Abstract Behavioral Specification: unifying modeling and programming

Bezirgiannis, N.

Chapter 4

Resource-aware Modeling in HABS

The standard ABS language, described in chapter 2, is adequate to represent models of concurrent object-oriented programs; the ABS user can make use of the ABS tool-suite to analyze, experiment, and execute such models. It becomes, however, more difficult for the user to express models which change their behaviour over *time*; such models are usually constructed during a simulation phase. The word *simulation* can take a broad meaning; here, we use the word to refer specifically to *computer simulation*: the (inexact) reproduction of a real-life process or system, performed with the aid of a computer. We implemented the timed extension of ABS with a real-time interpretation inside the HABS framework.

Furthermore, we model virtualized systems (named Deployment Components) directly inside ABS as first-class citizens of the language in section 4.3, as well as their virtualized resources (speed, memory, bandwidth) in section 4.3. At the end, we evaluate this extension to HABS in an industrial case-study by modeling and simulating real-world cloud environments.

4.1 Modeling time

[Bjørk et al., 2013] address the issue of time-varying models and simulation in ABS with a small extension of the language to deal with time; the entity *time* in their case is left abstract to accommodate all possible scenarios with different notions of time (symbolic or real-time) or units of time (seconds, milliseconds, days, etc.). The following ABS snippet encompasses the wholes new syntax of this "timed" ABS extension by means of an example:

```
{
  Rat i = 3.1;
  duration(i,i+1);
  await duration(i+1,i+2);
  Time n = now();
}
```

A duration(i,j); statement blocks the currently-executing Concurrent Object Group (COG) and all of its processes for less than j time and for the best case i amount of time; in other words, the blocked time is sampled from the interval $[i, j)$. The statement await duration(i,j); will instead block only the currently-executing ABS process for that amount of sampled time; the other processes of the COG can still be scheduled for execution in the meantime. Finally, there is the effectful expression now() which returns the current clock of the simulation in the abstract algebraic-datatype *Time*; this expression is used mostly for printing & debugging purposes. It is worth mentioning the fact that the rest statements of ABS do not "take" time — in the sense of abstract ABS time, they can still take perceived clock time — and treated by the timed extension of ABS as instantaneous.

For our case, we implement the Timed-ABS language extension as an extension of the HABS compiler & runtime, accordingly. We deviate from the initial work on Timed ABS ([Bjørk et al., 2013]) by providing a specific notion of time, that of the passage of real-world time — in short, real-time. This choice becomes important later on since it allows us to have *live* simulations where the human can interact with the computer's simulation, instead of having 'as-fast-as-possible" simulations. Another reason for implementing the Timed-ABS extension for HABS is that in the subsequent Chapter 5 that details the (cloud) distributed-computing part of HABS, the importance of time becomes more apparent in such a real-world setting, where the network latency of communication plays and workflows of cloud services dominate the structure of the model.

A different interpretation of time for Timed-ABS is that of symbolic time, which is is implemented in the Erlang-ABS backend. Specifically, the Erlang-ABS backend of ABS provides a symbolic interpretation of the abstractions modeling (CPU) time, that is, time is modeled by a symbolic clock which is advanced by the execution of a certain kind of statements, so-called duration statements. In contrast, in this thesis we introduce the new Haskell backend for ABS denoted by HABS, 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 [Johnsen et al., 2012], to refer to the ABS abstractions modeling (CPU) time themselves. In this section, 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.

4.2 Modeling virtualized hardware resources

Systems in ABS are composed of resources. Example of resources are the number of CPU cores, their speed, the total memory of the system, the network bandwidth, etc.. In this section we discuss how computing resources are modelled in ABS.

High-level annotations of the ABS code are used to specify the resource consumptions of the annotated statement ([Johnsen et al., 2012, Albert et al., 2014b]). For example to signal the overall-CPU resource consumption of a statement, we annotate it 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 an abstraction of a system, a so called deployment component (see section 4.3), 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.

