

Modeling Application-Level Management of Virtualized Resources in ABS ^{*}

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

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

Abstract. Virtualization motivates lifting aspects of low-level resource management to the abstraction level of modeling languages, in order to model and analyze virtualized resource usage for application-level services and its relationship to service-level QoS. In this paper we illustrate how the modeling language ABS may be used for this purpose by modeling a service deployed on the cloud. Virtual machines are provided on demand to the service, which distributes service requests between its available machines depending on its application-level load balancing scheme. The resulting ABS models are used to relate the accumulated usage cost for the virtual machines to the obtained QoS for the service.

ABS is an abstract behavioral specification language for designing executable models of distributed object-oriented systems. The language combines advanced concurrency and synchronization mechanisms based on concurrent object groups with a functional language for modeling data. ABS supports deployment variability by dynamically created deployment components which act as resource-restricted execution contexts for ABS objects, for example with respect to CPU resources. The use of these artefacts is demonstrated in this paper through an example of service-level management of virtualized resources on the cloud.

1 Introduction

The abstract behavioral specification language ABS is a formal modeling language which aims at describing systems on a level that abstracts from implementation details but captures essential behavioral aspects of the targeted systems [21]. 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 functional correctness of a targeted system largely depends on its high-level behavioral specification, independent of the platform on which the resulting code will be deployed. However, different deployment architectures may be envisaged for such a system, and the choice of deployment architecture may hugely influence the system's quality of service (QoS). For example, limitations in the processing

^{*} Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (<http://www.hats-project.eu>).

capacity of the CPU of a cell phone may restrict the applications that can be supported on the cell phone, and the capacity of a server may influence the response time for a service for peaks in the user traffic.

Recently, ABS has been extended with the concept of *deployment components* to capture the deployment architecture of target systems [24]. Whereas software components reflect the logical architecture of systems, deployment components reflect their deployment architecture. 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 they are parametric in the amount of resources they provide to their set of objects. This explicit representation of deployment architecture allows application-level response time and load balancing to be expressed in the system models in a very natural and flexible way, relative to the resources allocated to an application. This basic model of deployment architecture may be further extended by adding support for resource reallocation [23] or object mobility [25], allowing resources or objects to be moved from one deployment component to another.

The objective of this paper is to introduce and motivate the basic concepts developed to model deployment architecture in ABS in an informal way, rather than as a formal system. The formal syntax and semantics of ABS is discussed in [21] and the formalization of the integration of deployment components with ABS in [23–25]. In this paper we show how deployment components in ABS may be used to model virtualized systems, by developing an example inspired by cloud computing [8]. In this example, an abstract cloud provider offers virtual machines with given CPU capacities to client services, and bills the services based on an accounting scheme for their resource usage. The purpose of the model is not to make optimal use of hardware resources to provide these virtual machines (which would be the interest of the cloud provider) but rather to show how the developer may, at an early stage in the design of a service, gain insights into the resource needs of the service, and the trade-off between the cost of virtualized resources and the provided QoS for given customer scenarios.

The paper is structured as follows. Section 2 introduces timed ABS and shows how to model, e.g., response time in ABS models. Section 3 introduces the modeling concepts needed to capture deployment architecture and QoS in ABS. Section 4 introduces the case study developed in the paper and shows how the simulation tool for ABS can be used for rapid prototyping and analysis. Section 5 discusses related work and Section 6 concludes the paper.

2 Modeling Timed Behavior in ABS

ABS [21] is an executable object-oriented modeling language which targets distributed systems. The language is based on concurrent object groups, akin to concurrent objects (e.g., [9,12,22]), Actors (e.g., [1,19]), and Erlang processes [5]. A characteristic feature of concurrent object groups in ABS is that they internally support interleaved concurrency based on guarded commands. This makes

it very easy to combine active and reactive behavior in the concurrent object groups, based on cooperative scheduling of processes. 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. The processes stem from method activations. 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. For simplicity in this paper, we do not use other code structuring mechanisms. This section informally reviews the core ABS language and its timed extension (for further details, see [7, 21]).

