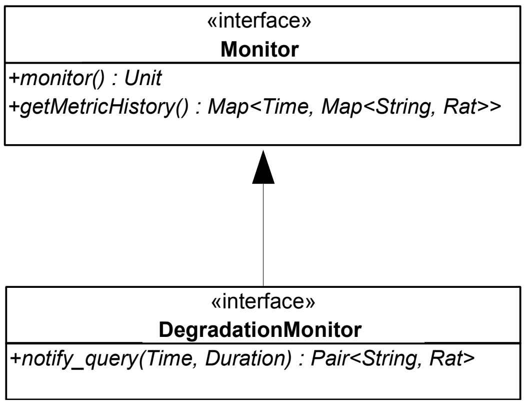

«interface»
Monitor
+monitor() : Unit
+getMetricHistory() : Map<Time, Map<String, Rat>>
«interface»
DegradationMonitor
+notify_query(Time, Duration) : Pair<String, Rat>

Figure 3.4: Monitor architecture

3.2 Monitoring Architecture

The architecture of the generated monitors consists of a general part shared by all monitors, and a specific part dedicated to each monitor. The generic part is captured by the `Monitor` interface depicted in Figure 3.4. It has two methods:

1. The `monitor()` method. The implementation of this method is provided by the user, and should take corrective actions to improve the value of the service metric function. This typically involves upscaling or downscaling. To determine the appropriate action, the user implementing this method can use the history of the metric values.

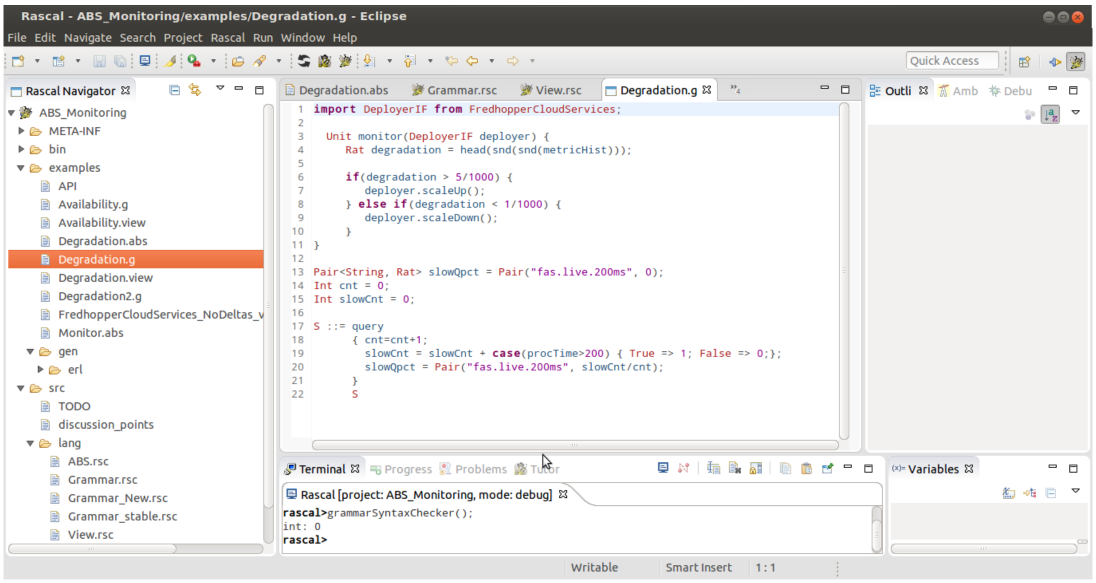
2. The `getMetricHistory()` method. The implementation of this method is automatically generated by the tooling (see Section 3.3). It returns the history of metric values over time, i.e., a Map that, given a time-stamp and metric name, it returns the value of that metric at that point in time.

A simple use case for such monitors is to call the `monitor()` method periodically with a user-specified delay. This ensures that appropriate actions, to improve the metric, are taken at well-defined time points.

The specific part of the monitoring architecture defines a method for each event, listed in a service view, to update the history. For example, for the Service Degradation view, there is only one event ("query") which has `Time t` and `Duration procTime` as attributes, so the interface of the `DegradationMonitor` has one additional method, besides the two general methods described above, `notify_query(Time, Duration)` that updates the history whenever it is notified of the "query" event. The implementation of this method is generated fully automatically from the view and grammar: the view determines its signature, and the grammar determines its body of (it corresponds to the action in the grammar production in which the event appears).

3.3 Tooling

We have implemented a tool SAGA that, given a service view and a grammar that includes the implementation of the `monitor()` method, fully automatically generates executable ABS monitors implementing the architecture discussed in Section 3.2. SAGA offers on the fly syntax highlighting and syntax checking of service views and grammars (see Figure 3.5). SAGA itself is implemented as a meta-program of ≈ 400 lines of code in the language Rascal [14].



```
import DeployerIF from FredhopperCloudServices;

  Unit monitor(DeployerIF deployer) {
     Rat degradation = head(snd(snd(metricHist)));

     if(degradation > 5/1000) {
        deployer.scaleUp();
     } else if(degradation < 1/1000) {
        deployer.scaleDown();
     }
}

Pair<String, Rat> slowQpct = Pair("fas.live.200ms", 0);
Int cnt = 0;
Int slowCnt = 0;

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

```
rascal>grammarSyntaxChecker();
int: 0
rascal>
```

Figure 3.5: Grammar Syntax Checking and Highlighting

As noted in Section 3.1, SAGA restricts to supporting regular grammars. From a regular grammar, SAGA synthesizes ABS code for a finite automaton with actions. The non-terminals become states of the automaton and the grammar terminals correspond to the automaton input symbols. Whenever the monitor is notified with a new event, it determines the new value of the attributes by taking a single transition in the automaton and executing the corresponding action given in the grammar. This means that there is no need to process or store the full event trace: we can calculate the new attribute value incrementally (on the fly) according to the grammar, by taking the step in the automaton and executing the corresponding action. No algorithm for incremental parsing in this sense is known for general context-free grammars (and is unlikely to exist, as this would yield a parsing algorithm with linear time complexity).

The restriction to regular grammars provides a good trade-off between expressiveness and performance: while such grammars are still Turing-complete in principle – one can use arbitrary ABS code in the actions if needed – in practice the actions are often simple (no loops, each action taking constant-time) thus giving linear resource consumption in the size of the trace. Section 4.3.2 provides a formal resource analysis.

Chapter 4

Use cases

We have implemented and applied a variety of use cases for the monitoring framework. In addition, an API that allows interacting with ABS models over HTTP was specified and implemented. This API allows, for example, invoking methods of an ABS model over HTTP. The API is attached in Appendix A, it forms an important building block for two of the use cases.

- **Visualization** (Section 4.1). We implemented a framework for visualizing, configuring and querying user-defined metrics from monitors on top of the well-established Grafana[1] and InfluxDB[2] platforms. Each monitor acts as a data-source by exposing its history of metric values over time as an HTTP endpoint.
- **Auto-scaling** (Section 4.2). We developed a rigorous basis for auto-scaling at the level of the SLA.
- **Formal analysis** (Section 4.3). We applied some ABS analyses on monitors to establish their correctness and determine their resource consumption.
- **Connecting external systems** (Section 4.4). We show how to connect external systems, by exposing methods in the ABS model as HTTP endpoints that act as data-sinks. This allows to replay real-world log files, or feed in data from external monitors of heterogeneous sources in real-time.

4.1 Visualisation

We implemented a visualization and query framework on top of Grafana (to visualize metrics) and InfluxDB (to store and query metric data) for user-defined monitors, generated using the monitoring framework described in Chapter 3. The monitoring framework automatically records the value of the service metric function over time as a time-series, and exposes the corresponding `getMetricHistory` method as an HTTP endpoint using the API described in Appendix A. We implemented a tool that periodically retrieves new metric values by invoking `getMetricHistory` at certain (user-specified) time intervals, stores them in an InfluxDB database and visualizes the metric in real-time using Grafana.

Figures 4.1 shows the Service Degradation metric, formalized in Section 3.1, in Grafana. Colors, time ranges and zooming are configurable through the interface shown in the figure. The metric data is generated from (the Service Degradation monitor in) the `FRH` ABS model, triggered with input data from a real-world log file (see Section 4.4). At start-up, the query processing is slow due to execution of background initialization processes. Therefore, the Service degradation, which measures the percentage of queries with `proctime>200 ms`, is very high (100% at the very beginning) as shown in Figure 4.1. Once these processes complete, the metric improves as Figure 4.2 shows.

Our framework allows users to set the starting time for the visualization in order to view metrics (as a graph) at different dates. Moreover, the user can define how ABS time is mapped to real time, e.g., the

[1] `http://grafana.org/`
[2] `https://influxdata.com/`

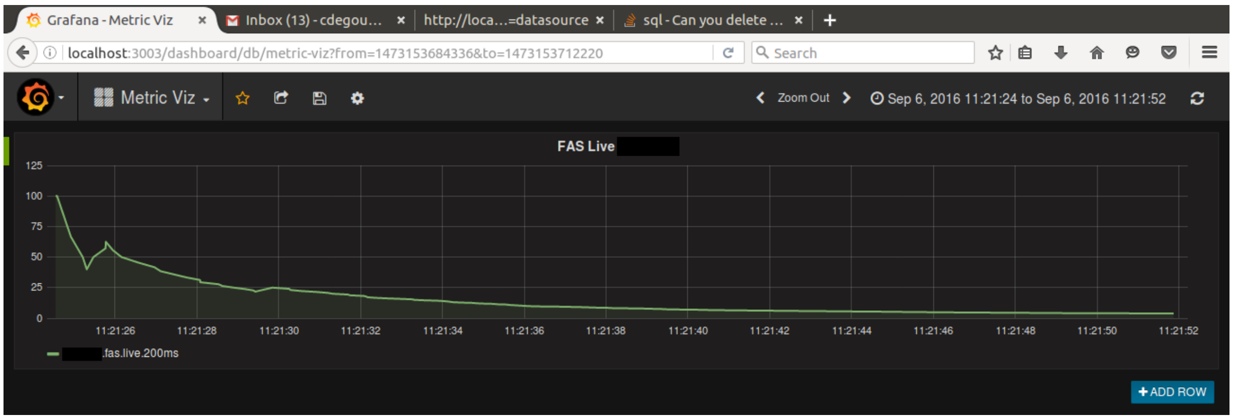


Figure 4.1: Service Degradation visualization at system start in Grafana

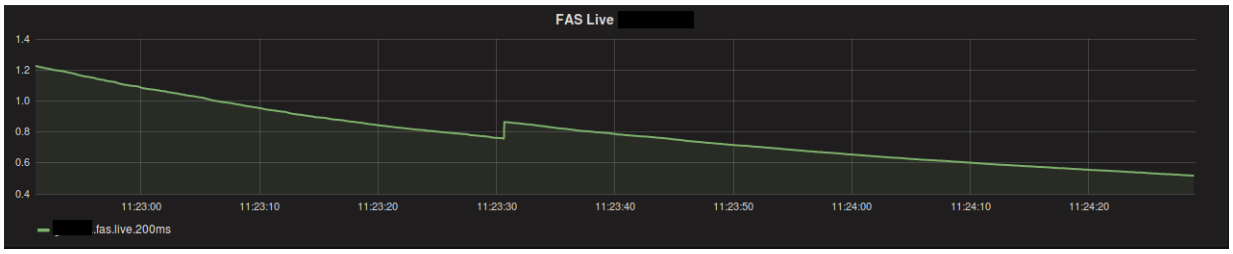


Figure 4.2: Service Degradation progression

user can specify that each clock cycle in the ABS model takes a certain number of milliseconds in real-time (the appropriate factor to use depends on the design choices for the ABS model). Our approach is back-end independent, in the sense that it supports all ABS back-ends that implement the API described in Appendix A.

4.2 Auto-scaling

In Chapter 3 we decribed how to generate executable monitors for user-defined, high-level metrics that are at the same abstraction level as the SLA. The `monitor()` method defined inside these generated monitors takes (if necessary) corrective actions, such as scaling or raising alerts to Cloud operators, aiming at improving the metric. This approach provides a rigorous basis for auto-scaling which is directly based on the SLA, instead of predefined lower level metrics such as CPU load which have no direct relation to the SLA.

The question arises in this context are: *how* to scale? How many new Service instances, and of what kind, should be deployed? On what virtual machines should they be deployed? How should they be configured? And how should we connect different service instances? For example, a query server in the US should be connected to a load balancer in US, and not to one in another region.

The above question are exactly the problems addressed by the SmartDeployer developed in T1.3. Therefore, the challenge was to find a way to integrate the SmartDeployer into the monitoring framework. In joint work with T1.3, we successfully developed a systematic approach to achieve this. We identified a simple API for a generic scaling deployer (see Figure 4.3), which should be used by the `monitor()` method whenever there is a need to scale.

With this API at hand, from high-level deployment requirements specifying the dynamic deployment constraints that should be satisfied, the SmartDeployer synthesizes a Deployer class with methods used to implement the above interface. It (dis)connects, configures and (un)deploys onto the correct virtual machines the Service instances to be created (when scaling up) or destroyed (when scaling down) fully automatically. The integration of the SmartDeployer into the monitors is realized by passing an instance of the deployment

```
interface DeployerIF {
        Unit scaleUp();
        Unit scaleDown();
}
```

Figure 4.3: API for Scaling Deployer

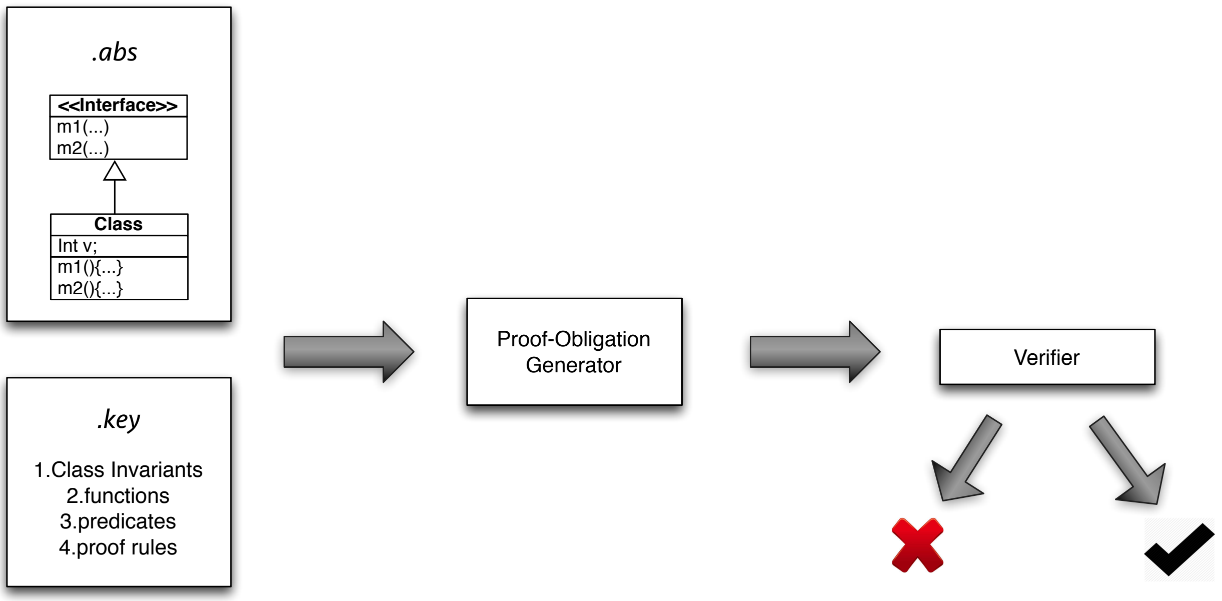


Figure 4.4: KeY ABS workflow

class to the monitor. The application of this approach to the FRH case study is described in D4.3.3.

4.3 Formal Analysis

The framework in Chapter 3 generates monitors in ordinary ABS, which makes them amenable to other formal analyses developed for ABS. In this section we explore the application of two such analyses using the Service Degradation monitor as a running example: verifying correctness of monitors with KeY-ABS (Section 4.3.1); and inferring resource consumption of monitors with CoFloCo and SACO (Section 4.3.2).

4.3.1 Functional Correctness

This section describes the efforts undertaken to ensure that the generated monitors for degradation detection and aversion are correct. By correctness we mean that i) a monitor detects when an SLA is threatened to be violated due to a degradation in response time; and ii) a monitor initiates the necessary steps to avert the violation.

Deductive Verification using KeY ABS

KeY ABS [8] is a deductive verification system that allows the functional and compositional verification of ABS programs. KeY ABS is a variant of the KeY verification system for Java [4]. We do not give a detailed description of the system and focus only on those parts, of the specification language and program logic, that are necessary to understand this section.

The verification work-flow (depicted in Figure 4.4) is as follows. The program and its specifications are loaded into KeY ABS. The user then selects the method and correctness guarantee, e.g., preservation of invariants, that should be verified for the selected method. The proof-obligation generator produces then a formula in the program logic of KeY ABS (called ABS Dynamic Logic, or short, ABSDL). This formula is then handed over to the verifier. If the formula can be shown valid, the method is guaranteed to be correct w.r.t. the chosen correctness guarantee.

Formulas in ABSDL contain programs directly as part of their syntax, for instance, the following formula

$$[\texttt{i = 2 * i + 1;}]\mathrm{even}(\texttt{i})$$

states that if the program `i = 2 * i + 1` terminates, then in its final state the value of `i` is even. The calculus used to verify such formulas is based on the symbolic execution paradigm. The program is first symbolically executed by applying the appropriate calculus rules until only first-order proof goals remain. These are then dealt with using classical first-order reasoning.

To be able to verify distributed and concurrent systems, ABSDL implements the concept of histories. Histories are sequences of system events created by method invocations and completions or object creation [10]. Concerning asynchronous method execution there are four different events:

1. The invocation event `InvocEv(callee, caller, future, methodLabel, arguments)` is generated once an asynchronous method invocation is executed. It records the `caller`, the `callee`, the `future` in which the method result will be put once computed, the invoked method (`methodLabel`) and the passed `arguments`.

2. While the invocation event marks the time in history when the invocation took place, the invocation reaction event `InvocREv(callee, caller, future, methodLabel, arguments)` marks the point in time, when the method is scheduled and actual execution of its body starts. The arguments of the event are identical to those of the invocation event.

3. Once an asynchronously called method completes its execution and returns, a completion event `compEv(callee, future, methodLabel, value)` is created, which records the object (`callee`) on which the method was invoked, the `future` (which now carries the result of the method invocation), the name of the method (`methodLabel`) and the result (`value`).

4. Finally, when a future gets queried for the result, the history is extended by a newly created completion reaction event `compREv(receiver, future, value)`. This event records the object (`receiver`) which queried the `future` and the result `value` carried by the future. As a future might be queried several times, this event is the only one that can occur repeatedly for the same future.

The future is the common element in all above events, and can be used to match all events related to a specific asynchronous method invocation during reasoning.

Histories, together with the concurrency model of ABS, allow us to reason about ABS code sequentially by considering one method at a time. The compositionality of ABSDL guarantees that the achieved correctness proof holds for any concurrent execution in any environment [1, 9].

Case Study

In the performed case study we verified an implementation of a degradation monitor implementing the following interface:

```
interface DegradationMonitorIf extends Monitor {
   Unit notify_query(Time t, Rat procTime);
   Unit monitor();
}
```

The method of interest is `monitor()`. This method is invoked regularly and checks the system state. In case it detects an over usage or under usage of resources it takes measures to allocate additional computing resources or to release existing resources. The generated monitor, which we verified implemented the following policy:

- If the most recent measurement shows that more than 0.5% of queries had an answer time more than 200ms, then additional resources have to be allocated by requesting them at the `Deploy` component.
- If the most recent measurement shows that less than 0.1% of queries had an answer time more than 200ms, then the `Deploy` component is asked to release computing resources.
- In all other case no action must be taken.

In a first step we had to specify the above policy as a class invariant that has to be established when creating the monitor and maintained by any method execution. Additional invariants were needed to ensure some properties related to the monitor's internal state, for instance, that the monitor must have a reference to an actual `Deploy` component (i.e., the reference must not be null).

The most important invariant for encoding the policy is shown below in a slightly beautified syntax:

