Research Report:
A Formal Model of the Kubernetes Container Framework

Gianluca Turin,
Andrea Borgarelli,
Simone Donetti,
Einar Broch Johnsen,
S. Lizeth Tapia Tarifa,
Ferruccio Damiani

Research report 496, June 2020

ISBN 978-82-7368-461-5
ISSN 0806-3036

A Formal Model of the Kubernetes Container Framework *

Gianluca Turin[1,2], Andrea Borgarelli[2], Simone Donetti[2], Einar Broch Johnsen[1], S. Lizeth Tapia Tarifa[1], and Ferruccio Damiani[2]

[1] Department of Informatics, University of Oslo, Oslo, Norway
{gianlutu,einarj,sltarifa}@ifi.uio.no
[2] Department of Computer Science, University of Turin, Turin, Italy
andrea.borgarelli@edu.unito.it
{simone.donetti,ferruccio.damiani}@unito.it

Abstract. Loosely-coupled distributed systems organized as collections of so-called cloud-native microservices are able to adapt to traffic in very fine-grained and flexible ways. For this purpose, the cloud-native microservices exploit containerization and container management systems such as Kubernetes. This paper presents a formal model of resource consumption and scaling for containerized microservices deployed and managed by Kubernetes. Our aim is that the model, developed in Real-Time ABS, can be used as a framework to explore the behavior of deployed systems under various configurations at design time—before the systems are actually deployed. We further present initial results comparing the observed behavior of instances of our modeling framework to corresponding observations of real systems. These preliminary results suggest that the modeling framework can provide a satisfactory accuracy with respect to the behavior of distributed microservices managed by Kubernetes.

1 Introduction

Software that was considered scalable yesterday, may now be perceived as inflexible and overly entangled compared to the suites of so-called microservices that are today widely used [4]. Microservices are loosely coupled, independently deployed, cloud-native small services [26]. Kubernetes [16] is a framework to resiliently run distributed systems built from such microservices; it takes care of scaling and failover for the application, provides deployment patterns, service discovery, load balancing and other development-related functionalities.

The underlying technology for orchestrating microservices with Kubernetes, is containerization [11]. Containers encapsulate a microservice environment, abstracting details of machines and operating systems from the application developer and the deployment infrastructure. Well-designed containers and container

* Supported by the Research Council of Norway through the project *ADAPt: Exploiting Abstract Data-Access Patterns for Better Data Locality in Parallel Processing* (www.mn.uio.no/ifi/english/research/projects/adapt/).

images are scoped to a single microservice, such that managing microservices means managing containers rather than machines. Thus, containerization enables a shift from machine-oriented to application-oriented orchestration of a system's deployment by managing containers to minimize the downtime for any deployed microservice, even when the system is flooded with requests.

In this paper, we develop a formal model of resource consumption and scaling for containerized microservices deployed and managed by Kubernetes. Although this model abstracts from many aspects of Kubernetes (e.g., self-healing, rollouts, rollbacks, and storage orchestration), it already allows system deployment under several configurations to be explored at the modeling level, *before the system is actually deployed*. Our objective with this work is to develop a modeling framework which can help the developer in finding a deployment strategy for a microservice-based system which meets the system's performance requirements. We aim to facilitate the comparison of different deployment strategies on a highly configurable and executable model. Although not addressed in this paper, the formal model can also be used to verify liveness and safety properties for workflows deployed as microservices with Kubernetes.

The Kubernetes model has been developed using Real-Time ABS [6, 22], a formal executable modeling language targeting distributed and cloud-based systems. We present a preliminary validation of our work by comparing results obtained with the Kubernetes model to observations of a real system running on HPC4AI [3], a cluster for deploying high-performance applications. The results of this comparison suggest that the model-based analysis of an application's deployment complies with the observed performance of its actual deployment.

The main contributions of this paper can be summarized as follows:

- **Formalization:** We develop a succinct formal executable model of Kubernetes, a state-of-the-art management framework for monitoring resources consumption and scalability of microservices;
- **Configurable modeling framework:** The developed Kubernetes model can be configured to different client workloads and to different microservices running in parallel and affecting each others performance. By means of simulations, system administrators can easily compare how different parameter configurations affect the performance of their deployed microservices at the modeling level;
- **Evaluation:** The proposed modeling framework is validated by comparing an instance of the framework, modeling a real system deployed using Kubernetes, to the modeled system. We consider several scenarios in which different workloads will trigger the need for automatic autoscaling. The results suggest that our modeling framework can provide a satisfactory accuracy with respect to the behavior of real systems.

Paper overview. Section 2 introduces microservices, Kubernetes and Real-Time ABS. Section 3 presents the developed Kubernetes model. Section 4 discusses how the model was validated. Section 5 surveys related work and Section 6 concludes the paper.

2 Background

2.1 Microservices, containers and their management via Kubernetes

Microservices [26] are small basic services which are easy to adapt to distributed hardware. They stem from service-oriented architectures (SOA) [14] and service-oriented computing (SOC) [17]. Microservices are so-called *cloud native*; i.e., they are built to run scalable applications on cloud infrastructure. An application consists of a collection of loosely coupled microservices. This decoupling makes them easier to develop, deploy, scale, monitor and maintain in isolation. Microservice architectures facilitate scalability since new instances of the same microservice can be launched to split the workload locally, without scaling the overall service.

Containers encapsulate execution environments for microservices, abstracting from details of physical and virtual machines and operating systems from the application developer and the deployment infrastructure. Containers have been proposed instead of heavy VMs, and raise the level of abstraction from running a service on virtual hardware to running it using *logical resources*. Containers keep the advantages of virtualization such as modularity, but the unit of deployment is the container and not a full VM, which opens for better utilization of resources. Containers offer better scalability and maintainability because they can be added or updated easily, such that resources can be shared in clusters to which containers can be added or removed on-demand.

By encapsulating microservices in containers, the services can be monitored with respect to service performance and resource utilization. In contrast to VMs, which run all components, including an operating system on top of virtualized hardware, containers are *lightweight* but still keep their own filesystem, CPU, memory, and process space similar to a VM. However, they are decoupled from the underlying infrastructure, and additional containers can be created at execution time rather than only at deployment time. They are also *portable* across clouds and OS distributions [15], therefore they require much less space and have faster booting time.

Kubernetes is an open-source system[3] for managing containerized applications across multiple hosts. It provides basic mechanisms for deployment, maintenance, and scaling of applications. Figure 1 depicts a logical representation of a Kubernetes instance in a public or private cloud. Among all the components implementing its functionalities, in the rest of this section, we briefly introduce the main Kubernetes components related to resource management, load balancing and autoscaling[4] (for further details, see [16]).