2.1 Core ABS

ABS is a modeling language which combines functional and imperative programming styles to develop high-level executable models. Concurrent object groups execute in parallel and communicate through asynchronous method calls. To intuitively capture internal computation inside a method, we use a simple functional language based on user-defined algebraic data types and functions. 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. At a high level of abstraction, concurrent object groups typically consist of a single concurrent object; other objects may be introduced into a group as required to give some of the algebraic data structures an explicit imperative representation in the model. In this paper, we aim at high-level models and the groups will consist of single concurrent objects.

The *functional sublanguage* of ABS consists of a library of 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 type variable `A`), and (parametric) functions over values of these data types. For example, we can define polymorphic sets using a type variable `A` and two constructors `EmptySet` and `Insert`, and a function `contains` which checks whether an element `el` is in a set `ss` recursively by pattern matching over `ss`:

```

data Set<A> = EmptySet | Insert(A, Set<A>);

def Bool contains<A>(Set<A> ss, A el) =
  case ss {
    EmptySet      => False ;
    Insert(el, _) => True;
    Insert(_, xs) => contains(xs, el);
  };

```

Here, the cases `p => exp` are evaluated in the listed order, underscore works as a wild card in the pattern `p`, and variables in `p` are bound in the expression `exp`.

The *imperative sublanguage* of ABS addresses concurrency, communication, and synchronization at the concurrent object level in the system design, and defines interfaces and methods with a Java-like syntax. ABS objects are *active*; i.e., their `run` method, if defined, gets called upon creation. *Statements* are standard for sequential composition `s1; s2`, assignments `x = rhs`, and for the **skip**,

if, **while**, and **return** constructs. The statement **suspend** unconditionally suspends the execution of the active process of an object by adding this process to the queue, from which an enabled process is then selected for execution. In **await** g , the guard g controls the suspension of the active process and consists of Boolean conditions b and return tests $x?$ (see below). Just like functional expressions e , guards g are side-effect free. If g evaluates to false, the active process is suspended, i.e., added to the queue, and some other process from the queue may execute. *Expressions* rhs include the creation of an object group **new cog** $C(e)$, object creation in the group of the creator **new** $C(e)$, method calls $o!m(e)$ and $o.m(e)$, future dereferencing $x.\text{get}$, and pure expressions e apply functions from the functional sublanguage to state variables.

Communication and *synchronization* are decoupled in ABS. Communication is based on asynchronous method calls, denoted by assignments $f=o!m(e)$ to future variables f . Here, o is an object expression and e are (data value or object) expressions providing actual parameter values for the method invocation. (Local calls are written **this**! $m(e)$.) 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 guard **await** $f?$ *suspends the active process* unless a return to 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 $f.\text{get}$, which *blocks all execution in the object* until the return value is available. The statement sequence $x=o!m(e); v=x.\text{get}$ encodes commonly used *blocking calls*, abbreviated $v=o.m(e)$ (often referred to as synchronous calls). If the return value of a call is without interest, the call may occur directly as a statement $o!m(e)$ with no associated future variable. This corresponds to message passing in the sense that there is no synchronization associated with the call.

2.2 Real-Time ABS

Real-Time ABS [7] is an extension of ABS which captures the timed behavior of ABS models. An ABS model is a model in Real-Time ABS in which execution takes zero time; thus, standard statements in ABS are assumed to execute in zero time. Timing aspects may be added incrementally to an untimed behavioral model. Our approach extends the distributed concurrent object groups in ABS with an integration of both *explicit* and *implicit* time.

Deadlines. The object-oriented perspective on timed behavior is captured by *deadlines* to method calls. Every method activation in Real-Time ABS has an associated deadline, which decrements with the passage of time. This deadline can be accessed inside the method body with the expression **deadline** (). Deadlines are *soft*; i.e., **deadline** () may become negative but this does not in itself block the execution of the method. By default the deadline associated with a method activation is infinite, so in an untimed model deadlines will never be missed. Other deadlines may be introduced by means of call-site *annotations*.

