# Modeling Resource-Aware Virtualized Applications for the Cloud in Real-Time ABS [⋆]

Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa

Department of Informatics, University of Oslo, Norway
{einarj,rudi,sltarifa}@ifi.uio.no

**Abstract.** An application's quality of service (QoS) depends on resource availability; e.g., response time is worse on a slow machine. On the cloud, a virtualized application leases resources which are made available on demand. When its work load increases, the application must decide whether to reduce QoS or increase cost. Virtualized applications need to manage their acquisition of resources. In this paper resource provisioning is integrated in high-level models of virtualized applications. We develop a Real-Time ABS model of a cloud provider which leases virtual machines to an application on demand. A case study of the Montage system then demonstrates how to use such a model to compare resource management strategies for virtualized software during software design. Real-Time ABS is a timed abstract behavioral specification language targeting distributed object-oriented systems, in which dynamic deployment scenarios can be expressed in executable models.

## 1 Introduction

The added value and compelling business drivers of cloud computing are undeniable [10], but considerable new challenges need to be addressed for industry to make an effective usage of cloud computing. As the key technology enabler for cloud computing, *virtualization* makes elastic amounts of resources available to application-level services deployed on the cloud; for example, the processing capacity allocated to a service may be changed on the demand. The integration of virtualization in general purpose software applications requires novel techniques for leveraging resources and resource management into software engineering. Virtualization poses challenges for the software-as-a-service abstraction concerning the development, analysis, and dynamic composition of software with respect to quality of service. Today these challenges are not satisfactorily addressed in software engineering. In particular, better support for the modeling and validation of application-level resource management strategies for virtualized resources are needed to help the software developer make efficient use of the available virtualized resources in their applications.

[⋆] Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu).

The abstract behavioral specification language ABS is a formalism which aims at describing systems at a level which abstracts from many implementation details but captures essential behavioral aspects of the targeted systems [25]. ABS targets the engineering of concurrent, component-based systems by means of executable object-oriented models which are easy to understand for the software developer and allow rapid prototyping and analysis. The extension Real-Time ABS integrates object orientation and timed behavior [8]. Whereas the functional correctness of a planned system largely depends on its high-level behavioral specification, the choice of deployment architecture may hugely influence the system's quality of service. For example, CPU limitations may restrict the applications that can be supported on a cell phone, and the capacity of a server may influence the response time of a service during peaks in the user traffic.

Whereas software components reflect the logical architecture of systems, *deployment components* have recently been proposed for Real-Time ABS to reflect the deployment architecture of systems [27, 28]. A deployment component is a resource-restricted execution context for a set of concurrent object groups, which controls how much computation can occur in this set between observable points in time. Deployment components may be dynamically created and are parametric in the amount of resources they provide to their objects. This explicit representation of deployment scenarios allows application-level response time and load balancing to be expressed in the software models in a very natural and flexible way, relative to the resources allocated to the software.

This paper shows how deployment components in Real-Time ABS may be used to model virtualized systems in a cloud environment. We develop a Real-Time ABS model of cloud provisioning and accounting for resource-aware applications: an abstract cloud provider offers virtual machines with given CPU capacities to client applications and bills the applications for their resource usage. We use this model in a case study of the Montage system [24], a cloud-based resource-aware application for scientific computing, and compare execution times and accumulated costs depending on the number of leased machines by means of simulations of the executable model. We show that our results are comparable to those previously obtained for Montage with the same deployment scenarios on specialized simulation tools [19] and thus that our formal model can be used to estimate cloud deployment costs for realistic systems. We then introduce dynamic resource management strategies in the Montage model, and show that these improve on the resource management strategies previously considered [19].

The paper is structured as follows. Section 2 presents the abstract behavioral specification language Real-Time ABS, Section 3 develops our model of cloud provisioning. Section 4 presents the case study of the Montage system. Section 5 discusses related work and Section 6 concludes the paper.

## 2 Abstract Behavioral Specification with Real-Time ABS

ABS is an executable object-oriented modeling language with a formal semantics [25], which targets distributed systems. The language is based on concurrent object groups, akin to concurrent objects (e.g., [14,17,26]), Actors (e.g., [1,23]), and Erlang processes [5]. Concurrent object groups in ABS internally support interleaved concurrency using guarded commands. This allows active and reactive behavior to be easily combined, based on cooperative scheduling of processes which stem from method calls. A concurrent object group has at most one active process at any time and a queue of suspended processes waiting to execute on an object in the group. Objects in ABS are dynamically created from classes but typed by interface; i.e., there is no explicit notion of hiding as the object state is always encapsulated behind interfaces which offer methods to the environment.

## 2.1 Modeling Timed Behavior in ABS

ABS combines functional and imperative programming styles with a Java-like syntax [25]. Concurrent object groups execute in parallel and communicate through asynchronous method calls. Data manipulation inside methods is modeled using a simple functional language. Thus, the modeler may abstract from the details of low-level imperative implementations of data structures, and still maintain an overall object-oriented design which is close to the target system.