Pods are the basic scheduling unit in Kubernetes. They are high-level abstractions for groups of containerized components. A pod consists of one or more containers that are guaranteed to be co-located on the host machine and can share resources. A pod is deployed according to its resource requirements

[3] https://github.com/kubernetes/kubernetes/

[4] https://kubernetes.io/docs/concepts/

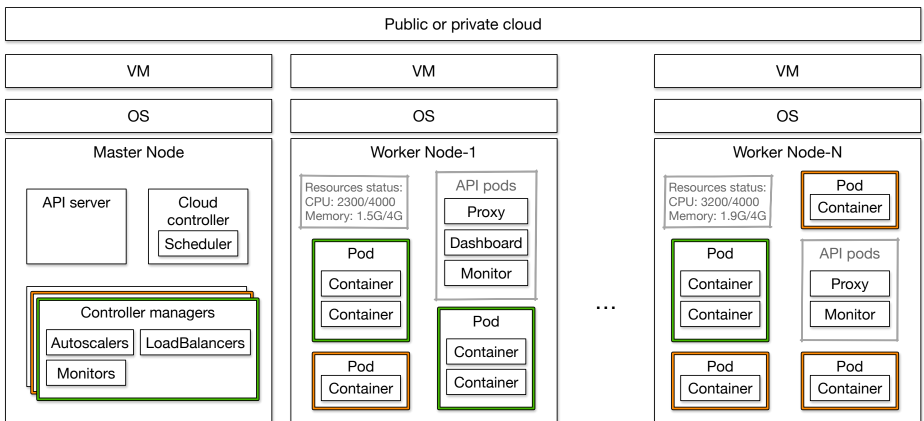


Fig. 1: A logical representation of Kubernetes components in a generic cloud infrastructure. The colors mark services deployed on the cluster, their controller managers reside on the master and their pods are distributed among the workers.

and has its own specified resource limits. For two or more pods to be deployed in the same node, the sum of the minimum amounts of resources required for the pods needs to be available in the node. All pods have unique IP address, which allows applications to use ports without the risk of conflict. Within the pod, containers can reference each other directly, but a container in one pod cannot address a container in another pod without passing through a reference to a service; the service then holds a reference to the target pod at the specific pod IP address. The IP addresses of pods are ephemeral; i.e., they are reassigned on pod creation and system boot.

Services represent components that act as basic internal load balancers and ambassadors for pods. A service groups together a logical collection of pods that perform the same function and presents them as a single entity. This allows the Kubernetes framework to deploy a service that can keep track of and route to all the back-end containers of a particular type. Internal consumers only need to know about the stable endpoint provided by the service. Meanwhile, the service abstraction enables the scaling or replacing of back-end work units as necessary. The IP address of a service remains stable regardless of changes to the pods to which it routes requests. By deploying a service, the associated pods gain discoverability, which simplifies container designs. Whenever access to one or more pods needs to be provided to another application or to external consumers, a service can be configured. Although services, by default, are only available using an internally routable IP address, they can be made available outside of the cluster.

Autoscalers are responsible for ensuring that the number of pods deployed in the cluster matches the number of pods in its configuration. There is one autoscaler for each service, managing a group of identical, replicated pods which

are created from pod templates and can be horizontally scaled. Autoscalers are processes that refer to a pod template and control parameters to scale identical replicas of a pod horizontally, i.e. by increasing or decreasing the number of running copies. Thus, autoscalers facilitate load distribution and increase availability natively within Kubernetes.

Nodes in a cluster are each given a role (master or worker) within the Kubernetes ecosystem. One node functions as the master node, it implements a server that acts as a gateway and controller for the cluster by exposing an API for developers and external traffic. It carries out scheduling, and orchestrates communication between other components. The master node acts as the primary point of contact with the cluster and is responsible for most of the centralized logic that Kubernetes provides. The workers host pods and form the larger part of a Kubernetes cluster. The worker nodes have explicit resource capabilities, which are known by the system. These are given as a set of labels attached to a worker node to specify its version, status and particular features.

Scheduler is in charge of assigning pods to specific nodes in the cluster. The scheduler matches the operating requirements of a pod's workload to the resources that are available in the current infrastructure environment, and places pods on appropriate nodes. The scheduler is responsible for monitoring the available capacity on each node to make sure that workloads are not scheduled in excess of the available resources. The scheduler needs to know the total capacity of each node as well as the resources already allocated to existing workloads on the nodes.

2.2 Real-Time ABS

The *a*bstract *b*ehavioral *s*pecification language (ABS)[5] is an actor-based, object-oriented modeling language targeting concurrent and distributed systems and supports the design, verification, and execution of such systems [18]. ABS has a Java-like syntax and a concurrency model, based on active objects, which decouples communication and synchronization using asynchronous method calls, futures and cooperative scheduling [7]. ABS is an open-source research project.[6]

The functional layer of ABS is used to model computations on the internal data of objects. It allows designers to abstract from the implementation details of imperative data structures at an early stage in the software design. The functional layer combines parametric algebraic data types (ADTs) and a simple functional language with case distinction and pattern matching. ABS includes a library with predefined datatypes such as Bool, Int, String, Rat, Float, Unit, etc. It also has parametric datatypes such as lists, sets and maps. All other types and functions are user-defined.

The imperative layer of ABS allows designers to express communication and synchronization between active objects. In the imperative layer, threads are encapsulated within COGs [18,28] (concurrent objects groups). Threads are created

[5] `www.abs-models.org`

[6] ABS can be found on GitHub at `github.com/abstools/abstools`.

automatically at reception of a method call and terminated after the execution of the method call is finished. ABS combines active (with a run method which is automatically activated) and reactive behavior of objects by means of cooperative scheduling: Inside COGs threads may suspend at explicitly defined scheduling points, after which control may be transferred to another thread. Suspension allows other pending threads to be activated. The suspending thread does not signal any other particular thread, instead the selection of the next thread to be executed is left to the scheduler. Between these scheduling points, only one thread is active inside a COG, which means that race conditions are avoided.

Real-Time ABS [6] extends ABS with support for the modeling and manipulation of dense time. This extension allows the logical execution time to be represented inside methods. The local passage of time is expressed in terms of **duration** statements (which constrain time advance, similar to guards in, e.g., UPPAAL [23] and Real-Time Maude [27]). To express dense time, we consider the two types Time and Duration Real-Time ABS provides. Time values capture points in time as reflected on a global clock during execution. In contrast, finite durations reflect the passage of time as local timers over time intervals.

ABS is supported by a range of analysis tools (see, e.g., [1]); for the analyses in this paper, we are using the simulation tool which generates Erlang code.