Real-Time ABS introduces two new data types into the functional sublanguage of ABS: `Time`, which has the constructor `Time(r)`, and `Duration`, which has the constructors `InfDuration` and `Duration(r)`, where `r` is a value of the type `Rat` of rational numbers. The accessor functions `timeVal` and `durationValue` returns `r` for time and duration values `Time(r)` and `Duration(r)`, respectively. Let `o` be an object which implements a method `m`. Below, we define a method `n` which calls `m` on `o` and specifies a deadline for this call, given as an annotation and expressed in terms of its own deadline. Remark that if its own deadline is `InfDuration`, then the deadline to `m` will also be unlimited. The function `scale(d, r)` multiplies a duration `d` by a rational number `r` (the definition of `scale` is straightforward).

```
Int n (T x){ [Deadline: scale(deadline(),0.9)] return o.m(x); }
```

Explicit time. In the explicit time model of Real-Time ABS [14], the execution time of computations is modeled using *duration statements* `duration(e1, e2)` with best- and worst-case execution times `e1` and `e2`. This is the standard approach to modeling timed behavior, known from, e.g., timed automata in UP-PAAL [27]. These statements are inserted into the model, and capture execution time which does not depend on the system’s deployment architecture. Let `f` be a function defined in the functional sublanguage of ABS, which recurses through some data structure `x` of type `T`, and let `size(x)` be a measure of the size of this data structure `x`. Consider a method `m` which takes as input such a value `x` and returns the result of applying `f` to `x`. Let us assume that the time needed for this computation depends on the size of `x`; e.g., the computation time is between a duration `0.5*size(x)` and a duration `4*size(x)`. An interface `I` which provides the method `m` and a class `C` which implements `I`, including the execution time for `m` using the explicit time model, are specified as follows:

```
interface I { Int m(T x) }
class C implements I {
  Int m (T x){ duration(0.5*size(x), 4*size(x)); return f(x);
}
}
```

Implicit time. In the implicit time model of Real-Time ABS, the execution time is not specified explicitly in terms of durations, but rather *observed* on the executing model. This is done by comparing clock values from a global clock, which can be read by an expression `now()` of type `Time`. We specify an interface `J` with a method `p` which, given a value of type `T`, returns a value of type `Duration`, and implement `p` in a class `D` such that `p` measures the time needed to call the method `m` above, as follows:

```
interface J { Duration p (T x) }
class D implements J (I o) {
  Duration p (T x){ Time start; Int y;
    start = now(); y=o.m(x); return timeDifference(now(), start);
}
}
```

Observe that by using the implicit time model, no assumptions about execution times are specified in the model above. The execution time depends on how

quickly the method call is effectuated by the called object. The execution time is simply measured during execution by comparing the time before and after making the call. As a consequence, the time needed to execute a statement with the implicit time model depends on the *capacity* of the chosen deployment architecture and on *synchronization* with (slower) objects.

3 Modeling Deployment Architectures in ABS

3.1 Deployment Components

A *deployment component* in Real-Time ABS captures the execution capacity associated with a number of concurrent object groups. Deployment components are first-class citizens in Real-Time ABS, and provide a given amount of resources which are shared by their allocated objects. Deployment components may be dynamically created depending on the control flow of the ABS model or statically created in the main block of the model. We assume a deployment component `environment` with unlimited resources, to which the root object of a model is allocated. When objects are created, they are by default allocated to the same deployment component as their creator, but they may also be allocated to a different component. Thus, a model without explicit deployment components runs in `environment`, which does not impose any restrictions on the execution capacity of the model. A model may be extended with other deployment components with different processing capacities.