The *functional* part of ABS allows user-defined algebraic data types such as the empty type `Unit`, Booleans `Bool`, integers `Int`; parametric data types such as sets `Set<A>` and maps `Map<A>` (given a value for the variable `A`); and user-defined functions over values of these types, with support for pattern matching.

The *imperative* part of ABS addresses concurrency, communication, and synchronization at the concurrent object level, and defines interfaces, classes, and methods. ABS objects are *active* in the sense that their `run` method, if defined, gets called upon creation. *Statements* for sequential composition $s_1; s_2$, assignment `x=rhs`, **`skip`**, **`if`**, **`while`**, and **`return`** are standard. The statement **`suspend`** unconditionally suspends the active process of an object by moving this process to the queue, from which an enabled process is selected for execution. In **`await`** $g$, the guard $g$ controls suspension of the active process and consists of Boolean conditions $b$ and return tests $x?$ (see below). Functional expressions $e$ and guards $g$ are side-effect free. If $g$ evaluates to false, the active process is suspended, i.e., moved to the queue, and some process from the queue may execute. *Expressions* `rhs` include the creation of an object group **`new cog`** `C(e)`, object creation in the creator's group **`new`** `C(e)`, method calls `o!m(e)` and `o.m(e)`, future dereferencing `x.`**`get`**, and functional expressions `e`.

*Communication* and *synchronization* are decoupled in ABS, which allows complex workflows to be modeled. Communication is based on asynchronous method calls, denoted by assignments `f=o!m(e)` where `f` is a future variable, `o` an object expression, and `e` are (data value or object) expressions. After calling `f=o!m(e)`, the future variable `f` refers to the return value of the call and the caller may proceed with its execution *without blocking* on the method reply. There are two operations on future variables, which control synchronization in ABS. First, the statement **`await`** `f?` *suspends the active process* unless a return value from the call associated with `f` has arrived, allowing other processes in the object group to execute. Second, the return value is retrieved by the expression `f.get`, which *blocks all execution in the object* until the return value is available. The statement sequence `x=o!m(e);v=x.get` encodes commonly used *blocking calls*, abbreviated `v=o.m(e)` (reminiscent of synchronous calls).

We work with Real-Time ABS [8], a timed extension of ABS with a run-to-completion semantics, which combines *explicit* and *implicit* time for ABS models. Real-Time ABS has an interpreter defined in rewriting logic [30] which closely reflects its semantics and which executes on the Maude platform [16]. In Real-Time ABS, explicit time is specified directly in terms of durations (as in, e.g., UPPAAL [29]). Real-Time ABS provides the statement **`duration`**`(b,w)` to specify a duration between the worst-case `w` and the best case `b`. A process may also suspend for a certain duration, expressed by **`await duration`**`(b,w)`. For the purposes of this paper, it is sufficient to work with a discrete time domain, and let `b` and `w` be of type `Int`. In contrast to explicit time, implicit time is *observed* by measurements of the executing model. Measurements are obtained by comparing clock values from a global clock, which can be read by an expression **`now`**`()` of type `Time`. With implicit time, no assumptions about execution times are hard-coded into the models. The execution time of a method call depends on how quickly the call is effectuated by the server object. In fact, the execution time of a statement varies with the *capacity* of the chosen deployment architecture and on *synchronization* with other (slower) objects.

## 2.2 Modeling Deployment Architectures in Real-Time ABS

*Deployment components* in Real-Time ABS abstractly capture the resource capacity at a location [27, 28]. Deployment components are first-class citizens in Real-Time ABS and share their resources between their allocated objects. The root object of a model is allocated to the deployment component `environment`, which has unlimited resources. Deployment components with different resource capacities may be dynamically created depending on the control flow of the model or statically created in the main block of the model. When created, objects are by default allocated to the same deployment component as their creator, but they may also be explicitly allocated to a different component by an annotation.

Deployment components have the type `DC` and are instances of the class `DeploymentComponent`. This class takes as parameters a name (the name of the location, mostly used for monitoring purposes), given as a string, and a set of restrictions on resources. Here we focus on resources reflecting the components' *CPU processing* capacity, which are specified by the constructor `CPUCapacity(r)`, where `r` of type `Resource` represents the amount of available abstract processing resources between observable points in time. The expression **`thisDC`**`()` evaluates to the deployment component of the current object. The method `total("CPU")` of a deployment component returns the total amount of CPU resources allocated to that component.

The *CPU processing capacity* of a deployment component determines how much computation may occur in the objects allocated to that component. The CPU resources of a component define its capacity between observable (discrete) points in time, after which the resources are renewed. Objects allocated to the component compete for the shared resources in order to execute. With the run-to-completion semantics, the objects may execute until the component runs out of resources or they are otherwise blocked, after which time will advance [28].

