



HAL
open science

Human-in-the-Loop Simulation of Cloud Services

Nikolaos Bezirgiannis, Frank De Boer, Stijn De Gouw

► **To cite this version:**

Nikolaos Bezirgiannis, Frank De Boer, Stijn De Gouw. Human-in-the-Loop Simulation of Cloud Services. 6th European Conference on Service-Oriented and Cloud Computing (ESOCC), Sep 2017, Oslo, Norway. pp.143-158, 10.1007/978-3-319-67262-5_11 . hal-01677612

HAL Id: hal-01677612

<https://inria.hal.science/hal-01677612>

Submitted on 8 Jan 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

Human-in-the-Loop Simulation of Cloud Services ^{*}

Nikolaos Bezirgiannis¹, Frank de Boer², and Stijn de Gouw³

¹ Leiden Institute for Advanced Computer Science, Leiden, the Netherlands
n.bezirgiannis@umail.leidenuniv.nl

² Centrum Wiskunde & Informatica (CWI), Amsterdam, Netherlands
f.s.de.boer@cwi.nl

³ Open University, Netherlands
stijn.degouw@ou.nl

Abstract. In this paper we discuss an integrated tool suite for the simulation of software services which are offered on the Cloud. The tool suite uses the Abstract Behavioral Specification (ABS) language for modeling the software services and their Cloud deployment. For the real-time execution of the ABS models we use a Haskell backend which is based on a source-to-source translation of ABS into Haskell. The tool suite then allows Cloud engineers to interact in real-time with the execution of the model by deploying and managing service instances. The resulting human-in-the-loop simulation of Cloud services can be used both for training purposes and for the (semi-)automated support for the real-time monitoring and management of the actual service instances.

Keywords: Human-in-the-loop simulation, Cloud services, monitors, Service Level Agreement

1 Introduction

The Abstract Behavioral Specification (ABS) language⁴ is an executable modeling language which features powerful abstractions of virtualized resources [7] like CPU time, memory, and bandwidth. As such it is particularly tailored towards modeling and simulation of software services offered on the Cloud [1]. Further, a variety of tools⁵ which include simulation with visualization support, deadlock analysis, cost analysis, deployment synthesis, and test case generation, supports the formal development and analysis of software models and their deployment as executable ABS.

The Erlang backend of ABS provides a symbolic interpretation of the abstractions modeling (CPU) time, that is, time is modeled by a symbolic clock which

^{*} Partly funded by the EU project FP7-610582 ENVISAGE. This work was carried out on the Dutch national e-infrastructure with the support of SURF Foundation.

⁴ <http://docs.abs-models.org>

⁵ <http://abs-models.org/abs-tools>

is advanced by the execution of a certain kind of statements, so-called duration statements. In contrast, in this paper we introduce a new Haskell backend, in the sequel denoted by `ABS_RT`, which is based on a source-to-source translation of ABS into Haskell and which directly relates the ABS abstractions of time to the underlying hardware clock. It should be noted that the term “real-time ABS” has also been used, for example in [8], to refer to the ABS abstractions modeling (CPU) time themselves. In this paper however we use the term “real-time” to refer to the implementation of these abstractions with respect to some external clock, e.g., the hardware clock. This implementation allows for a different kind of simulation, so-called human-in-the-loop simulation, abbreviated in the sequel by HITL. In general this kind of simulations require human interaction and are used for training purposes. A typical example is that of flight simulations where trainees interact in real-time with a model of a plane in flight. Clearly, for such training to be effective the human interactions should be processed by the model in real-time as measured by the hardware clock.

In this paper we introduce the `ABS_RT` Haskell backend of ABS and present its use by Cloud engineers so that they can interact in real-time with the execution of the model of the services offered on the Cloud. This interaction consists of deploying and managing service instances and allows Cloud engineers to acquire knowledge of the real-time consequences of their decisions. We illustrate this use of HITL simulation of Cloud services by an industrial case study based on the Fredhopper Cloud Services.

Main contribution and related work There exists a variety of cloud simulation tools including CloudSim [4], GreenCloud [9], and iCanCloud [10]; although all of these tools offer finer-grained analysis (e.g. network configuration and energy consumption in the Cloud) they rely on discrete-event computer simulation engines, which do not permit live HITL intervention on a running simulation. To the best of our knowledge HITL simulation of Cloud services has not been investigated before. As already stated above, HITL simulation allows Cloud engineers to acquire knowledge of the real-time consequences of their decisions directly in an interactive manner.

Our overall contribution is an integrated tool suite which supports HITL simulations of Cloud services. This suite integrates the SAGA tool [3] for the declarative specification of service metric functions, and SmartDeployer [6] for the formalization of deployment requirements and the automatic generation of provisioning scripts. At the core of this suite is a new Haskell backend `ABS_RT` of the ABS modeling language which supports a real-time interpretation of the timing constructs of ABS. We further illustrate the use of our tool-suite by an industrial case study based on the Fredhopper Cloud Services. The underlying ABS model of the Fredhopper Cloud Services builds on the one presented in [6] which focuses on automated generation of deployment actions. Here we extend that model to support HITL simulation and for the generation of more realistic deployment recommendations.