Given the interfaces `I` and `J` and classes `C` and `D` defined in Section 2.2, we can for example specify a deployment architecture in which two `C` objects are deployed on different deployment components `server1` and `server2`, and interact with the `D` objects deployed on a deployment component `clientServer`. Deployment components in Real-Time ABS have the type `DC` and are instances of the class `DeploymentComponent`. This class takes as parameters a name, given as a string, and a set of restrictions on resources. The name is mainly used for monitoring purposes. Here we focus on resources reflecting the components' processing capacity, which are specified by the constructor `CPUCapacity(r)`, where `r` represents the amount of abstract processing resources available between observable points in time. Below, we create three deployment components `Server1`, `Server2`, and `ClientServer`, with the processing capacities 6, 3, and unlimited (i.e., `ClientServer` has no resource restrictions). The local variables `server1`, `server2`, and `clientServer` refer to these three deployment components, respectively. Objects are explicitly allocated to the servers by annotations; below, `object1` is allocated to `Server1`, etc.

```
{ // This main block initializes a static deployment architecture:
  DC server1 = new DeploymentComponent("Server1",set[CPUCapacity(6)]);
  DC server2 = new DeploymentComponent("Server2",set[CPUCapacity(3)]);
  DC clientServer = new DeploymentComponent("ClientServer", EmptySet);
  [DC: server1] I object1 = new cog C;
  [DC: server2] I object2 = new cog C;
  [DC: clientServer] J client1monitor = new cog D(object1);
  [DC: clientServer] J client2monitor = new cog D(object2);
}
```

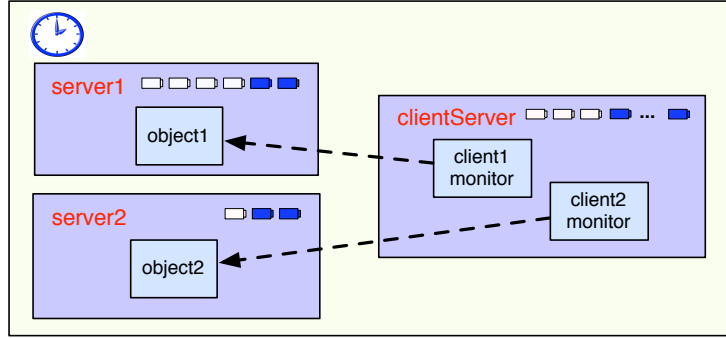


Fig. 1. A deployment architecture in Real-Time ABS, with three deployment components `server1`, `server2`, and `clientServer` (see Section 3.1). In each deployment component, we see its allocated objects and the “battery” of allocated and available processing resources (top right of each deployment component).

Figure 1 depicts this deployment architecture and the artefacts introduced into the modeling language. Since all objects are allocated to a deployment component (which is `environment` unless overridden by an annotation), we let the expression `thisDC()` evaluate to the deployment component of an object. For convenience, a call to the method `total("CPU")` of a deployment component returns its total amount of allocated CPU resources.

3.2 Resource Costs

The available resource capacity of a deployment component determines how much computation may occur in the objects allocated to that component. Objects allocated to the component compete for the shared resources in order to execute, and they may execute until the component runs out of resources or they are otherwise blocked. For the case of CPU resources, the resources of the component define its processing capacity between observable (discrete) points in time, after which the resources are renewed.

Cost models. The cost of executing statements in the ABS model is determined by a default value which is set as a compiler option (e.g., `defaultcost=10`). However, the default cost does not discriminate between statements and we may want to introduce a more refined cost model. For example, if e is a complex expression, then the statement $x=e$ should have a significantly higher cost than `skip` in a realistic model. For this reason, more fine-grained costs can be inserted into Real-Time ABS models by means of annotations. For example, let us assume that the cost of computing the function $f(x)$ defined in Section 2.2 may be given as a function g which depends on the size of the input value x . In the context of deployment components, we may redefine the implementation of interface `I`

above to be *resource-sensitive* instead of having a predefined duration as in the explicit time model. The resulting class C2 can be defined as follows:

```
class C2 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 provide more realistic resource-sensitive behavior. For the computation of the 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 input value x based on static analysis techniques, when given the ABS 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 a further refinement of a behavioral model. Therefore, cost annotations may be used by the modeler to abstractly represent the cost of some computation which remains to be fully specified. For example, the class C3 below represents a draft version of our method m in which the worst-case cost of the computation is specified although the function f has yet to be introduced:

```
class C3 implements I {  
  Int m (T x){ [Cost: size(x)*size(x)] return 0;  
  }  
}
```

4 Case Study: Application-Level Management of Virtualized Resources

A common strategy for web applications these days, especially in early development and deployment, is to acquire the needed resources (server, storage, bandwidth) from a cloud infrastructure provider such as Amazon instead of purchasing server hardware and data center space. In that way, initial costs can be kept low while still keeping the flexibility to react quickly to demand growth [8]. In this case study, we develop a model of a web application which distributes user requests to a number of servers deployed on the cloud. To clarify terminology, we shall refer to the clients of the web service as *users*, and the clients of the cloud provider (such as the web service) as *clients*.

A cloud infrastructure provider leases *virtual servers* to its clients by the CPU hour. Typically, the client application can select different configurations with respect to virtualized resources such as processing capacity, memory size, etc. The cost of leasing a virtual server depends on the configuration of these virtual resources, and in particular on the processing capacity of the virtual server. To keep costs down, it is in the interest of the client application that virtual servers are kept running only when they are busy processing requests from users, and that they are stopped and returned to the cloud provider otherwise.

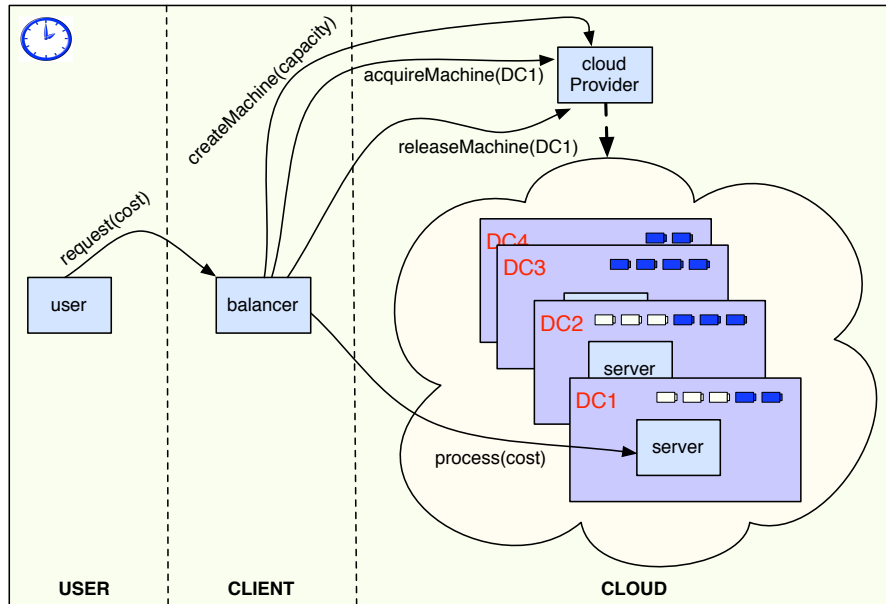


Fig. 2. An on-demand deployment architecture for the client application in Real-Time ABS. Neither user nor cloud provider contribute to the cost of running the system, and we assume the request processing costs dwarf the resources needed to run the balancer. Hence, only the servers are running in dedicated deployment components.

Consequently, a cloud-enabled application will typically have a component which handles the management of virtualized resources at the application level. This component monitors the user demand, provisions servers as needed, and distributes user requests between the active servers in order to meet the deadlines of the user requests while keeping the costs of leasing virtual servers down.

In this section, we develop a Real-Time ABS model of a *client* application which interacts with a *cloud provider* and with a *user*. The model is depicted in Figure 2. This client application consists of a (dynamic) number of *servers* and one *balancer* which is the main focus of our case study. The balancer is in charge of the management of the virtualized resources acquired by the client application. The user sends processing requests to the balancer, which sends them to an active server. To keep the focus on the balancer, we do not model the details of these requests; instead, they carry a deadline and a processing cost that represent an abstraction of QoS and computing requirements. For the same reason, we do not provide the details of the cloud provider model in this paper.

It is the responsibility of the balancer to implement a resource management strategy which both minimizes the cost of running the client application on the cloud and maximizes the application's QoS (i.e., minimizes the number of deadline misses for user requests). Note that this model does not aim for precise