3 A Kubernetes Model in Real Time ABS

In this section, we present the Real-Time ABS model of Kubernetes, with a focus on resource management and autoscaling, by modeling the Kubernetes components involved in the deployment of a service. We aim for the model to be *executable* and to faithfully *reproduce* the behavior of Kubernetes. The precision of this model determines the predictive capabilities of the simulations of real world scenarios.

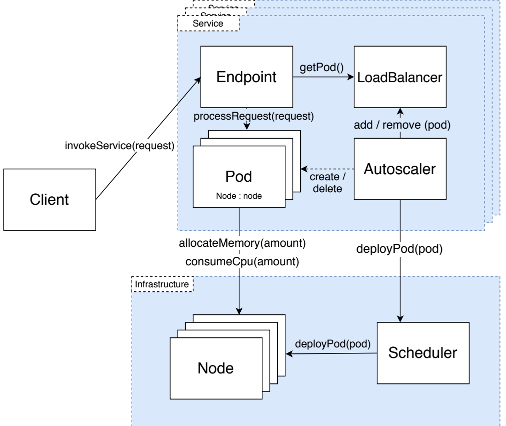


Fig. 2: The architecture of the modeled Kubernetes cluster

Figure 2 shows the structure of a modelled cluster. A service is composed from its pods, an endpoint, a load balancer and an autoscaler. Clients invoke the service by sending a request to the endpoint which gets a selected pod using the load balancer. A pod is deployed on a node and consumes its resources while processing a request. The scheduler manages the number of pods for the service and calls the autoscaler to deploy new pods. In the remainder of this section, we discuss some selected aspects of this model.[7]

[7] The full model is available at `https://doi.org/10.5281/zenodo.3975006`.

3.1 Modeling of Pods

A service is carried out by its pods, for simplicity in the proposed model pods are assumed to consist of a single container (a pod with many containers would correspond to a pod running one container which consumes the sum of their consumed resources). They are deployed onto nodes whose resources are consumed while processing requests.

Figure 3 shows the model of a pod using the PodObject class: a PodObject is instantiated by passing the configuration parameters which are serviceName, id, compUnitSize, cpuRequest, cpuLimit, monitor (used by the method processRequest in Fig. 4) and insufficientMemCooldown. After the underlying node is set by the setNode method, the refreshAvailableCpu cycle starts. The PodObject class has a custom scheduler which executes refreshAvailableCpu as the first method of every time interval. Note that the auxiliary function reset_availCpu_scheduler, which is set as custom scheduler of the COG by the expression inside square brackets (Fig. 3 Line 1), ensures that the scheduler gives priority to the execution of method refreshAvailableCpu and guarantees that every consumed CPU unit is counted in the right time interval. If availableCpu falls to zero the pod has reached its cpuLimit meaning no more CPU will be consumed within that time interval. The allocateMemory and releaseMemory methods manage memory allocation and deallocation on the Node, they are both called in the processRequest method. If the Node's free memory is not sufficient, the allocateMemory method waits for insufficientMemCooldown time before retrying.

Figure 4 shows processRequest method (called by clients) in a PodObject, which models resource consumption while processing a request. In our model, a Request is modeled as a pair of CPU and memory costs. The method first stores information about the CPU and memory cost, a time stamp started, for the calling time of a request, and a deadline for the request to be processed. At lines 2 and 3 the required memory is allocated, the request cost is then consumed one step at a time in the loop of lines 6–17. The size of the step is compUnitSize and is set in the pod configuration, which determines the amount of CPU the pod can consume in a round, having the same compUnitSize for every pod achieves fair CPU scheduling on a node. If the Node runs out of CPU resources, the consumption is suspended (consumeCpu sets the variable blocked to True) for that time interval, it is then resumed in the next time interval after the pod's monitor is updated. At line 9 availableCpu is checked, if it is equal to zero the pod limit is reached and no more cost is consumed within that time interval. Once the request cost is entirely consumed, at line 19 the previously allocated memory is released and at line 21 the total time spent in the process is computed subtracting started to the actual time. Line 22 shows how the spentTime is then compared to the deadline This approximates the quality of service related to the response time, separating served requests between successes and failures. The passing of time is a consequence of the limited amount of available CPU on a node in every time interval. As explained in Sect. 2.2, the value of time during the model execution is managed by the functions provided by Real-Time ABS.

```
1  [Scheduler: reset_availCpu_scheduler(queue)] class PodObject(String serviceName, Int id,
2  Rat compUnitSize, Rat cpuRequest, Rat cpuLimit, ResourcesMonitor monitor,
3  Rat insufficientMemCooldown) implements Pod {
4    Bool blocked = False; Node node = null; Rat availableCpu = 0;
5
6    Unit setNode(Node n){ this.node = n; this!refreshAvailableCpu(); }
7
8    Unit refreshAvailableCpu(){
9      this.availableCpu = cpuLimit; // sets max available CPU for this time interval
10     this.blocked = False;
11     await duration(1,1);
12     this!refreshAvailableCpu();}
13
14   Bool processRequest(Request request, Time started, Duration deadline){ ... }
15
16   Rat allocateMemory(Rat requiredMemory){
17     Bool memoryAllocated = False;
18     Rat givenMemory = 0;
19
20     while (!memoryAllocated){
21       givenMemory = await node!allocateMemory(requiredMemory);
22       if (givenMemory > 0){ memoryAllocated = True; }
23       else {await duration(insufficientMemCooldown,insufficientMemCooldown);}}
24     return givenMemory;}
25
26   Rat releaseMemory(Rat amount){ Rat v = await node!releaseMemory(amount); return v;}
27   ...}
```

Fig. 3: PodObject class

```
1  Bool processRequest(Request request, Time started, Duration deadline) {
2    Rat cost = requestCost(request); Rat requiredMemory = memory(request);
3    this.allocateMemory(requiredMemory); // memory allocation
4    monitor!consumedMemoryUpdate(requiredMemory);
5
6    while (cost > 0){
7      ...
8      if (cost >= compUnitSize){
9        await this.availableCpu > 0; // check on the pod limit
10       await node!consumeCpu(compUnitSize,this); // consume node CPU
11       await !this.blocked; // refresh sync
12       availableCpu = availableCpu − compUnitSize;
13       monitor!consumeCpu(compUnitSize);
14       cost = cost − compUnitSize; // cost decreases
15     } else if (cost > 0){ ... } // consume remaining cost
16     ... suspend;}
17
18   this.releaseMemory(requiredMemory); // memory release
19   monitor!consumedMemoryUpdate(−requiredMemory);
20   Rat spentTime = timeDifference(now(),started); // deadline check
21   Bool success = (spentTime <= durationValue(deadline));
22   return success; }
```