The general methodology underlying the use of `ABS_RT` in the HITL simulation of Cloud services involves the formalization of Service Level Agreements

(SLA 's) as a property of a service metric function, as described in [5], with a new framework in ABS which captures various monitoring concepts from QoS and SLAs to lower-level metrics, metric policies, and listenable and billable events. The monitoring framework allows the formal development and analysis of monitors as executable ABS.

Outline of the paper In the next section we introduce the ABS_RT backend. In Section 3 we describe the ABS model of the Fredhopper Cloud Services. The use of ABS_RT in the HITL simulation of this model is described in Section 4. The experimental results are described in Section 5. Finally, in Section 6 we draw some conclusions.

2 The ABS language

ABS is an executable resource-aware modeling language which at its core integrates an imperative layer based on concurrent objects and a functional layer based on algebraic data types. Concurrent objects are (strongly) typed by interfaces and communicate via asynchronous method calls. Such calls generate messages which are queued to be processed sequentially (by the object callee) up to method completion or deliberate yield of control (cooperative scheduling).

ABS further provides a high-level model of *deployment components* which encapsulate virtualized resources of a computer system like CPU time, memory, and bandwidth. These components are expressed by concurrent objects themselves and as such are an integral part of an ABS model. Objects dynamically deployed onto these components share their resources. Usually the ABS user does not create deployment component objects directly (by calling `new`), but instead through a higher object abstraction named *CloudProvider*, which serves both as a factory of deployment components as well as a communication endpoint to an infrastructure service (IaaS):

```
CloudProvider cp = new AmazonCloudProvider(params);
DeploymentComponent vm1 = cp.createInstance(map[Pair(
    Cores,4), Pair(Speed,35), Pair(Memory,16)]);
[DC: vm1] new WebServer(8080); // deployed object
```

High-level annotations of the ABS code are used to specify the corresponding cost model. A statement in ABS can be annotated by `[Cost: intExp()] stmt`; which means in practice that *stmt* will be only completed (and its side-effects instantaneously realised) after some time where *intExp* amount of resource *Speed* has been provided and consumed by the currently executing deployment component. This model of deployment as executable ABS allows for a formal analysis of the constraints induced by the shared resources in terms of a formal cost model and its relation to a formalization of Service Level Agreements (SLA 's) as a property of a service metric function.

Whereas the *Cost* annotation induces the passage of time *locally* inside the deployment component, the timed-ABS extension of the language enables time to pass globally (over the whole model), always with respect to an external clock.

The statement `await duration(min,max)` means that the current process will be rescheduled for execution only after *min* and less than *max* time steps from now have passed on the clock; the statement `duration (min,max)` will accordingly block the object and all of its process for that time. If the ABS clock refers to symbolic (abstract) time — used for synchronizing distinct parts of the model — then the models’ execution is essentially a computer simulation; however, a model running on the real (hardware) clock defines a user-interactive simulation.

Finally, since ABS was primarily designed as a modeling language, it lacks the common I/O functionality found in mainstream programming languages. To allow user interaction a new language extension was introduced to the language built around a REST API. The ABS user may annotate any object declaration with `[HTTPName: strExp()] I o = new ...` to make the object and its fields accessible from the outside as an HTTP endpoint. Any such object can have some of its method definitions annotated with `[HTTPCallable]` to allow them to be called from the outside; the arguments passed and the method’s result will be serialized according to a standard JSON format.

The ABS_RT backend The original Haskell backend of ABS was designed with speed in mind, as well as to offer distributed computing on the cloud [2]. The choice of Haskell was made since it provides language features that closely match those of ABS, and also certain runtime facilities that make the ABS straightforward to implement (e.g. first-class continuations).

At runtime, each ABS concurrent object (or ABS concurrent object group) is associated with one Haskell green thread. Each such thread listens to its own queue for new or re-activated processes and executes 1 at a time up to their next release point (`await` or `return`). The GHC runtime (Haskell’s standard compiler) preempts over these green threads, which are automatically load-balanced to system threads to support Symmetric Multi-Processing (multi-core).

During an asynchronous method call, a caller creates a new process by applying the corresponding function to its arguments and ships its body (function closure) to the end of the callee’s queue. This shipment is done for the parallel runtime through shared-memory, or for the distributed-runtime through Cloud Haskell (TCP/IP). To complement cooperative scheduling, awaiting on futures is implemented a-top of extra temporary green threads and utilizing an asynchronous I/O event library (e.g. `epoll` on Linux); `await` on boolean conditions are optimized to avoid unnecessary busy-wait polling through a more notification-like protocol.

Algebraic-datatypes, parametric polymorphism, interfaces, pure functions are all one-to-one mapped down to Haskell. Haskell’s type system lacks subtyping polymorphism, and as such we implement this in the ABS_RT compiler itself through means of implicit coercive subtyping. The REST API extension of ABS utilizes WARP: a high-performance, high-throughput server library written in Haskell.