```
autoScale : DegradationView.DegradationMonitorImpl {

 \forall Future fut;
  ( HistoryLabel::seqGet(history, seqLen(history)-1) =
                   compEv(self, fut, DegradationMonitorIf::monitor, Unit)
    -> \if (head(snd(head(
            List::select(heap, self, DegradationMonitorImpl::metricHist))))
           > rat(5,1000))
       \then (
          isInvocationEvFor(HistoryLabel::seqGet(history, seqLen(history)-2), Deploy::scaleUp)
       ) \else ( /* cases for scaling down and no action */ )
}
```

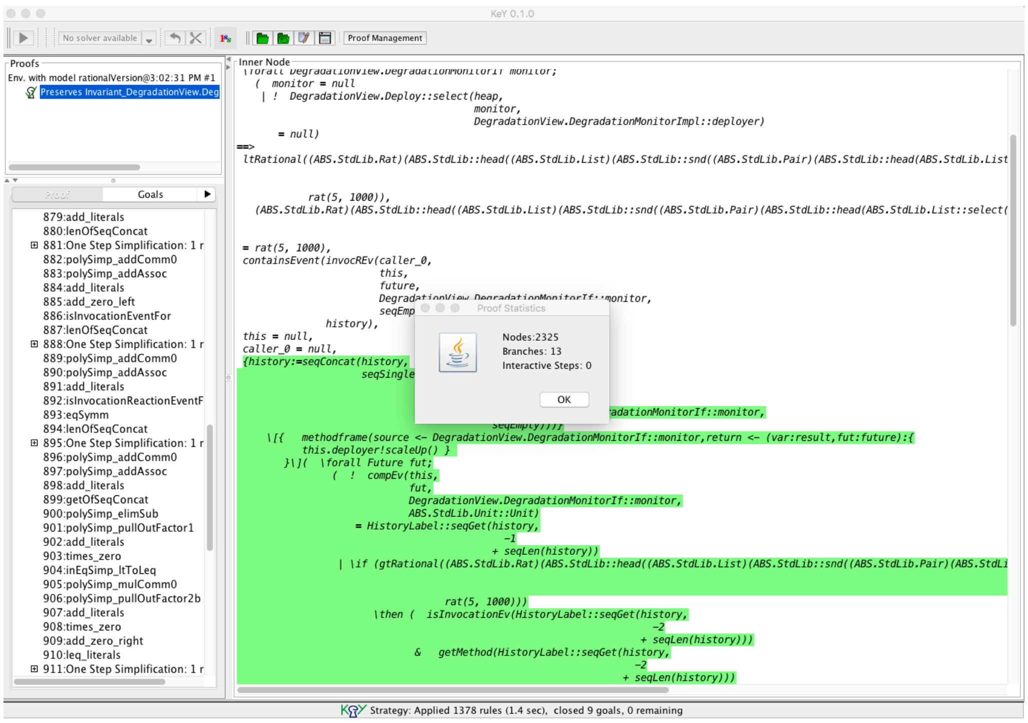
It expresses that at any time when there is a completion event in the history that belongs to an invocation of the `monitor` method, then, according to the metric, the allocation of additional computing resources (by calling `scaleUp`) must have been initiated (if the most recent measurement showed that more than 0.5% of the queries had a slower respond time than 200ms). Not shown in the excerpt, but deallocation of resources is specified in a similar manner; and in case that metric is in the green range, the invariant ensures that no action has been taken between the time when the execution of the method started and its completion. To match the invocation reaction event with the completed method event, the future identity is used (not shown in the excerpt).

The ABS model and the above specification were loaded into KeY ABS and the proof obligation *Preserves Class Invariant* for method `monitor()` was chosen. KeY ABS was able to verify successfully the correct behavior of the monitor w.r.t. to the specified policy. Except for providing the specification, the verification itself could be completed fully automatically in 2235 proof steps. The resulting proof tree consisted of 13 branches. The closed proof is depicted in Figure 4.5.

To enable verification of the case study, KeY ABS had to be extended with basic support for reasoning about rational numbers as up-to-then only integers were supported. The case study contributed thus to the further development of the tool and could successfully be completed.

4.3.2 Resource Analysis

An important aspect of monitors is the amount of overhead that they introduce at runtime. If this overhead is high, it can affect significantly the performance of the application rendering its use impractical. The objective of this section is to obtain CPU and memory consumption upper bounds of the generated monitors.



Env. with model rationalVersion@3:02:31 PM #1
Preserves Invariant_DegradationView.Deg

```
879:add_literals
880:lenOfSeqConcat
⊞ 881:One Step Simplification: 1 r
882:polySimp_addComm0
883:polySimp_addAssoc
884:add_literals
885:add_zero_left
886:isInvocationEventFor
887:lenOfSeqConcat
⊞ 888:One Step Simplification: 1 r
889:polySimp_addComm0
890:polySimp_addAssoc
891:add_literals
892:isInvocationReactionEventF
893:eqSymm
894:lenOfSeqConcat
⊞ 895:One Step Simplification: 1 r
896:polySimp_addComm0
897:polySimp_addAssoc
898:add_literals
899:getOfSeqConcat
900:polySimp_elimSub
901:polySimp_pullOutFactor1
902:add_literals
903:times_zero
904:inEqSimp_ltToLeq
905:polySimp_mulComm0
906:polySimp_pullOutFactor2b
907:add_literals
908:times_zero
909:add_zero_right
910:leq_literals
⊞ 911:One Step Simplification: 1 r
```

```
\forall DegradationView.DegradationMonitorIf monitor;
  (  monitor = null
   | !  DegradationView.Deploy::select(heap,
                                      monitor,
                                      DegradationView.DegradationMonitorImpl::deployer)
      = null)
==>
 ltRational((ABS.StdLib.Rat)(ABS.StdLib::head((ABS.StdLib.List)(ABS.StdLib::snd((ABS.StdLib.Pair)(ABS.StdLib::head(ABS.StdLib.List


           rat(5, 1000)),
  (ABS.StdLib.Rat)(ABS.StdLib::head((ABS.StdLib.List)(ABS.StdLib::snd((ABS.StdLib.Pair)(ABS.StdLib::head(ABS.StdLib.List::select(


 = rat(5, 1000),
 containsEvent(invocREv(caller_0,
                        this,
                        future,
                        DegradationView.DegradationMonitorIf::monitor,
                        seqEmp
              history),
 this = null,
 caller_0 = null,
 {history:=seqConcat(history,
                    seqSingle



                                          adationMonitorIf::monitor,
                                seqEmpty)))}
    \[{   methodframe(source <- DegradationView.DegradationMonitorIf::monitor,return <- (var:result,fut:future):{
          this.deployer!scaleUp() }
       }\](  \forall Future fut;
               (  !  compEv(this,
                            fut,
                            DegradationView.DegradationMonitorIf::monitor,
                            ABS.StdLib.Unit::Unit)
                  = HistoryLabel::seqGet(history,
                                          -1
                                        + seqLen(history))
                | \if (gtRational((ABS.StdLib.Rat)(ABS.StdLib::head((ABS.StdLib.List)(ABS.StdLib::snd((ABS.StdLib.Pair)(ABS.StdLi


                          rat(5, 1000)))
                    \then (  isInvocationEv(HistoryLabel::seqGet(history,
                                                                  -2
                                                                + seqLen(history)))
                          &   getMethod(HistoryLabel::seqGet(history,
                                                              -2
                                                            + seqLen(history)))
```

Nodes:2325
Branches: 13
Interactive Steps: 0

Strategy: Applied 1378 rules (1.4 sec), closed 9 goals, 0 remaining

Figure 4.5: Finished correctness proof for the generated monitor

In order to obtain such upper bounds we used the tools SACO and CoFloCo. SACO [2] is a static analysis framework that can perform resource analysis of ABS programs. CoFloCo [12, 11] is a cost relation solver that can be used as a back-end by SACO to analyze models where SACO's default solver is not powerful enough. We focus our analysis on the CPU and memory consumption of the monitor for service degradation, in particular of its method `notify_query`.

CPU Consumption Analysis

The CPU consumption of `DegradationMonitorImpl.notify_query` is relatively straightforward. Its code makes a single call to the method `DegradationMetricImpl.notify_query` and adds the result to the list `metricHist`:

```
Unit notify_query(Time t, Rat procTime) {
  Pair<String, Rat> measurement = metric.notify_query(t, procTime);
  metricHist = Cons(Pair(t, list[measurement]), metricHist);
}
```

`DegradationMetricImpl.notify_query` does not contain loops or recursive functions and thus we can expect the CPU consumption to be constant.

We applied SACO+CoFloCo and we obtained a constant upper bound of 34. This upper bound refers to the number of evaluation steps needed to execute the method `DegradationMetricImpl.notify_query` and serves as a proxy for the CPU consumption.

Memory Consumption Analysis

The analysis of the memory consumption was considerably more challenging. In this case we were interested in the monitors' memory footprint. That is, the amount of memory required by the monitor's objects at any given time. Additionally, we wanted to infer how this memory usage is affected by each call to `DegradationMonitorImpl.notify_query`.

```
Unit count_list_size(List<Pair<Time,List<Pair<String,Rat>>>> list){
  List< Pair<String, Rat > > inner_list=Nil;
  while(list!=Nil){
     case list {
        Cons(Pair(time,head),tail) => {
          list=tail;
          [Cost: 1] inner_list=head; //cost of storing Time
           while(inner_list != Nil){
              [Cost : 2] inner_list=tail(inner_list); //cost of storing Pair<String, Rat >
           }
        }
        Nil => {}
     }
  }
}
```

Figure 4.6: Auxiliary method `count_list_size` to compute the memory consumption of the monitors.

SACO allows the user to select among multiple *cost models* that determine the resource to be measured. Unfortunately, none of these pre-defined cost models was adequate for our analysis. Instead we opted to select the "user" cost model which allows us to specify the cost of each statement through annotations.

A monitor is composed by two objects: `DegradationMonitorImpl` and `DegradationMetricImpl`. The fields in the `DegradationMetricImpl` object have a constant size (they are integer or pairs of integers and rationals) and thus we are interested in their size at any given time. Therefore, we annotated each field update in `DegradationMetricImpl.notify_query` with a constant cost representing the amount of memory required by that field.

```
...
[Cost : 1] curState = S;
 { [Cost : 1]cnt=cnt+1;
   { [Cost: 1]slowCnt = slowCnt + case(procTime>200) { True => 1; False => 0;};
     [Cost: 2]slowQpct = Pair("slow", slowCnt/cnt);
...
```

The case of `DegradationMonitorImpl` is more elaboraed since it contains a field `metricHist` of type `List<Pair<Time,List<Pair<String,Rat>>>>`. In this case we are interested in knowing how the size of this data structure varies with each call to `notify_query`. For this purpose, we create an auxiliary method `notify_query_aux` that receives the same arguments as `notify_query` plus the `metricHist` and returns the new `metricHist`. This way, `metricHist` becomes an input parameter of the method and we can obtain upper bounds as a function of its size:

```
List<Pair<Time,List<Pair<String,Rat>>>> notify_query_aux(Time t, Rat procTime,
                                       List<Pair<Time,List<Pair<String,Rat>>>> metricHistAux) {
    Pair<String, Rat> measurement = metric.notify_query(t, procTime);
    metricHistAux = Cons(Pair(t, list[measurement]), metricHistAux);
    this.count_list_size(metricHistAux);
    return metricHistAux;
}
```

Additionally, after updating `metricHistAux`, we call the auxiliary method `count_list_size`. Method `count_list_size` (see Figure 4.6) iterates over the data structure `metricHist` and consumes a constant amount of resources for each variable of type *Time* and each pair `Pair<String,Rat>` inside the data structure (in order to "simulate" corresponding memory consumption). This way we can set the resource consumption to the size of `metricHist` after the execution of one call to `notify_query`, and then use the resource analysis to measure such resource consumption in terms of the initial size of `metricHist`.

We applied SACO+CoFloCo with the following options:

- Option *backend* is set to *cofloco*. This allows selecting CoFloCo as a backend.
- Option *cost_centers* is set to *class*. This allows us to obtain separate bound for the resources consumed in each class.
- Option *size_abst* is set to *typed_norms*. This option specifies that the data structures are abstracted (to corresponding size measure) according to the techniques detailed in [3].

According to these options the data structure `metricHistAux` is abstracted into two numeric variables:

- `metricHistAux_1` represents the maximum size of the inner lists of type `List<Pair<String,Rat>>`. The size of each list is considered to be the number of constructors used to form the list, including the constructor `Nil` at the end of the list. This means that the size of the list is 1 plus the length of the list.
- `metricHistAux_2` represents the size of the outer list (the length of `metricHistAux` plus 1).

The results of the analysis that we have obtained are as follows:

```
5 within cost-center 'DegradationMetricImpl'
max(2*metricHistAux_2,2*metricHistAux_2*nat(metricHistAux_1-1)]+metricHistAux_2
  within cost-center 'DegradationMonitorImpl'
```

Where `nat(metricHistAux_1-1)` represents `max(metricHistAux_1-1,0)`. This gives us a constant memory cost of 5 units for `DegradationMetricImpl` which corresponds to the size of the object's fields. In the result for `DegradationMonitorImpl` we can use the fact that the inner lists in `metricHistAux` always have length 1 (and correspondingly `metricHistAux_1=2`) to simplify the resulting expression to:

```
  max(2*metricHistAux_2,2*metricHistAux_2*nat(metricHistAux_1-1)]+metricHistAux_2
= max(2*metricHistAux_2,2*metricHistAux_2*nat(2-1)]+metricHistAux_2
= 2*metricHistAux_2+metricHistAux_2
= 3*metricHistAux_2
```

Furthermore, we know that `metricHistAux_2=length(metricHistAux)+1` so we can also express the memory cost as `3*length(metricHistAux)+3`. This result indicates that the memory consumption for object `DegradationMonitorImpl` is proportional to the length of the event trace (`3*length(metricHistAux)`) and that after each call to `notify_query`, the memory consumption is incremented by 3 which accounts for the size of the new element added to `metricHistAux`.

Analysis of the Extended Service Degradation Monitor

We applied the same analysis to a different monitor called "Extended Service Degradation Monitor". This monitor implements the same metric but it is more fine-grained and closer to the real-world. The main difference between this monitor and the previous one is method `DegradationMetricImpl.notify_query` which now returns a list of type `List<Rat>` with two elements which is added to `metricHist` as in the previous case. The type of `metricHist` also changes to `List<Pair<Time,List<Rat>>>` instead of the previous type `List<Pair<Time,List<Pair<String,Rat>>>>`.

The results are 63 units of CPU usage, 6 memory units for the object `DegradationMetricImpl` and the same result, as for the previous monitor, for the memory consumption of `DegradationMonitorImpl`. Once simplified, it is `3*length(metricHistAux)+3`. This is not surprising as we have substituted a list of pairs (`List<Pair<String,Rat>>`) with only one element by a list of rationals (`List<Rat>`) with two elements (under the assumption that all basic types such as `String` and `Rat` require a single memory unit).

Conclusion and Future Work of Resource Analysis

We have successfully analyzed the CPU and memory consumption of two monitors that implement different versions of the degradation metric. The conclusion is that the CPU overhead is constant and the memory overhead grows proportionally to the length of the event trace (`metricHist`).

During the analysis of monitors we have detected and corrected several bugs both in SACO and CoFloCo. In addition to that, we have identified several aspects of the tools that could be improved and new opportunities for research. For instance, we found the ability to define the cost model using cost annotations very useful. However, at the moment, SACO only supports annotations with constant values. Extending SACO's support for cost annotations based on variables and their sizes would simplify the analysis greatly. We would not need the auxiliary method `count_list_size` but instead a simple annotation would suffice.

In order to analyze the second monitor, we had to perform a slight modification of the code of method `DegradationMetricImpl.notify_query`. This method has a condition that depends on a field and is always true. That constitutes a class invariants but SACO is unable to detect this. SACO could be extended to infer this kind of invariants or allow the user to provide them. Furthermore, KeY ABS could be used to verify the validity of the such invariants before they are fed into SACO.

4.4 Connecting External Systems

The API for interacting with an ABS model over HTTP (see Appendix A) provides a way to connect external systems: define a method that acts as a data-sink by marking it with a `[HTTPCallable]` annotation (this exposes it as an HTTP endpoint, i.e., the method can now be invoked over HTTP), and publish the desired data from the external system to it. For example, the interface in Figure 4.7 exposes a method `invokeWithSize` as a data-sink endpoint that allows to replay a real world log file, taking as inputs the original processing time of a query (`proctime`) and the CPU execution capacity (`amazonECU`) of the machine that processed it.

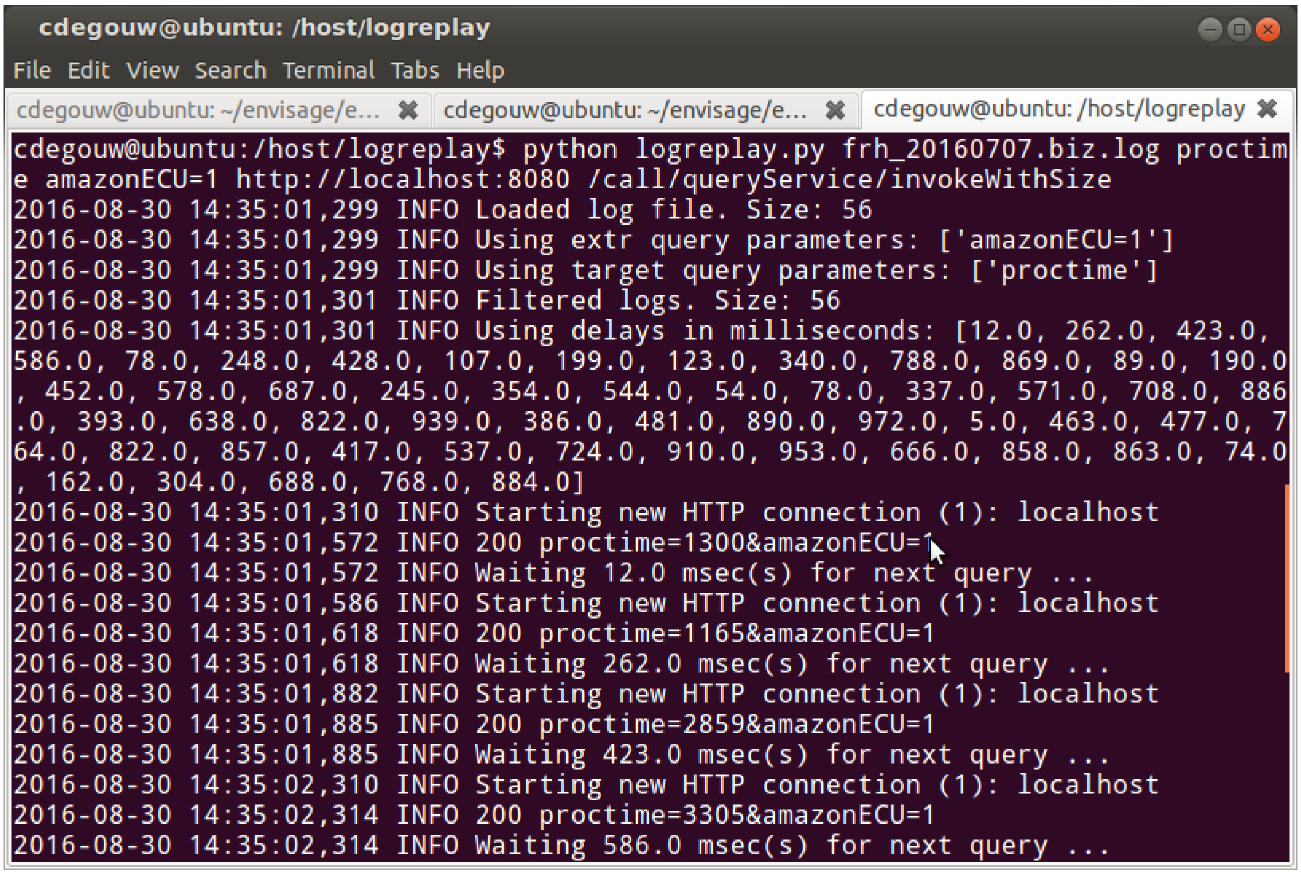
```
interface MonitoringQueryEndpoint extends EndPoint {
       [HTTPCallable] Unit invokeWithSize(Int proctime, Int amazonECU);
}
```

Figure 4.7: Data-sink endpoint

We implemented a general log-replay tool, that replaces logged events from a log-file according to the original timing by invoking a specified endpoint (see Figure 4.8). It supports abstracting away irrelevant attributes and applying custom transformation functions on attribute values before generating the request.

Another use-case for connecting external systems is plugging in external monitors. External monitors can be implemented in an entirely different language and run on external machines controlled by different organizations. This allows to monitor properties that should not be monitored at the server side, such as Availability (FRH uses Pingdom[3] to monitor Availability) and take infrastructure failures into account (FRH uses Amazon[4], we can simply retrieve the state of a virtual machine). By forwarding the data of these heterogeneous external systems to ABS data-sinks, all data-sources are available in real-time in the ABS model. This makes it possible to define monitors for more complex properties that combine different external monitors, we can leverage the ABS formal analyses, and all monitors are accessible and configurable through a unified visualization and management interface (Section 4.1).

[3]https://www.pingdom.com
[4]https://aws.amazon.com/ec2/


cdegouw@ubuntu: /host/logreplay
File Edit View Search Terminal Tabs Help
cdegouw@ubuntu: ~/envisage/e...
cdegouw@ubuntu: ~/envisage/e...
cdegouw@ubuntu: /host/logreplay