4.3 Modeling systems

We extend the ABS language with syntactic and library support for Deployment Components. A *Deployment Component* (DC), first described in [Johnsen et al., 2010b], is "an abstraction from the number and speed of the physical processors available to the underlying ABS program by a notion of concurrent resource". Over time, in ABS a DC has further evolved to include other virtualized resources of a computer system seen in the previous section, like CPU time, memory, and bandwidth, which allows to model virtual machines and in general other technologies, e.g. Docker containers,

unikernels. We want to be able to deploy and execute objects on a Deployment Component and that requires at least the presence of CPU resources. In this section we only deal with "simulated" Deployment Components, i.e. machines that do not have actual computing resources but instead simulated ones, for tracking and predicting the possible utilization of real machines.

To be able to programmatically (at will) create and delete machines in any language would require modeling them as first-class citizens of that language. As such, we introduce DCs as first-class citizens to the already-existing language of ABS in the least-intrusive way: by modeling them as objects. Since Deployment Components are expressed by concurrent objects themselves they become an integral part of any ABS model. All created DC objects are typed by the interface `DC`. The minimal interface for deployment components contains the methods `shutdown` for shutting down and releasing the cloud resources of a virtual machine, and `load` for probing its average *system load*, i.e. a metric for how busy the underlying computing-power stays in a period of time. We use the Unix-style convention of returning 3 average values of 1, 5 and 15 minutes. After calling `shutdown()`, the DC object will point to `null`. The DC interface resides in the augmented standard library:

```
module StandardLibrary.CloudAPI;
interface DC {
    Unit shutdown();
    Triple<Rat,Rat,Rat> load();
}
```

Similar to this identifier, a method context contains the thisDC read-only variable (with type DC) that points to the machine host of the currently executing object. A running ABS node can thus control itself (or any other nodes), by getting its system load or shutting down its own machine. However, after its creation, a running ABS node will remain effectively "idle" until some objects are created/assigned to it. The DC annotation can be used in conjunction with the new keyword to specify in which (possibly remote) DC the newly created objects which "live" and run:

```
[DC: dc1] Interf1 o1 = new Cls1(args..);
o1 ! method1(args..);
this.method2(o1);
```

Such objects dynamically deployed onto deployment components are named *remote objects* and share their resources. The `DC` annotation does not change the behaviour of the `new` keyword: it still creates a new object (inside a new COG), initializes it, and optionally calls its run method. Indeed, the unannotated expression `new Cls1(params)` is equivalent (as in syntactic sugar) to the annotated `[DC: thisDC] new Cls1(params)`. References to remote objects are indistinguishable to local object references and can be normally passed around or called for their methods. The ABS language specification and its cloud extension do not dictate a

particular Garbage Collection policy — a specific implementation is provided for distributed HABS at section 5.1.3, but we assume that holding a reference to a remote object or future means that the object is alive, if its DC is alive as well.

Usually the ABS user does not create deployment components directly (i.e. by calling new DC), but instead through a higher object abstraction named *CloudProvider*, which serves both as a factory of deployment components as well as a communication endpoint (in the real and not simulated world this corresponds to the infrastructure service, e.g. Amazon AWS, OpenStack, Azure):

```
CloudProvider cp = new CloudProvider(params);
this.addInstanceDescription(Pair("c4_2xlarge_eu", map(Cons(Pair(CostPerInterval,419),
  Cons(Pair(Cores, 8), Cons(Pair(Memory, 1500), Cons(Pair(Speed, 31), Nil )))))));
this.addInstanceDescription(Pair("m4_large_eu", map(Cons(Pair(CostPerInterval, 120),
  Cons(Pair(Cores, 2), Cons(Pair(Memory, 800), Cons(Pair(Speed, 6), Nil ))))) ));

DeploymentComponent vm1 = cp.launcInstanceNamed("m4_large_eu");
[DC: vm1] new WebServer(8080); // deployed object
```

4.4 A real-time implementation

In this section 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.