Compared to some other backends (Erlang,Java), the Haskell backend does not treat active ABS processes as individual system threads, but instead as data (closures) that are stored in the queue of the concurrent object, which leads to a

```

Unit executeCost(Int cost) {
    Int remaining = cost;
    while (remaining > this.instrPS) {
        duration(1,1);
        suspend;
        remaining = remaining - this.instrPS;
    }
    Rat last = remaining / this.instrPS;
    duration(last,last);
}

```

Fig. 1. The implementation of Cost annotation for the ABS_RT backend

smaller memory footprint. This “data-oriented” implementation preserves local message ordering of method activations, although the ABS language specification cares to leave this unspecified.

We augment the original Haskell backend with support for the timed-ABS language extension, and name the resulting backend ABS_RT. The clock that ABS_RT uses is the available real-time hardware clock underneath. This means that compared to the backends with a symbolic clock (Erlang, Maude), the passage of time is not influenced by timed-ABS calls but instead by the real clock itself. The `duration` statement is implemented as a *sleep* call on the concurrent object’s thread, whereas the `await duration` creates a new extra lightweight thread which will re-schedule its continuation back to the original object thread after the specified time. The `[Cost: x]` annotations are translated to a `executeCost()` method call on the deployment component object as seen in Fig. 1. The `instrPS` field refers to the number of instructions the particular deployment component is able to execute per second. The unit of time (default is seconds) is tunable as a runtime option.

It is worth noting that the GHC runtime scheduler dictates that any “sleeping” thread will be re-activated (preempted) no sooner than the specified time, but may be later than prescribed (not precise). This does affect the reproducibility, among the fact that there is no notion of simultaneous method calls (no specific ordering, thus non-deterministic hardware-dependent process-enqueuing of simultaneous callers) as it can be done with total ordering of symbolic time. Finally, we would like to mention that this real-time implementation as shown in Fig. 1 is generic for any ABS backend that uses the hardware clock and implements *duration/await duration* as a *sleep()* system call. Indeed, it would be straightforward to port it to the Erlang and Java backends as well.

3 FRH case study

Fredhopper⁶ provides the Fredhopper Cloud Services to offer search and targeting facilities on a large product database to e-Commerce companies as services

⁶ <https://www.fredhopper.com/>

(SaaS) over the cloud computing infrastructure (IaaS). 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 to 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*. The advantages of offering software as a service on the cloud over on-premise deployment include the following: to increase fault tolerance; to handle dynamic throughputs; to provide seamless service update; to increase service testability; and to improve the management of infrastructure. To fully utilize the cloud computing paradigm, software must be designed to be *horizontally* scalable⁷. Typically, software services are deployed as *service instances*. Each instance offers the same service and is exposed via the Load Balancing Service, which in turn offers a service endpoint (Fig. 2). Requests through the endpoint are then distributed over the 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 located, plays an important role. Typical lows in demand are observed daily between 2 am and 5 am. In the event of varying throughput, a different number of instances may be deployed and be exposed through the same endpoint. Moreover, at any time, if an instance stops accepting requests, a new instance may be deployed in place.

3.1 Architecture of the Fredhopper Cloud Services

Each service instance offers the same service and is exposed via Load Balancer endpoints that distribute requests over the service instances. Fig. 2 shows a block diagram of the Fredhopper Cloud Services.

Load Balancing Service The Load Balancing Service is responsible for distributing requests from service endpoints to their corresponding instances. Currently at Fredhopper, this service is implemented by HAProxy (www.haproxy.org), a TCP/HTTP load balancer.

Platform Service The Platform Service provides an interface to the Cloud Engineers to manage customer information, deploy and manage service instances associated to the customers, and associate service instance to endpoints (load balancers). The Platform Service takes a service specification, which includes a *resource configuration* for the service, and creates and deploys the specified service. A service specification from a customer determines which type of service is being offered, the number of service instances to be deployed initially for that customer, and the kinds of *virtualized resources* on which the service instances should be deployed.

⁷ en.wikipedia.org/wiki/Scalability#Horizontal_and_vertical_scaling

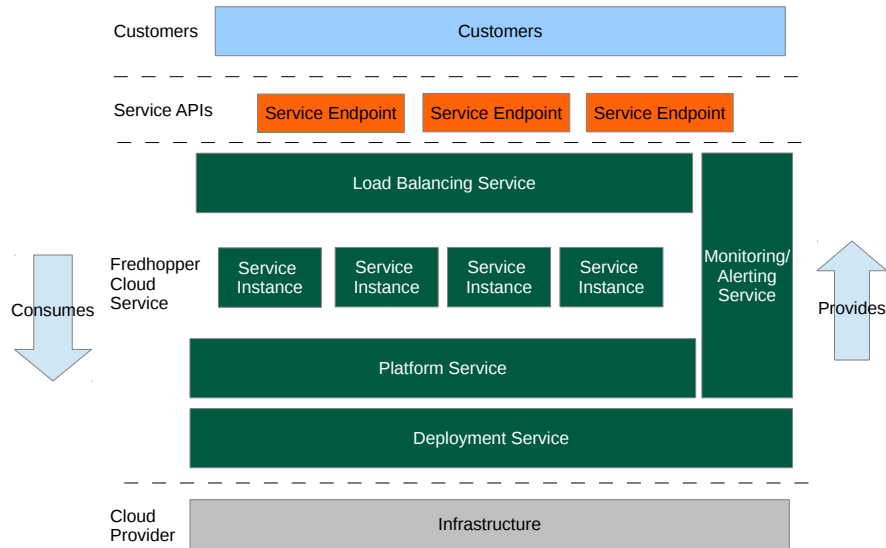


Fig. 2. The architecture of the Fredhopper Cloud Services

Deployment Service The Deployment Service provides an API to the Platform Service to deploy service instances (using a dedicated Deployment Agent) onto specified virtualized resources provided by the *Infrastructure Service*. The API also offers operations to control the life-cycle of the deployed service instances. The Deployment Service allows the Fredhopper Cloud Services to be independent of the specific infrastructure that underlies the service instances.