The *cost* of executing statements is given by a cost model. A default cost value for statements can be set as a compiler option (e.g., `defaultcost=10`). This default cost does not discriminate between different statements. For some statements a more precise cost expression is desirable in a realistic model; e.g., if `e` is a complex expression, then the statement `x=e` should have a significantly higher cost than the statement **`skip`**. For this reason, more fine-grained costs can be introduced into the models by means of annotations, as follows:

```
class C implements I {
  Int m (T x) { [Cost: g(size(x))] return f(x); }
}
```

It is the responsibility of the modeler to specify an appropriate cost model. A behavioral model with default costs may be gradually refined to obtain more realistic resource-sensitive behavior. To provide cost functions such as `g` in our example above, the modeler may be assisted by the COSTABS tool [2], which computes a worst-case approximation of the cost for `f` in terms of the size of the input value `x` based on static analysis techniques, when given the definition of the expression `f`. However, the modeler may also want to capture resource consumption at a more abstract level during the early stages of system design, for example to make resource limitations explicit before further behavioral refinements of a model. Therefore, cost annotations may be used to abstractly represent the cost of some computation which remains to be fully specified.

## 3 Resource Management and Cloud Provisioning

An explicit model of cloud provisioning allows the application developer to interact in a simple way with a provisioning and accounting system for virtual machines. This section explains how such cloud provisioning may be modeled, for Infrastructure-as-a-Service [10] cloud environments. Consider an interface `CloudProvider` which offers three methods for resource management to client applications: `createMachine`, `acquireMachine`, and `releaseMachine`.

The method `createMachine` prepares and returns an abstract virtual machine with a specified processing capacity, after which the client application may deploy objects on the machine. This method models the provisioning and configuration part of a cloud-based application, and corresponds roughly to instancing and configuring a virtual machine on a cloud, without starting up the machine.

Before running a computation on a machine created with `createMachine`, the client application must first call the method `acquireMachine`. The cloud provider then starts *accounting* for the time this machine is kept running; the client calls the method `releaseMachine` to “shut down” the machine again. (For simplicity it is currently not checked whether processes are run before calling `acquireMachine` or after `releaseMachine`; this is a straightforward extension of the approach which could be useful to model “cheating” clients.)

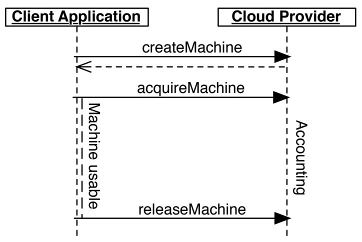

**Fig. 1.** Interaction between a client application and the cloud provider.

For a later reactivation of the same machine, only `acquireMachine` needs to be called. Fig. 1 shows one such sequence of interactions between a client application and a cloud provider.

In addition, the interface offers a method `getAccumulatedCost` which returns the cost accumulated so far by the client application. This method can be used in load balancing schemes to implement various trade-offs between quality of service and the cost of running the application, or to implement operator alerts when certain QoS or cost budgets are bypassed.

*A Model of Cloud Provisioning in Real-Time ABS.* A class which implements the `CloudProvider` interface is given in Fig. 2. Abstract virtual machines are modeled as deployment components. The class has two formal parameters to allow easy configuration: `startupTime` sets the length of the startup procedure for virtual machines and `accountingPeriod` sets the length of each accounting period. In addition, the class has four fields: `accumulatedCost` stores the cost incurred by the client application up to present time, the set `billableMachines` contains the machines to be billed in the current time interval, and the sets `availableMachines` and `runningMachines` contain the created but not currently running and the running machines, respectively. The empty set is denoted `EmptySet`. Let $s$ be a set over elements of type $T$ and let $e : T$. The following functions are defined in the functional part of Real-Time ABS: `insertElement`$(s, e)$ returns $\{e\} \cup s$, `remove`$(s, e)$ returns $s \setminus \{e\}$, and `take`$(s)$ returns some $e$ such that $e \in s$.