```
cdegouw@ubuntu:/host/logreplay$ python logreplay.py frh_20160707.biz.log proctim
e amazonECU=1 http://localhost:8080 /call/queryService/invokeWithSize
2016-08-30 14:35:01,299 INFO Loaded log file. Size: 56
2016-08-30 14:35:01,299 INFO Using extr query parameters: ['amazonECU=1']
2016-08-30 14:35:01,299 INFO Using target query parameters: ['proctime']
2016-08-30 14:35:01,301 INFO Filtered logs. Size: 56
2016-08-30 14:35:01,301 INFO Using delays in milliseconds: [12.0, 262.0, 423.0,
586.0, 78.0, 248.0, 428.0, 107.0, 199.0, 123.0, 340.0, 788.0, 869.0, 89.0, 190.0
, 452.0, 578.0, 687.0, 245.0, 354.0, 544.0, 54.0, 78.0, 337.0, 571.0, 708.0, 886
.0, 393.0, 638.0, 822.0, 939.0, 386.0, 481.0, 890.0, 972.0, 5.0, 463.0, 477.0, 7
64.0, 822.0, 857.0, 417.0, 537.0, 724.0, 910.0, 953.0, 666.0, 858.0, 863.0, 74.0
, 162.0, 304.0, 688.0, 768.0, 884.0]
2016-08-30 14:35:01,310 INFO Starting new HTTP connection (1): localhost
2016-08-30 14:35:01,572 INFO 200 proctime=1300&amazonECU=1
2016-08-30 14:35:01,572 INFO Waiting 12.0 msec(s) for next query ...
2016-08-30 14:35:01,586 INFO Starting new HTTP connection (1): localhost
2016-08-30 14:35:01,618 INFO 200 proctime=1165&amazonECU=1
2016-08-30 14:35:01,618 INFO Waiting 262.0 msec(s) for next query ...
2016-08-30 14:35:01,882 INFO Starting new HTTP connection (1): localhost
2016-08-30 14:35:01,885 INFO 200 proctime=2859&amazonECU=1
2016-08-30 14:35:01,885 INFO Waiting 423.0 msec(s) for next query ...
2016-08-30 14:35:02,310 INFO Starting new HTTP connection (1): localhost
2016-08-30 14:35:02,314 INFO 200 proctime=3305&amazonECU=1
2016-08-30 14:35:02,314 INFO Waiting 586.0 msec(s) for next query ...
```

Figure 4.8: Log replay

Bibliography

[1] Wolfgang Ahrendt and Maximilian Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programming*, 77(12):1289–1309, October 2012.

[2] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, Germán Puebla, and Guillermo Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *Lecture Notes in Computer Science*, pages 562–567. Springer-Verlag, 2014.

[3] Elvira Albert, Samir Genaim, and Raúl Gutiérrez. Towards a Transformational Approach to Resource Analysis with Typed-Norms (Extended Abstract). In *23rd International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'13)*, pages 85–96, September 2013.

[4] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software. The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[5] Frank S. de Boer and Stijn de Gouw. Run-time deadlock detection. In *ProCoS Workshop on Provably Correct Systems*, To appear.

[6] Frank S. de Boer, Stijn de Gouw, Einar Broch Johnsen, Andreas Kohn, and Peter Y. H. Wong. Run-time Assertion Checking of Data- and Protocol-oriented Properties of Java Programs: An Industrial Case Study. *T. Aspect-Oriented Software Development*, 11:1–26, 2014.

[7] Stijn de Gouw, Jacopo Mauro, Behrooz Nobakht, and Gianluigi Zavattaro. Declarative elasticity in ABS. In *Service-Oriented and Cloud Computing - 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings*, pages 118–134, 2016.

[8] Crystal Chang Din, Richard Bubel, and Reiner Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In Amy P. Felty and Aart Middeldorp, editors, *Proceedings of the 25th International Conference on Automated Deduction (CADE 2015)*, volume 9195 of *Lecture Notes in Computer Science*, pages 517–526. Springer-Verlag, 2015.

[9] Crystal Chang Din, Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming*, 81(3):227–256, 2012.

[10] Crystal Chang Din and Olaf Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, 27(3):551–572, 2015.

[11] Antonio Flores-Montoya. Upper and lower amortized cost bounds of programs expressed as cost relations. In *21st International Symposium on Formal Methods (FM'16)*, Lecture Notes in Computer Science, 2016. to appear.

[12] Antonio Flores-Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In Jacques Garrigue, editor, *12th Asian Symposium on Programming Languages and Systems*

(APLAS'14), volume 8858 of *Lecture Notes in Computer Science*, pages 275–295. Springer, November 2014.

[13] Elena Giachino, Stijn de Gouw, Cosimo Laneve, and Behrooz Nobakht. Statically and dynamically verifiable SLA metrics. In *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, pages 211–225, 2016.

[14] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. EASY meta-programming with rascal. In *Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*, pages 222–289, 2009.

[15] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[16] Behrooz Nobakht, Stijn de Gouw, and Frank S. de Boer. Formal verification of service level agreements through distributed monitoring. In Schahram Dustdar, Frank Leymann Schahram, and Massimo Villari, editors, *Service Oriented and Cloud Computing – 4th European Conference, ESOCC 2015, Taormina, Italy, September 15-17, 2015*, Lecture Notes in Computer Science, pages 125–140. Springer-Verlag, 2015.

Glossary

AWS Amazon Web Services.

Cloud Engineer A Cloud Engineer handles the day-to-day operation of services. She deploys/updates services through PaaS and IaaS according to incomplete *service requirements* from Consultants, diagnoses issues at service-level and either resolves them at real time or informs the Support Engineers and/or the Software Engineers. She manages the up and down scaling of service resources according to alerts and metric visualizations provided by the monitoring system. She also performs any necessary infrastructural changes

Endpoint Used interchangeably with URL that refers to an HTTP address that a user can reach to invoke a certain operation

IaaS Infrastructure as a Service

Infrastructure as a Service A provision model in which an organization outsources the equipment used to support IT operations, including storage, hardware, servers and networking components. The service provider owns the equipment and is responsible for housing, running and maintaining it. The client typically pays on a per-use basis

JSON JavaScript Object Notation. A data format that uses human-readable text to transmit data objects consisting of attribute-value pairs.

PaaS Platform as a Service

Platform as a Service A category of cloud service offerings that facilitates the deployment of applications without the cost and complexity of buying and managing the underlying hardware and software and provisioning hosting capabilities

QoS Quality of Service

Quality of Service Generic term encapsulating all the non-functional aspects of a service delivery

Resource Any entity that provides computation power or storage facilities

Resource Configuration A description of the number of service instances initially required for a service offered to a Customer and the virtualized resource to be allocated initially to those service instances

SaaS Software as a Service

Service Level Agreement A legal contract between a service provider and his customer. It records a common understanding about services, priorities, responsibilities, guarantees, and warranties

SLA Service Level Agreement

Software as a Service A software delivery model in which software and associated data are centrally hosted on the cloud. SaaS is typically accessed by users using a thin client via a web browser

Appendix A

HTTP API

The HTTP API provides a means of querying and accessing a simulated model from the outside via HTTP requests. HTTP was chosen for its ubiquity and ease of use. The following features are implemented:

- Objects are registered via an annotation on `new` expressions;
- Methods are made callable via an annotation in an interface definition;
- Objects are protected from garbage collection as long as they are registered;
- Field values of registered objects can be accessed (read-only);
- Callable methods of registered objects can be called via the API, and the return value can be read;
- Registered objects and their fields and callable methods can be enumerated via the API.

This Appendix presents an interim version of the HTTP API; changes in supported parameter and return values and other aspects will be documented in the ABS manual.

A.1 Registering Objects

The annotation

```
[HTTPName: "name"] new C();
```

registers the new object as `name`. In case `name` is already in use, the previous object is unregistered and will not be accessible via the API anymore.

A.2 Making Methods Callable

In an interface declaration, the following

```
[HTTPCallable] String method(Int parameter);
```

declares the method `method` to be callable via the API on registered objects.

- It is a compile-time error if the method takes parameters whose types are not supported. Currently supported types are the ABS types `Int`, `String` and `Bool`.
- The method can have an arbitrary return type. Integers, strings and boolean values will be returned as the equivalent JSON types, other values will be returned as strings via the ABS `toString()` function.

A.3 Querying for Objects

The HTTP request

```
GET http://localhost:8080/o
```

returns a list of registered object names.

A.4 Reading Object State

The HTTP request

```
GET http://localhost:8080/o/name
```

returns a JSON map of state of the object registered as `foo`, as `{ "field": "value", "field2": "value2"}`. Field values are converted in the same way as method return values.

The HTTP request

```
GET http://localhost:8080/o/foo/field
```

returns a JSON map with a singleton entry `{ "field" : "value" }`. In addition:

- Unknown object requests result in a 404 response code
- Unknown field names result in a 404 response code

A.5 Calling Methods

For a registered name `name`, the HTTP request

```
GET http://localhost:8080/call/name
```

returns a, JSON array with metadata about callable functions, with each element of the list being a map with the following entries:

`name` method name

`parameters` array with one object per parameter, each with the following entries:

> `name` name of the parameter
>
> `type` type of the parameter

`return` return type of the method

The HTTP request

```
GET http://localhost:8080/call/name/method?param1=value&param2=50
```

calls `method` on the object registered as `name`, which is equivalent to performing the ABS method call `foo.method("value", 50)`. The following datatypes can be used for parameters:

`Bool` given as literal upper- or lowercase "true" / "false", i.e. `?p=True`, `?p=true`, `?p=False`, `?p=false`

`String` URLEncoded text, e.g., `?p=Hello%20World!`

`Int` Integer, e.g., `?p=42`

- Successful calls produce a 200 response code and a JSON map with a single entry `result` mapping to the result value.
- Unknown object requests produce 404 response code
- Unknown method name produces 404 response code
- Invalid parameters produce a 400 response code
- Errors during method invocation produce 500 code

Appendix B

Formal Verification of Service Level Agreements Through Distributed Monitoring [16]

Formal verification of service level agreements through distributed monitoring[⋆]

Behrooz Nobakht[1,2], Stijn de Gouw[2,3], and Frank S. de Boer[1,3]

[1] Leiden Advanced Institute of Computer Science
Leiden University
bnobakht@liacs.nl
[2] SDL Fredhopper
{bnobakht,sgouw}@sdl.com
[3] Centrum Wiskunde en Informatica
{frb,cdegouw}@cwi.nl

Abstract. In this paper, we introduce a formal model of the availability, budget compliance and sustainability of distributed services, where service sustainability is a new concept which arises as the composition of service availability and budget compliance. The model formalizes a distributed platform for monitoring the above service characteristics in terms of a parallel composition of task automata, where dynamically generated tasks model asynchronous events with deadlines. The main result of this paper is a formal model to optimize and reason about service characteristics through monitoring. In particular, we use schedulability analysis of the underlying timed automata to optimize and guarantee service sustainability.

Keywords: runtime monitoring, service availability, budget compliance, service sustainability, distributed architecture, cloud computing, service level agreement

1 Introduction

Cloud computing provides the elastic technologies for virtualization. Through virtualization, software itself can be offered as a service (Software as a Service, SaaS). One of the aims of SaaS is to allow service providers to offer reliable software services while scaling up and down allocated resources based on their availability, budget, service throughput and the Service Level Agreements (SLA). Thus, it becomes essential that virtualization technologies facilitate elasticity in a way that enables business owners to *rapidly* evolve their systems to meet their customer requirements and expectations.

The fundamental technical challenge to a SaaS offering is maintaining the quality of service (QoS) promised by its SLA. In SaaS, providers must ensure a

[⋆] This paper is funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services, http://www.envisage-project.eu.

consistent QoS in a dynamic virtualized environment with variable usage patterns. Specifically, virtualized environments such as the cloud provide elasticity in resource allocation, but they often do not offer an SLA that can guarantee constant resource availability. As a result, SaaS providers are required to react to resource availability at runtime. Furthermore, by offering a 24/7 software service, SaaS providers must be able to react to certain service usage patterns, such as an increase in throughput to ensure the SLA is maintained.

Runtime monitoring [20,4] is a dynamic analysis approach based on extracting relevant information about the execution. Runtime monitoring may be employed to collect statistics about the service usage over time, and to detect and react to service behavior. This latter ability is fundamental in the SaaS approach to guarantee the SLA of a service and is the focus of this paper.

The monitoring model that is presented in this paper is designed to *observe* in real-time certain service characteristics and *react* to them to ensure the evolution of the system towards its SLA. Asynchronous communication is an essential feature of a monitoring model in a distributed context. Asynchronous communication accomplishes non-intrusive observations of the service runtime. Further, the monitoring model is expected to operate according to certain real-time constraints specified by the SLA of the service. Satisfying the real-time constraints is the main challenge in a distributed monitoring model.

In this paper, we formalize service availability and budget compliance in a distributed deployment environment. This formalization is based on high-level task automata models [1,9,13]. The automata capture the real-time evolution of the resources provided by a distributed deployment platform and the above two main service characteristics. These task automata represent the real-time generation of the asynchronous events extended with deadlines [3,22] by the monitoring platform for managing resources (i.e. allocation or deallocation). The main result of this paper is a formal model to optimize and reason about the above service characteristics through monitoring. In particular, the *schedulability* of the underlying timed automata implies service availability and budget compliance. Furthermore, we introduce a composition of service availability and budget compliance which captures service sustainability. We show that service sustainability presents a multi-objective optimization problem.

2 Related Work

Vast research work present different aspects of runtime monitoring. We focus on those that present a line of research for distributed deployment of services.

MONINA [12] is a DSL with a monitoring architecture which supports certain mathematical optimization techniques. A prototype implementation is available. Accurately capturing the behavior of an in-production legacy system coded in a conventional language seems challenging: it requires developing MONINA components, which generate events at a specified fixed rate, there are no control structures (if-else, loops), the data types that can be used in events are predefined, and there are no OO-features. We use ABS [15], an executable modeling language that supports all of these features and offers a wide range of tool-supported analyses [5,25]. The mapping from ABS to timed automata [1] allows to exploit the state-of-the-art tools for timed automata, in particular for reasoning about real-time properties (and, as we show, SLAs using schedulability analysis [9]). MONINA offers *two pre-defined* parameters that can be used in monitoring to adapt the system: cost and capacity. Our service metric function generalizes this to *arbitrary user-defined* parameters, including cost and capacity.

Hogben and Pannetrat examine in [11] the challenges of defining and measuring availability to support real-world service comparison and dispute resolution through SLAs. They show how two examples of real-world SLAs would lead one service provider to report 0% availability while another would report 100% for the same system state history but using a different period of time. The transparency that the authors attempt to reach is addressed in our work by the concept of monitoring window and expectation tolerance in Section 4. Additionally, the authors take a continuous time approach contrasted with ours that uses discrete time advancements. Similarly, they model the property of availability using a two-state model.

The following research works provide a language or a framework that allows to formalize service level agreements (SLA). However, they do not study how such SLAs can be used to monitor the service and evolve it as necessary. WSLA [18] introduces a framework to define and break down customer agreements into a technical description of SLAs and terms to be monitored. In [21], a method is proposed to translate the specification of SLA into a technical domain directed in SLA@SOI EU project. In the same project, [8] defines terms such as availability, accessibility and throughput as notions of SLA, however, the formal semantics and properties of the notions are not investigated. In [6], authors describe how they introduce a function how to decompose SLA terms into measurable factors and how to profile them. Timed automata is used in [24] to detect violations of SLA and formalize them.

Johnsen [16] introduce “deployment components” using Real-Time ABS [3]. A deployment component enables an application to acquire and release resources on-demand based on a QoS specification of the application. A deployment component is a high level abstraction of a resource that promotes an application to a resource-aware level of programming. Our work is distinguished by the fact that we separate the monitors from the application (service) themselves. We argue that we aim to design the monitoring model to be as *non-intrusive* as possible to the service runtime. Thus, we do not deploy the monitors inside the service runtime.

In Quanticol EU project[4], authors in [7] and [10] use statistical approaches to observe and guarantee service level agreements for public transportation. We also present that service characteristics can be composed together. This means that evolving a system based on SLAs turns into a multi-object optimization problem. In addition, in COMPASS EU project[5], CML [26] defines a formal language to

[4] Quanticol EU project with no. 600708: `http://quanticol.eu/`
[5] COMPASS EU project with no. 287829: `http://www.compass-research.eu/`

model systems of systems and the contracts between them. CML studies certain properties of the model and their applications. CML is used in the context of a Robotics technology to model and ensure how emergency sensors should react and behave according to the SLAs defined for them. Our approach is similar to provide a generic model for service characteristics definition, however, we utilize timed and task automata.

3 SDL Fredhopper Cloud Services

In this section, we introduce a running example in the context of SDL Fredhopper. We use the example in different parts of the paper and also in the experiments.

SDL Fredhopper develops the Fredhopper Cloud Services to offer search and targeting facilities on a large product database to e-Commerce companies as services (SaaS) over the cloud computing infrastructure (IaaS). Fredhopper Cloud Services provides several SaaS offerings on the cloud. These services are exposed at endpoints. In practice these endpoints typically are implemented to accept connections over HTTP. For example, one of the services offered by these endpoints is the Fredhopper Query API, which allows users to query over their product catalog via full text search[6] and faceted navigation[7].

A customer of SDL Fredhopper using Query API owns a *single* HTTP endpoint to use for search and other operations. However, internally, a number of resources (virtual machines) are used to deliver Query API for the customer. The resources used for a customer are managed by a load balancer. In this model of deployment, each resource is launched to serve *one* instance of Query API; i.e. resources are *not* shared among customers.

When a customer signs a contract with SDL Fredhopper, there is a clause in the contract that describes the minimal QoS levels of the Query API. For example, we have a notion of query per second (`QPS`) that defines the number of completed queries per second for a customer. An agreement is a bound on the expected `QPS` and forms the basis of many decisions (technical or legal) thereafter. The agreement is used by the operations team to set up an environment for the customer which includes the necessary resources described above. The agreement is additionally used by the support team to manage communications with the customer during the lifetime of the service for the customer.

Maintaining the services for more than 250 customers on more than 1000 servers is not an easy operation task [8]. Thus, to ensure the agreements in a customer's contract:

– The operation team maintains a monitoring platform to get notifications on the current metrics.

[6] `http://en.wikipedia.org/wiki/Full_text_search`

[7] `http://en.wikipedia.org/wiki/Faceted_navigation`

[8] Figures are indication of complexity and scale. Detailed confidential information may be shared upon official request.

- The operation team performs *manual* intervention to ensure that sufficient resources are available for a customer (launching or terminating).
- The monitoring platform depends on *human* reaction.
- The cost that is spent for a customer on the basis of safety can be *optimized*.

In this paper, we use the notion of QPS as an example in the concepts that are presented in this research. We use the example here to demonstrate how the model that is proposed in this research can address the issues above and alleviate the *manual* work with *automation*. The manual life cycle depends on the domain-specific and contextual knowledge of the operations team for every customer service that is maintained in the deployment environment. This is labor-intensive as the operations team stands by 24×7. In such a manual approach, the business is forced to over-spend to ensure service level agreements for customers.

4 Distributed Monitoring Model

We introduce a distributed monitoring platform and its components and discuss some underlying assumptions and definitions. Further, we define the notion of service availability and service budget compliance. In the deployment environment (e.g., "the cloud"), every server from the IaaS provider is used for a *single* service of a customer, such as the Query Service API for a customer of SDL Fredhopper (c.f. Section 3). Typically, multiple servers are allocated to a single customer. The number of servers allocated for a customer is not visible to the customer. The customer uses a single endpoint - in the load balancer layer - to access all their services.

The ultimate goal is to maintain the environment in such a way that customers and their end users experience the delivered services up to their expectations while minimizing the cost of the system. The first objective can be addressed by adding resources; however, this conflicts with the second goal since it increases the cost of the environment for the customer. In this section, we formalize the above intuitive notions as *service availability* and *service budget compliance*.

We then develop a distributed monitoring platform that aims to optimize these service characteristics in a deployment environment. The monitoring platform works in two cyclic phases: *observation* and *reaction*. The observation phase takes measurements on services in the deployment environment. Subsequently, the corresponding levels of the service characteristics are calculated. In the reaction phase, if needed, a platform API is utilized to make the necessary changes to the deployment environment (e.g. adjust the number of allocated resources) to optimize the service characteristics. The monitoring platform builds on top of a real-time extension of the actor-based language ABS [15]. To ensure non-intrusiveness of the monitor with the running service, each monitor is an active object (actor) running on a separate resource from that which runs the service itself, and the components of the monitoring platform communicate through *asynchronous messages* with deadlines [16].

Below, we discuss assumptions and basic oncepts that will be used in the analysis of the formal properties of the monitoring platform and corresponding theorems. We assume that the external infrastructure provider has an *unlimited* number of resources. Further, we assume that all resources are of the *same type*; i.e. they have the same computing power, memory, and IO capacity. Finally, we assume that every resource is initialized within at most t_i amount of time.

In our framework time T is a universally shared clock based on the NTP [9] that is used by all elements of the system in the same way. T is discrete. We fix that the unit of time is *milliseconds*. This level of granularity of time unit means that between two consecutive milliseconds, the system is not observable. For example, we use the UTC time standard for all services, monitors and platform API. We refer to the current time by t_c.

We denote by r a resource which provides computational power and storage and by s a general abstraction of a service in the deployment environment. A service exposes an API that is accessible through a delivery layer, such as HTTP. In our example, a service is the Query API (c.f. Section 3) that is accessible through a single HTTP endpoint.

In our framework, *monitoring platform* P is responsible for (de-)allocation of resources for computation or storage. We abstract from a specific implementation of the monitoring platform P through an API in Listing 1. There is only *one* instance of P available. In this paper, P internally uses an external infrastructure provisioning API to provide resources (e.g. AWS EC2). The term "platform" is interchangeably used for monitoring in this paper. The platform provides a method `getState(Service s)` which returns the number of resources allocated to the given service s at time t_c.

Listing 1: Platform API