Infrastructure Service The Infrastructure Service offers an API to the Deployment Service to acquire and release virtualized resources. At the time of writing the Fredhopper Cloud Services utilizes virtualized resources from the Amazon Web Services (aws.amazon.com), where processing and memory resources are exposed through Elastic Compute Cloud instances (<https://aws.amazon.com/ec2/instance-types/>).

Monitoring and Alerting Service The Monitoring and Alerting Service provides 24/7 monitoring services on the functional and non-functional properties of the services offered by the Fredhopper Cloud Services, the service instances deployed by the Platform Service, and the healthiness of the acquired virtualized resources.

If a monitored property is violated, an alert is raised to the Cloud Engineers via emails and SMS messages, and Cloud Engineers can react accordingly. For example, if the query throughput of a service instance is below a certain threshold, they increase the amount of resources allocated to that service. For broken functional properties, such as a run-time error during service up-time, Cloud Engineers notify Software Engineers for further analysis. Fig. 6a shows a visualization of monitors in Grafana, the visualization framework used by ABS.

3.2 Human in the Loop

A dedicated team of Cloud Engineers is in charge of the day to day operation of the Fredhopper Cloud Services. Cloud Engineers keep track of alerts raised by the monitors and the value of monitored metrics over time. Based on their interpretation of this information, using their domain knowledge, Cloud Engineers decide if, when and how to scale up, down or restart services instances and Virtual Machines. Manual scaling rather than auto-scaling is used, as any bug or imprecision in an auto-scaling approach may have disastrous consequences:

1. Automatically scaling up too much jeopardizes the continuity of the business: the infrastructure provider charges running Virtual Machines.
2. Automatically scaling down too much may break the Service Level Agreement(s) (SLAs) between Fredhopper and customers. In the most extreme case, the web shop of a customer may become unavailable, resulting in financial and reputation damage.

The Cloud Engineers must take into account many factors when deciding if, when and how to scale. Most importantly:

- The target QoS values for service metrics specified in the SLA between Fredhopper and the customer.
- Logical and resource requirements on the deployment⁸.
- General business KPIs.

Finding scaling actions resulting in a deployment satisfying all above desiderata, and applying them at the right time is a challenging task due to several reasons.

SLAs traditionally are informal natural language documents, not represented at the software level. Thus, metrics tracked by the monitoring system (i.e., memory consumption), are not directly related to SLAs between Fredhopper and its customers. The Cloud Engineer must manually infer a relation between a combination of the metrics from the monitoring system (typically lower-level), and the metrics in the SLA (typically higher-level, aggregated at the customer level).

Synthesizing a deployment satisfying all logical and resource requirements is a computationally complex task for Cloud Engineers. Even taking only the resource requirements into consideration, it is an instance of the NP-hard multi-dimensional multi-knapsack problem, where the items are service instances (whose weights are the resource requirements for the service, like the amount of memory needed, minimal speed of CPU, etc), and the knapsacks are virtual machines. Logical requirements must also be taken into account. For example, which service instances should be co-located on the same VM, and which to deploy on a dedicated VM? For example, the Query service requires the presence of the Deployment service to function properly. Another logical requirement is to scale with multiple VMs simultaneously in different available zones (locations) in each region. This is mandated by most infrastructure providers to be eligible for compensation for faulty VMs.

In the next section we describe how HITL simulation of ABS models can be used to improve the above practice of Cloud engineers.

⁸ A deployment associates service instances to Virtual Machines

4 Human-in-the-loop Framework

Our tool suite for HITL simulations of Cloud services integrates several different tools.

- The SAGA tool [3] was tweaked for monitoring SLA metrics and the Grafana framework visualizes the metrics
- The SmartDeployer [6] for synthesizing deployment actions
- A logreplay tool for replaying real-world log files
- The new Haskell ABS_RT backend for real-time simulations (Section 2).

We discuss below how each of these tools was exploited to contribute to the support for realistic HITL simulations.

We defined a new layered declarative generic framework in ABS which captures various monitoring concepts from QoS and SLAs to lower-level metrics, metric policies, and listenable and billable events. This framework exploits the SAGA tool for the declarative specification of service metric functions which are used to formalize SLA's. A service metric function is defined by a mapping of (time-stamped) 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. Each monitor captures a single metric, and based on the value of that metric, suggest scaling actions to improve that metric. The `MonitoringService` periodically polls the registered monitors at a user-configured interval to retrieve its suggested scaling actions. An `await duration(1,1)` statement is used to advance the clock and determine which monitors to poll at the current time.