The methods for resource management move machines between these sets. Any machine which is either created or running within an accounting period, is billable in that period; i.e., a machine may be both acquired and released in a period, so there may be more billable than running machines. The method `createMachine` creates a new deployment component of the given capacity and adds it to `availableMachines`. The method `acquireMachine` moves a machine from `availableMachines` to `runningMachines`. Since the machine becomes billable, it is placed in `billableMachines`. The method suspends for the duration of the `startupTime` before it returns, so the accounting includes the startup time of the machine. The method `releaseMachine` moves

```
interface CloudProvider {
  DC   createMachine(Int capacity);
  Unit acquireMachine(DC machine);
  Unit releaseMachine(DC machine);
  Int  getAccumulatedCost();
}
class CloudProvider (Int startupTime, Int accountingPeriod)
    implements CloudProvider {
  Int accumulatedCost = 0; Set<DC> billableMachines = EmptySet;
  Set<DC> availableMachines = EmptySet;
  Set<DC> runningMachines = EmptySet;

  DC createMachine(Int r) {
    DC dc = new DeploymentComponent("", set[CPUCapacity(r)]);
    availableMachines = insertElement(availableMachines, dc);
    return dc;
  }
  Unit acquireMachine(DC dc) {
    billableMachines = insertElement(billableMachines, dc);
    availableMachines = remove(availableMachines, dc);
    runningMachines = insertElement(runningMachines, dc);
    await duration(startupTime, startupTime);
  }
  Unit releaseMachine(DC dc) {
    runningMachines = remove(runningMachines, dc);
    availableMachines = insertElement(availableMachines, dc);
  }
  Int getAccumulatedCost(){ return accumulatedCost; }
  Unit run() {
    while (True) {
      await duration(accountingPeriod, accountingPeriod);
      Set<DeploymentComponent> billables = billableMachines;
      while (~(billables == EmptySet)) {
        DeploymentComponent dc = take(billables);
        billables = remove(billables,dc); Int capacity = dc.total("CPU");
        accumulatedCost = accumulatedCost+(accountingPeriod*capacity);
      }
      billableMachines = runningMachines;
}}}
```

**Fig. 2.** The CloudProvider class in Real-Time ABS.

a machine from `runningMachines` to `availableMachines`. The machine remains billable for the current accounting period.

The `run` method of the cloud provider implements the accounting of incurred resource usage for the client application. The method suspends for the duration of the accounting period, after which all machines in `billableMachines` are billed by adding their resource capacity for the duration of the accounting period to `accumulatedCost`. Remark that Real-Time ABS has a run-to-completion semantics which guarantees that the loop in `run` will be executed after every accounting period. After accounting is finished, only the currently running machines are already billable for the next period. These are copied into `billableMachines` and the `run` method suspends for the next accounting period.

| Module | Description |
|---|---|
| **mImgtbl** | Extract geometry information from a set of FITS headers and create a metadata table from it. |
| **mOverlaps** | Analyze an image metadata table to determine which images overlap on the sky. |
| **mProject** | Reproject a FITS image. |
| **mProjExec** | Reproject a set of images, running *mProject* for each image. |
| **mDiff** | Perform a simple image difference between a pair of overlapping images. |
| **mDiffExec** | Run *mDiff* on all the overlap pairs identified by *mOverlaps*. |
| **mFitplane** | Fit a plane (excluding outlier pixels) to an image. Used on the difference images generated by *mDiff*. |
| **mFitExec** | Run *mFitplane* on all overlapping pairs. Creates a table of image-to-image difference parameters. |
| **mBgModel** | Modeling/fitting program which uses the image-to-image difference parameter table to interactively determine a set of corrections to apply to each image to achieve a "best" global fit. |
| **mBackground** | Remove a background from a single image |
| **mBgExec** | Run *mBackground* on all the images in the metadata table. |
| **mAdd** | Co-add the reprojected images to produce an output mosaic. |

**Fig. 3.** The modules of the Montage case study.

## 4 Case Study: The Montage Toolkit

Montage is a portable software toolkit for generating science-grade mosaics by composing multiple astronomical images [24]. Montage is modular and can be run on a researcher's desktop machine, in a grid, or on a cloud. Due to the high volume of data in a typical astronomical dataset and the high resolution of the resulting mosaic, as well as the highly parallelizable nature of the needed computations, Montage is a good candidate for cloud deployment. In [19], Deelman et al. present simulations of cloud deployments of Montage and the cost of creating mosaics with different deployment scenarios, using the specialized simulation tool GridSim [9].

This section describes the architecture of the Montage system and how it was modeled in Real-Time ABS. We explain how costs were associated to the different parts of the model. The results obtained by simulations of the model in the Real-Time ABS interpreter are compared to those obtained in the specialized simulator. Finally, more fine-grained dynamic resource management, not considered in the previous work [19], is proposed and compared to previous scenarios.

### 4.1 The Problem Description

Creating a mosaic from a set of input images involves a number of tasks: first reprojecting the images to a common projection, coordinating system and scale, then rectifying the background radiation in all images to a common flux scale

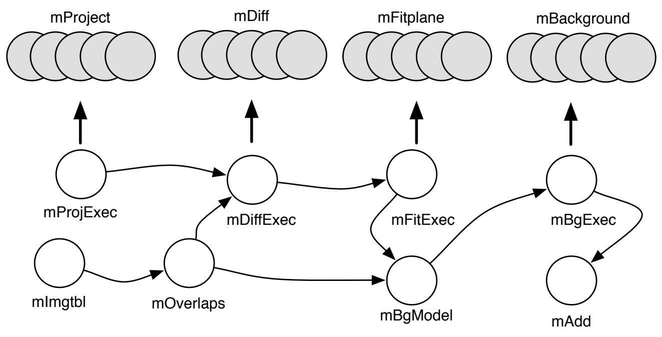

**Fig. 4.** Montage abstract workflow.

and background level, and finally co-adding the reprojected background-rectified images into a final mosaic. The tasks exchange data in the format FITS, which encapsulates image data and meta-data. These tasks are implemented by a number of Montage modules [24], which are listed and described in Fig. 3. These modules can be run individually or combined in a workflow, locally or remotely on a grid or a cloud. Fig. 4 depicts the dataflow dependencies between the modules in a typical Montage workflow [19]. These dependencies show which jobs can be parallelized on multiprocessor systems, grids, or cloud services.