We augment the original HABS 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-ABS, Maude-ABS), 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 reschedule 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 Listing 4.1. 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.

```
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);
  }
```

Listing 4.1: The implementation of Cost annotation for the ABS_RT backend

It is worth nothing 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, in addition to 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 Listing 4.1 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-ABS and Java-ABS backends as well.

4.4.1 Comparison with symbolic-time execution

As briefly discussed in section 4.1, the Erlang-ABS backend also implements the Timed-ABS extension but with a symbolic clock as notion of time. The Erlang manual of ABS (at `http://docs.abs-models.org`) says that:

> Time only advances when all processes are blocked or suspended and no process is ready to run. This means that for time to advance, all processes are in one of the following states: the process is awaiting for a guard that is not enabled, the process is blocked on a future that is not available the process is suspended waiting for time to advance, the process is waiting for some resources, In practice this means that all processes run as long as there is work to be done.

At implementation side, the Erlang-ABS backend will execute all processes that are enabled in the current clock to completion, and will advance the time only if all of the processes of the system are blocked (as in idling). Then, the Erlang-ABS runtime will advance the clock to the smallest amount of time of a duration or await duration ABS statement.

The described above Erlang-ABS execution resembles that of timed automata, for example as is done in the model checker UPPAAL. There are certain reproducibility problems attached to this execution method. First of all, there exist the problem of "granularity of concurrency": the Cost resources although being rational numbers, are always distributed to processes of a COG with a granularity of 1 unit. For example, in a hypothetical situation of a DC with Speed=3, one process may

"consume" 2 cost resources, while the other process can only "grab" 1 cost resource. A better approach would be to distribute the resources evenly to all the processes (for the example 1.5 to each process). This is currently hardcoded in the Erlang-ABS runtime and is not parameterizable. A further problem with reproducibility is that the Erlang-ABS runtime does not provide any "local" method ordering; the scheduled processes of a COG do not follow a queue pattern, where a process that arrived earlier will execute also earlier (FIFO), instead the processes are picked up for execution in arbitrary order. In other words, the scheduling policy of Erlang-ABS is non-deterministic. This leads to the inherent problem of non-reproducibility for certain ABS models running with the Erlang-ABS backend. Consider the artificial example of spawning a number of asynchronous methods:

```
module Test;

class C {
  Unit run() {
    Int i = 0;
    while (i<10) {
      this!m(i);
      i=i+1;
    }
   }
  Unit m(Int n) {
    println(toString(n));
  }
}

{
  new C();
}
```

The output of the above model's execution is non-deterministic with the Erlang-ABS runtime, varying between successive runs, e.g. 0 2 3 4 6 8 7 5 1 9 and 4 9 7 2 1 0 6 5 3 8.

However, even with assumption of method ordering inside the COG the execution of an ABS model remains non-deterministic (thus non-reproducible simulation) since there is no fixed scheduler for which COG will execute next. In fact, certain runtimes (Erlang-ABS, HABS) execute the COGs simultaneously for the benefit of parallelism. Consider the following example of i number of COGs:

```
module Test;

interface R {
  Unit m(Int n);
```

```
}
class R implements R{
   Unit m(Int n) {
     println(toString(n));
  }
}

class S(R r, Int i) {
  Unit run() { r!m(i); }
}

{
  R r = new R();
  Int i=0;
  while (i<10) {
    new S(r,i);
    i=i+1;
  }
}
```

The output of the above example will again vary on successive runs. This leads us to consider for future work a simulation of Timed-ABS models where the execution is driven by a discrete-event simulation (DES) engine. In this way, we could achieve reproducibility since every event will be marked with its timestamp and all events are executed in total order. Another theoretical benefit is that such discrete-event simulations can be executed in parallel or distributed over different computers which may improve the execution performance compared to the real-time approach (HABS) as well as that of timed automata (Erlang-ABS).

4.5 Case study: DevOps-in-the-Loop

In this section, we evaluate the ABS_RT backend on an industrial case study. We integrated ABS_RT in a new tool-suite for human-in-the-loop simulations for cloud engineers. Other tools in the suite include the SAGA tool [Boer and Gouw, 2014] for the declarative specification of service metric functions, and SmartDeployer [Gouw et al., 2016] 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 [Gouw et al., 2016] 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 [Giachino et al., 2016a], 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.