```
interface Platform {
  void    allocate(Service s);
  void    deallocate(Service s);
  Number  getState(Service s);
  boolean verifyα(Service s);
  boolean verifyβ(Service s);
}
```

We use monitoring to observe the external behavior of a service. We formalize the external behavior of a service with its service-level agreement (SLA). An SLA is a contract between the customer (service consumer) and the service provider which defines (among other things) the minimal quality of the offered service, and the compensation if this minimal level is not reached. To formally analyze an SLA, we introduce the notion of a service metric function. We make basic measurements of the service externally in a given monitoring window (a duration). The service metric function aggregates the basic measurements into a single number that indicates the quality of a certain service characteristic (higher numbers are better).

Basic measurement $\mu(s, r, t)$ is a function that produces a real number of a *single* monitoring check on a resource r allocated to service s at some time t. For example, for SDL Fredhopper cloud services, a basic measurement is the number of completed queries at the current time.

[9] `https://tools.ietf.org/html/rfc1305`

Service Metric f_s is a function that aggregates a sequence of basic non-negative measurements to a single non-negative real value: $f_s : \bigcup_n \mathbb{R}^n \to \mathbb{R}$. For example, for SDL Fredhopper cloud services, the service metric function f_s calculates the average number of queries per second (QPS) given a list of basic measurements.

Monitoring Window is a duration of time τ throughout which basic measurements for a service are taken.

Monitoring Measurement is a function that aggregates the basic measurements for a service over its resources in the last monitoring window. The last monitoring window is defined as $[t_c - \tau, t_c]$. To produce the monitoring measurement, f_s is applied. Formally:

$$\mu(s, r, \tau) = f_s\big(\langle \mu_i(s, r, t) \rangle_{i=0}^{\infty}\big) \;\text{ where }\; t \in [t_c - \tau, t_c]$$

in which $\mu_i(s, r, t)$ is the i-th basic measurement of services s on resource r at time t where $t \in [t_c - \tau, t_c]$.

Definition 1 (Service Availability $\alpha(s, \tau, t_c)$)**.** First, we need a few auxiliary definitions before we can define service availability.

Service Capacity $\kappa_\sigma(s, \tau) = \sum_{r \in \sigma(s)} \mu(s, r, \tau)$ denotes the capability of service s that is the aggregated monitoring measurements of its resources over the monitoring window τ and $\sigma(s)$ is the number of allocated resources to service s.

Agreement Expectation $E(s, \tau, t_c)$ is the minimum number of requests that a customer expects to complete in a monitoring window τ. The agreement expectation depends on the current time t_c because the expectation may change over time. For example, SDL Fredhopper customers expect a different QPS during Christmas.

We define the availability of a service $\alpha(s, \tau, t_c)$ in every monitoring window τ as:

$$\alpha(s, \tau, t_c) = \frac{\kappa_\sigma(s, \tau)}{E(s, \tau, t_c)}$$

Capacity Tolerance $\varepsilon_\alpha(s, \tau)) \in [0, 1]$ defines how much $\kappa_\sigma(s, \tau)$ can deviate from $E(s, \tau, t_c)$ in every time span of duration τ.

Service Guarantee Time t_G is the duration within which a customer expects service availability reaches an acceptable value after a violation. Typically, t_G is an input parameter from the customer's contract.

Example 1. Intuitively, $\alpha(s, \tau, t_c)$ presents the actual capability of a service s over a time period τ compared to the expectation on the service $E(s, \tau)$. For values $\alpha(s, \tau, t_c) \ll 1 - \varepsilon_\alpha(s, \tau))$, the resource for service s are at "under-capacity" while for values $\alpha(s, \tau, t_c) \gg 1 + \varepsilon_\alpha(s, \tau))$, there is "over-capacity". The goal is optimize $\alpha(s, \tau, t_c)$ towards a value of 1.

For example, we expect a query service to be able to complete 10 queries per second. We define the monitoring window $\tau = 5$ minutes; thus, $E(s, \tau, t_c) = 10 \times 60 \times 5 = 3000$. Suppose we allocate only one resource to the service, measure the service during a single monitoring window τ and find $\mu(s, r, \tau) = 2900$. Then $\alpha(s, \tau, t_c) = \frac{2900}{3000} = 0.966$. If we have $\varepsilon_\alpha(s, \tau)) = 0.03$, this means that service s is under-capacity because $\alpha(s, \tau, t_c) < 1 - \varepsilon_\alpha$.

Definition 2 (Budget Compliance $\beta(s, \tau)$)**.** We first provide a few auxiliary definitions.

Resource Cost $€(r, \tau) \in \mathbb{R}^+$ is the cost of resource r in a monitoring window τ which is determined by a fixed resource cost per time unit.

Service Cost $€_\sigma(s, \tau) \in \mathbb{R}^+$ is the cost of a service s in a monitoring window τ and defined as $€_\sigma(s, \tau) = \sum_{r \in \sigma(s)} €(r, \tau)$.

Service Budget $B(s, \tau)$ specifies an upper bound of the expected cost of a service in the time span τ. Intuitively $B(s, \tau)$ is the allowed budget that can be spent for service s over the time span τ. The service budget is typically chosen to be fixed over any time span τ.

We are now ready to define service budget compliance $\beta(s, \tau)$ that, intuitively, represents how a service complies with its allocated budget:

$$\beta(s, \tau) = \frac{€_\sigma(s, \tau)}{B(s, \tau)}$$

Budget Tolerance $\varepsilon_\beta(s, \tau) \in [0, 1]$ specifies how much the service cost $€(s, \tau)$ can deviate from $B(s, \tau)$ in every time span of duration τ.

Service Guarantee Time t_G is similar to that defined for service availability.

Example 2. Assume every resource on the environment costs 1 (e.g. €) per hour. Suppose we set a budget of 1.5 per hour for every service, allocate *one* resource to the service and define a monitoring window of $\tau = 5$ minutes. Every hour has 12 monitoring windows. This means that each resource costs $€(r, \tau) = \frac{1}{12} \approx 0.08$ per monitoring window. Since there is only one resource, the service cost is $€(s, \tau) = \sum_{r \in \sigma(s)} €(s, \tau) \approx 0.08$ per monitoring window. On the other hand, if we calculate the budget for one monitoring window, we have $B(s, \tau) = \frac{1.5}{12} = 0.125$ per monitoring window. This yields budget compliance as $\beta(s, \tau) = \frac{0.08}{0.125} = 0.64$.

The formal definitions of service availability and budget compliance provide a rigorous basis for automatic deployment of resource-aware services with an appropriate quality of service, taking costs into account. This in particular includes automated scaling up or down of the service with the help of monitoring checks that are installed for the service. The fundamental challenge in ensuring service availability and budget compliance is that they have *conflicting* objectives:

$$\alpha(s, \tau, t_c) \uparrow \iff \beta(s, \tau) \downarrow$$

Intuitively, if more resources are used to ensure the availability of a service; then $\alpha(s, \tau, t_c)$ increases. However, at the same time, the service costs more; i.e. budget compliance $\beta(s, \tau)$ decreases.

5 Service Characteristics Verification

In this section, we use timed automata and task automata to model the behavior of a monitoring platform P, the deployment environment E, and the monitoring components for service availability $\alpha(s, \tau, t_c)$ and budget compliance $\beta(s, \tau)$. [13] defines a task automata as an extension of timed automata in which each task is a piece of executable program with (b, w, d): best/worst time and deadline of the task. A task automata uses a scheduler for the tasks to schedule each task with a location on a queue.

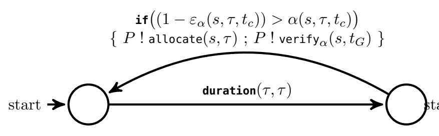
Modeling the elements of the monitoring platform is necessary to be able to study certain properties of the system. The most important goal of a monitoring platform is to enable the autonomous operation of a set of services according to their SLA. Thus, it is essential how to analyze that the monitoring platform can provide certain guarantees about the service and its SLA. In addition, it is important be able to verify the monitoring platform through model checking and schedulability analysis. Using timed automata and task automata facilitates model checking and verification through formal method tools such as UPPAAL [2] supporting advanced methods such as state-space reduction [19].

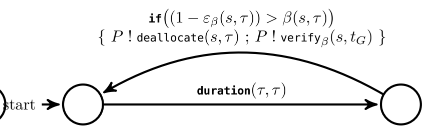
We use task automata as defined in [9,14,13]. Task automata are an extension of timed automata [1]. In addition, we design the automata for the monitoring platform using the real-time extension of task automata presented in [13] p. 92 in which the author presents a mapping from Real-Time ABS [16] to the equivalent task automata.

A task type is a piece of executable program/code represented by a tuple (b, w, d), where b and w respectively are the best-case and worst-case execution times and d is the deadline. In a task automata, there are two types of transitions: *delay* and *discrete*. A delay transition models the execution of a running task by idling for other tasks. A discrete transition corresponds to the arrival of a new task. When a new task is triggered, it is placed into a certain position in the queue based on a scheduling policy [23,22]. Examples of a scheduling policy are FIFO or EDF (earliest deadline first). The scheduling policy is modeled as a timed automaton Sch. Every task has its own stop watch. The scheduler also maintains a separate stop watch for each task to determine if a task misses its deadline. All stop watches work at the same clock speed specified by T.

We design separate automata for each service s characteristic: service availability $\alpha(s, \tau, t_c)$ by an automata M_{α_s} and service budget compliance $\beta(s, \tau)$, by an automata M_{β_s}. Each automaton is responsible for one goal: to optimize the service characteristic. M_{α_s} aims to improve $\alpha(s, \tau, t_c)$ whereas M_{β_s} aims to improve $\beta(s, \tau)$. M_{α_s} uses `allocate` to launch a new resource in the environment and improve the service s. In contrast, M_{β_s} uses `deallocate` to terminate a resource to decrease the cost of the service.

We use task automata to design M_{α_s}. Periodically, M_{α_s} checks whether the service availability is within the thresholds, taking tolerance into account (Definition 1). If the condition fails, M_{α_s} generates a task for monitoring platform P to allocate a new resource to service s with a deadline of τ. We define the period to be τ. We use the semantics of a task automata in [13] p. 92 in the transitions of the task automata. Figure Fig. 1a and Fig. 1b present M_{α_s} and M_{β_s}. Both M_{α_s} and M_{β_s} share state with the monitoring platform P. The state keeps the current number of resources for a service s that is denoted by $\sigma(s)$. All timed automata and task automata in the monitoring platform have shared access to $\sigma(s)$. In the automata, we use a conditional statement to check the service characteristics $\alpha(s,\tau,t_c)$ or $\beta(s,\tau)$. If the condition fails, M_{α_s} requests P to `allocate` a new resource to s and M_{β_s} requests P to `deallocate` a resource. In addition, M_{α_s} triggers a new task $\texttt{verify}_\alpha$ with deadline t_G. Intuitively, this means the service characteristic $\alpha(s,\tau,t_c)$ is verified to be within the expected thresholds after at most t_G time.


if((1 − ε_α(s, τ, t_c)) > α(s, τ, t_c))
{ P ! allocate(s, τ) ; P ! verify_α(s, t_G) }
start
duration(τ, τ)
Fig. 1a: M_{α_s} task automata for $\alpha(s,\tau,t_c)$


if((1 − ε_β(s, τ)) > β(s, τ))
{ P ! deallocate(s, τ) ; P ! verify_β(s, t_G) }
start
duration(τ, τ)
Fig. 1b: M_{β_s} task automata for $\beta(s,\tau)$

We use a separate task automaton for each service characteristic to verify the SLA of the service after t_G time. Respectively, M_{V}^{α} and M_{V}^{β} execute tasks $\texttt{verify}_\alpha$ and $\texttt{verify}_\beta$ (Figures Fig. 2a and Fig. 2b). M_{V}^{α} uses **`await`** to ensure the condition of the SLA. In addition, the task is controlled by the scheduler using a deadline that is specified as t_G in the generated task $\texttt{verify}_\alpha(s,t_G)$ in M_{α_s}. If t_G passes before the guard statement in **`await`** statement holds, it leads to a *missed deadline.*

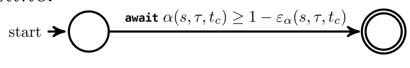


Fig. 2a: M_{V}^{α} to execute $\texttt{verify}_\alpha$

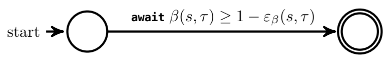


Fig. 2b: M_{V}^{β} to execute $\texttt{verify}_\beta$

Both M_{α_s} and M_{β_s} are specific to one particular service s. A generalized automaton for all services is obtained as their parallel composition:$M_\alpha = (\|_s M_{\alpha_s})$ and $M_\beta = (\|_s M_{\beta_s})$. The tasks generated by M_α and M_β (triggered by the calls to `allocate` and `deallocate`) are executed by the task automata for platform M_P.

We model monitoring platform P by a task automata M_P. The task types are $\{\texttt{A}(\texttt{allocate}), \texttt{D}(\texttt{deallocate})\}$. For task type `A` in M_P, we use $(b,w,d) = (t_i,\tau,\tau)$; i.e. the best-case execution time of a task is the resource initialization time, the worst-case is the length of the monitoring window, and the deadline is the length of the monitoring window. For task type `D` in M_P, we use $(b,w,d) = (0,\tau,\tau)$. We do not fix the scheduling policy Sch. The error state q_{err} in M_P is defined when either a deadline is missed or when the platform fails to provision a resource. Thus the monitoring platform P contains the following ingredients:

$$M_P = \langle M_{\mathsf{A}} \parallel M_{\mathsf{D}} \parallel M_{\mathsf{V}}^{\alpha} \parallel M_{\mathsf{V}}^{\beta}, \mathsf{Sch}, \tau \rangle$$

We define M_{A_s} as the timed automata to execute the tasks of type `allocate` in M_P. We use the model semantics presented in [13] p. 92 to design M_{A_s}. The resulting automata is presented in Figure 3.

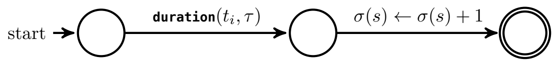


Fig. 3: M_{A_s}: Timed Automaton to execute task type `allocate` in M_P

Then, we define M_{A} in M_P as: $M_{\mathsf{A}} = \|_s M_{\mathsf{A}_s}$; i.e. the composition of all timed automata to execute a task `allocate` for some service s. Similarly, we design M_{D_s}

to execute task type `deallocate` in Figure 4. Therefore, we also have M_{D} in M_P as: $M_{\mathsf{D}} = \|_s M_{\mathsf{D}_s}$.

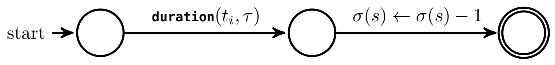


Fig. 4: M_{D_s}: Timed Automaton to execute task type `deallocate` in M_P

For a particular service s, its automaton M_{α_s} regularly measures the service characteristics and calculates $\alpha(s, \tau, t_c)$. When s is under-capacity, M_{α_s} requests to `allocate` a new resource for s through monitoring platform P. This generates a new task in M_P that is executed by M_{A_s}. When the task completes, the state of the service $\sigma(s)$ is updated; strictly increased. Thus, in isolation, the combination of M_{α_s} and M_{A_s} increase the value of service availability $\alpha(s, \tau, t_c)$ for service s over time. Similarly, in isolation, the combination of M_{β_s} and M_{D_s} increase the value of service budget compliance $\beta(s, \tau)$ for service s over time. Because in the latter, `deallocate` is used to decrease the cost of the service and as such increases $\beta(s, \tau)$.

In reality, resources might fail in the environment. The failure of a resource is not and cannot be controlled by the monitoring platform P. However, the failure of a resource affects the state of a service and its characteristics. Thus, we model the environment, including failures, as an additional timed automata, M_E. In M_E, in every monitoring window, there is a probability that some resources fail. For example, we present a particular instance of M_E in Figure 5. In this environment, in every monitoring, an unspecified constant (c) number of resources fail.

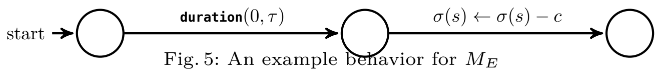


Fig. 5: An example behavior for M_E

We define system automata [13] (p. 33, Definition 3.2.7) for each service characteristic; $\mathcal{S}_\alpha$ for $\alpha(s, \tau, t_c)$ and $\mathcal{S}_\beta$ for $\beta(s, \tau)$:

$$\mathcal{S}_\alpha = M_\alpha \parallel M_E \parallel M_P \qquad \textit{and} \qquad \mathcal{S}_\beta = M_\beta \parallel M_E \parallel M_P$$

With the above automata that we designed for $\alpha(s, \tau, t_c)$ and $\beta(s, \tau)$, we are now ready to present the main results.

Theorem 1. If the SLA for service s on $\alpha(s, \tau, t_c)$ is violated, either:

– $\mathcal{S}_\alpha$ re-establishes the condition $\alpha(s, \tau, t_c) \geq 1 - \varepsilon_\alpha(s, \tau)$ (thereby satisfying the SLA) within t_G time, or,
– there exists at least one task $\texttt{verify}_\alpha$ in M^α_{V} with a missed deadline.

Proof. At any given time in T:

– If $\alpha(s, \tau, t_c) \geq 1 - \varepsilon_\alpha(s, \tau)$, then the SLA for service availability α is satisfied.
– If the above condition does not hold, on every monitoring window τ, M_α generates a new task `allocate` in M_{A}. In addition, a new task $\texttt{verify}_\alpha$ is generated with a deadline t_G. After a duration of t_G, the **`await`** statement allows M^α_{V} to complete the task $\texttt{verify}_\alpha$ only if the condition $\alpha(s, \tau, t_c) \geq 1 - \varepsilon_\alpha(s, \tau)$ holds. If this is not the case, since t_G has passed, the scheduler generates a missed deadline (moving to its error state).

□

Theorem 2. If the SLA for service s on $\beta(s,\tau)$ is violated, either:

– $\mathcal{S}_\beta$ re-establishes the condition $\beta(s,\tau) \geq 1 - \varepsilon_\beta(s,\tau)$ (thereby satisfying the SLA) within t_G time, or,
– there exists at least one task $\texttt{verify}_\beta$ in M^β_{V} with a missed deadline.

Proof. Similar to the proof of Theorem 1. □

In practice, the guarantee of $\mathcal{S}_\alpha$ and $\mathcal{S}_\beta$ in isolation to eventually evolve the system to satisfy the SLA is not enough. In reality, a service provider tries ensure both simultaneously to reduce their cost of service delivery while ensuring the delivered service is of the expectations agreed upon with the customer. However, these goals conflict. When $\alpha(s,\tau,t_c)$ increases because of adding a new resource, it means that service s costs more, hence $\beta(s,\tau)$ decreases. The same applies in the other direction: increasing $\beta(s,\tau)$ negatively affects $\alpha(s,\tau,t_c)$.

To capture the combined behavior of service availability and budget compliance, we compose them. We define *service sustainability* $\gamma(s,\tau)$ as the composition of $\alpha(s,\tau,t_c)$ and $\beta(s,\tau)$. We present the composition by system automata $\mathcal{S}_\gamma$ as:

$$\mathcal{S}_\gamma = \mathcal{S}_\alpha \parallel \mathcal{S}_\beta$$

Authors in [9] define that a task automata is *schedulable* if there exists no task on the queue that misses its deadline. The next theorem presents the relationship between schedulability analysis of service sustainability and satisfying its SLA.

Theorem 3. If $\mathcal{S}_\gamma$ is *schedulable* given input parameters (τ, t_i, t_G), then the SLA for both service characteristics $\alpha(s,\tau,t_c)$ and $\beta(s,\tau)$ is satisfied within t_G time after a violation.

Proof. When a violation of the SLA occurs in $\mathcal{S}_\gamma$, either $\mathcal{S}_\alpha$ or $\mathcal{S}_\beta$ (or both) start to evolve the service based on Theorems 1 and 2. Therefore, there exists at least one task of $\texttt{verify}_\alpha$ or $\texttt{verify}_\beta$ with a deadline t_G. Hence, if $\mathcal{S}_\gamma$ is schedulable, then neither $\texttt{verify}_\alpha$ nor $\texttt{verify}_\beta$ miss their deadline. Thus, both $\mathcal{S}_\alpha$ and $\mathcal{S}_\beta$ are schedulable. This means that both $\texttt{verify}_\alpha$ and $\texttt{verify}_\beta$ complete successfully. Therefore, the SLA of the service is guaranteed within t_G after a violation in $\mathcal{S}_\gamma$. □

Using the algorithm presented in Chapter 6 [13], we translate the above task automata into traditional timed automata. This allows to leverage well-established model checking techniques such as UPPAAL [2] to determine the schedulability of $\mathcal{S}_\gamma$. Moreover, the results of the schedulability analysis serves as a method to optimize the input parameters of the monitoring model including τ and t_G.

6 Evaluation of the monitoring model

In this section, we evaluate the implementation of the monitoring model.

We set up an environment to evaluate how the monitoring evolves a service according to its SLA. In the environment, a single instance of monitoring platform is present to provide new resources as necessary. Every resource hosts only one service. We define two customers in the environment. For both customers, we deploy the same service, Fredhopper Query API. For every resource that hosts a service, we set up a monitor that measures QPS and reports it to the platform. Both customers run with the same SLA: the QPS expectation is $E(s, \tau, t_c) = 10$ and $\varepsilon_\alpha(s, \tau, t_c) = 0.1$. We launch every customer service with only one resource. Monitors observe the customer service and calculate the service availability of every customer service $\alpha(s, \tau, t_c)$.

We run the environment setup for different monitoring windows $\tau \in \{1, 5, 10\}$ (seconds). We fix the initialization time of a resource to $t_i = 2.5$ seconds. We set $t_G = 300$ seconds; i.e. we verify the service after this time and evaluate if the service is guaranteed based on its SLA.

Figure 6 plots the service availability $\alpha(s, \tau, t_c)$ over time with the different monitoring windows. The following summarizes the behavior:

- As the monitoring window τ increases, the system converges with a slower pace towards the expected $\alpha(s, \tau, t_c)$.
- When the monitoring window is chosen such that $\tau < t_i$, the evolution of the system becomes *non-deterministic*.
- The setting $\tau < t_i$ causes a missed deadline in $\texttt{verify}_\alpha$ because after a duration of t_G the service availability has not yet reached the expected value.

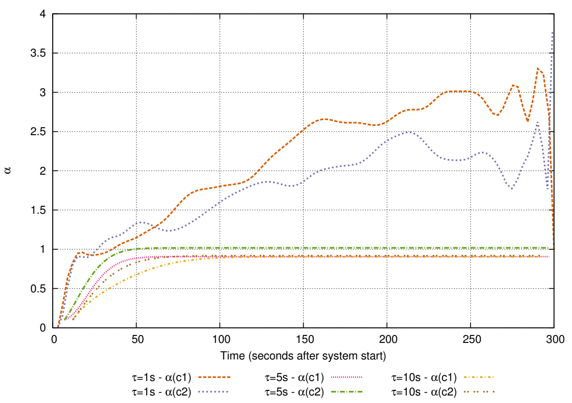


Fig. 6: Evolving $\alpha(s, \tau, t_c)$ with different τ

Every monitoring measurement is performed in a monitoring window τ. Monitoring measurements are aggregated and calculated in every window and form the basis of reactions necessary to evolve the service to meet their SLA. Thus, selection of an appropriate monitoring window length τ is crucial, as we also discussed how schedulability analysis can be used to optimize it. The authors in [11] present that for the same setup and deployment of services, measurements using different monitoring windows yield to very different understanding of service properties such as service availability. Therefore, it is essential to choose the value of τ such that monitoring measurements do not lead to *unrealistic* understanding and inappropriate reactions.

If $\tau < t_i$, Theorem 1 does not hold because every task `allocate` in M_{A} misses its deadline. Thus, it is essential that $\tau \geq t_i$. Analogously, choosing monitoring window as $\tau \gg 2 \times t_i$ also has a counter-productive effect on the service deployments. In a real setting, different services may use different types of resources. In such a setting, the monitoring window should be chosen as the largest t_i of any resource type that is available in the platform: $\tau \geq \mathsf{max}(t_i)\ \forall r \in P$.

7 Future work

We continue to generalize the notion of the distributed service characteristics and investigate how the composition of an arbitrary number of such properties can be formalized and reasoned about. In the context of the ENVISAGE project, industry partners define their service characteristics in this framework and monitor the service evolution. Moreover, the work will be extended to generate parts of the monitoring platform based on an input of different SLA formalizations such as SLA⋆ [17]. Currently, we are integrating our automated monitoring infrastructure into the in-production SDL Fredhopper cloud services (cf. Section 3).

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.
2. G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.
3. J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. T. Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.
4. K. Bratanis, D. Dranidis, and A. J. H. Simons. Towards Run-Time Monitoring of Web Services Conformance to Business-Level Agreements. volume 6303, pages 203–206. Springer, 2010.
5. R. Bubel, A. Flores-Montoya, and R. Hähnle. Analysis of executable software models. In *SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*, pages 1–25, 2014.
6. Y. Chen, S. Iyer, X. Liu, D. Milojicic, and A. Sahai. SLA decomposition: Translating service level objectives to system level thresholds. In *Autonomic Computing, 2007. ICAC'07. Fourth International Conference on*, pages 3–3. IEEE, 2007.
7. A. Coles, A. J. Coles, A. Clark, and S. Gilmore. Cost-sensitive concurrent planning under duration uncertainty for service-level agreements. In *ICAPS*, 2011.
8. M. Comuzzi, C. Kotsokalis, G. Spanoudakis, and R. Yahyapour. Establishing and monitoring SLAs in complex service based systems. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 783–790. IEEE, 2009.
9. E. Fersman, P. Krcal, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.
10. S. Gilmore, L. Gönczy, N. Koch, P. Mayer, M. Tribastone, and D. Varró. Non-functional properties in the model-driven development of service-oriented systems. *Software & Systems Modeling*, 10(3):287–311, 2011.

11. G. Hogben and A. Pannetrat. Mutant Apples: A Critical Examination of Cloud SLA Availability Definitions. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 379–386. IEEE, 2013.
12. Inzinger, Christian and Hummer, Waldemar and Satzger, Benjamin and Leitner, Philipp and Dustdar, Shahram. Generic event-based monitoring and adaptation methodology for heterogeneous distributed systems. *Software – Practice and Experience*, 2014.
13. M. M. Jaghoori. *Time at your service: schedulability analysis of real-time and distributed services*. PhD thesis, Leiden University, 2010.
14. M. M. Jaghoori. Composing real-time concurrent objects refinement, compatibility and schedulability. In *Fundamentals of Software Engineering*, pages 96–111. Springer Berlin Heidelberg, 2012.
15. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects*, pages 142–164. Springer, 2012.
16. E. B. Johnsen, R. Schlatte, and S. L. T. Tarifa. Modeling resource-aware virtualized applications for the cloud in Real-Time ABS. In *Formal Methods and Software Engineering*, pages 71–86. Springer, 2012.
17. K. T. Kearney, F. Torelli, and C. Kotsokalis. SLA$\star$: An abstract syntax for Service Level Agreements. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 217–224. IEEE, 2010.
18. 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.
19. K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: compact data structure and state-space reduction. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 14–24. IEEE, 1997.
20. X. Logean, F. Dietrich, H. Karamyan, and S. Koppenhöfer. Run-time monitoring of distributed applications. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, pages 459–474, 1998.
21. K. Mahbub, G. Spanoudakis, and T. Tsigkritis. Translation of SLAs into monitoring specifications. In *Service Level Agreements for Cloud Computing*, pages 79–101. Springer, 2011.
22. B. Nobakht, F. S. de Boer, and M. M. Jaghoori. The future of a missed deadline. In *Coordination Models and Languages*, pages 181–195. Springer, 2013.
23. B. Nobakht, F. S. de Boer, M. M. Jaghoori, and R. Schlatte. Programming and deployment of active objects with application-level scheduling. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1883–1888. ACM, 2012.
24. F. Raimondi, J. Skene, and W. Emmerich. Efficient online monitoring of web-service SLAs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 170–180. ACM, 2008.
25. P. Y. H. Wong, R. Bubel, F. S. de Boer, M. Gómez-Zamalloa, S. de Gouw, R. Hähnle, K. Meinke, and M. A. Sindhu. Testing abstract behavioral specifications. *STTT*, 17(1):107–119, 2015.
26. J. Woodcock, A. Cavalcanti, J. Fitzgerald, S. Foster, and P. G. Larsen. Contracts in CML. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, pages 54–73. Springer, 2014.

Appendix C

Run-time Deadlock Detection [5]

Run-time Deadlock Detection

Frank S. de Boer[1,3] and Stijn de Gouw[1,2]

[1] CWI, Amsterdam, The Netherlands
[2] SDL, Amsterdam, The Netherlands
[3] Leiden University, The Netherlands

Abstract. This paper reports research that is partly funded by the EU project FP7-610582 Envisage. It describes a method for detecting at run-time deadlock in both multi-threaded Java programs and systems of concurrent objects. The method is based on attribute grammars for specifying properties of message sequences. For multi-threaded Java programs we focus on the actual tool-development which extends the run-time checking of assertions. For concurrent objects which communicate via asynchronous message passing and synchronize on futures which store the return values, we present the underlying theory and sketch its implementation.

1 Introduction

As early as in 1949, Alan Turing suggested the use of assertions in a talk "Checking a Large Routine" at Cambridge, for specifying and proving program correctness. This use of assertions in the logical specification of the mathematical relations between the values of the program variables was further developed by Floyd in inductive assertion networks and by Hoare in a programming logic. Furthermore, checking assertions at run-time is an important practical method for finding bugs.

In [5] we enhanced run-time assertion checking with attribute grammars [11] for describing properties of *histories*, e.g., sequences of method calls and returns. This supports strict programming to interfaces because it allows for interface specifications abstracting from the state as represented by the program variables. In [4] we extended this approach to multi-threaded Java programs which avoids interference problems in a natural manner. In this paper we show how we can express, and detect at run-time, deadlock in both multi-threaded Java programs and Actor-based programs (as for example introduced in [10]) by means of attribute grammars.

Related Work In [4] we showed how our approach to run-time assertion checking can be extended to multi-threaded Java programs while avoiding in a natural manner interference problems. As an example of the generality of our approach, we showed in [4] how to express deadlocks in multi-threaded Java programs. In this paper we detail the actual implementation of this application to deadlock detection in multi-threaded Java programs. We further show how to express and detect deadlock arising in systems of concurrent objects which communicate via *asynchronous method calls* and so-called *futures* which store the return values (as described in [3]).

One of the main related works [1] describes how to detect deadlock *potentials* in multi-threaded programs which may give rise to *false positives* and *false negatives*. We however focus on detecting *actual* deadlocks at run-time.

There exist a variety of *static* techniques for deadlock analysis. Such techniques analyze the source code *without* executing it and aim at establishing absence of deadlock in *all* executions or finding a counter-example, i.e., a deadlocking computation. In general the computational complexity of the algorithms underlying these techniques is a major obstacle to their application to large software systems. Furthermore, their application in general requires certain abstractions which give rise to imprecision. For example, in [6] a CFL-reachability analysis[4] for deadlock in multi-threaded Java programs is introduced which is based on a finite abstraction provided by the underlying call-graphs. As another example, in [7] Dynamic Push down Networks (DPNs) are introduced as an abstract model for parallel programs with (recursive) procedures and dynamic process creation. Further, in [9], [2], and [8] different techniques for the deadlock analysis of systems of concurrent objects are introduced based on a variety of abstractions, e.g., abstract descriptions of methods behaviours.

[4] Here CFL stands for “Context Free Language”.

2 The Framework

This section briefly summarizes the use of attribute grammars in run-time verification as presented in [5]. We use the interface of the Java `BufferedReader` (Figure 1) as a running example to explain the basic modeling concepts.