Our tool suite further integrates SmartDeployer [6] for the formalization of deployment requirements, and the automatical derivation of an executable (in ABS) provisioning script that synthesizes a deployment satisfying all specified requirements. By further integrating SmartDeployer actions into the executable, SLA-level monitors generated by SAGA, we have a formalized model that automatically suggests appropriate scaling actions at the right time: when the values of the SLA metrics give rise to it.

The simulation itself consists of replaying a log file generated by the actual system on the ABS model of the system. The logreplay tool is responsible for firing at appropriate times a REST API call (as explain in Sec. 2) to the running simulation for each request recorded in the log file. These requests will trigger ABS code that contains *Cost* annotations (Fig. 3), which has the effect of the real-time simulation as defined for the ABS_RT backend.

This model includes automatically generated monitors in ABS which integrate the declarative specification of service metric functions of SAGA and the provisioning scripts of SmartDeployer. In the simulation Cloud engineers then can interactively select the scaling actions recommended by the different monitors and thus acquire realtime knowledge of their consequences. In general, these selections requires specific domain knowledge which includes knowledge of past behavior. For simplicity, Cloud Engineers can interact with a running HITL simulation via an HTML/Javascript graphical user interface; a live screenshot

```

Bool invoke(Int request){
    print("Executing request in service:"+serviceId);
    [Cost : cost(request)] reqCount = ( reqCount + 1 );
    return True;
}

```

Fig. 3. ABS method that process each incoming request from the log-file is shown in Fig. 5. This interface makes also use of the REST API (Fig. 4) extension as implemented in the ABS_RT backend, for fetching the metric history and recommendations.

```

{ // ... main block header omitted
[HTTPName:"monitoringService"] IMonitoringService ms
    =new MonitoringService();
[HTTPName:"monitor1"] IDegradationMonitor dm
    =new DegradationMonitor(deployer1);
ms!addMonitor(Rule(5000,dm)); // registers a new monitor
[HTTPName:"queryService"] IMonitoringQueryEndpoint ep
    =new MonitoringQueryEndpoint(loadBalancerEndPoints,dm);
println("Endpoints set up. Waiting for requests...");
}

```

Fig. 4. The main ABS block exposing the FRH services through the REST API.

This model-based approach of ABS and its toolset can also be used by the Cloud Engineers as a semi-automated support system: the Engineer still interacts with the Fredhopper Cloud Services to perform at the right time the desired scaling actions suggested by the framework. To achieve this the REST API can be used to forward queries in real-time from the production system to the ABS monitors, whereas the CloudProvider interface deploys actual IaaS virtual machines. Hence to allow the Cloud Engineer to engage in simulating real-world scenarios, or simply to interact with the system in a meaningful manner, we believe it is crucial that the simulation executes *in real-time*.

5 Experimental Results

The FRH case study and its ABS model (≈ 2.000 lines of code⁹) forms the basis of our experimental results. We focus on the following metric, which is part of the SLA negotiated between Fredhopper and its customers (the exact percentages are not fixed, they can be negotiated by customers):

“Services must maintain 95% of the queries with less than 200 milliseconds of processing time, and 99% with less than 500 milliseconds, subtracting the 2% slowest queries.”

⁹ The source code for the FRH model is at <http://github.com/abstools/habs-frh>

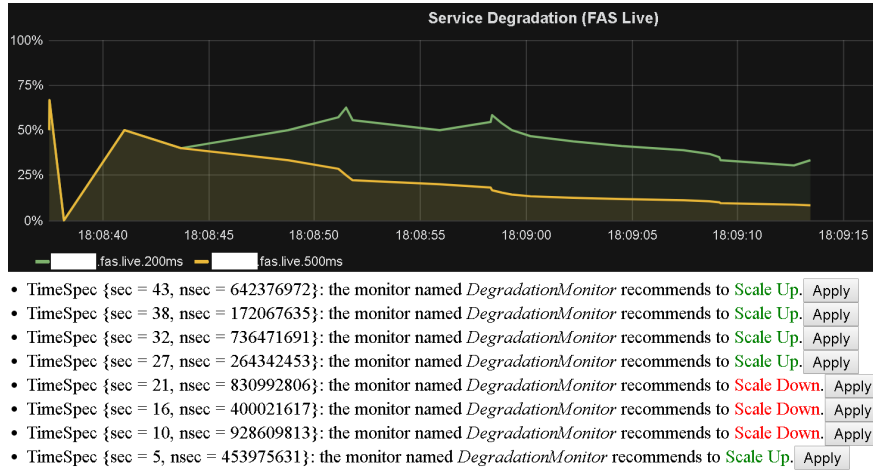
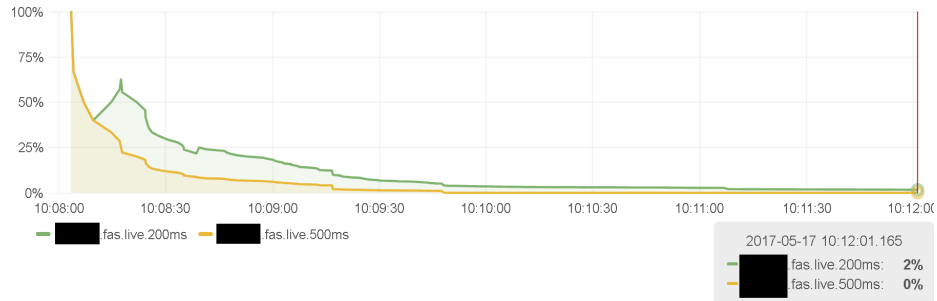