Fredhopper[1] 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). 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* scalable[2]. 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. 4.1). 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 two am and five 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.

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. 4.1 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. Cur-

[1] `https://www.fredhopper.com/`

[2] `en.wikipedia.org/wiki/Scalability#Horizontal_and_vertical_scaling`

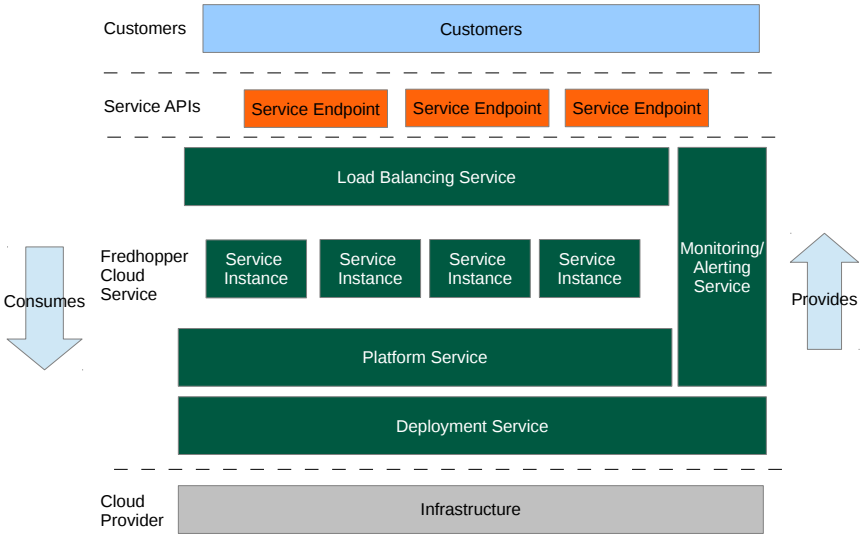


Figure 4.1: The architecture of the Fredhopper Cloud Services

rently 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.

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 writ-

ing 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. 4.3a shows a visualization of monitors in Grafana, the visualization framework used by ABS.

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 deployment[3].
- General business KPIs.

[3]A deployment associates service instances to Virtual Machines

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.

4.5.1 The tool

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

- The SAGA tool [Boer and Gouw, 2014] was tweaked for monitoring SLA metrics and the Grafana framework visualizes the metrics
- The SmartDeployer [Gouw et al., 2016] for synthesizing deployment actions
- A logreplay tool for replaying real-world log files
- The new Haskell ABS_RT backend for real-time simulations.

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,

suggests 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 [Gouw et al., 2016] 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 recorded by the actual system on the ABS model of the system. The logreplay tool is responsible for firing at appropriate times a HTTP API call (as explain in section 3.2.5) to the running simulation for each request recorded in the log file. These requests will trigger ABS code that contains *Cost* annotations (Listing 4.2), which has the effect of the real-time simulation as defined for the ABS_RT backend.

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

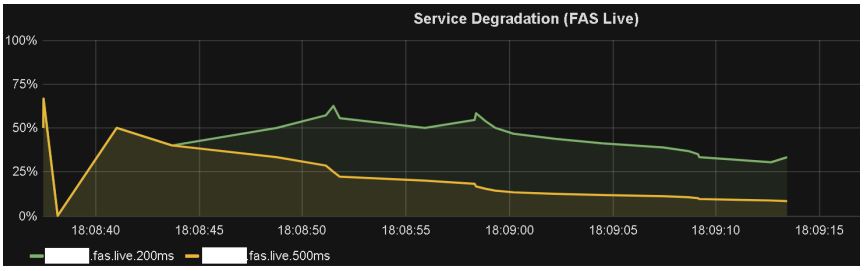
Listing 4.2: ABS method that process each incoming request from the log-file