Fig. 4: Pod processRequest method

```
1  interface ServiceLoadBalancer{
2    Pod getPod();
3    Unit addPod(Pod p, ResourcesMonitor rm);
4    Unit removePod(Pod p);
5    List⟨Pair⟨Pod,ResourcesMonitor⟩ ⟩ getPods();
6    ServiceState getConsumptions(); // Total service consumption
7    List⟨PodState⟩ getPodsConsumptions();}
```

Fig. 5: ServiceLoadBalancer interface

3.2 Modeling of Services

A service is invoked through its endpoint which provides the service reference for the clients. As explained in Sec. 2.1, every service has its own load balancer that chooses the pod to which the endpoint forwards the request. The load balancer's policy for work distribution between all the pods of the service is round robin. Figure 5 shows the ServiceLoadBalancer interface of our model: getPod returns the pod for forwarding a request, addPod and removePods add and remove pods from the pods of the service, getPods returns the service's available pods and getConsumptions and getPodConsumptions return the total consumption and per pod consumption values in the current time interval.

Like in a real Kubernetes installation, a service in our model is configurable. Several parameters are passed on service instantiation as PodConfig and ServiceConfig. PodConfig specifies the CPU request and limit for the pods, the cool-down time for insufficient memory and the computation unit size. The memory cool-down is the time awaited before retrying in case there's not enough free memory on the node. The computation unit size is the amount of cost computed every time the pod is given the CPU. For example, if CompUnitSize for Service A is 1 and for Service B is 2, the pods of Service B will execute twice the cost of the pods of Service A every time they are scheduled. This allows control over the CPU time scheduling, setting all unit sizes to the same amount will provide a fair scheduling, while setting different values allows to set different priorities for the pods. ServiceConfig specifies the initial number of pods, the minimum and maximum number of pods for the service and the configuration of the autoscaler.

3.3 Modeling of Autoscalers

Every service in our model has also its own Autoscaler which creates and deletes pods. On service initialization it creates the specified starting number of pods and then periodically checks the average load on the pods. In case the given thresholds for scaling are reached, it creates or deletes pods accordingly. After creating a pod the Autoscaler calls the Scheduler to deploy it on a node. Figure 6 shows the resize method of the Autoscaler, it fetches the average pod CPU consumption ratio in the current time interval, waits for the next time interval to apply the scaling, then starts over. The Autoscaler has its own configuration: cycle period gives the frequency of resize execution, the thresholds for scaling (percentages of requested CPU) up and down are modeled by downscaleThreshold and upscaleThreshold and finally, downscalePeriod specifies how long a pod set has to stay idle before shrinking. While scaling up is immediate as soon as the threshold is hit, for scaling down the load is required to stay below the threshold for a configurable period of time before any pod is deleted.

3.4 Modeling of Nodes

The Kubernetes master node is not explicitly modeled, its functionalities are implemented in the model logic, while Node models the Kubernetes worker node,

```
1  Unit resize(){
2    ServiceState ss = await lb!getConsumptions(); // get service consumption
3    Rat serviceRatio = cpuRatio(ss);
4
5    if (serviceRatio < downscaleThreshold){ // updates the cumulative counter
6      underDsThresholdCounter = underDsThresholdCounter + 1;
7    } else { // reset it
8      underDsThresholdCounter = 0;}
9    await duration(cycle, cycle); // scale in the successive time interval
10   if (serviceRatio >= upscaleThreshold && nPods < maxPods){
11     ... // scale up}
12   if (underDsThresholdCounter >= downscalePeriod && nPods > minPods){
13     ... // scale down}
14   this!resize();}
```

Fig. 6: ServiceAutoscaler resize method

which has a given amount of resources (CPU and memory) to be consumed by its running pods. CPU and memory capacities for a node are specified upon node creation:

- CPU is refreshed every time interval, the total amount of computed costs on a node in the time interval cannot exceed the node's CPU capacity.
- Memory is time independent, it can be decreased and restored, it is decreased when a pod starts the processing of a request and allocates memory cost on the node memory. If there is enough free memory then it is decreased for the whole computation time and the allocated amount is restored on request completion. In case the free memory is insufficient, the request remains pending until enough memory is available.

The available resources of the node are statically reserved when a pod is scheduled. The amount of CPU required by the pod serves as discriminant for the scheduler to find a suitable node. (This easily extends to matching over multiple resource capabilities using the aforementioned label mechanism, which we have left for future work.) Hence a node can be fully occupied while actually idle, since there can be many pods deployed on it, but none is receiving requests.

3.5 Modeling of Scheduler

The Scheduler deploys pods on nodes. Figure 7 shows the deployPod method of the Scheduler: it checks the pod CPU request and compares it to the available CPU in the least busy node. If there is enough available CPU, the pod is scheduled on that node, otherwise it remains pending, to be scheduled in another time interval.

4 Validating the Model

We report on initial experiments to assess the precision of our model with respect to real microservices managed by Kubernetes.

```
1  Node deployPod(Pod p, ResourcesMonitor rm){
2    Bool deployed = False;
3    Rat requestedCpu = await rm!getCpuRequest();
4    Node result = null;
5
6    while (!deployed){
7      result = head(activeNodes);
8      Rat maxCpu = await result!getAvailableCpu(); // total cpu − total requested CPU
9      List⟨Node⟩ nodesToCheck = tail(activeNodes);
10     foreach ( n in nodesToCheck){ ... // get the node with maximum available CPU}
11     if (maxCpu >= requestedCpu){await result!addPod(p,rm); deployed = True;}
12     else{await duration(1,1);} }
13   return result;}
```

Fig. 7: Scheduler deployPod method

4.1 Experimental setup

We set up experiments in which we compare two simple scenarios of microservices running on a cluster to simulations in our model.

HPC4AI. The experiments have been performed on the HPC4AI infrastructure. HPC4AI [3] is a centre on High-Performance Computing for Artificial Intelligence at the University of Turin and the Polytechnic University of Turin, which offers on-demand provisioning of AI and BDA cloud services to a heterogeneous industrial community of Small-Medium Enterprises (SMEs) active in many different sectors and leaning towards Industry 4.0. The centre aims at an increasingly connected ecosystem of devices that produce digital data of increasing variety, volume, speed and volatility. To fully exploit its potential, the next generation of AI applications must embrace distributed High-Performance Computing (HPC) techniques and platforms, where computing and data management capabilities of distributed HPC are readily and easily accessible on-demand to data scientists, who are more used to perform their work locally on interactive platforms. The centre is currently looking at using containerized microservices for this purpose. The preliminary results of this paper contribute towards a modeling framework to equip HPC4AI with deployment decisions for this complex setup.