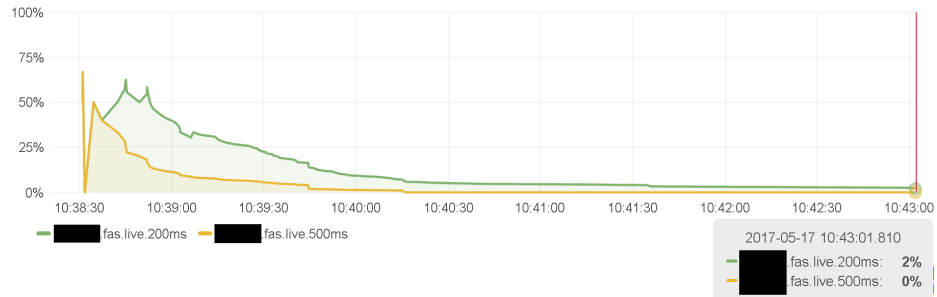
Fig. 5. The GUI of the HITL framework intended for training Cloud Engineers.

Initially, our experiments were focused on the FRH case study behavior when simulating its model (expressed in ABS) without any human intervention. A provisioning script generated by SmartDeployer automatically instantiated all services of the Cloud Architecture (Fig. 2), requested suitable VMs from the CloudProvider and deployed the various kinds of Service instances shown in the diagram on it. For the QueryService, a minimal setup was used with a single instance (co-located with a DeploymentService instance) deployed to an Amazon `m4.large` VM. The input to the simulation was a real-world log file of a particular customer with length of 4 minutes and 30 seconds, coming from a single production VM (of type `m4.large`). Fig. 6a visualizes the Service Degradation of that log file (customer names are anonymized); We then proceeded with simulating the FRH system on the Haskell and Erlang backends of ABS, inputted with the same exact log and using the same deployment scenario.

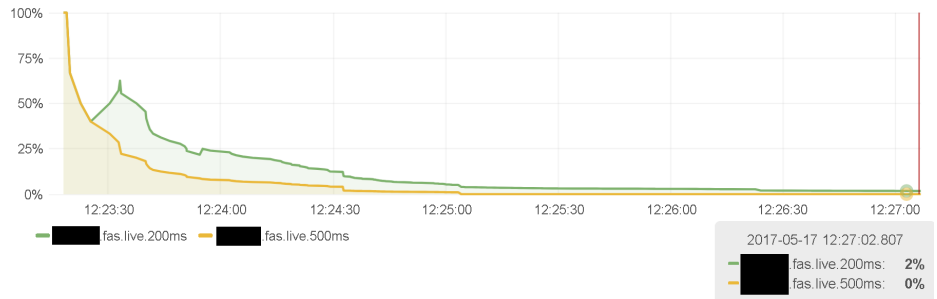
The simulation of the FRH model on the Haskell-ABS backend took 4 minutes and 30 seconds to complete, which matches the log’s length and encourages us to believe that the simulation is done in real-time. The output of the simulation on the Haskell backend is shown in Fig. 6b. There is a deviation that can be seen when comparing it to the original graph of Fig. 6a: the Haskell output reports higher degradation than what would be expected from the real-world log. This can be attributed to three causes; first, there is the overhead of processing the log file itself (network communicating to the logreplay tool). Secondly, the simulation of the real-time measurements of the log file involves *sleep* system calls, which as explained in Section 2, dictates that any “sleeping” thread will be re-activated no sooner than the specified time, but most likely later than prescribed, which depends on factors such as backend implementation, hardware configuration, or the workload of the particular model. Fortunately none of these had great effect on the models we tested, and the reported degradation is negligibly affected by this. The last cause which however has a larger effect on



(a) Original degradation from production system



(b) Haskell simulation of the degradation when simulating the original log



(c) Erlang simulation of the degradation when simulating the original log

Fig. 6. Degradation in the production system and as simulated on different backends

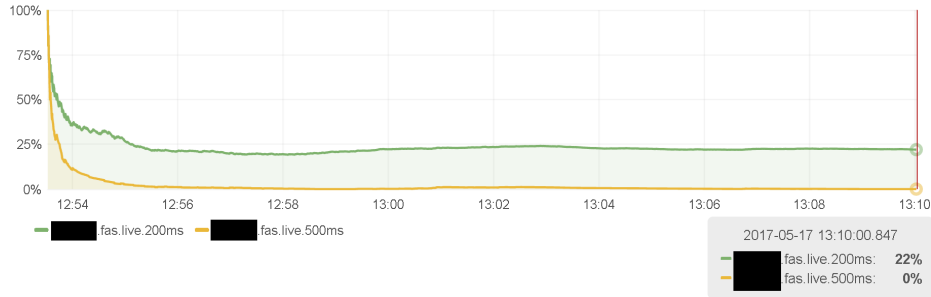
the degradation is that the log file contains a certain number of concurrent requests (requests on a single machine that were served concurrently in time). The recorded processing time of the requests are translated into *Cost* annotations (taking into account the resource capacities of the machine that has processed the request), and therefore the concurrent execution of such requests in the simulation as described in Fig. 1 will further increase the *simulated* processing time of the individual requests. In general, the recorded processing time of the individual requests includes the overhead of time sharing and as such do not specify their “intrinsic” processing time. In practice we think one can obtain a “correct” model by approximating these intrinsic processing time of the individual requests by averaging over different log files and different deployment scenarios.