```
interface BufferedReader {
  void close();
  void mark(int readAheadLimit);
  boolean markSupported();
  int read();
  int read(char[] cbuf, int off, int len);
  String readLine();
  boolean ready();
  void reset();
  long skip(long n);
}
```

Fig. 1. Methods of the BufferedReader Interface

Communication View A communication view is a (possibly partial) mapping which associates a name to each event. Partiality makes it possible to filter out irrelevant events and event names are convenient in referring to events.

Suppose we wish to formalize the following property of the `BufferedReader`:

> The `BufferedReader` may only be closed by the same object which created it, and reads may only occur between the creation and closing of the `BufferedReader`.

This property must hold for the local history of all instances. The intuitive idea behind this property is that the object that opened (created) the buffer "owns" it, and is as such responsible for closing it, but it may pass the buffer on to clients that can read from it (so in particular, reads are allowed by multiple other objects). The communication view in Figure 2 selects the relevant events and associates them with intuitive names: *open*, *read* and *close*.

All return and call events not listed in the view are filtered. Note how the view identifies two different events (calls to the overloaded read methods) by giving them the same name *read*. Though the above communication view contains only provided methods (those listed in the `BufferedReader` interface), required methods (e.g. methods of other interfaces or classes) are also supported. Since messages to such methods are sent to objects of a different class (or interface), one must include the appropriate type explicitly in the method signature. For example, if we additionally include the following event in the view:

```
local view BReaderView grammar BReader.g
specifies java.util.BufferedReader {
  BufferedReader(Reader in) open,
  BufferedReader(Reader in, int sz) open,
  call void close() close,
  call int read() read,
  call int read(char[] cbuf, int off, int len) read
}
```

Fig. 2. Communication view of a BufferedReader

```
call void C.m() out
```

then all call-messages to the method `m` of class `C` sent by a `BufferedReader` are selected and named *out*. In general, incoming messages received by an object correspond to calls of provided methods and returns of required methods. Outgoing messages sent by an object correspond to calls of required methods and returns of provided methods. Incoming call-messages of local histories never involve static methods, as such methods do not have a callee.

Local communication views, such as the one in Figure 2, select messages sent and received by *a single object* of a particular class, indicated by 'specifies java.util.BufferedReader'. In contrast, global communication views select messages sent and received by *any* object during the execution of the Java program. This is useful to specify global properties of a program. In addition to instance methods, calls and returns of static methods can also be selected in global views.

In contrast to interfaces of the programming language, communication views can contain constructors, required methods, static methods (in global views) and can distinguish methods based on return type or method modifiers such as 'static', or 'public'. The following features are supported: constructors, inheritance, dynamic binding, overloading, static methods, access modifiers. In addition to these features, in Section 4 we add support for multi-threading. We associate a grammar to each view. The grammar keyword, followed by a file name indicates the file containing the grammar associated to the view (i.e. Figure 2 refers to the grammar in the file `BReader.g`). The next section discusses grammars in detail.

Grammars The context-free grammar underlying the attribute grammar in Figure 3 generates the valid histories for `BufferedReader`, describing the prefix closure of sequences of the terminals 'open', 'read' and 'close' as given by the regular expression (open read* close). In general, the event names form the terminal symbols of the grammar, whereas the non-terminal symbols specify the structure of valid sequences of events. In our approach, a communication history is valid if and only if it and all its prefixes are generated by the grammar.

We extend the grammar with attributes for specification of the *data-flow* of the valid histories. Each terminal symbol has *built-in* attributes named `caller`, `callee` and the parameter names for respectively the object identities of the caller, callee and actual parameters. Terminals corresponding to method returns additionally have an attribute `result` containing the return value. Non-terminals have *user-defined* attributes to define data properties of sequences of terminals. We extend the attribute grammar with assertions to specify properties of attributes. For example, in the attribute grammar in Figure 3 a user-defined synthesized attribute ‘c’ for the non-terminal ‘C’ is defined to store the identity of the object which closed the `BufferedReader` (and is `null` if the reader was not closed yet). Synthesized attributes define the attribute values of the non-terminals on the left-hand side of each grammar production, thus the ‘c’ attribute is not set in the productions of the start symbol ‘S’.

The assertion allows only those histories in which the object that opened (created) the reader is also the object that closed it. Throughout the paper the start symbol in any grammar is named ‘S’. For clarity, attribute definitions are written between parentheses ‘(’ and ‘)’ whereas assertions over these attributes are surrounded by braces ‘{’ and ‘}’. We use subscripts to distinguish different occurrences of the same non-terminal, i.e., in the grammar below C and C_1 are different occurrences of the non-terminal C.

```
S ::= open R  {assert (open.caller == null || open.caller == R.c ||
                       R.c == null);}
   |  ε
R ::= read R1 (R.c = R1.c;)
   |  C       (R.c = C.c;)
C ::= close C1 (C.c = C1.caller;)
   |  close   (C.c = close.caller;)
   |  ε       (C.c = null;)
```

Fig. 3. Attribute Grammar which specifies that ‘read’ may only be called in between ‘open’ and ‘close’, and the reader may only be closed by the object which opened it.

Assertions can be placed at any position in a production rule and are evaluated at the position they were written. Note that assertions appearing directly before a terminal can be seen as a precondition of the terminal, whereas post-conditions are placed directly after the terminal. This is in fact a generalization of traditional pre- and post-conditions for methods as used in design-by-contract: a single terminal ‘call-m’ can appear in multiple productions, each of which is followed by a different assertion. Hence different preconditions (or post-conditions) can be used for the same method, depending on the context (grammar production) in which the event corresponding to the method call/return appears.

Figure 4 shows a parse tree of the sequence of terminals ‘open read read close’, where the caller of open and close is the same object o_1, but the second read operation is triggered by another object o_2. Terminals - corresponding read, open or close events - are shown as rectangles in the parse tree with a built-in attribute ‘caller’. A circle denotes a non-terminal, with a user-defined attribute ‘c’ for the non-terminals C and R to store the object that last closed it.

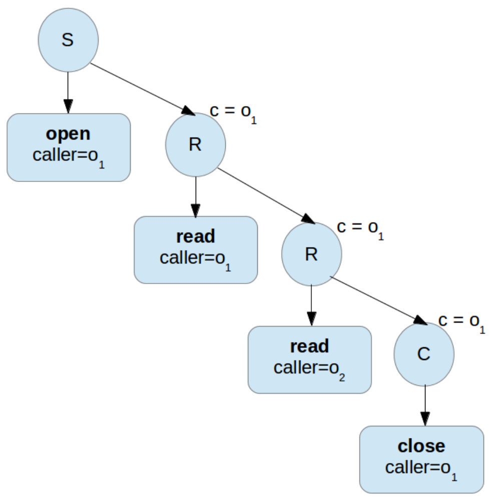


Fig. 4. Parse tree of ‘open read read close’

3 Deadlock Detection for Concurrent Objects

In this section we discuss the run-time detection of deadlock in systems of concurrent objects as described in [3]. Such systems consist of objects which communicate via *asynchronous method calls* and so-called *futures* which store the return values. An asynchronous method call $v = e!m(\bar{e})$ (where $\bar{e}$ denotes the sequence of actual parameters of the call of method m of the called object denoted by e, and where v denotes a future), generates a corresponding *closure* which is stored in the process queue of the callee. A closure consists of a (sequential) statement, e.g., the body of a method, and a local environment specifying the values of the local variables (including the formal parameters). The future variable v stores a reference to the return value (as such it can be passed around). The operation v.get *blocks* the current active closure till the return value has been generated. On the other hand, the operation v.await *suspends* the current active closure by storing it in the process queue till the return value has been generated. This allows so-called *cooperative* scheduling of another closure for execution. All objects are executing their active closures concurrently and fully encapsulate their local data. See Fig. 5 for the formal syntax with the following non-terminals: T for types, P for programs, L for classes, M for methods, sr for statements which return a value, s for any other statement, v for fields and local variables, f for fields, and, finally, x for local variables. By $\overline{X}$, where X denotes a sequence of symbols, we denote a sequence of X's. Types include class names C and types $\uparrow T$ of a future reference to a return value of type T. A program P consists of a sequence of class definitions L which supports class inheritance. A class definition consists of a sequence of method definitions M. A method is defined by its

$$\begin{array}{lll@{\qquad}lll}
T & ::= & C \mid !T \mid \ldots & P & ::= & \overline{L}\ \{\overline{Tx}; sr\} \\
L & ::= & \texttt{class}\ C\ \texttt{extends}\ C\ \{\overline{Tf}; \overline{M}\} & M & ::= & T\ m\ (\overline{Tx})\{\overline{Tx}; sr\} \\
sr & ::= & s; \texttt{return}\ e & s & ::= & v = e \mid \\
 & & & & & v = \texttt{new}\ C() \mid v = v!m(\bar{e}) \mid v = v.\texttt{get} \\
 & & & & & v.\texttt{await} \mid \ldots \\
v & ::= & f \mid x & & &
\end{array}$$

Fig. 5. The language syntax. Variables v are fields (f) or local variables (x), and C is a class name.

return type, the types of the formal parameters, and its body which terminates in a return statement. We abstract from the syntax of the side-effect free expressions e. The main statements of interest are side-effect free assignments $v = e$ to either a field f or a local variable x, object creation $v = \texttt{new}\ C()$, asynchronous method calls $v = v!m(\bar{e})$ and statements $v = v.\texttt{get}$ and $v.\texttt{await}$ which involve polling a future, as described informally above. We assume distinguished local variables `this` and `dest` which denote the executing object and the future `dest` uniquely identifying the corresponding method invocation.

Figure 6 presents a publisher-subscriber pattern which is taken from [3] and (quoting [3]) "wherein an event observed by a sensor is published to objects subscribed to a service. To avoid bottlenecks when publishing an event, the service delegates to a chain of proxy objects, where each proxy object informs both the next proxy and up to a specified limit of subscribing clients. We assume these classes exist: Sensor with method detectEvent, Client with method signal, and a list parametric in type T, with method add."

The operational semantics is defined by a transition relation between global configurations (γ, δ, θ), where γ is the set of active closures, δ the set of suspended closures, and θ represents the (global) heap. A global heap assigns a local state to both the existing objects and futures. The local state $\theta(o)$ of an object o is an assignment of values to its fields, whereas the local state $\theta(r)$ of a future (reference) r is simply a value of the corresponding type or the value $\perp$ which stands for "undefined" (or "uninitialized"). A closure c is a pair (τ, sr), where τ is an assignment of values to the local variables.

Fig. 7 gives the main operational rules. Here $\theta_\tau(e)$ denotes the value of the (side-effect free) expression e in the global heap θ and local environment τ, e.g., $\theta_\tau(x) = \tau(\texttt{this})$, for every local variable x (including `this`) , and $\theta_\tau(f) = \theta(\tau(\texttt{this}))(f)$, for every field f. For any sequence of expressions $\bar{e}$, we denote by $\theta_\tau(\bar{e})$ the corresponding sequence of values. Further, by $\theta[o.f \mapsto d]$ we denote the update of θ resulting from assigning the value d to the field f of object o, e,g., $\theta[o.f \mapsto d](o)(f) = d$. Similarly, by $\theta[r \mapsto d]$ we denote the update of θ resulting from assigning the return value d to the future reference r. The above notation is extended in the obvious manner to simultaneous updates. The rule CALL describes an asynchronous method call. It generates a fresh future reference r and a closure $cl(C, m, o, r, \theta_\tau(\bar{e}))$ which consists the body of the method (as defined in class

```
class Service {
       Sensor sensor; Proxy proxy;
       Service(int val) {
              sensor = new Sensor; proxy = new Proxy(val);
       }
       void subscribe(Client cl) { proxy!add(cl) }
       void process() {
         while (true) {
              !Event fut = sensor!detectEvent();
              proxy!publish(fut);
              await fut?;}
       }
}

class Proxy {
       List<Clients> myClients; Proxy nextProxy;
       Event ev; int limit;
       Proxy(int k) {
               limit = k; myClients = new List(); nextProxy = null;
       }
       void add(Client cl) {
              if myClients.length < limit { myClients.add(cl); }
              else { if nextProxy == null nextProxy = new Proxy(limit);
                     nextProxy.add(cl); }
       }
       void publish(!Event fut) {
              await fut?;
              if nextProxy != null { nextProxy!publish(fut); }
              ev = fut.get();
              for Client client : myClients { client!signal(ev); }
       }
}
```

Fig. 6. Publisher-Subscriber Pattern

$$\text{(CALL)}\quad \frac{r \notin \mathrm{dom}(\theta) \quad \theta_\tau(v) = o \quad c = cl(C, m, o, r, \theta_\tau(\bar{e}))}{(\gamma \cup \{(\tau, u = v!m(\bar{e}); sr)\}, \delta, \theta) \to (\gamma \cup \{(\tau, u = r; sr)\}, \delta \cup \{c\}, \theta[r \mapsto \perp])}$$

$$\text{(AWAIT1)}\quad \frac{\theta_\tau(v) \neq \perp}{(\gamma \cup \{(\tau, v.\texttt{await}; sr)\}, \delta, \theta) \to (\gamma \cup \{(\tau, sr)\}, \delta, \theta)}$$

$$\text{(AWAIT2)}\quad \frac{\theta_\tau(v) = \perp}{(\gamma \cup \{(\tau, v.\texttt{await}; sr)\}, \delta \cup \{c\}, \theta) \to (\gamma, \delta \cup \{(\tau, v.\texttt{await}; sr)\}, \theta)}$$

$$\text{(SCHED)}\quad \frac{c = (\tau, sr) \quad \forall (\tau', sr') \in \gamma : \tau'(\texttt{this}) \neq \tau(\texttt{this})}{(\gamma, \delta \cup \{c\}, \theta) \to (\gamma \cup \{c\}, \delta, \theta)}$$

$$\text{RETURN}\quad (\gamma \cup \{(\tau, \texttt{return}\ e)\}, \delta, \theta) \to (\gamma, \delta, \theta[\tau(\texttt{dest}) \mapsto \theta_\tau(e)])$$

Fig. 7. The operational semantics.

C) and a local environment τ' such that $\tau'(\texttt{this}) = o$, $\tau(\texttt{dest}) = r$, and $\tau'(\bar{x}) = \theta_\tau(\bar{e})$, where $\bar{x}$ are the formal parameters. This closure is added to the set of suspended closures and the value of r is set to $\perp$. The rule AWAIT1 describes the continuation of the flow of control in case the polled future stores a returned value, whereas rule AWAIT2 describes suspending the active closure, in case the polled future is still undefined. The rule SCHED allows to schedule a suspended closure in case the object is idle, i.e., it has no active closure. This rule abstracts from the particular scheduling policy used and possible optimizations avoiding busy waiting, i.e., scheduling blocked await/get statements. The last rule RETURN describes the effect of the return statement in terms of the initialization of the corresponding future `dest`.

Polling futures gives rise to a dependency relation between method invocations, e.g., a method invocation executing an await statement $v.\texttt{await}$ depends on the execution of the method invocation uniquely identified by the future v to return a value. A cycle in this dependency relation between method invocations implies that we have a deadlock in the set of involved method invocations.

Definition 1 (Deadlock). *Deadlock arises in a global configuration* (γ, δ, θ) *when there exist closures* $c_i = (\tau_i, s_i; sr_i) \in \gamma \cup \delta$, *where* s_i *either denotes an await statement* $v_i.\texttt{await}$ *or a get operation* $v = v_i.\texttt{get}$, *such that* $\tau_i(v_i) = \tau_{i \oplus 1}(\texttt{dest})$, $i = 1, \ldots, n$ *(*$\oplus$ *here denotes addition modulo* n*).*

In order to detect deadlock, the built-in attributes of events generated by asynchronous method calls denote, besides the caller, callee and the parameters, the generated future uniquely identifying the corresponding method invocation, which is denoted by the attribute name `dest`. The built-in attributes of events generated by return statements consist of the executing object (denoted by the attribute name `this`), the value returned (denoted by the name `val`) and the corresponding future (denoted by `dest`). The built-in attributes of events generated by await statements and assignments involving the `get` operation consist of the polled future (denoted by `fut`) and the future uniquely identifying the executing ("polling") method invocation (denoted by `dest`). In a (asynchronous) communication view we then can specify which synchronization events, i.e., , `await`/`get` operations on futures which refer to the return value of a certain method, we want to observe by means of the specifications `await` $C.m$ (and `get` $C.m$). By `await any` (`get any`) we refer to any `await` (`get`) operation.

Surprisingly, we can detect deadlock by *only* observing `await` and `get` operations, by means of the built-in attributes `fut` and `dest`, which denote the future which is polled and the future uniquely identifying the polling method invocation, respectively. This results in the following (global) communication view which maps every await/get operation on the same grammar token `poll`.