Simulations. We replicated two simple scenarios in the model, each simulating the execution of a stress test on a microservice system deployed on the HPC4AI cluster. The stress tests have been created with Apache Jmeter, a tool generating traffic to test web services. To reproduce the same circumstances, we modeled the cluster infrastructure and measured the load generated during the stress test for any type of service request. To this end, we represented stress tests as waves of requests (see Figs. 8a and 10c). To reproduce the load of a wave, the model instantiates a certain number of clients; by duplicating the number it will simulate twice the load of the original wave.

We consider single workload and mixed workload scenarios. In the *single workload scenario* the flow of requests is generated by three succeeding groups of threads targeting the same service and running at different speed, such that the central wave, with the highest load (see Fig. 8a) delivers twice the number of requests than the first and the third. Simulating complicated stress tests requires more measurements to be taken. In the *mixed workload scenario* we therefore considered two services sharing the available resources and affecting each others performance. In this setting, each service is targeted by a thread group generating a certain load for the service.

To provide a baseline for the resource consumption of the model, we tuned the model by stressing each service in isolation. After that, it is possible to simulate mixed workloads. To generate a group of clients that reproduces a certain load in the model, we needed to find the balance between the number and the cost of the requests sent at any step. Here we decide also on the granularity of the model: a large set of requests in the real system will be simulated in the model with few costly ones, as done with batch processing. This will keep the granularity of the simulations coarse, instead of fine-grained with many cheap requests, and will allow us to run big workloads in the model in a short amount of time.

The duration of a time interval in the model is decided during the model calibration, where the size of the waves in the requests determines the length of the stress test on the cluster, and the granularity of the model the number of time intervals of the simulation.

Experiments. We set up two experiments with a time interval corresponding to 2 seconds.

Experiment 1. The purpose of Experiment 1 is to check the precision of the modeling framework. We do this by running a single service stress test 10 minutes long, in order to measure the model's ability to reproduce Kubernetes autoscaling while the service is processing requests and then compare the load experienced on the cluster with the one seen in the simulation. In more detail, the cluster setting was one service deployed and three nodes available with 4000 millicores of CPU capacity each.

We start with one pod requiring 1000 millicores of CPU and limited to 2500. The autoscaling threshold was set to 80% of the required CPU busy and the downscale time was 300 seconds of inactivity. The load on the cluster has been generated with Jmeter:[8] a group of 50 threads send requests with a given timing for a three minutes, flooding the system with a wave. Then for the next three minutes they generate requests at twice the speed (the second wave) before sending the final requests again at their initial speed (see Fig. 8a). This stress test has been replicated in the model by bulking clients up to reach the first wave load, then twice that number of clients has been used to flood during the second wave, and finally return to the initial amount of request in the third wave (see Fig. 8c), to obtain the same total load in the simulation and real Kubernetes

[8] `https://jmeter.apache.org/`

deployment. The ABS code emulating the clients can be found in the repository of the simulator along with the Jmeter stress test descriptor.[9]

Experiment 2. The purpose of Experiment 2 is to test the prediction ability of the modeling framework. Namely, given the same load of requests of two services to the model and the real system, can we predict the scaling behavior of the real system? We run a second stress test 13 minutes long in a scenario with the two services, where each service goes through variable load, so that we can simulate scheduling and autoscaling in a resource-intensive scenario under different configurations. The first service is the same as in Experiment 1 and configured similarly; the second service has a different profile of resource consumption, its pods require 1000 millicores of CPU and limited to 2500, but have an upscale threshold of 95% and a downscale time of 300 seconds. The stress test load can be divided into four phases and is different for both services: one uses the same thread group as the first simulation, but inverting the load of the three waves: it starts with a high load of requests, then reduces the load to half and finally increases it again to the double. The load on the second service has been calibrated separately, it starts low, turns high, then drops down again before finishing with a demand that is much higher than the previous high traffic wave.

4.2 Results of the Experiments

Experiment 1. Figure 8 reports on the results of the first experiment. The load in the real system, shown in Fig. 8a, triggers the scaling of pods, shown in Fig. 8b. The approximated load in the model, shown in Fig. 8c, triggers the scaling of pods, shown in Fig. 8d. The graphs suggest that the model is properly calibrated and can reproduce the scalability scenario with reasonable accuracy.

Experiment 2. The results of the second experiment are shown in Figs. 10, 11 and 12. In this case we first reproduced the load in the Kubernetes model, based on the measurements of the separate loads of the two services, then we executed the model with the two services, potentially affecting each others performance, and therefore affecting also the scalability of the services. Figure 10 shows the result of this calibration.

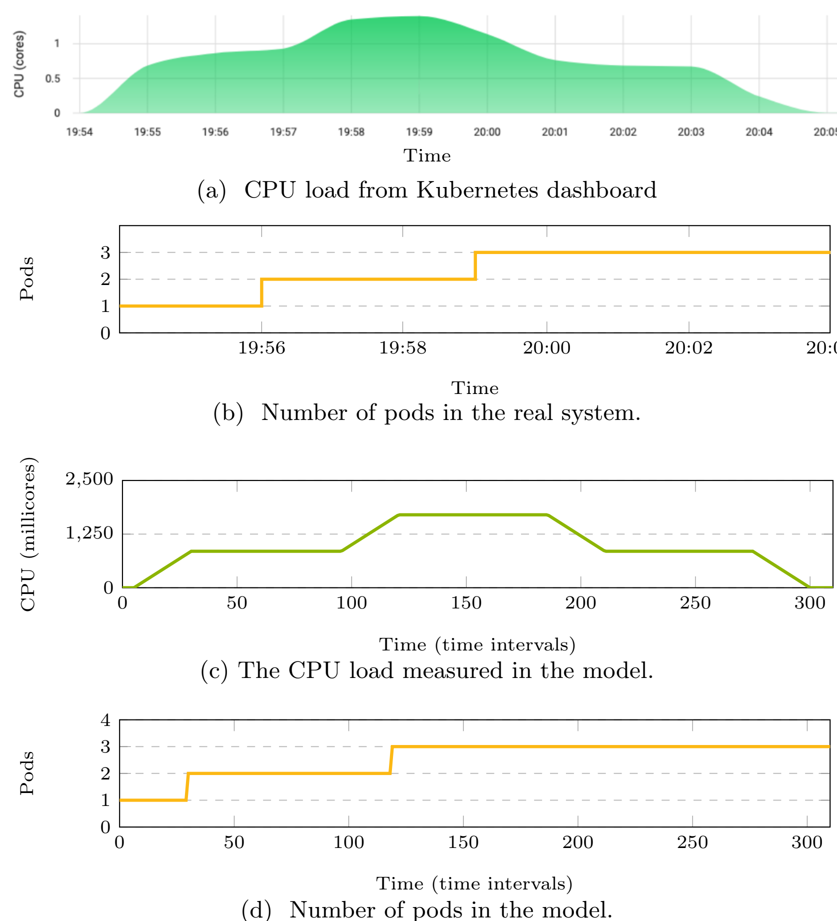
We then tested *different* configurations of the model before doing the corresponding runs on the real cluster. This carried two main benefits: it lowered the resources needed to test several configurations, and it emulated a real time interval with few seconds of computation time.