Simulation results for running Montage on the *Amazon* cloud with the workflow depicted in Fig. 4 have been published in [19], including cost measurements for CPU and storage resources. The simulation tool GridSim [9] was used to study the trade-offs between cost and performance for different execution and resource provisioning scenarios when running Montage in a cloud service provider.

We model and analyze the same abstract workflow architecture of Montage based on the model of cloud provisioning presented in Section 3, as a means to validate the presented formal model of cloud provisioning in Real-Time ABS. In particular, we consider the case in which Montage processes multiple input images in parallel. Our model abstracts from the implementation details of the manipulation of images, replacing them with abstract statements and cost annotations. One important result of [19] is that computation cost dominates storage and data transfer cost for the Montage workload by 2-3 orders of magnitude, which allows us to focus on CPU usage alone.

## 4.2 A Model of the Montage Workflow in Real-Time ABS

*The Core Modules.* The Montage core modules that execute atomic tasks (i.e., `mProject`, `mDiff`, `mFitplane`, `mBgModel`, `mBackground`, `mAdd`, `mImgtbl`, and `mOverlaps`) are modeled as methods inside a class `CalcServer` which implements the `CalcServer` interface shown in Fig. 5. In the methods of this class, cost annotations are used to specify the costs of executing atomic tasks. The images considered in the case study have a constant size, so it is sufficient to use a constant cost for the atomic tasks. Lacking precise cost estimates for

```
interface CalcServer {
  DeploymentComponent getDC();
  MetadataT mImgtbl(List<FITS> i);
  MetadataT mOverlaps(MetadataT mt);
  FITS mProject(FITS image);
  FITSdf mDiff (FITS image1, FITS image2);
  FITSfit mFitplane (FITSdf df);
  CorrectionT mBgModel(Image2ImageT diffs, MetadataT ovlaps);
  FITS mBackground (Int correction,FITS image );
  FITS mAdd (List<FITS> images); }

class CalcServer implements CalcServer {
  ...
  FITS mBackground (Int correction,FITS image ){
    [Cost: 1] FITS result = correctFITS(image,correction);
    return result;
  }
... }
```

**Fig. 5.** `CalcServer` interface and class in Real-Time ABS.

the individual tasks, we consider an abstract cost model in which each atomic task is assigned the cost of 1 resource. (This cost model could be further refined; although some timing measurements are given in [24], these are not detailed enough for this purpose.) The code for one such atomic task inside the `CalcServer` class is shown in Fig. 5.

*Resource Management.* The workflow process does not interact with the different instances of `CalcServer` directly. Instead, tasks are sent to an instance of `ApplicationServer` which acts a broker for the preallocated machine instances and distributes tasks to free machines. The `ApplicationServer` interface, partly shown in Fig. 6, provides the workflow with means to start the parallelizable tasks (i.e., `mProjExec`, `mDiffExec`, `mFitExec` and `mBgExec`) and distributes the atomic tasks (e.g., `mDiff`) to instances of `CalcServer`. Atomic tasks are sent directly to one calculation server. Two fields `activeMachines` and `servers` keep track of the number of active jobs on each created machine and the order in which servers get jobs, respectively. Surrounding every call to a calculation server the auxiliary methods `getServer` and `dropServer` do the bookkeeping and resource management of the virtual machines. Asynchronous method calls to the future variables `fimage` and `fnewimages`, and task suspension are used to keep the application server responsive.