```
global view DeadlockMyProgram grammar deadlock.g {
   await any poll
   get any poll
}
```

Fig. 8. Global asynchronous communication view

The following grammar then generates, for each sequence of `poll` tokens, a corresponding graph of futures and checks absence of cycles.

$$
\begin{array}{rl}
S ::= & \texttt{poll} \ \{ \ \texttt{g.addEdge(poll.dest,poll.fut);} \ \} \\
\mid & \epsilon \ \{\texttt{assert g.noCycle();}\}
\end{array}
$$

Fig. 9. Attribute Grammar for Deadlock Detection

At run-time a given program instrumented with history updates which consist of adding a `poll` token just *before* every execution of an `await`/`get` operation then can be checked for absence of deadlock by simply parsing the history according to the above attribute grammar. Clearly, a deadlock will generate an assertion failure. It is less obvious that an assertion failure indeed corresponds with a deadlock. Note for example that edges are *not* removed when a future is initialized. However, because futures are assigned to only once we can argue as follows. Let (γ, δ, θ) result from the execution of an active closure which generates an assertion failure caused by the addition of an edge (r, r') in the graph denoted by g. Let $r' = r_0, \ldots, r_{n-1} = r$ be the nodes in g such that between r_i and $r_{i\oplus 1}$ ($\oplus$ here denotes addition modulo n) there exists an edge. We have to show that for $i = 0, \ldots, n-1$ there exist closures $c_i = (\tau_i, sr_i) \in \gamma \cup \delta$ such that $\tau_i(\texttt{dest}) = r_i$ and the initial statement of sr_i involves an await or get operation on the future $r_{i\oplus 1}$. We show by induction that there exists such a closure c_i. For $i = n-1$ let $c_{n-1} = (\tau, sr)$ be the closure in γ such that $\tau(\texttt{dest}) = r$ and the initial statement of sr involves an await or get operation on the future r'. Next let $0 < i < n-1$ and $c_i = (\tau_i, sr_i)$ be the closure in $\gamma \cup \delta$ such that $\tau_i(\texttt{dest}) = r_i$ and the initial statement of sr_i involves an await or get operation on the future $r_{i \oplus 1}$. Let $c_{i-1} = (\tau_{i-1}, sr_{i-1})$ be the closure that resulted from the generation of the edge (r_{i-1}, r_i), i.e., $\tau_{i-1}(\texttt{dest}) = r_{i-1}$. Since $\tau_i(\texttt{dest}) = r_i$ and $c_i \in \gamma \cup \delta$ it follows from the operational semantics that $\theta(r_i) \neq \perp$. Therefore $c_{i-1} \in \gamma \cup \delta$ and the initial statement of sr_{i-1} involves an await or get operation on the future r_i.

4 Deadlock Detection for Multi-threaded Java Programs

Deadlocks in multi-threaded Java programs can arise from `Lock` objects, or from `synchronized` methods and statements. Deadlocks caused through using `Lock` objects can be detected in a straightforward manner by tracking calls to the `lock()` and `unlock()` methods, and do not require an extension to the framework introduced in the previous section. Thus we focus on deadlocks arising from `synchronized` methods. The program in Figure 10 will be used as a running example. Depending on the scheduling, it can contain a deadlock: if the first thread starts executing `alphonse.bow(gaston)` but does not execute the call to `bowBack` before the second thread executes `gaston.bow(alphonse)`, the program deadlocks.

We specify different aspects of a multi-threaded program with the help of the following three perspectives:
Thread view: here we specify the behavior of each thread in isolation.
Object view : here we specify the behavior of objects individually.
Global view: here we specify global properties of a program.
All of the above views can be supported by a single formalism: attribute grammars extended with assertions, but the underlying history on which the grammar is evaluated differs between the various perspectives. The next subsection discusses multi-threaded events, and the required extensions to communication views to support the perspectives.

All grammars in this section are given in ANTLR [12] syntax: the input format of the underlying tool implementation (all grammars have been fully implemented and were used for run-time checking). The syntax of ANTLR grammars is close to Java: comments start with '//', and the actions (attribute definitions or assertions) in the grammar are ordinary Java statements, surrounded by the braces '' and ''. The left-hand side and right-hand side of a production are separated by a colon. ANTLR supports Extended BNF (EBNF): operators from regular expressions can be used in productions, such as the '*' (zero or more repetitions) and '?' (an optional symbol). Figure 12 shows an example grammar (the figure is discussed in more detailed in Section 4.2).

4.1 Multi-threaded Events

In a multi-threaded environment, events occur in different threads. Thus the first new ingredient compared to Section 2 is to keep track of the thread identity for

```java
package nl.cwi.saga.deadlock;

import java.io.*;

public class Deadlock {
    public static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s"
                    + " has bowed to me!%n",
                this.name, bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s"
                    + " has bowed back to me!%n",
                this.name, bower.getName());
        }
    }

    public static void main(String[] args) {
        final Friend alphonse = new Friend("Alphonse");
        final Friend gaston = new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
```

Fig. 10. Example program with a potential deadlock. Source: `https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html`

each event. This is achieved with a new built-in attribute `Long threadId`. This attribute will be used in the deadlock detector to determine the events that wait on the completion of other events in a different thread.

In multi-threaded programs, due to scheduling and locking, there can be a delay between when a method is called, and when its body starts executing. For synchronized methods, a method call indicates that a lock was requested, whereas the start of the execution of a method body indicates that the lock was acquired successfully. To distinguish these two events, we introduce an 'exec' event, that indicates the start of execution of a method body (and thus implies acquisition of the lock). Returns of synchronized methods indicate the release of the lock.

4.2 Multi-threaded Perspectives

Thread View In the thread perspective, we specify the behavior of each thread in isolation. Each thread has its own dedicated history, and the grammar generates the set of valid histories of the thread. Semantically, such thread-local histories can be obtained from the global history by projection on the value of the `threadId` attribute (which , as mentioned above, stores the identity of the thread in which the event occurred).

We illustrate the thread view using the running example (Figure 10). Figure 11 presents the corresponding communication view, introducing the grammar terminals "BOW" and "BOWBACK" for the corresponding events. Only events from implementations of the `Fork` interface with `synchronized` versions of `get` and `release` are selected.

```
thread view BowHistory grammar Bow.g {
    call public synchronized void
         Deadlock.Friend.bow(Friend bower) BOW,
    call public synchronized void
         Deadlock.Friend.bowBack(Friend bower) BOWBACK
}
```

Fig. 11. Communication view of `bow` and `bowBack`

We will specify that each person bows back to the same person that bowed to them. This intuitive property is formalized by the ANTLR grammar in Figure 12. It specifies that each thread must first call `bow`, then `bowBack`, and (using an assertion) that the parameter of `bow` denotes the same object as the callee of `bowBack`.

Object View In the object view of a Java program, we specify the interactions of a single object with a communication view and corresponding grammar. The

```
grammar Bow;

///////////// HEADERS

//////////////////////// start ::= s EOF //////////////////////////////
start : s EOF;

/////////////////////// s ::= BOW BOWBACK? | /\ //////////////////////
s : BOW
    (BOWBACK {assert $BOW.bower() == $BOWBACK.callee();})?
  | ;
```

Fig. 12. ANTLR attribute grammar specifying bowing behavior

grammar generates the set of all valid traces of events that the object engages in. In a multi-threaded setting, several threads can be active (executing) in a single object, thus the object view is particularly useful for specifying (constrain) the order between events from different threads active in the same object. Intuitively, the local object histories can be obtained from the global history by projection on the values of the built-in attributes `caller` (for calls made by the object) and `callee` (for calls to the object).

For the bow-bowBack example, the object view is uninteresting: all interleavings/orderings between bows and bowBacks from different threads are allowed. A useful application of the object view is illustrated by the communication view in Figure 2 and grammar in Figure 3 in Section 2.

Global View The global view treats the Java program as a single entity that we wish to specify. The grammar generates the set of all valid *global* traces of the entire program. The user can specify the desired interleavings between events from different threads.

We use the global view for our deadlock detector. A thread blocks if it calls a synchronized method on an object that is already locked by another thread. The general idea is to build a directed "wait-for" graph to capture such dependencies between threads. A deadlock corresponds to a cycle in the wait-for graph.

In more detail, the nodes of the graph are thread id's, and there is an edge from t_1 to t_2 if t_1 calls a method on some object that is locked by t_2.

The view depicted in Figure 13 selects the events relevant for deadlock detection. Note that we do not need to distinguish whether a certain event arose from `bow` or `bowBack`: the only information needed to identify deadlocks is which thread has requested/acquired/released the lock for which objects. Thus the calls to `bow` and `bowBack` are identified (mapped to the same terminal). The terminal "REQ" signifies requesting a lock, "ACQ" events are generated if a lock was acquired, and "REL" denotes the release of a lock.

Figure 14 shows an ANTLR attribute grammar that asserts no deadlock has occurred. To that end, a wait-for graph is built in the grammar productions

```
global view DeadlockHistory grammar Deadlock.g {
    call public synchronized void
            Deadlock.Friend.bow (Deadlock.Friend bower) REQ,
    call public synchronized void
            Deadlock.Friend.bowBack(Deadlock.Friend bower) REQ,

    exec public synchronized void
            Deadlock.Friend.bow (Deadlock.Friend bower) ACQ,
    exec public synchronized void
            Deadlock.Friend.bowBack(Deadlock.Friend bower) ACQ,

    return public synchronized void
            Deadlock.Friend.bow (Deadlock.Friend bower) REL,
    return public synchronized void
            Deadlock.Friend.bowBack(Deadlock.Friend bower) REL
}
```

Fig. 13. Global communication view

with the help of two inherited attributes (syntactically in the ANTLR grammar, those are passed as parameters to the “s” non-terminal):

– An attribute `reqLock` of type `Map<Long, Object>` that maps a thread id (a `Long`) to the object for which it requested, but has not yet acquired the lock.
– An attribute `hasLock` of type `Map<Long, Map<Object, Integer> >`. Given a thread id and an object, this map returns the number of times the lock on that object has been acquired but not released by the thread[5].

The attributes are updated in the grammar productions. In particular, the two maps are initialized to empty by the `start` production (line 6–7). Further:

– The production with the “REQ” terminal (line 12–16) signifies the request of a lock on the callee, correspondingly, in the grammar production we insert the thread identity and callee identity into the `reqLock` map.
– The production with terminal “ACQ” (line 18–31) signifies that the thread has successfully acquired the lock on the callee. Since the lock request for the callee is not pending anymore, the thread id is removed from the `reqLock` map. Additionally we increase the number of locks (due to re-entrance, a lock may have been acquired for that object already by the thread) that that thread has on the callee in the `hasLock` map.
– The “REL” terminal (line 33-42) signifies the release of a lock. In the grammar production we therefore decrease the number of locks that the thread has on the object. If the count becomes 0, the entry is removed.
– The last production (the empty production, line 44-60) builds the wait-for graph: an edge is drawn from thread t_1 to thread t_2 if t_1 requests a lock

[5] Due to re-entrance, locks in Java can be acquired more than once by the same thread.

```
grammar Deadlock;

///////////// HEADERS

//////////////////////// start ::= s EOF////////////////////////////////
start : s[new HashMap<Long, Object>(),
         new HashMap<Long, Map<Object, Integer> >()]
       EOF;

//////////////////////// s ::= (REQ | ACQ | REL)* /////////////////////
s[Map<Long, Object> reqLock, Map<Long, Map<Object, Integer> > hasLock] :
    REQ
    {
           reqLock.put($REQ.threadId(), $REQ.callee());
    }
    s[reqLock,hasLock]

  | ACQ
    {
        reqLock.remove($ACQ.threadId());
        Map<Object, Integer> m = hasLock.get($ACQ.threadId());
        int newCnt = 1;
        if(m == null) {
            m = new HashMap<Object, Integer>();
            hasLock.put($ACQ.threadId(), m);
        } else if(m.get($ACQ.callee()) != null)
            newCnt = m.get($ACQ.callee())+1;
        m.put($ACQ.callee(), newCnt);
    }

    s[reqLock,hasLock]

  | REL
    {
        Map<Object, Integer> m = hasLock.get($REL.threadId());
        Integer cnt = m.get($REL.callee());
        if(cnt == 1)
            m.remove($REL.callee());
        else
            m.put($REL.callee(), cnt-1);
    }
    s[reqLock,hasLock]

  |
  {
    DirectedGraph<Long, DefaultEdge> g =
        new DefaultDirectedGraph<Long, DefaultEdge>(DefaultEdge.class);
    for(Long rl : reqLock.keySet() ) {
        for(Long hl : hasLock.keySet() ) {
            if(rl != hl && hasLock.get(hl).containsKey(reqLock.get(rl))) {
                g.addVertex(rl);
                g.addVertex(hl);
                g.addEdge(rl, hl);
            }
        }
    }

    CycleDetector<Long, DefaultEdge> d =
        new CycleDetector<Long, DefaultEdge>(g);
    assert !d.detectCycles();
  };
```

owned by t_2. If $t_1 = t_2$ then t_1 has requested a lock that it already owns. In that case the lock can be acquired (no deadlock), thus we insert the edge only if $t_1 \neq t_2$. Since a cycle now corresponds to a deadlock, the assertion (line 60) is true if and only if there is no deadlock.

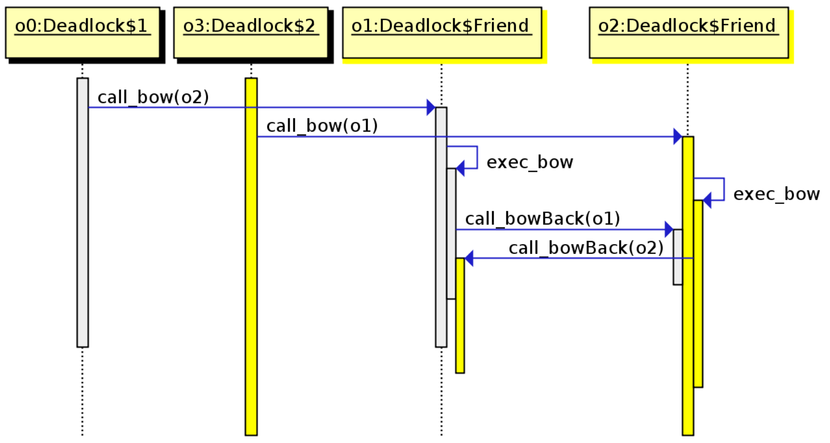


Fig. 15. Sequence diagram of a deadlocking executing of program Figure 10.

As observed previously, there are schedulings for which the program in Figure 10 deadlocks. We executed the program, checking for deadlocks using the given attribute grammar and encountered a deadlocking scenario. Our run-time checker prints certain information to aid debugging and isolate errors when an assertion fails or a parse error occurs: a stack trace that indicates the line of code where the error occurred, and a textual representation of the history that violated the specification. For example, the stack trace in Figure 16 shows that execution failed at line 18 in the file Deadlock.java (Figure 10).

That textual representation of the history can be visualised by the Quick Sequence Diagram Editor sdEdit. Figure 15 shows a visualization by sdEdit of a deadlocking trace. sdEdit gives each thread a color: in our case, gray(ish) and yellow. After the `exec_bow`-events, the gray thread owns the lock on o_1 and the yellow thread has the lock on o_2. With the two `call_bowBack`-events, the gray thread requests the lock for o_2 and the yellow thread requests the lock for o_1, thereby causing a deadlock.

5 Tool Architecture

For practical purposes, an important design goal of our run-time checker SAGA was to allow the use of up-to-date versions of the Java language. In particular,