We considered two different configurations, changing the upscale threshold for the services. In the first configuration, shown in Fig. 11, we obtained similar graphs for the model and for the

	Service 1	**Service 2**
Calibration	80%	95%
Configuration 1	95%	80%
Configuration 2	95%	95%

Fig. 9: Upscale thresholds for Experiment 2.

[9] `https://github.com/giaku/abs-k8s-model`



(a) CPU load from Kubernetes dashboard

(b) Number of pods in the real system.

(c) The CPU load measured in the model.

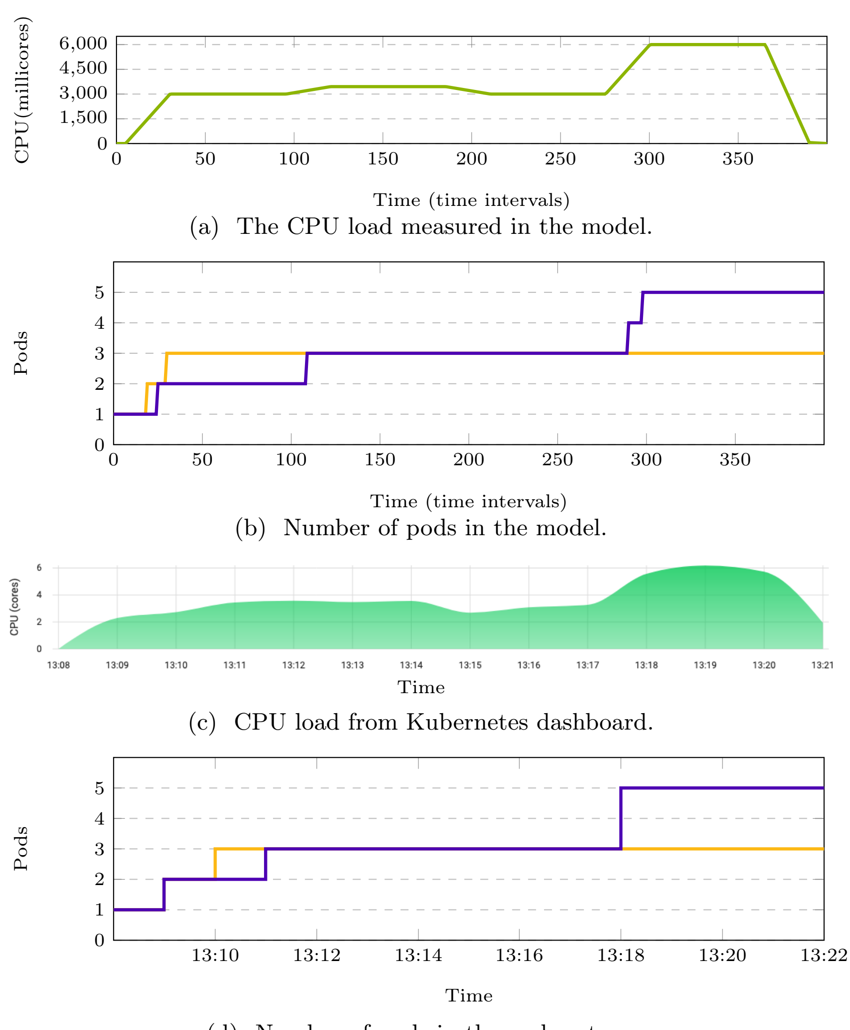
(d) Number of pods in the model.

Fig. 8: The results of the first experiment.

real system. The yellow line represents the number of pods for the first service, which stayed beneath 2 as a result of having a 95% scaling threshold, the blue line represents the second service, which raised up to six with an upscale threshold of 80%. In the second configuration, shown in Fig. 12, we tested 95% as the threshold for the second service as well, its number of pod grew at most at 5 (blue line) both in the simulator and on the real cluster. Figure 9 summarizes the different configurations.

5 Related Work

Cloud-based models in ABS. Whereas there are many cloud modeling languages (see, e.g., [5]), this paper is part of a line of work on formal modeling of virtualized systems in ABS. The perspective on virtualized systems we have taken, is to focus



(a) The CPU load measured in the model.

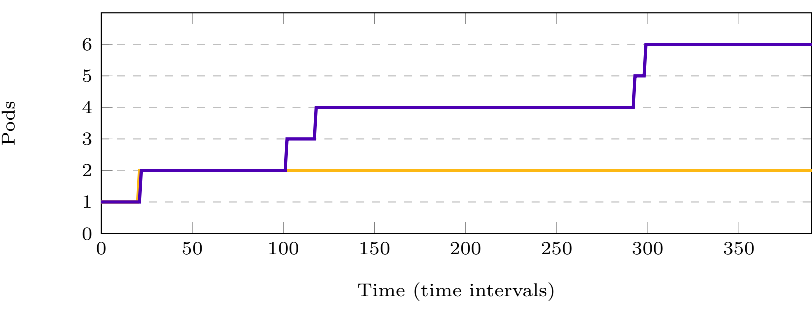
(b) Number of pods in the model.

(c) CPU load from Kubernetes dashboard.

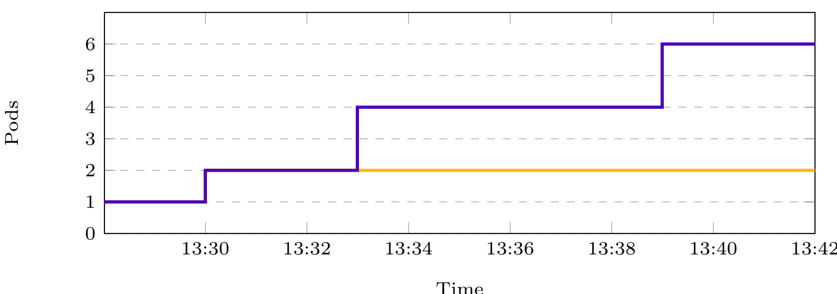
(d) Number of pods in the real system.