Moving on to the Erlang symbolic-time simulation, we observe slight inaccuracies of the output (Fig. 6c) compared to the original graph. These inaccuracies can be attributed to two reasons: first, the monitors act autonomously (`while (True){await duration(1,1);...}`), so they may uncontrollably advance the symbolic time by themselves between REST calls of the logreplay tool; as a result the graph is slightly “stretched” because of extra erroneous time advancements. We propose two ways to mitigate this at the ABS language level: a) having a statement `every(intExp()){body}`; which will register the body as a callback to be executed with the period given or b) a statement `await until(t)`; which will resume the process only after the specific time given. In either case the two statements do not advance the time by themselves. The other reason which leads to inaccuracies is that the concurrent requests of the log are processed sequentially (as opposed to Haskell) because of practical difficulties of synchronizing an external tool that uses the real-world clock (logreplay) and the Erlang-ABS runtime which uses the symbolic clock. Since, as mentioned before part of the requests in the log happen to be concurrent, the resulted degradation of the Erlang simulation may differ from the expected original.

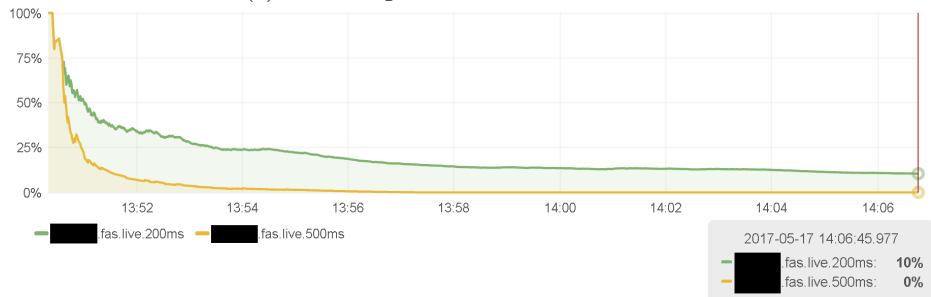
The Erlang-ABS backend took 15min and 30 seconds to complete the simulation of real-world 4min and 30 seconds of the log. This may be attributed to the fact that the granularity of the request timestamps is per *ms* (as given in the log file). We could speed it up by having a more coarse-grained (less accurate) timestamps. Furthermore, the Erlang backend does not use a (parallel) Discrete-Event simulation runtime (called also as-fast-as-possible computer simulation) but a timed-automata inspired runtime for the advancement of the clock, which requires a computationally-heavier continuous global administration of the simulation. Given the reasons above, the code for the monitors `while True {await duration(1,1)}` affects the execution speed. A way to mitigate this is again to have a coarser periodicity for the monitors. Based on these experimental findings, we believe in general simulation frameworks based on symbolic time are not suited for HITL simulations of Cloud applications.

To evaluate the HITL simulation of FRH case study, a training exercise was carried out for the Cloud Engineers. Using our framework, we first visualized the Service Degradation of a different real-world log file, but include the same Service Degradation metric from the SLA as above. The deployment configuration used for that customer was the initial default configuration used by the Cloud Ops team, which provisions the minimum number of VM’s, and each VM has as few resources as needed by the services running on the VM. In particular, aside from the Service instances shared between different customers, such as the `PlatformService` and `LoadbalancerService`, the non-shared initial default per-customer setup consisted of 1 query service instance and a corresponding deployment service instance in every availability zone (in the region of the customer), and those were deployed on an Amazon VM with instance type `m4.large`.

Fig. 7a shows the resulting Service Degradation for that customer on this deployment configuration. The graph shows that in the beginning, performance is low (and Service Degradation is high). This is caused by the fact that after



(a) No scaling - 200ms metric breaks SLA



(b) Performing a Scale-up after 1 minute

Fig. 7. No-scaling versus Scaling during the Haskell simulation

a service is started, an initialization phase is triggered, and performance is (as expected) low during this phase. After a few minutes, initialization finishes and the service degradation metrics stabilize to around 20% queries slower than 200ms and 0% queries slower than 500ms (subtracting the two percent slowest queries). This means that while the target QoS as agreed in the SLA for the category “slower than 500ms” is achieved, this is (by far) not the case for the category “slower than 200ms”.

After establishing that the initial default deployment configuration was not sufficient to satisfy the SLA as agreed with that customer (on that real-world query log file), the training exercise continued. The Cloud Ops were tasked with selecting and executing appropriate scaling actions to mitigate the situation. The scaling actions could be selected through the ABS REST API, or in a very simple front-end (Fig. 5).