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 can then 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 is shown in Fig. 4.2. This interface makes also use of the HTTP API (Listing 4.3) extension as implemented in the HABS 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 ...");
}
```

Listing 4.3: The main ABS block exposing the FRH services through the HTTP API.



- TimeSpec {sec = 43, nsec = 642376972}: the monitor named *DegradationMonitor* recommends to Scale Up. Apply
- TimeSpec {sec = 38, nsec = 172067635}: the monitor named *DegradationMonitor* recommends to Scale Up. Apply
- TimeSpec {sec = 32, nsec = 736471691}: the monitor named *DegradationMonitor* recommends to Scale Up. Apply
- TimeSpec {sec = 27, nsec = 264342453}: the monitor named *DegradationMonitor* recommends to Scale Up. Apply
- TimeSpec {sec = 21, nsec = 830992806}: the monitor named *DegradationMonitor* recommends to Scale Down. Apply
- TimeSpec {sec = 16, nsec = 400021617}: the monitor named *DegradationMonitor* recommends to Scale Down. Apply
- TimeSpec {sec = 10, nsec = 928609813}: the monitor named *DegradationMonitor* recommends to Scale Down. Apply
- TimeSpec {sec = 5, nsec = 453975631}: the monitor named *DegradationMonitor* recommends to Scale Up. Apply

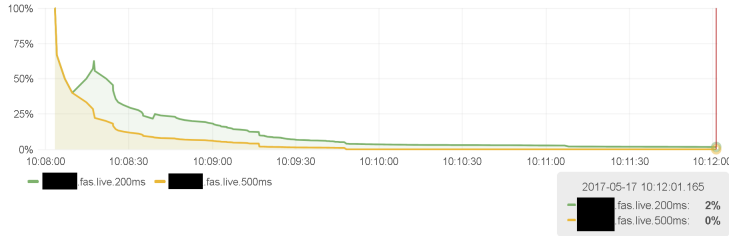
Figure 4.2: The GUI of the HITL framework intended for training Cloud Engineers.

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 HTTP 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*.

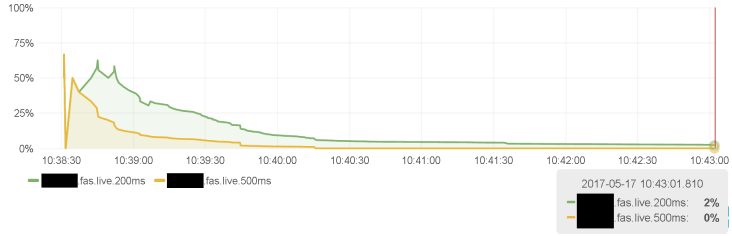
4.5.2 Benchmark

The FRH case study and its ABS model ($\approx$ 2.000 lines of code[4]) 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):

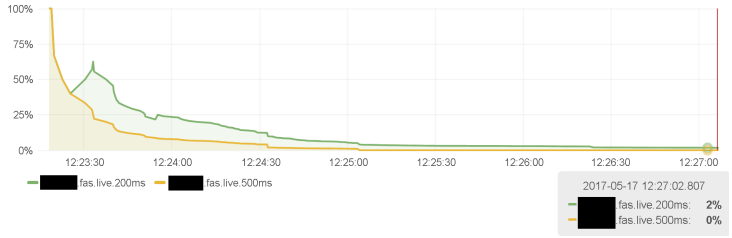
[4]The source code for the FRH model is at `http://github.com/abstools/habs-frh`



(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

Figure 4.3: Degradation in the production system and as simulated on different backends

> *"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."*

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. 4.1), requested suitable VMs from the CloudProvider

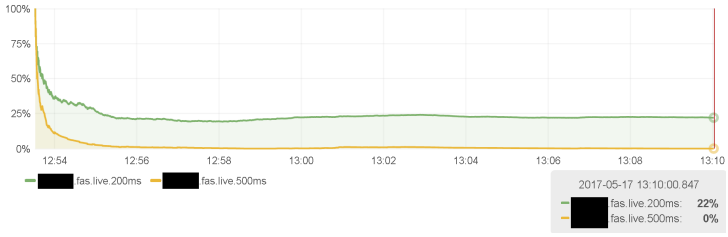
and deployed the various kinds of Service instances shown in the diagram on it. For the QueryService, a minimal setup was used with a single instance (co-located with a DeploymentService instance) deployed to an Amazon m4.large VM. The input to the simulation was a real-world log file of a particular customer with length of 4 minutes and 30 seconds, coming from a single production VM (of type m4.large). Fig. 4.3a 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 HABS 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 HABS backend is shown in Fig. 4.3b. There is a deviation that can be seen when comparing it to the original graph of Fig. 4.3a: the HABS 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 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 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 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.

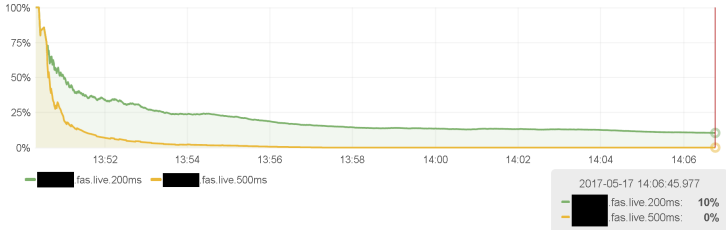
Moving on to the Erlang-ABS symbolic-time simulation, we observe slight inaccuracies of the output (Fig. 4.3c) compared to the original graph. These inaccuracies can be attributed to two reasons: first, the monitors act autonomously (`while (True) {await duration (1,1);...}`), so they may uncontrollably advance the symbolic time by themselves between HTTP 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-ABS 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-ABS 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.