Fig. 10: Calibration for the mixed workload scenario of the second experiment.

on resource provisioning and quality-of-service, which typically affects the timing behavior of systems on the cloud. The underlying technical idea is to introduce a separation of concerns between resource-needs for different computational tasks, and resource-provisioning in the infrastructure [20–22]. This approach has been successfully applied to different kinds of virtualization infrastructure, including Amazon AWS [19], Hadoop YARN [25] and Hadoop Spark Streaming [24]. The concurrency model of ABS, based on actors, has also been used for verification to industrial case studies in a DevOps setting [1] and for parallel cost analysis [2], a novel static analysis method related to parallelism and maximal span. The formal



(a) Required number of pods as predicted in the model.

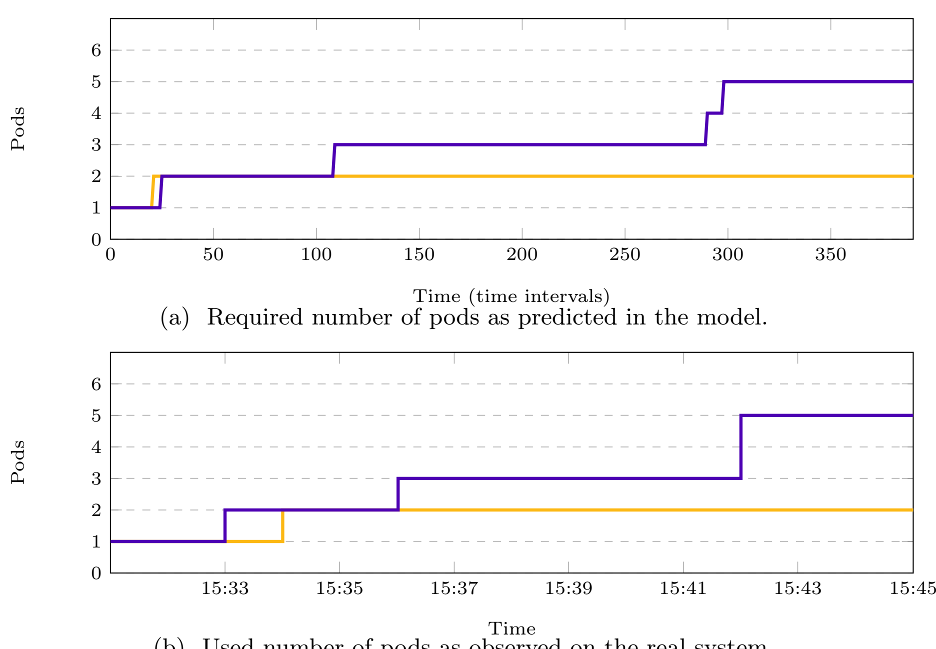


(b) Used number of pods as observed on the real system.

Fig. 11: Results for the first configuration, comparing the predicted need for pods in the model the observed use of pods on the real cluster.

model of Kubernetes presented in this paper differs from previous work in its *nested* virtualization; i.e., the containerization of microservices lead to two levels of book-keeping in the resource-sensitive architecture, corresponding to the pods ond nodes of the Kubernetes framework. Furthermore, the notion of indirection due to the service-concept and the auto-scaling groups add complexity compared to previous work.

Optimization of microservice management. It has been shown that deployment management can be formalized as finite state machines, such as the Aeolus [13] and TOSCA-compliant deployment models [10], which can be adapted to formally reason about the static deployment of microservices; i.e., to express component resilience and static links between components. For example, the static deployment of microservices can be encoded as a constraint problem [9]. This work, which is based on Aeolus, takes an ABS model as its starting point. In contrast to our work, the authors are not restricted to modelling and simulation but are able to decide on optimal deployment. However, the optimization can only handle very limited forms of reconfiguration, and does not address dynamic scaling as modelled in our work.



(a) Required number of pods as predicted in the model.

(b) Used number of pods as observed on the real system.

Fig. 12: Results for the second configuration, comparing the predicted need for pods in the model the observed use of pods on the real cluster.

Testing environments for Cloud-based services. In order to perform tests on the real Kubernetes platform, we looked for a tool that allowed to simulate multiple requests in parallel to the service and to simulate behavior that varies over time. After a brief investigation about which tools are available on the market (eg: Apache JMeter[10], Locust[11], Tsung[12], etc.) we decided to use Apache JMeter since it is an open-source tool, it is multiplatform, multiprotocol, it comes with a simple GUI for configuration and to run the simulation from a shell, it presents the simulation results in textual or graphical format. Apache JMeter is used both by companies and in the scientific field to emulate traffic to network services [8, 12].

The most significant KPIs that we are looking for in order to evaluate the service performance are the response time over time (which gives us an indication about the quality of the offered service) and the number of requests per second (to have an evaluation about the load our service undergoes).

[10] https://jmeter.apache.org/
[11] https://locust.io/
[12] http://tsung.erlang-projects.org/

6 Conclusion and Future Work

In this paper, we present a formal model of resource consumption and scaling for containerized microservices deployed and managed by Kubernetes. The model focuses on how the deployment of such systems can behave under various configurations to be explored at design time and abstract from other aspects of Kubernetes such self-healing, rollouts, rollbacks, and storage orchestration. This preliminary model and results contribute towards the development of a modeling framework which can help developers in finding a deployment strategy for a microservice-based system which meets the system's performance requirements. The model is implemented in Real-Time ABS, it can be configured with different client workloads and different microservices running in parallel and affecting each others performance. The presented model can be used to explore different configurations for loosely coupled microservices at design time.

In future work, we plan to extend the model with aspects related to resiliency and reconfiguration of distributed and decoupled system, adding possible failures, volumes and stateful Kubernetes components. In particular, we plan to use the resulting model to assess quality-of-service aspects of different configuration choices by, e.g., predicting their response time and resource consumption. We also plan to trace data movement within the cluster and predict how this may affect the performance. We plan to validate such models using workloads collected from hours, weeks or months of running real systems. We further plan to investigate how such resiliency and reconfiguration can affect data access times and patterns.