Our model defines algebraic data types `FITS`, `FITSdf`, `FITSfit`, as well as the list `MetadataT` and the maps `CorrectionT` and `Image2ImageT` to represent the input and output data at the different stages of the workflow; for example, `FITS` is a data type which represents image archives in FITS format, which is constructed from an abstract representation of metadata and of image data. This data can be used to keep track of data flow and abstractions of calculation results. The empty list and map are denoted `Nil` and `EmptyMap`. On lists, the constructor `Cons`$(h, t)$ takes as arguments an element $h$ and a list $t$;

```
interface ApplicationServer {
  FITS mAdd (List<FITS> images);
  List<FITS>  mProjExec(List<FITS> images);
  List<FITSdf> mDiffExec (MetadataT metatable, List<FITS> images);
  Image2ImageT mFitExec(List<FITSdf> dfs);
  List<FITS> mBgExec (CorrectionT corrections, List<FITS> images);
  ... }

class ApplicationServer(CloudProvider provider)
    implements ApplicationServer {
  List<CalcServer> servers = Nil; Map<DC,Int> activeMachines = EmptyMap;
  ...
  List<FITS> mBgExec(CorrectionT corrections,List<FITS> images) {
    List<FITS> newimages = Nil;
    if (isEmpty(images)==False) {
      FITS image = head(images);
      Int correction = lookupDefault(corrections,getId(image), 0);
      CalcServer b = this.getServer();
      Fut<FITS> fimage = b!mBackground (correction,image);
      Fut<List<FITS>> fnewimages=this!mBgExec(corrections,tail(images));
      await fimage?; FITS tmpimage = fimage.get;
      this.dropServer(b);
      await fnewimages?; List<FITS> newtmpimages = fnewimages.get;
      newimages = Cons(tmpimage, newtmpimages);}
    return newimages;}
... }
```

**Fig. 6.** The `ApplicationServer` interface and class (abridged).

`head`(`Cons`$(h,t)$) $= h$ and `tail`(`Cons`$(h,t)$) $= t$. The function `isEmpty`$(l)$ returns true if $l$ is the empty list. On maps, the function `lookupDefault`$(m,k,v)$ returns the value bound to $k$ in $m$ if the key $k$ is bound in $m$, and otherwise it returns the default value $v$.

## 4.3 Simulation Results

We simulated a workload equivalent to the *Montage 1* scenario described in [19]. As in that paper, the simulations were run on deployment scenarios ranging from 1 to 128 virtual machine instances, where all the machines were started up prior to the simulations (i.e., the `startupTime` parameter of the `CloudProvider` class in our model has value 0). Both simulation approaches exhibit the expected geometric downward progression of execution time when going from 1 to 128 machines, and roughly half an order of magnitude increase in cost. In our first simulation runs, the execution cost (measured in simulated machine-minutes) increased a little over two-fold over the full simulation range, versus closer to a six-fold increase ("60 cents [...] versus almost 4$") in [19]. To explain this difference, we theorized that the observed lower cost may have resulted from better machine allocation strategies in our model—the virtual machines were eagerly released by the `ApplicationServer` class when no more work was available to them, instead of being kept running until all computations finished.

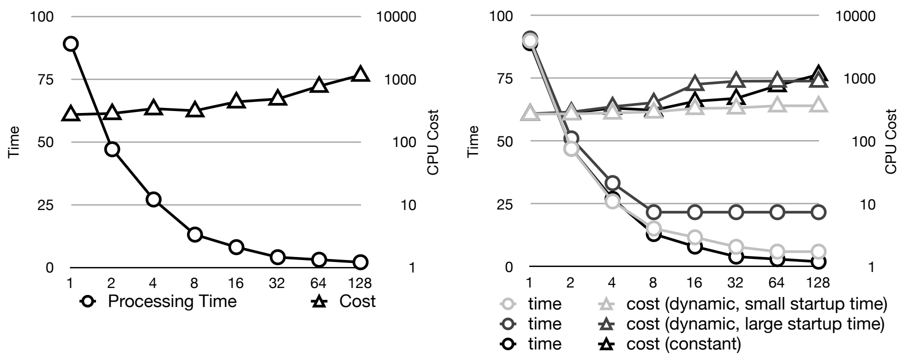

**Fig. 7.** Execution costs and times of simulation. The *Montage 1* scenario (left figure) is compared to dynamic resource management (right figure). The costs are presented on a logarithmic scale for easier comparison with the results of [19].

To test this hypothesis, the `ApplicationServer` class was modified to keep all instances running during the whole computation task. Using this allocation strategy, we observed a cost increase of 4.27 from 1 to 128 computation servers, which is more in line with the results obtained using GridSim. Fig. 7 (left) shows the simulation results of the modified model. The authors of [19] later confirmed in private communication that our hypothesis about the setup of the GridSim simulation scenario was indeed correct.

In order to further investigate the initial results involving dynamic startup and shutdown of machine instances, we refine our model by introducing startup times for virtual machines. Fig. 7 (right) compares the previous static deployment scenario (constant) with two dynamic resource management scenarios with varying startup times for virtual machines. One scenario models machine startup times of roughly one tenth of the time needed for performing a basic task, the other startup times roughly as large as basic task times. It can be seen that the cost of running a single job in the Montage system can be substantially reduced by switching off unused machines, given that the cost of starting machines is dominated by the actual calculations taking place, with almost no loss in time. On the other hand, if starting a machine is significantly slower than executing a basic task, it can be seen that both cost and time of the dynamic scenario are worse than when initially starting all machines in the static scenario of the considered workflow except in the case of severe over-provisioning of machines.