```
java.lang.AssertionError
 at DeadlockParser.s(DeadlockParser.java:229)
 at DeadlockParser.s(DeadlockParser.java:146)
 at DeadlockParser.s(DeadlockParser.java:146)
 at DeadlockParser.s(DeadlockParser.java:176)
 at DeadlockParser.s(DeadlockParser.java:176)
 at DeadlockParser.s(DeadlockParser.java:146)
 at DeadlockParser.s(DeadlockParser.java:146)
 at DeadlockParser.start(DeadlockParser.java:68)
 at DeadlockHistoryAspect$DeadlockHistory.parse
                               (DeadlockHistoryAspect.java:502)
 at DeadlockHistoryAspect$DeadlockHistory.update
                                (DeadlockHistoryAspect.java:603)
 at DeadlockHistoryAspect.ajc$before$DeadlockHistoryAspect$5$e8e2469d
                                (DeadlockHistoryAspect.java:242)
 at Deadlock$Friend.bow(Deadlock.java:18)
 at Deadlock$2.run(Deadlock.java:36)
 at java.lang.Thread.run(Thread.java:745)
```

Fig. 16. Assertion failure in attribute grammar.

updates to the compilers should not break SAGA (in contrast, previous run-time checkers for JML specifications used a proprietary Java compiler which was not kept in sync with the Java language). The input of SAGA consists of a specification in the form of an attribute grammar with assertions, accompanied by a communication view. The output of SAGA is an AspectJ program for the generation of the events specified by the communication views (see [5]).

Choosing AspectJ as the output language of SAGA, allows the use of modern Java language versions, including the latest Java 8. AspectJ is tailored to the interception of events and as such is a most natural target language. An alternative approach would to instrument the program with a self-developed component of SAGA. But this is difficult because in general the instrumentation cannot be restricted locally to the methods that must be monitored. For example, since the identity of the caller is a built-in attribute of the grammar terminal, we cannot get away with instrumenting only the monitored methods, as one does not have access to the low-level stack in Java. Thus the identity of the callee is not accessible. This means that all call-sites should be instrumented.

However, the use of AspectJ raises certain challenges: we are now bound by limitations of Java. For a debugging tool such as a run-time checker, it would be convenient to have some control or access to various elements from the underlying execution platform, but this is often prohibited or even impossible in Java. For example: in a multi-threaded environment, during the evaluation of the specifications (i.e. the attribute grammar), another thread can potentially modify the heap. This would mean that different parts of the specification are evaluated in different states. Consider for example the assertion `assert x==x;`, where x is a field of an object. If after retrieving the value of the first occurrence of x *another* thread modifies x then the assertion may evaluate to false! This problem can be prevented if the run-time checker had control over the execution platform: it could then stall the other threads while a specification is evaluated. In [4] we discuss how we solved this without having control over the execution platform, and without stalling other threads (since this can cause a severe performance loss). A second implication arising from using AspectJ as target language is that to print an accurate sequence diagram, we must distinguish objects with different identities in the diagram. In Java, one can *test* objects for equality (using "=="), but in general there is no string that identifies each object uniquely (for example, the memory location for the object would qualify, but it is not accessible in Java). Thus SAGA generates a unique ID itself for each object appearing in the history.

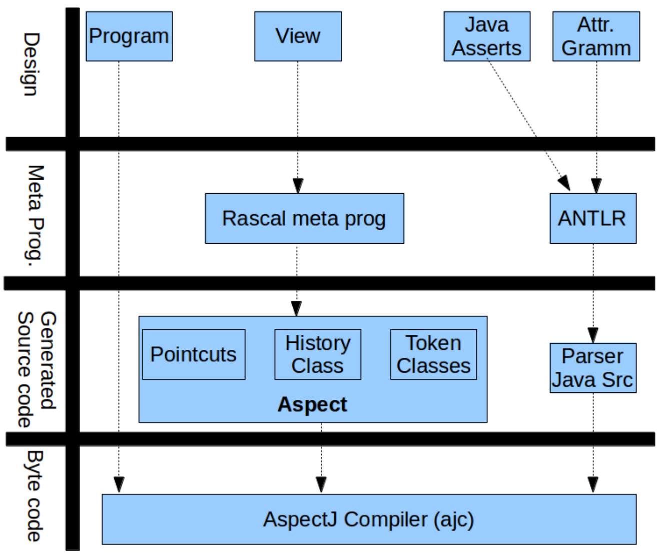


Fig. 17. SAGA Tool Architecture

Figure 17 shows an overview of the resulting tool architecture. It consists of an integration of four different components: a state-based assertion checker, a parser generator, a monitor to intercept events and a general tool for meta-programming. This architecture is further discussed in [4].

6 Conclusion and Future Work

We discussed a method for the run-time detection of deadlock in both multi-threaded Java programs and systems of concurrent objects. The new version of

SAGA which implements this method for multi-threaded Java programs can be obtained from `https://github.com/cwi-swat/saga`. Although we illustrated our framework for detecting deadlock for multi-threaded Java programs which use synchronized methods, general locks as provided in the package `java.util.concurrent.locks` can be handled just as easily by tracking the methods `lock`, `tryLock` and `unlock` in the communication view. What remains to be done is extending SAGA to deadlock detection of concurrent objects as described in this paper. In general, future work will focus on further improving and extending the method by applying it to (industrial) case studies.

Acknowledgement

We thank the anonymous reviewers for their most constructive and helpful comments.

References

1. R. Agarwal, S. Bensalem, E. Farchi, K. Havelund, Y. Nir-Buchbinder, S. D. Stoller, S. Ur, and L. Wang. Detection of deadlock potentials in multithreaded programs. *IBM Journal of Research and Development*, 54(5):3, 2010.
2. F. S. de Boer, M. Bravetti, I. Grabe, M. D. Lee, M. Steffen, and G. Zavattaro. A petri net based analysis of deadlocks for active objects and futures. In *Formal Aspects of Component Software, 9th International Symposium, FACS 2012, Mountain View, CA, USA, September 12-14, 2012. Revised Selected Papers*, pages 110–127, 2012.
3. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 316–330, 2007.
4. F. S. de Boer and S. de Gouw. Run-time checking multi-threaded java programs. In *42nd International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM.* Lecture Notes in Computer Science, 2016.
5. F. S. de Boer, S. de Gouw, E. B. Johnsen, A. Kohn, and P. Y. H. Wong. Run-time Assertion Checking of Data- and Protocol-oriented Properties of Java Programs: An Industrial Case Study. *T. Aspect-Oriented Software Development*, 11:1–26, 2014.
6. F. S. de Boer and I. Grabe. Automated deadlock detection in synchronized reentrant multithreaded call-graphs. In *SOFSEM 2010: Theory and Practice of Computer Science, 36th Conference on Current Trends in Theory and Practice of Computer Science, Spindleruv Mlýn, Czech Republic, January 23-29, 2010. Proceedings*, pages 200–211, 2010.
7. T. M. Gawlitza, P. Lammich, M. Müller-Olm, H. Seidl, and A. Wenner. Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, pages 199–213, 2011.

8. E. Giachino, C. A. Grazia, C. Laneve, M. Lienhardt, and P. Y. H. Wong. Deadlock analysis of concurrent objects: Theory and practice. In *Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings*, pages 394–411, 2013.
9. E. Giachino and C. Laneve. Analysis of deadlocks in object groups. In *Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*, pages 168–182, 2011.
10. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag, 2011.
11. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.
12. T. Parr. *The Definitive ANTLR Reference.* Pragmatic Bookshelf, 2007.

Appendix D

Declarative Elasticity in ABS [7]

Declarative Elasticity in ABS[⋆]

Stijn de Gouw[1], Jacopo Mauro[3], Behrooz Nobakht[2], and Gianluigi Zavattaro[4]

[1] Fredhopper, Netherlands
[2] Leiden University, Netherlands
[3] University of Oslo, Norway
[4] University of Bologna/INRIA, Italy

Abstract. Traditional development methodologies that separate software design from application deployment have been replaced by approaches such as continuous delivery or DevOps, according to which deployment issues should be taken into account already at the early stages of development. This calls for the definition of new modeling and specification languages. In this paper we show how deployment can be added as a first-class citizen in the object-oriented modeling language ABS. We follow a declarative approach: programmers specify deployment constraints and a solver synthesizes ABS classes exposing methods like `deploy` (resp. `undeploy`) that executes (resp. cancels) configuration actions changing the current deployment towards a new one satisfying the programmer's desiderata. Differently from previous works, this novel approach allows for the specification of incremental modifications, thus supporting the declarative modeling of elastic applications.

1 Introduction

Software applications deployed and executed on cloud computing infrastructures should flexibly adapt by dynamically acquiring or releasing computing resources. This is necessary to properly deliver to the final users the expected services at the expected level of quality, maintaining an optimized usage of the computing resources. For this reason, modern software systems call for novel engineering approaches that anticipate the possibility to reason about deployment already at the early stages of development.

Modeling languages like TOSCA [21], CloudML [16], and CloudMF [13] have been proposed to specify the deployment of software artifacts, but they are mainly intended to express deployment of already developed software. An integration of deployment in software modeling is still far from being obtained in the current practices. To cover this gap, in this paper we address the problem of extending the ABS (Abstract Behavioural Specification) language [2] with linguistic constructs and mechanisms to properly specify deployment. Following [9]

[⋆] Supported by the EU projects FP7-610582 *Envisage: Engineering Virtualized Services* (`http://www.envisage-project.eu`) and H2020-644298 *HyVar: Scalable Hybrid Variability for Distributed, Evolving Software Systems* (`http://www.hyvar-project.eu`).

our approach is declarative: the programmer specifies deployment constraints and a solver computes actual deployments satisfying such constraints. In previous work [10] we presented an external engine able to synthesize ABS code specifying the initial static deployment; in this paper we fully integrate this approach in the ABS language allowing for the declarative specification of the incremental upscale/downscale of the modeled application depending, e.g., on the monitored workload or the current level of resource usage.

ABS is an object-oriented modeling language with a formally defined and executable semantics. It includes a rich tool-chain supporting different kinds of analysis (like, e.g., logic-based modular verification [11], deadlock detection [15], and cost analysis [3]). Executable code can be automatically obtained from ABS specifications by means of code generation. ABS has been mainly used to model systems based on asynchronously communicating concurrent objects, distributed over Deployment Components corresponding to containers offering to objects the resources they need to properly run. For our purposes, we adopted ABS because it allows the modeling of computing resources and it has a real-time semantics reflecting the way in which objects consume resources. This makes ABS particularly suited for modeling and reasoning about deployment.

Our initial proposal for the declarative modeling of deployment into ABS [10] was based on three main pillars: (i) classes are enriched with annotations that indicate functional dependencies of objects of those classes as well as the resources they require, (ii) a separate high-level language for the declarative specification of the deployment, (iii) an engine that, based on the annotations and the programmer's requirements, computes a fully specified deployment that minimizes the total cost of the system. The computed deployment is expressed in ABS and can be manually included in a main block.

The work in [10] had two main limitations: (i) there was no way to express incremental deployment decisions like, e.g., the need to upscale or downscale the modeled system at run-time and (ii) there was no real integration of the code synthesized by the engine in the corresponding ABS specification. In this paper we address these limitations by promoting the notion of deployment as a first-class citizen of the language. During a pre-processing phase, the new tool SmartDepl generates classes exposing the methods `deploy` and `undeploy` to upscale and downscale the system. The deployment requirements can now also reuse already deployed objects just specifying which existing objects could be used, and how they should be connected with new objects to be freshly deployed. This has been the fundamental step forward that allowed us to support incremental modification of the current deployment. Moreover, other relevant contributions of this paper are (i) a more natural high-level language for the specification of requirements that now supports universal and existential quantifiers, and (ii) the usage of the delta modules and the variability modeling features of the ABS framework [7] to automatically and safely inject the deployment instructions into the existing ABS code.

Our ABS extension and the realization of the corresponding SmartDepl tool have been driven by Fredhopper Cloud Services, an industrial case-study of the

European FP7 Envisage project. The Fredhopper Cloud Services offer search and targeting facilities on a large product database to e-Commerce companies. Depending on the specific profile of an e-Commerce company Fredhopper has to decide the most appropriate customized deployment of the service. Currently, such decisions are taken manually by an operation team which decides customized, hopefully optimal, service configurations taking into account the tension among several aspects like the level of replications of critical parts of the service to ensure high availability. The operators manually perform the operations to scale up or down the system and this usually causes the over-provision of resources for guaranteeing the proper management of requests during a usage peak. With our extension of ABS, we have been able to realize a new modeling of the Fredhopper Cloud Services in which both the initial deployment and the subsequent up- and down-scale is expected to be executed automatically. This new model is a first fundamental step towards a new more efficient and elastic deployment management of the Fredhopper Cloud Services.

Structure of the paper Section 2 describes the Fredhopper Cloud Services case-study. Section 3 reports the ABS deployment annotations that we already defined in [10]. Section 4 presents the new high-level language for the specification of deployment requirements while Section 5 discusses the corresponding solver. Finally, the application of our technique to the Fredhopper Cloud Services use-case is reported in Section 6. Section 7 discuss the related literature while in Section 8 we draw some concluding remarks.

2 The Fredhopper Cloud Services

Fredhopper provides the Fredhopper Cloud Services to offer search and targeting facilities on a large product database to e-Commerce companies as services (SaaS) over the cloud computing infrastructure (IaaS). The Fredhopper Cloud Services drives over 350 global retailers with more than 16 billion in online sales every year. A customer (service consumer) of Fredhopper is a web shop, and an end-user is a visitor of the web shop.

The services offered by Fredhopper are exposed at endpoints. In practice, these services are implemented to be RESTful and accept connections over HTTP. Software services are deployed as *service instances*. Each instance offers the same service and is exposed via Load Balancer endpoints that distribute requests over the service instances.

The number of requests can vary greatly over time, and typically depends on several factors. For instance, the time of the day in the time zone where most of the end-users are plays an important role (typical lows in demand are observed between 2 am and 5 am). Figure 1 shows a real-world graph for a single day (with data up to 18:00) plotting the number of queries per second (y-axis, ranging from 0-25 qps, the horizontal dotted lines are drawn at 5,10,15 and 20 qps) over the time of the day (x-axis, starting at midnight, the vertical dotted lines indicate multiples of 2 hours). The 2a - 5am low is clearly visible.

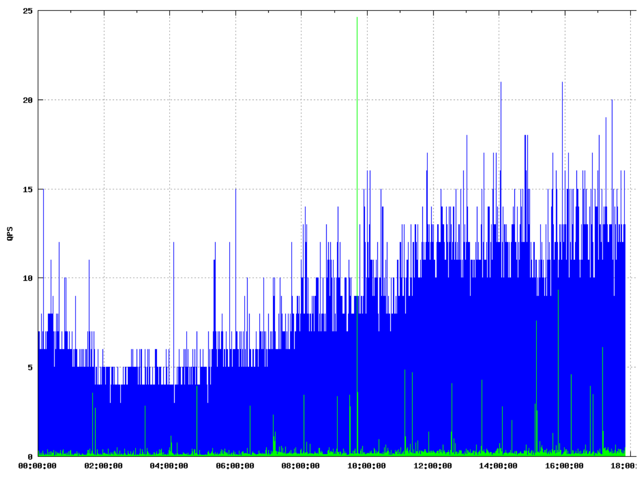


Fig. 1. Number of queries per second (in green the query processing time).

Peaks typically occur during promotions of the shop 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). For example, based on the negotiated SLA with a customer, services must maintain 100 queries per seconds with less than 200 milliseconds of response time over 99.5% of the service uptime, and 99.9% with less than 500 milliseconds.

Previous work reported in [10] aimed to compute an optimal initial deployment configuration (using the size of the product catalogue, number of expected visitors and cost of the required virtual machines). The computation was based on an already available model of the Fredhopper Cloud Services written in the ABS language. In this paper we address the problem of maintaining a high quality of service after this initial set-up by taking dynamic factors into account, such as fluctuating user-demand and unexpectedly failing virtual machines.

The solution that we propose is based on a tool named SmartDepl that, when integrated in the ABS model of the Fredhopper Cloud Services, enables the modeling of automatic upscaling or downscaling. When the decision to scale up or down is made, SmartDepl indicates how to automatically evolve the deployment configuration. This is not a trivial task: the desired deployment configuration should satisfy various requirements, and those can trigger the need to instantiate multiple service instances that furthermore require proper configuring to ensure they function correctly.

The requirements can originate from both business decisions or technical reasons. For instance, for security reasons, services that operate on sensitive customer data should not be deployed on machines shared by multiple customers. Below we list some of these requirements.

– To increase fault-tolerance, we aim to spread virtual machines across geographical locations. Amazon allows specifying the desired region (a geographical area) and availability zone (a geographical location in a region) for a virtual machine. Fault tolerance is then increased by balancing the number of machines between different availability zones. Thus, when scaling, the number of machines should be adjusted in all zones simultaneously. Effectively this means that with two zones, we scale up or down with an even number of machines.
– Each instance of a Query service is in one of two modes: 'live' mode to serve queries, or 'staging' mode to serve as an indexer (i.e., to publish updates to the product catalogue). There always should be at least one instance of Query service in staging mode.
– The network throughput and latency between the PlatformService and indexer is important. Since the infrastructure provider gives better performance for traffic between instances in the same zone, we require the indexer and PlatformService to be in the same zone.
– Installing an instance of the QueryService requires the presence of an instance of the DeploymentService on the same virtual machine.
– For performance reasons and fault tolerance, load balancers require a dedicated machine without other services co-located on the same virtual machine.

3 Annotated ABS

The ABS language is designed to develop executable models. It targets distributed and concurrent systems by means of concurrent object groups and asynchronous method calls. Here, we will recap just the specific linguistic features of ABS to support the modeling of the deployment; for more details we refer the interested reader to the ABS project website [2] and [10] for the cost annotations.

The basic element to capture the deployment in ABS is the *Deployment Component* (DC), which is a container for objects/services that, intuitively, may model a virtual machine running those objects/services. ABS comes with a rich API that allows the programmer to model a cloud provider of deployment components.

```
1 CloudProvider cProv = new CloudProvider("Amazon");
2 cProv.addInstanceDescription(Pair("c3",
3   InsertAssoc(Pair(CostPerInterval,210),
4     InsertAssoc(Pair(Memory,7500),
5       InsertAssoc(Pair(Cores,4), EmptyMap)))));
6 DeploymentComponent dc = cProv.prelaunchInstanceNamed("c3");
7 [DC: dc] Service s = new QueryServiceImpl();
```

In the ABS code above, the cloud provider "Amazon" is modeled as the object `cProv` of type `CloudProvider`. The fact that "Amazon" can provide a virtual machine of type "c3" is modeled by calling `addInstanceDescription` in Line 2. With this instruction we also specify that c3 virtual machines cost 0,210 cents per hour, provide 7.5 GB of RAM and 4 cores. In Line 5 an instance of "c3" is launched and the corresponding deployment component is saved in the variable `dc`. Finally, in Line 6, a new object of type `QueryServiceImpl` (implementing interface `Service`) is created and deployed on the deployment component `dc`.

ABS supports declaring interface hierarchies and defining classes implementing them.

```
interface Service { ... }
interface IQueryService extends Service { ... }
class QueryServiceImpl(DeploymentService ds, Bool staging)
  implements IQueryService { ... }
```

In the excerpt of ABS above, the `IQueryService` service is declared as an interface that extends `Service`, and the class `QueryServiceImpl` is an implementation of this interface. Notice that the initialization parameters required at object instantiation are indicated as parameters in the corresponding class definition.

Classes can be annotated with the cost and requirements of an object of that class.

```
[Deploy: scenario[Name("staging"), Cost("Cores", 2),
 Cost("Memory",7000), Param("staging", Default("True")),
 Param("ds", Req)] ]
[Deploy: scenario[Name("live"), Cost("Cores", 1),
 Cost("Memory",3000), Param("staging", Default("False")),
 Param("ds", Req)] ]
```

The above two annotations, to be included before the declaration of the class `QueryServiceImpl` in the above ABS code, describe two possible deployment scenarios for objects of that class. The first annotation models the deployment of a Query Service in staging mode, the second one models the deployment in live mode. A Query Service in staging mode requires 2 cores and 7GB of RAM. In live mode, 1 core and 3GB of RAM suffices. Creating a Query Service object requires the instantiation of its two initialization parameters `ds` and `staging`. The second parameter should be instantiated with `True` or `False` depending on the deployment scenario. The first parameter is required (keyword `Req` in the annotation): this means that the Query Service requires a reference to an object of type `DeploymentService` passed via the `ds` initialization parameter.

4 The Declarative Requirement Language DRL

Computing a deployment configuration requires taking into account the expectations of the ABS programmer. For example, in the Fredhopper Cloud Services, one initial goal is to deploy with reasonable cost a given number of Query Services and a Platform Service, possibly located on different machines to improve fault tolerance, and later on to upscale (or subsequently downscale) the system according to the monitored traffic. Each desiderata can be expressed with a corresponding expression in *Declarative Requirement Language* (DRL): a new language for stating constraints a configuration to be computed should satisfy.

```
1  b_expr : b_term (bool_binary_op b_term )* ;
2  b_term : ('not')? b_factor ;
3  b_factor : 'true' | 'false' | relation ;
4  relation : expr (comparison_op expr)? ;
5  expr : term (arith_binary_op term)* ;
6  term : INT                                              |
7    ('exists' | 'forall') VARIABLE 'in' type ':' b_expr |
8    'sum' VARIABLE 'in' type ':' expr                    |
9    (( ID | VARIABLE | ID '[' INT ']' ) '.')? objId      |
10   arith_unary_op expr                                  |
11   '(' b_expr ')'                                       ;
12 objId :  ID | VARIABLE | ID '[' ID ']' | ID '[' RE ']';
13 type : 'obj' | 'DC' | RE ;
14 bool_binary_op : 'and' | 'or' | 'impl' | 'iff' ;
15 arith_binary_op : '+' | '-' | '*' ;
16 arith_unary_op : 'abs' ; // absolute value
17 comparison_op : '<=' | '=' | '>=' | '<' | '>' | '!=' ;
```

Table 1. DRL grammar.

As shown in Table 1, that reports an excerpt of the DRL grammar,[5] a desiderata is a (possibly quantified) Boolean formula `b_expr` obtained by using the usual logical connectives over comparisons between arithmetic expressions. An atomic arithmetic expression is an integer (Line 6), a sum statement (Line 8) or an identifier for the number of deployed objects (Line 9). The number of objects to deploy using a given scenario is defined by its class name and the scenario name enclosed in square brackets (Line 12). For example, the below formula requires deploying at least one object of class `QueryServiceImpl` in staging mode.

```
QueryServiceImpl[staging] > 0
```

The square brackets are optional (Line 12 - first option) for objects with only one default deployment scenario. Regular expressions (`RE` in Line 12) can match objects deployed using different scenarios. The number of deployed objects can be prefixed by a deployment component identifier to denote just the number of objects defined within that specific deployment component. As an example, the deployment of only one object of class `DeploymentServiceImpl` on the first and second instance of a "c3" virtual machine can be enforced as follows.

```
c3[0].DeploymentServiceImpl = 1 and
  c3[1].DeploymentServiceImpl = 1
```

[5] The complete grammar defined using the ANTLR compiler generator is available at https://github.com/jacopoMauro/abs_deployer/blob/smart_deployer/decl_spec_lang/DeclSpecLanguage.g4.

Here the 0 and 1 numbers between the square brackets represent respectively the first and second virtual machine of type "c3". To shorten the notation, the `[0]` can be omitted (Line 9).[6]

It is possible to use also quantifiers and sum expressions to capture more concisely some of the desired properties. Variables are identifiers prefixed with a question mark. As specified in Line 13, variables in quantifiers and sums can range over all the objects (`'obj'`), all the deployment components (`'DC'`), or just all the virtual machines matching a given regular expression (`RE`). In this way it is possible to express more elaborate constraints such as the co-location or distribution of objects, or limit the amount of objects deployed on a given DC.[7] As an example, the constraint enforcing that every Query Service has a Deployment Service installed on its virtual machine is as follows.

```
forall ?x in DC: (
  ?x.QueryServiceImpl['.*'] > 0  impl
  ?x.DeploymentServiceImpl > 0 )
```

Here `impl` stands for logical implication. The regular expression `'.*'` allows us to match with both deployment modalities for the Query Service (`staging` and `live`). Finally, specifying that the load balancer must be installed on a dedicated virtual machine (without other Service instances) can be done as follows.

```
forall ?x in DC: (
  ?x.LoadBalancerServiceImpl > 0  impl
  (sum ?y in obj: ?x.?y) = ?x.LoadBalancerServiceImpl )
```

5 Deployment Engine

SmartDepl is the tool that we have implemented to realize automatic deployment. The key idea of SmartDepl is to allow the user on the one hand to declaratively specify the desired deployments and, on the other hand, to develop its program abstracting from concrete deployment decisions. More concretely, deployment requirements are specified as program annotations. SmartDepl processes each of these annotations and generates for each of them a new class that specifies the deployment steps to reach the desired target. Then this class can be used to trigger the execution of the deployment, and to undo it in case the system needs to downscale.

As an example, imagine that an initial deployment of the Fredhopper Cloud Services has been already obtained and that, based on a monitor decision, the

[6] We assume that every deployment desiderata expressed in DRL deals with only a bounded number of deployment components (the bound is a configuration parameter for SmartDepl). Notice that this does not mean that the total number of deployment components in an application is bound, as the deployment can be repeated an unbounded number of times.

[7] DRL improves on the specification language presented in [10] because the addition of the quantifiers and sums allow to write the desiderata more concise and naturally.

```
 1 { "id": "AddQueryDeployer",
 2   "specification": "QueryServiceImpl[live] = 1",
 3   "obj": [ { "name": "platformObj",
 4              "provides": [ {
 5                "ports": [ "MonitorPlatformService",
 6                           "PlatformService" ],
 7                "num": -1 } ],
 8              "interface": "PlatformService" },
 9            { "name": "loadBalancerObj",
10              "provides": [ {
11                "ports": [ "LoadBalancerService" ],
12                "num": -1 } ],
13              "interface": "LoadBalancerService" },
14            { "name": "serviceProviderObj",
15              "provides": [ {
16                "ports": [ "ServiceProvider" ],
17                "num": -1 } ],
18              "interface": "ServiceProvider" } ],
19   "DC": [] }
```

Table 2. An example of a deployment annotation.

user wants to add a Query Service instance in live mode. The annotation that describes this requirement is the JSON object defined in Table 2.[8]

In Line 1, the keyword `"id"` specifies that the name of the class with the deployment code, to be synthesized by SmartDepl, is `AddQueryDeployer`. As we will see later, this class exposes methods to be invoked to actually execute deployment actions that modifies the current deployment according to the requirements in the deployment annotation. The second line contains the declarative specification of the desired configuration in DRL. Deploying a new instance of the Query Service may involve other relevant objects from the surrounding environment, such as the `PlatformService` or a `LoadBalancerService`. Which objects are relevant may come from business, security or performance reasons, thus in general it may be undesirable to select or create automatically a Service instance of the right type. SmartDepl is flexible in this regard: the user supplies the appropriate ones. By using the keyword `"obj"`, Lines 3-18 list the appropriate objects. Since these object are already available, they need not be deployed again. The names of these objects are specified with the keyword `"name"` (Lines 3,9,14), the provided interfaces with the keyword `"port"` (Lines 5-6,11,16) with the amount of services that can use it (keyword `"num"` in Lines 7,12,17 — in this case a -1 value

[8] To facilitate the interoperability between ABS and SmartDepl we have adopted a JSON syntax for the deployment annotations. For the interested reader the formal specification of the JSON annotations is defined in `https://github.com/jacopoMauro/abs_deployer/blob/smart_deployer/spec/smart_deploy_annotation_schema.json`.

means that the object can be used by an unbounded number of other objects), and the object interface with keyword `"interface"` (Lines 8,13,18). Finally, with the keyword `"DC"`, the user specifies if there are existing deployment components with free resources that can be used to deploy new objects. In this case, for fault tolerance reasons the user wants to deploy the Query Service in a new machine and therefore the `"DC"` is empty (Line 19).

Once the annotation is given, the user may freely use this class. For instance, the below ABS code scales the system up or down based on a monitor decision.