During the training exercise, several different scenarios were trained; Fig. 7b shows one scenario of the effect on the Service Degradation after the engineer decided to scale up with 2 query services instances (and corresponding deployment service instance) in two zones on a (simulated) Amazon `m4.xlarge` instance after 1 minute (13:51) into the simulation. At time 13:54 the new machines have finished initializing, and the services deployed on them have been started. After that time, the 200ms metric quickly improves, and after about 25 minutes reaches the target $\leq 5\%$ degradation.

The integrated tool suite described in Section 4 considerably simplified the task of the Cloud Engineers in managing the day-to-day operation of the Cloud services. In particular:

- The support for real-time simulation was critical in providing a realistic training experience for the cloud engineers. It allowed the Ops to evaluate and view metrics of the system and apply corrective actions to the system *at the same speed as they do in the production environment*.
- The high abstraction level of the metrics captured by the ABS monitoring framework enables *SLA-based scaling*, simplifying the decision process of the Cloud ops in selecting the appropriate corrective scaling actions. Still, domain knowledge of the Cloud operator is crucial to properly “translate” their interpretation of multiple (possibly conflicting) metrics over time into corrective actions. The direct relation of the metrics to SLAs and business KPIs in our tool suite eliminated the burden on the Cloud ops to manually interpret how traditional lower-level metrics (such as CPU usage, memory consumption) relate to the higher-level SLA/KPI metrics.
- By suggesting to the Cloud ops only a limited number of possible corrective actions (synthesized by SmartDeployer), the number of choices the Cloud Op has to take in real-time (i.e.: which and how many services to deploy, how to link them, on what kind of VM to deploy them, etc) was reduced substantially. Since the SmartDeployer actions are synthesized based on the deployment requirements and Smartdeployer generates a corresponding provisioning script, the numerous deployment requirements are satisfied automatically “by construction”. However, the quality of the suggestions (actions) proposed by the framework should be improved.

In principle, the suggested SmartDeployer scaling actions could be exploited for a full auto-scaling approach, without any human intervention. We carried out initial experiments, but it turned out to be very complex how to deal with different monitors from heterogeneous sources that give conflicting scaling suggestions, taking into account machine booting time, upcoming promotions from web-shops where peaks in demand are expected, historic data, etc. Thus keeping the human in the loop - the cloud engineers with their domain knowledge - still is crucial to optimize the day-to-day management of services.

6 Conclusion

Our initial experimental results on the use of the presented tool suite provides clear evidence for the viability of HITL simulation of Cloud services for training purposes. The training sessions themselves can further be used to provide feedback to the underlying ABS models of the Cloud services and the monitors. Ultimately, the resulting fine-tuning of these models may reach a level of maturity and confidence that allows their deployment in the real-time monitoring and management of the actual service instances.

In general, we believe that HITL simulation of Cloud services provides a variety of interesting and challenging research problems, for example mining the log files to calculate an approximation of the “intrinsic” processing time of the individual service requests, cancelling the effect of time sharing.

References

1. Albert, E., de Boer, F.S., Hähle, R., Johnsen, E.B., Schlatte, R., Tarifa, S.L.T., Wong, P.Y.H.: Formal modeling and analysis of resource management for cloud architectures: an industrial case study using real-time ABS. *Service Oriented Computing and Applications* 8(4), 323–339 (2014)
2. Bezirgiannis, N., de Boer, F.: ABS: A high-level modeling language for cloud-aware programming. In: *SOFSEM 2016: Theory and Practice of Computer Science*, pp. 433–444. Springer (2016)
3. de Boer, F.S., de Gouw, S.: Combining monitoring with run-time assertion checking. In: *Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*. pp. 217–262 (2014)
4. Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A.F., Buyya, R.: Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw. Pract. Exper.* 41(1), 23–50 (Jan 2011)
5. Giachino, E., de Gouw, S., Laneve, C., Nobakht, B.: 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*. pp. 211–225 (2016)
6. de Gouw, S., Mauro, J., Nobakht, B., Zavattaro, G.: Declarative elasticity in ABS. In: *Service-Oriented and Cloud Computing - 5th IFIP WG 2.14 European Conference, ESOC 2016, Vienna, Austria, September 5-7, 2016, Proceedings*. pp. 118–134 (2016)
7. Johnsen, E.B.: Separating cost and capacity for load balancing in ABS deployment models. In: *Formal Methods for Components and Objects - 11th International Symposium, FMCO 2012, Bertinoro, Italy, September 24-28, 2012, Revised Lectures*. pp. 145–167 (2012)
8. Johnsen, E.B., Schlatte, R., Tarifa, S.L.T.: Modeling resource-aware virtualized applications for the cloud in real-time ABS. In: *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012, Proceedings*. pp. 71–86 (2012)
9. Kliazovich, D., Bouvry, P., Audzevich, Y., Khan, S.U.: Greencloud: A packet-level simulator of energy-aware cloud computing data centers. In: *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*. pp. 1–5 (Dec 2010)
10. Núñez, A., Vázquez-Poletti, J.L., Caminero, A.C., Castañé, G.G., Carretero, J., Llorente, I.M.: icancloud: A flexible and scalable cloud infrastructure simulator. *Journal of Grid Computing* 10(1), 185–209 (2012)