## 5 Related Work

The concurrency model of ABS is based on concurrent objects and Actor-based computation, in which software units with encapsulated processors communicate asynchronously (e.g., [1,5,14,23,26]). Their inherent compositionality allows concurrent objects to be naturally distributed on different locations, because only the local state of a concurrent object is needed to execute its methods. In previous work, the authors have introduced *deployment components* as a formal modeling concept to capture restricted resources shared between concurrent object groups and shown how components with parametric resources naturally model different deployment architectures [28], extended the approach with resource reallocation [27], and combined it with static cost analysis [4]. This paper complements our previous work by using deployment components to model cloud-based scenarios and the development of the Montage case study. A companion paper [18] further applies the approach of this paper to an industrial case study.

Techniques for prediction or analysis of non-functional properties are based on either *measurement* or *modeling*. Measurement-based approaches apply to existing implementations, using dedicated profiling or tracing tools like JMeter or LoadRunner. Model-based approaches allow abstraction from specific system intricacies, but depend on parameters provided by domain experts [20]. A survey of model-based performance analysis techniques is given in [7]. Formal systems using process algebra, Petri Nets, game theory, and timed automata have been used in the embedded software domain (e.g., [15, 21]). Real-Time ABS combines *explicit* time modeling with duration statements with *implicit* measurements of time already at the modeling level, which is made possible by the combination of costs in the application model and capacities in the deployment components.

Work on modeling object-oriented systems with resource constraints is more scarce. Using the UML SPT profile for schedulability, performance, and time, Petriu and Woodside [32] informally define the Core Scenario Model (CSM) to solve questions that arise in performance model building. CSM has a notion of resource context, which reflects an operation's set of resources. CSM aims to bridge the gap between UML and techniques to generate performance models [7]. Closer to our work is M. Verhoef's extension of VDM++ for embedded real-time systems [33], in which static architectures are explicitly modeled using CPUs and buses. The approach uses fixed resources targeting the embedded domain, namely processor cycles bound to the CPUs, while we consider more general resources for arbitrary software. Verhoef's approach is also based on abstract executable modeling, but the underlying object models and operational semantics differ. VDM++ has multi-thread concurrency, preemptive scheduling, and a strict separation of synchronous method calls and asynchronous signals, in contrast to our work with concurrent objects, cooperative scheduling, and caller-decided synchronization.

Related work on simulation tools for cloud computing are typically reminiscent of network simulators. A number of testing techniques and tools for cloud-based software systems are surveyed in [6]. In particular, CloudSim [13] and ICanCloud [31] are simulation tools using virtual machines to simulate cloud environments. CloudSim is a fairly mature tool which has already been used for a number of papers, but it is restricted to simulations on a single computer. In contrast, ICanCloud supports distribution on a cluster. Additionally CloudSim was originally based on GridSim [9], a toolkit for modeling and simulations of heterogeneous Grid resources. EMUSIM [12] is an integrated tool that uses AEF [11]

(Automated Emulation Framework) to estimate performance and costs for an application by means of emulations to produce improved input parameters for simulations in CloudSim. Compared to these approaches, our work is based on a formal semantics and aims to support the developer of software applications for cloud-based environments at an early phase in the development process.

Another interesting line of research is static cost analysis for object-oriented programs (e.g., [3,22]) Most tools for cost analysis only consider sequential programs, and assume that the program is fully developed before cost analysis can be applied. COSTABS [2] is a cost analysis tool for ABS which supports concurrent object-oriented programs. Our approach, in which the modeler specifies cost in cost annotations, could be supported by COSTABS to automatically derive cost annotations for the parts of a model that are fully implemented. In collaboration with Albert *et al.*, we have applied this approach for memory analysis of ABS models [4]. However, the full integration of COSTABS in our tool chain and the software development process remain future work.

## 6 Conclusion

This paper develops a model in Real-Time ABS of a cloud provider which offers virtual machines with given CPU capacities to a client application. Virtual machines are modeled as deployment components with given CPU capacities, and the cloud provider offers methods for resource management of virtual machines to client applications. The proposed model has been validated by means of a case study of the Montage toolkit, in which a typical Montage workflow was formalized. This formalization allows different user scenarios and deployment models to easily expressed and compared by means of simulations using the Real-Time ABS interpreter. The results from these simulations were comparable to those obtained for the Montage case study using specialized simulators, which suggests that models using abstract behavioral specification languages such as Real-Time ABS can be used to estimate cloud deployment costs for realistic systems.

Real-Time ABS aims to support the developer of client applications for cloud-based deployment, and in particular to facilitate the development of strategies for virtualized resource management at early stages in the development process. We are not aware of similar work addressing the formal modeling of virtualized resource management and cloud computing from the client application perspective. With the increasing focus on cloud-based deployment of general purpose software, such support could become very useful for software developers.