```
1  while ( ... ) {
2   if ( monitor.scaleUp() ) {
3     SmartDeployInterface depObj = new AddQueryDeployer(
4       cProv, platformService, loadBalancerService, serviceProvider);
5     depObj.deploy();
6     depObjList = Cons(depObj,depObjList);
7    } else if ( (monitor.scaleDown()) && (depObjList != Nil) ) {
8      SmartDeployInterface depObj = head(depObjList);
9      depObjList = tail(depObjList);
10     depObj.undeploy(); } }
```

Every time an upscale is needed, an object of class AddQueryDeployer (the name associated with the annotation previously discussed) is created. The idea is to store the references to these deployment objects in a list called depObjList. We now discuss the initialization parameters for such objects. The first parameter is the cloud provider, as defined for instance in Section 3. The next parameters are the objects already available for the deployment that do not need to be re-deployed. These are given according to the order they are defined in the annotation in Table 2. The generated class implements the SmartDeployInterface with: i) a deploy method to realise the deployment of the desired configuration, ii) an undeploy method to undo the deployment gracefully by removing the virtual machine created with the deploy method, iii) getter methods to retrieve the list of new objects and deployment components created by running the deploy method (e.g., a call depObj.getIQueryService() retrieves the list of all the Query Services created by depObj.deploy()). The actual addition of the Query Service is performed in Line 5 with the call of the deploy method. If the monitor decides to downscale (Line 7), the last deployment solution is retrieved (Line 8), and the corresponding deployment actions are reverted by calling the undeploy method.[9]

Technically, SmartDepl is written in Python (∼1k lines of code) and relies on Zephyrus2, a configuration optimizer that given the user desiderata and a universe of components, computes the optimal configuration satisfying the user needs.[10] The cost annotations (see Section 3) are used to compute a configuration

[9] Since ABS does not have an explicit operation to force the removal of objects the undeploy procedure just removes the references to these objects leaving the garbage collector to actually remove them. The deployment components created by the deploy methods are removed instead using an explicit kill primitive provided by ABS.

[10] SmartDepl uses Zephyrus2 (freely available at https://jacopomauro@bitbucket.org/jacopomauro/zephyrus2.git) since it allows the use of a new expressive lan-

that satisfies the constraints, minimizes the cost of the deployment components that need to be created and, in case of ties, minimizes the number of created objects. The user is notified if no configuration exists that satisfies the desiderata. Once a configuration is obtained, SmartDepl uses topological sorting to take into account all the object dependencies and computes the sequence of deployment instructions to realise the desirable configuration. SmartDepl exploits Delta Modeling [7] to generate the code of the classes and methods to inject into the interface. SmartDepl also notifies the user when it is unable to generate a sequence of deployment actions due to mutual dependencies between the objects.[11]

As an example the deploy code generated by SmartDepl for the annotation defined in Table 2 is the following.

```
1  Unit deploy() {
2    DeploymentComponent c3_0 = cloudProvider.prelaunchInstanceNamed("c3");
3    ls_DeploymentComponent = Cons(c3_0,ls_DeploymentComponent);
4    [DC: c3_0] DeploymentService oDef___DeploymentServiceImpl_0_c3_0 =
5      new DeploymentServiceImpl(platformObj);
6    ls_DeploymentService = Cons(oDef___DeploymentServiceImpl_0_c3_0,
7      ls_DeploymentService);
8    [DC: c3_0] IQueryService olive___QueryServiceImpl_0_c3_0 = new
9      QueryServiceImpl(oDef___DeploymentServiceImpl_0_c3_0, False);
10   ls_IQueryService = Cons(olive___QueryServiceImpl_0_c3_0, ls_IQueryService);
11   ls_Service = Cons(olive___QueryServiceImpl_0_c3_0, ls_Service);
12   ls_EndPoint = Cons(olive___QueryServiceImpl_0_c3_0, ls_EndPoint);
13 }
```

At Line 3, a new deployment component c3_0 is created. In Lines 4-5 an object of class DeploymentService is created, since every Query Service requires a corresponding Deployment Service (it is one of the required parameters, cf. Section 3) to be deployed before the Query Service. In Lines 8-9 the desired object of class IQueryService is created. Both objects are deployed on c3_0.

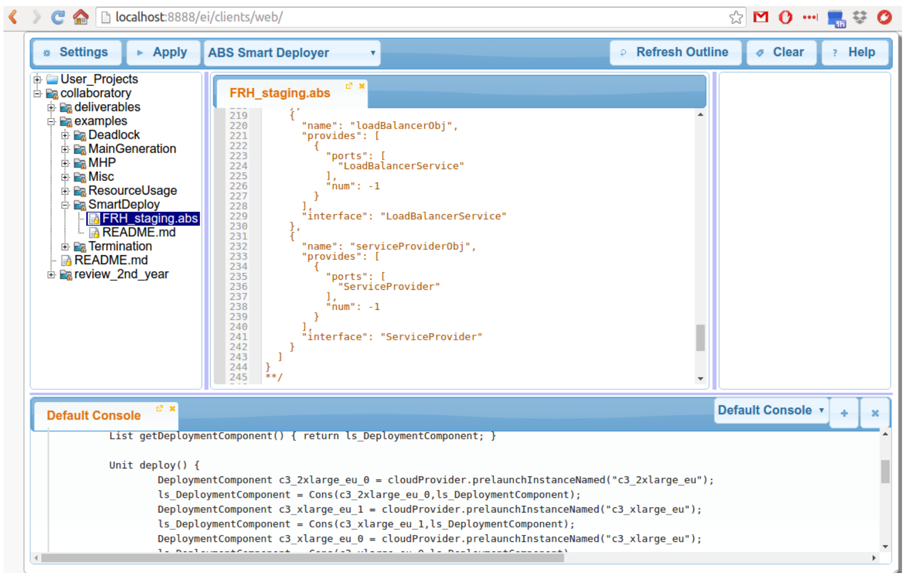
Even though for the sake of the presentation this is just a simple example, it is immediately possible to notice that SmartDepl alleviates the user from the burden of the deployment decisions. Indeed, she can specify the desired configuration without worrying about the dependencies of the various objects and their distributed placement for obtaining the cheapest possible solution.

SmartDepl is open source, available at https://github.com/jacopoMauro/abs_deployer/tree/smart_deployer and to increase its portability it can be installed also by using the Docker container technology [12]. As illustrated in Figure 2, SmartDepl has also been integrated into the ABS toolchain,[12] an IDE for a collection of tools for writing, inspecting, checking, and analyzing ABS programs developed within the Envisage European project.

guage and because it relies on MiniSearch [24], a new efficient and flexible framework for planning the search strategies. Zephyrus2 is a completely new re-engineering of the previous Zephyrus solver [8, 9].

[11] This occurs when the creation of an object requires the execution of a complex protocol, such as what happens for the boostrapping of Linux distributions [1].

[12] http://abs-models.org/installation/



```
    {
      "name": "loadBalancerObj",
      "provides": [
        {
          "ports": [
            "LoadBalancerService"
          ],
          "num": -1
        }
      ],
      "interface": "LoadBalancerService"
    },
    {
      "name": "serviceProviderObj",
      "provides": [
        {
          "ports": [
            "ServiceProvider"
          ],
          "num": -1
        }
      ],
      "interface": "ServiceProvider"
    }
  ]
}
**/
```

```
        List getDeploymentComponent() { return ls_DeploymentComponent; }

        Unit deploy() {
                DeploymentComponent c3_2xlarge_eu_0 = cloudProvider.prelaunchInstanceNamed("c3_2xlarge_eu");
                ls_DeploymentComponent = Cons(c3_2xlarge_eu_0,ls_DeploymentComponent);
                DeploymentComponent c3_xlarge_eu_1 = cloudProvider.prelaunchInstanceNamed("c3_xlarge_eu");
                ls_DeploymentComponent = Cons(c3_xlarge_eu_1,ls_DeploymentComponent);
                DeploymentComponent c3_xlarge_eu_0 = cloudProvider.prelaunchInstanceNamed("c3_xlarge_eu");
```

Fig. 2. SmartDepl execution within the ABS toolchain IDE.

6 Application to the Fredhopper use case

In this section we report on the modeling with SmartDepl of the concrete deployment requirements of the Fredhopper Cloud Services, previously introduced in Section 2. We decided to apply our techniques to the Fredhopper Cloud Services use case because it was already modeled in ABS, and thanks to extensive profiling of the in-production system, the cost of its services are known.

SmartDepl was used twice: to synthesize the initial static deployment of the entire framework and to add (and later remove) instances of the Query Service if the system needs to scale. Since the Fredhopper Cloud Services uses Amazon EC2 Instance Types, we used two types of deployment components corresponding to the "xlarge" and "2xlarge" instances of the Compute Optimized instances (version 3)[13] of Amazon. For fault tolerance and stability, Fredhopper Cloud Services uses instances in multiple regions in Amazon (regions are geographically separate areas, so even if there is a force majeure in one region, other regions may be unaffected). We model the instance types in different regions as follows: "c3_xlarge_eu", "c3_xlarge_us", "c3_2xlarge_eu", "c3_2xlarge_us" ("eu" refers to a European region, "us" is an American region).

The static deployment of the Fredhopper Cloud Services requires deploying a Load Balancer, a Platform Service, a Service Provider and 2 Query Services with at least one in staging mode. This is expressed as follows.

$$\mathsf{LoadBalancerServiceImpl} = 1 \text{ and } \mathsf{PlatformServiceImpl} = 1 \text{ and}$$

[13] https://aws.amazon.com/ec2/instance-types/

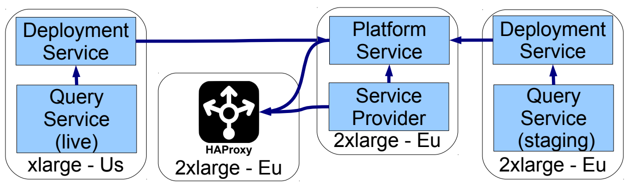


Fig. 3. Example of automatic objects allocation to deployment components.

```
ServiceProviderImpl = 1 and QueryServiceImpl[staging] > 0 and
QueryServiceImpl[staging] + QueryServiceImpl[live] = 2
```

For the correct functioning of the system, a Query Service requires a Deployment Service installed on the same machine. This constraint is expressed as shown in Section 4. The requirement that a Service Provider is present on every machine containing a Platform Service is expressed by:

```
forall ?x in DC: (?x.PlatformServiceImpl > 0 impl ?x.ServiceProviderImpl > 0)
```

Not all services can be freely installed on an arbitrary virtual machine. To increase resilience, we require that the Load Balancer, the Query/Deployment Services, and the Platform Service/Service Provider are never co-located on the same virtual machine. The end of Section 4 shows how this is expressed.

To handle catastrophic failures, the Fredhopper Cloud Services aim to balance the Query Services between the regions (see Section 2). This is enforced by constraining the number of the Query Services in the different data centers to be equal. In DRL this is expressed with regular expressions as follows.

```
(sum ?x in '.*_eu': ?x.QueryServiceImpl['.*']) =
(sum ?x in '.*_us': ?x.QueryServiceImpl['.*'])
```

As described in Section 4, for performance reasons, the Query Service in Staging mode should be located in the zone of the Platform Service, since Amazon connects instances in the same region with low-latency links. For the European data-center this is expressed by:

```
(sum ?x in '.*_eu': ?x.QueryServiceImpl[staging]) > 0) impl
(sum ?x in '.*_eu': ?x.PlatformServiceImpl ) > 0)
```

From this specification SmartDepl computes the initial configuration in Figure 3, which minimizes the total costs per interval. It deploys the Load Balancer, Platform Service and one staging Query Service on three "2xlarge" instances in Europe, and deploys a live Query service on an "xlarge" instance in US.

After this initial deployment, the Cloud engineers of Fredhopper Cloud Services rely on feedback provided by monitors to decide if more Query Services in live mode are needed. Figure 4 and 5 show some of the main metrics for a single customer used to determine the scaling. The timescale in the figures is 1

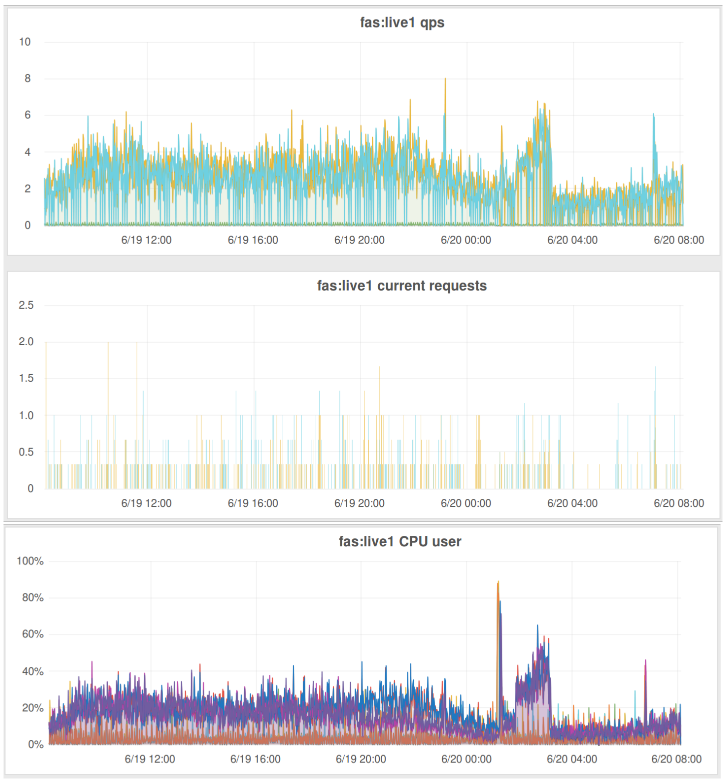


Fig. 4. Metrics graphed over a single day for a customer (a).

day, but this can be adjusted to see trends over longer periods, or zoom in on a short period. The figures show that the number of queries served per second (qps, first graph of Figure 4) is relatively high and the requests (Figure 4, second graph) are fairly low, so requests are not queuing. Furthermore the CPU usage (Figure 4, third graph) and memory consumption with small swap space used (Figure 5, second and third graphs) look healthy. Hence, no scaling is needed.

If we would have needed to scale up, *two* Query Service instances are added: one in an EU region, and one in an US region for balancing across regions. In contrast, if there is unnecessary overcapacity, the most recent ones can be shut down. Since the Cloud operations team currently manually decides to scale, and Fredhopper has very aggressive SLAs, the team is typically conservative with downscaling, leading to potential over-spending. The ability of SmartDepl to deploy in the programming language (ABS) itself allows to leverage the extensive tool-supported analyses available for ABS [3, 11, 15, 25]. For example, by using monitors to track the quality of services, SmartDepl allows to reason on a rigorous basis on the scaling decisions and their impact on the SLA agreed with the customers.

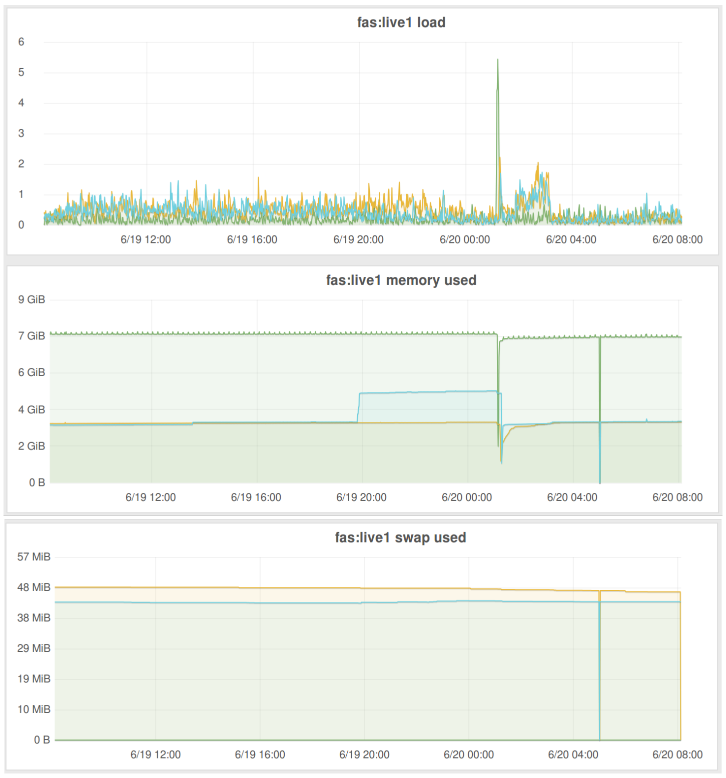


Fig. 5. Metrics graphed over a single day for a customer (b).

Furthermore, while the operations team currently use ad-hoc scripts to configure newly added or removed service instances, and these scripts are specific to the infrastructure provider, SmartDepl automatically generates code that accomplishes this (for example, see Table 2). SmartDepl is flexible in the sense that it is infrastructure independent, allowing to seamlessly switch between different infrastructure providers: virtual machines are launched and terminated through a generic Cloud API offered by ABS for managing virtual resources. Executable code is automatically generated from ABS for any of the infrastructures for which an implementation of the Cloud API exists (e.g., Amazon, Docker, OpenStack).

To automatically generate the scaling deployment configuration, SmartDepl uses all the previous specifications, except that now instead of requiring a Platform Service and a Load Balancer we simply require two Query services in live mode. In this case, as expected after the deployment of the initial framework, the best solution is to deploy one Query Service in Europe and one in US using "xlarge" instances. The ABS model used with all the annotations and specifications and an example of generated code is available at `https://github.com/jacopoMauro/abs_deployer/blob/smart_deployer/test/`.

7 Related Work

Many management tools for bottom-up deployment exist, e.g., CFEngine [6], Puppet [19], MCollective [23], and Chef [22]. Such tools allow for the declaration of components, by indicating how they should be installed on a given machine, together with their configuration files, but they are not able to automatically decide where components should be deployed and how to interconnect them for an optimal resource allocation. The alternative holistic approach allows modeling the entire application and derives the deployment plan top-down. In this context, one prominent work is represented by the TOSCA (Topology and Orchestration Specification for Cloud Applications) standard [21]. Following a similar philosophy, we can mention Terraform [17], JCloudScale [26], Apache Brooklyn [4], and tools supporting the Cloud Application Management for Platforms protocol [20]. A first attempt to combine the holistic and bottom-up approaches is reported in [5]: a global deployment plan expressed in TOSCA is checked for correctness against local specifications of the deployment lifecycle of the single components.

Similarly to our approach, ConfSolve [18] and Engage [14] use a solver to plan deployment starting from the local requirements of components, but these approaches were not incorporated in fully-fledged specification languages (including also behavioral descriptions as in our case with ABS).

8 Conclusions

We presented an extension of the ABS specification language that supports modeling deployment in a declarative manner: the programmer specifies deployment constraints, and a solver synthesizes ABS classes with methods that execute deployment actions to reach an optimal deployment configuration that satisfies the constraints. Our approach, which is inspired by [9] and significantly improves our initial work [10], can be easily applied to any other object-oriented language that offers primitives for the acquisition and release of computing resources.

As a future work we plan to investigate the possibility to invoke at run time the external deployment engine. In this way, it could be possible to dynamic re-define the deployment constraints by means of a dynamic tuning of the engine. Nevertheless, dynamically computing the deployment steps may require additional elements such as the support of new reflection primitives to get a snapshot of the running application, and possibly the use of sub-optimal solutions when computing the optimal configuration takes too much time.

References

1. P. Abate and S. Johannes. Bootstrapping Software Distributions. In *CBSE'13*, 2013.
2. Abstract behavioral specification language. `http://www.abs-models.com/`.
3. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *ETAPS*, 2014.

4. Apache Software Foundation. Apache Brooklyn. https://brooklyn.incubator.apache.org/.
5. A. Brogi, A. Canciani, and J. Soldani. Modelling and Analysing Cloud Application Management. In *ESOCC*, 2015.
6. M. Burgess. A Site Configuration Engine. *Computing Systems*, (2), 1995.
7. D. Clarke, R. Muschevici, J. Proença, I. Schaefer, and R. Schlatte. Variability Modelling in the ABS Language. In *FMCO*, 2010.
8. R. D. Cosmo, M. Lienhardt, J. Mauro, S. Zacchiroli, G. Zavattaro, and J. Zwolakowski. Automatic Application Deployment in the Cloud: from Practice to Theory and Back. In *CONCUR*, 2015.
9. R. D. Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, J. Zwolakowski, A. Eiche, and A. Agahi. Automated synthesis and deployment of cloud applications. In *ASE*, 2014.
10. S. de Gouw, M. Lienhardt, J. Mauro, B. Nobakht, and G. Zavattaro. On the Integration of Automatic Deployment into the ABS Modeling Language. In *ESOCC*, 2015.
11. C. C. Din, R. Bubel, and R. Hähnle. Key-abs: A deductive verification tool for the concurrent modelling language ABS. In *CADE*, 2015.
12. Docker Inc. Docker. https://www.docker.com/.
13. N. Ferry, F. Chauvel, A. Rossini, B. Morin, and A. Solberg. Managing multi-cloud systems with CloudMF. In *NordiCloud*, 2013.
14. J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: a deployment management system. In *PLDI*, 2012.
15. E. Giachino, C. Laneve, and M. Lienhardt. A framework for deadlock detection in core ABS. *CoRR*, 2015.
16. G. E. Gonçalves, P. T. Endo, M. A. Santos, D. Sadok, J. Kelner, B. Melander, and J. Mångs. CloudML: An Integrated Language for Resource, Service and Request Description for D-Clouds. In *CloudCom*, 2011.
17. HashiCorp. Terraform. https://terraform.io/.
18. J. A. Hewson, P. Anderson, and A. D. Gordon. A Declarative Approach to Automated Configuration. In *LISA*, 2012.
19. L. Kanies. Puppet: Next-generation configuration management. *;login: the USENIX magazine*, (1), 2006.
20. OASIS. Cloud Application Management for Platforms. http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.html.
21. OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0. http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html.
22. Opscode. Chef. http://www.opscode.com/chef/.
23. Puppet Labs. Marionette collective. http://docs.puppetlabs.com/mcollective/.
24. A. Rendl, T. Guns, P. J. Stuckey, and G. Tack. MiniSearch: A Solver-Independent Meta-Search Language for MiniZinc. In *CP*, 2015.
25. P. Y. H. Wong, R. Bubel, F. S. de Boer, M. Gómez-Zamalloa, S. de Gouw, R. Hähnle, K. Meinke, and M. A. Sindhu. Testing abstract behavioral specifications. *STTT*, 17(1):107–119, 2015.
26. R. Zabolotnyi, P. Leitner, W. Hummer, and S. Dustdar. JCloudScale: Closing the Gap Between IaaS and PaaS. *ACM Trans. Internet Techn.*, 15(3):10, 2015.