(a) No scaling - 200ms metric breaks SLA



(b) Performing a Scale-up after 1 minute

Figure 4.4: No-scaling versus Scaling during the Haskell simulation

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 one 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. 4.4a 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 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 HTTP API, or in a very simple front-end (Fig. 4.2).

During the training exercise, several different scenarios were trained; Fig. 4.4b shows one scenario of the effect on the Service Degradation after the engineer decided to scale up with two query services instances (and corresponding deployment service instance) in two zones on a (simulated) Amazon m4.xlarge instance after one 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 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.

4.6 Related Work

There exists a variety of cloud simulation tools including CloudSim [Calheiros et al., 2011], GreenCloud[Kliazovich et al., 2010], and iCanCloud [Núñez et al., 2012]; 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 runnning 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.

The Timber language [Black et al., 2002] `http://timber-lang.org` is a Haskell-like language but with strict semantics. It offers a limited form of concurrent objects with the extra feature of attaching baselines and deadlines to methods. The execution is non-deterministic and according to the Chemical Abstract Machine [Berry and Boudol, 1990].

The most known use of real-time in computing comes in the form of a real-time operating system (OS). Such OSes (e.g. QNX, FreeRTOS) use certain schedulers to maximize the responsiveness of the system, while minimizing any deadline misses: these criteria are paramount in the world of embedded systems. A real-time OS, compared to real-time language like ABS, operates on the level of system processes (executable programs) and not inside a concurrent program itself, thus it may be agnostic of any inner program characteristics.

Hardware description languages such as Verilog and VHDL and multicore simulation tools such as Graphite and Sniper are used to also construct software that "talks" about hardware. However their purposes are in contrast with the "resource-aware" programming that we detail in this chapter. Specifically, whereas such languages focus more on the description, design, architecture and implementation of computing hardware (e.g. CPUs, caches), the ABS tries to create models and simulations of software that take explicit control and monitor its hardware (cloud) resources.

On the side of ABS, related work revolves around the extension of Standard ABS with real-time concepts [Johnsen et al., 2012] and its further refinement [Bjørk et al., 2013] which adds language support for custom (user-defined) process schedulers. This refinement permits the containment of the non-deterministic scheduling of ABS processes inside the COG, programmatically; yet this is not enough to make ABS programs reproducible (deterministic), because (hard/soft)-deadline misses can still occur and more importantly there is no global — only local, inside the COG — ordering of processes.