This paper focused on the formalization of cloud provisioning and simulations of the executable model. The presented work can be extended in a number of directions. In particular, we are interested in how to combine different virtualized resources in the same model to estimate combined costs of, e.g., computations, storage, bandwidth, and power consumption. Another extension is to strengthen the tool-based analysis support for Real-Time ABS. An integration with cost analysis tools such as COSTABS would assist the developer in providing cost annotations in the model. Furthermore, we plan to investigate symbolic execution techniques for Real-Time ABS, which would allow stronger automated analysis results than those considered here. Finally, an integration of QoS contracts with the interfaces of Real-Time ABS could form a basis for analysis abstract behavioral specifications with respect to service-level agreements.

**Acknowledgment.** We thank G. Bruce Berriman and Ewa Deelman for helping us with additional details of the Montage case study.

## References

1. G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems.* The MIT Press, Cambridge, Mass., 1986.
2. E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. COSTABS: a cost and termination analyzer for ABS. In *Proc. Workshop on Partial Evaluation and Program Manipulation (PEPM'12)*, pages 151–154. ACM, 2012.
3. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *Proc. ESOP'07*, *LNCS* 4421, pages 157–172. Springer, 2007.
4. E. Albert, S. Genaim, M. Gómez-Zamalloa, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Simulating concurrent behaviors with worst-case cost bounds. In *Proc. Formal Methods (FM'11)*, *LNCS* 6664, pages 353–368. Springer, June 2011.
5. J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.
6. X. Bai, M. Li, B. Chen, W.-T. Tsai, and J. Gao. Cloud testing tools. In *Proc. 6th Symposium on Service Oriented System Engineering*, pages 1–12. IEEE, 2011.
7. S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.
8. J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 2012. `http://dx.doi.org/10.1007/s11334-012-0184-5`
9. R. Buyya and M. Murshed. GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience*, 14:1175–1220, 2002.
10. R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.
11. R. N. Calheiros, R. Buyya, and C. A. F. De Rose. Building an automated and self-configurable emulation testbed for grid applications. *Software: Practice and Experience*, 40(5):405–429, Apr. 2010.
12. R. N. Calheiros, M. A. Netto, C. A. D. Rose, and R. Buyya. EMUSIM: an integrated emulation and simulation environment for modeling, evaluation, and validation of performance of cloud computing applications. *Software: Practice and Experience*, pages 00–00, 2012.
13. R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, and R. Buyya. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software, Practice and Experience*, 41(1):23–50, 2011.
14. D. Caromel and L. Henrio. *A Theory of Distributed Objects.* Springer, 2005.

15. A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Resource interfaces. In *Proc. EMSOFT'03*, *LNCS* 2855, pages 117–133. Springer, 2003.
16. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude*, *LNCS* 4350. Springer, 2007.
17. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Proc. ESOP'07*, *LNCS* 4421, pages 316–330. Springer, 2007.
18. F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, and P. Y. H. Wong. Formal Modeling of Resource Management for Cloud Architectures: An Industrial Case Study. In *Proc. European Conference on Service-Oriented and Cloud Computing (ESOCC)*, To appear in *LNCS*. Springer, Sep. 2012.
19. E. Deelman, G. Singh, M. Livny, G. B. Berriman, and J. Good. The cost of doing science on the cloud: The Montage example. In *Proc. High Performance Computing (SC'08)*, pages 1–12. IEEE/ACM, 2008.
20. I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by runtime parameter adaptation. In *Proc. ICSE'09*, pages 111–121. IEEE, 2009.
21. E. Fersman, P. Krcál, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.
22. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. POPL'09*, pages 127–139. ACM, 2009.
23. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.
24. J. C. Jacob, D. S. Katz, G. B. Berriman, J. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. A. Prince, and R. Williams. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *Intl. Journal of Computational Science and Engineering*, 4(2):73–87, 2009.
25. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proc. Symposium on Formal Methods for Components and Objects (FMCO)*, *LNCS* 6957, pages 142–164. Springer, 2011.
26. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
27. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Dynamic resource reallocation between deployment components. In *Proc. Intl. Conference on Formal Engineering Methods (ICFEM'10)*, *LNCS* 6447, pages 646–661. Springer, 2010.
28. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Validating timed models of deployment components with parametric concurrency. In *Formal Verification of Object-Oriented Software (FoVeOOS)*, *LNCS* 6528, pages 46–60. Springer, 2011.
29. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Intl. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.
30. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
31. A. Nuñez, J. Vázquez-Poletti, A. Caminero, G. Castañé, J. Carretero, and I. Llorente. iCanCloud: A flexible and scalable cloud infrastructure simulator. *Journal of Grid Computing*, 10:185–209, 2012.
32. D. B. Petriu and C. M. Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Software and System Modeling*, 6(2):163–184, 2007.
33. M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In *Proc. Formal Methods (FM'06)*, *LNCS* 4085, pages 147–162. Springer, 2006.