References

1. Albert, E., de Boer, F.S., Hähnle, R., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L., Wong, P.Y.H.: Formal modeling of resource management for cloud architectures: An industrial case study using Real-Time ABS. Journal of Service-Oriented Computing and Applications **8**(4), 323–339 (Dec 2014). https://doi.org/10.1007/s11761-013-0148-0
2. Albert, E., Correas, J., Johnsen, E.B., Pun, K.I, Román-Díez, G.: Parallel cost analysis. ACM Trans. Comput. Logic **19**(4), 31:1–31:37 (Nov 2018), http://doi.acm.org/10.1145/3274278
3. Aldinucci, M., Rabellino, S., Pironti, M., Spiga, F., Viviani, P., Drocco, M., Guerzoni, M., Boella, G., Mellia, M., Margara, P., Drago, I., Marturano, R., Marchetto, G., Piccolo, E., Bagnasco, S., Lusso, S., Vallero, S., Attardi, G., Barchiesi, A., Colla, A., Galeazzi, F.: HPC4AI: an AI-on-demand federated platform endeavour. In: Proc. 15th International Conference on Computing Frontiers (CF 2018). pp. 279–286. ACM (2018). https://doi.org/10.1145/3203217.3205340
4. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables DevOps: Migration to a cloud-native architecture. IEEE Software **33**(3), 42–52 (2016). https://doi.org/10.1109/MS.2016.64
5. Bergmayr, A., Breitenbücher, U., Ferry, N., Rossini, A., Solberg, A., Wimmer, M., Kappel, G., Leymann, F.: A systematic review of cloud modeling languages. ACM Comput. Surv. **51**(1), 22:1–22:38 (2018), https://doi.org/10.1145/3150227

6. Bjørk, J., de Boer, F.S., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: User-defined schedulers for real-time concurrent objects. Innovations in Systems and Software Engineering **9**(1), 29–43 (2013), http://dx.doi.org/10.1007/s11334-012-0184-5
7. de Boer, F.S., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C.C., Johnsen, E.B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., Yang, A.M.: A survey of active object languages. ACM Comput. Surv. **50**(5), 76:1–76:39 (Oct 2017). https://doi.org/10.1145/3122848
8. Brady, J.F., Gunther, N.J.: How to emulate web traffic using standard load testing tools. CoRR **abs/1607.05356** (2016), http://arxiv.org/abs/1607.05356
9. Bravetti, M., Giallorenzo, S., Mauro, J., Talevi, I., Zavattaro, G.: Optimal and automated deployment for microservices. In: Hähnle, R., van der Aalst, W. (eds.) Proc. 22nd International Conference on Fundamental Approaches to Software Engineering (FASE 2019). Lecture Notes in Computer Science, vol. 11424, pp. 351–368. Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_21
10. Brogi, A., Canciani, A., Soldani, J.: Modelling and analysing cloud application management. In: Dustdar, S., Leymann, F., Villari, M. (eds.) Proc. 4th European Conference on Service Oriented and Cloud Computing (ESOCC 2015). Lecture Notes in Computer Science, vol. 9306, pp. 19–33. Springer (2015). https://doi.org/10.1007/978-3-319-24072-5_2
11. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, Omega, and Kubernetes. Queue **14**(1), 70–93 (Jan 2016). https://doi.org/10.1145/2898442.2898444
12. Curiel, M., Pont, A.: Workload generators for web-based systems: Characteristics, current status, and challenges. IEEE Communications Surveys Tutorials **20**(2), 1526–1546 (2018). https://doi.org/10.1109/COMST.2018.2798641
13. Di Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Aeolus: A component model for the cloud. Inf. Comput. **239**, 100–121 (2014), https://doi.org/10.1016/j.ic.2014.11.002
14. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall (2005)
15. Fazio, M., Celesti, A., Ranjan, R., Liu, C., Chen, L., Villari, M.: Open issues in scheduling microservices in the cloud. IEEE Cloud Computing **3**(5), 81–88 (2016). https://doi.org/10.1109/MCC.2016.112
16. Hightower, K., Burns, B., Beda, J.: Kubernetes: Up and Running Dive into the Future of Infrastructure. O'Reilly (2017)
17. Huhns, M.N., Singh, M.P.: Service-oriented computing: Key concepts and principles. IEEE Internet Computing **9**(1), 75–81 (Jan 2005). https://doi.org/10.1109/MIC.2005.21
18. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B., de Boer, F.S., Bonsangue, M.M. (eds.) Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010). Lecture Notes in Computer Science, vol. 6957, pp. 142–164. Springer (2011). https://doi.org/https://doi.org/10.1007/978-3-642-25271-6_8
19. Johnsen, E.B., Lin, J.-C., Yu, I.C.: Comparing AWS deployments using model-based predictions. In: Margaria, T., Steffen, B. (eds.) Proc. 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: (ISoLA 2016). Lecture Notes in Computer Science, vol. 9953, pp. 482–496. Springer (2016). https://doi.org/https://doi.org/10.1007/978-3-319-47169-3_39

20. Johnsen, E.B., Pun, K.I, Tapia Tarifa, S.L.: Modeling deployment decisions for elastic services with ABS. In: Behjati, R., Elmokashfi, A. (eds.) Proc. First International Workshop on Formal Methods for and on the Cloud. Electronic Proceedings in Theoretical Computer Science, vol. 228, pp. 16–26. Open Publishing Association (2016). https://doi.org/10.4204/EPTCS.228.3
21. Johnsen, E.B., Pun, K.I, Tapia Tarifa, S.L.: A formal model of cloud-deployed software and its application to workflow processing. In: Begusic, D., Rozic, N., Radic, J., Saric, M. (eds.) Proc. 25th International Conference on Software, Telecommunications and Computer Networks (SoftCOM 2017). pp. 1–6. IEEE (2017). https://doi.org/10.23919/SOFTCOM.2017.8115501
22. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Integrating deployment architectures and resource consumption in timed object-oriented models. Journal of Logical and Algebraic Methods in Programming **84**(1), 67–91 (2015). https://doi.org/10.1016/j.jlamp.2014.07.001
23. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer **1**(1–2), 134–152 (1997). https://doi.org/https://doi.org/10.1007/s100090050010
24. Lin, J.-C., Lee, M.-C., Yu, I.C., Johnsen, E.B.: A configurable and executable model of spark streaming on apache YARN. IJGUC **11**(2), 185–195 (2020). https://doi.org/10.1504/IJGUC.2020.10026548
25. Lin, J.-C., Yu, I.C., Johnsen, E.B., Lee, M.-C.: ABS-YARN: A formal framework for modeling hadoop YARN clusters. In: Stevens, P., Wasowski, A. (eds.) Proc. 19th International Conference on Fundamental Approaches to Software Engineering (FASE 2016). Lecture Notes in Computer Science, vol. 9633, pp. 49–65. Springer (2016). https://doi.org/10.1007/978-3-662-49665-7_4
26. Newman, S.: Building microservices - designing fine-grained systems, 1st Edition. O'Reilly (2015), `http://www.worldcat.org/oclc/904463848`
27. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of real-time maude. High. Order Symb. Comput. **20**(1-2), 161–196 (2007). https://doi.org/10.1007/s10990-007-9001-5
28. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: European Conference on Object-Oriented Programming (ECOOP 2010). Lecture Notes in Computer Science, vol. 6183, pp. 275–299. Springer (2010). https://doi.org/10.1007/978-3-642-14107-2_13