```

interface CloudProvider {
    DC    createMachine(Int capacity);
    Unit  acquireMachine(DC machine);
    Unit  releaseMachine(DC machine);
    Int   getAccumulatedCost();
}

interface Balancer {
    Bool  request (Int cost); // called from User
}

interface Server {
    Unit  process(Int cost); // called from Balancer
    DC    getDC();
}

```

Fig. 3. Interfaces of the case study.

measurements, but rather for a rough understanding of the system behavior. Hence, no precise costs of running the system are obtained via simulations (which would depend on the varying price of CPU hours). Rather, *different balancing strategies* can be compared by evaluation against different usage scenarios, for example a user with a *steady request rate* or with an unexpected five-fold *load spike*.

Figure 3 shows the interfaces of the entities of the case study (the user needs no interface since it is not referenced by any object). Each `Server` has a method `process`, which incurs run-time costs on the server’s deployment component, which can be found via the `getDC` method. The `Balancer`’s `request` method is called from the `User`. The balancer is responsible for creating `Server` objects on deployment components acquired from the `CloudProvider` via the method `createMachine`. Two methods `acquireMachine` and `releaseMachine` start and stop virtual machines (modeled by deployment components) so that the `Server` objects can process requests.

4.1 The Server and the Cloud Provider

The server and the cloud provider are implemented by two classes `Server` and `CloudProvider`, which do not change as we vary strategies and user behavior. The class `Server`, shown in Figure 4, implements the `Server` interface and is quite straightforward. The method `process` consumes resources according to its cost argument, and the method `getDC` simply returns the deployment component on which the server object is deployed.

The class `CloudProvider` implements the `CloudProvider` interface with methods for creating, acquiring and releasing virtual machines. This is done by creating deployment components on which the client application can deploy objects. In addition, the cloud provider keeps track of the accumulated costs incurred by the client application. The cost is calculated in terms of the sum of the processing capacities of the *active* virtual machines; i.e., a call to

```

class Server implements Server {
    Unit process (Int cost) {
        while (cost > 0) { [Cost: 1] skip; cost = cost - 1; }
    }

    DC getDC() { return thisDC(); }
}

```

Fig. 4. Implementation of the Server class.

`acquireMachine(dc)` starts accounting for the virtual machine `dc` and a call to `releaseMachine(dc)` stops the accounting again for `dc`. The method `getAccumulatedCost` returns the accumulated cost of the client application. Inside the cloud provider, an active run method does the accounting for every time interval. Since our focus is the application-level management of virtualized resources, as implemented by the balancer, and not on specific strategies for cloud provisioning, we do not detail the cloud provider further in this paper.

4.2 The User Scenarios

We consider two user scenarios: *steady load* and *load spike*. The two scenarios are modeled by the corresponding classes `SteadyLoadUser` and `LoadSpikeUser`, given in Figure 5. The two classes have fields `numRequests` and `numFailures`, which are used for counting the number of sent requests and the number of missed deadlines for these requests, respectively. Both classes implement the method `sendRequest` which calls `request` with a given deadline on the balancer, suspends execution while waiting for the reply to the call, and does the bookkeeping after the reply has been received by incrementing the fields `numRequests` and `numFailures` as appropriate. The frequency of these requests is controlled by the active run method which differs between the two classes. In the `SteadyLoadUser` class, the run method asynchronously calls `sendRequest` and then suspends for a fixed duration. In contrast the run method of `LoadSpikeUser` has the same steady load behavior except for a window of time (between time 60 and 80 according to the clock), during which there is a load spike in which asynchronous calls to `sendRequest` are sent with much shorter intervals.

4.3 Balancing strategies

In this case study, we model three different balancers for the application-level management of the virtualized resources. The balancers provide the front end to our web application, which receives user requests, and uses backend servers, deployed on the cloud, for processing these user requests. The different balancers reflect different strategies for interacting with the cloud provider to achieve the resource management, and may be described as follows:

```

class SteadyLoadUser(Balancer b) {
  Int numRequests = 0;
  Int numFailures = 0;
  Unit run() {
    while (True) {
      this!sendRequest();
      await duration(5, 5);
    }
  }
  Unit sendRequest() {
    [Deadline: Duration(2)] Fut<Bool> s = b!request(3);
    await s?; Bool success = s.get;
    numRequests = numRequests + 1;
    if (~success) numFailures = numFailures + 1;
  }
}

class LoadSpikeUser(Balancer b) {
  Int numRequests = 0;
  Int numFailures = 0;
  Unit run() {
    while (True) {
      if (timeVal(now()) > 60 && timeVal(now()) < 80) {
        this!sendRequest();
        await duration(1, 1);
      } else {
        this!sendRequest();
        await duration(5, 5);
      }
    }
  }
  Unit sendRequest() {
    [Deadline: Duration(2)] Fut<Bool> s = b!request(3);
    await s?; Bool success = s.get;
    numRequests = numRequests + 1;
    if (~success) numFailures = numFailures + 1;
  }
}

```

Fig. 5. Different user behavior modeled by the two classes `SteadyLoadUser` and `LoadSpikeUser`.

- the **constant balancer** simply allocates one server sufficient for the expected load and keeps it running;
- the **as-needed balancer** calculates the server size needed to fulfill a specific request within the deadline, and allocates the needed resources disregarding the cost; and
- the **budget-aware balancer** operates with a given budget of CPU resources per time unit. Unused resources can be “saved for later” to cope with unexpected load spikes, but the cost of running the system is still bounded.

The Constant Balancer captures over-provisioning by processing all requests on a single server which should have sufficient capacity, and is modeled by the class `ConstantBalancer` in Figure 6. It initializes the web application by

```

class ConstantBalancer(CloudProvider provider, Int serverSize)
implements Balancer {
  Server server;
  DC dc;
  Bool initialized = False;
  Unit run() {
    Fut<DC> f = provider!createMachine(serverSize);
    await f?; dc = f.get;
    [DC: dc]server = new cog Server();
    initialized = True;
  }

  Bool request (Int cost) {
    await initialized;
    Fut<Unit> r = server!process(cost);
    await r?; return (durationValue(deadline()) > 0);
  }
}

```

Fig. 6. The Real-Time ABS model of the constant balancer.

requesting a single machine from the cloud provider, on which it deploys a concurrent object group consisting of a `Server` object. After initialization, the constant balancer uses this server to process all user requests, and returns success to a user request if it was processed within the deadline.

The As-Needed Balancer is modeled by the class `DynamicBalancer` in Figure 7. This class maintains a data structure `sleepingMachines` which sorts available machines (with deployed servers) by CPU processing capacity. We omit the (straightforward) definitions of the following auxiliary functions on this data structure: `hasMachine(s, i)` checks if a machine of capacity `i` is available in the structure `s`; `addMachine(s, i, m)` adds a machine `m` to the set associated with capacity `i` in `s`; and `removeMachine(s, i, m)` removes the machine `m` from the set associated with `i` in `s`.

When the `DynamicBalancer` receives a request, it calculates the machine capacity resources needed to fulfill the request, and requests a server deployed on a machine of appropriate size by calling `this.getMachine(resources)`. When it gets the server, it asynchronously calls `process` on this server and suspends. Once the reply is available, it calls `this.dropMachine(server)` and returns success to the user if the processing happened within the deadline.

The method `getMachine` first checks in `sleepingMachines` if there are available servers deployed on machines of appropriate size, in which case such a server is returned. (The auxiliary function `take(s)` selects an element of the set `s`.) Otherwise, the balancer requests a new machine from the cloud provider by calling `createMachine` and deploys a server on the new machine. The method `dropMachine` asks the cloud provider to stop running the machine on which the server is deployed and returns the server to the `sleepingMachines` set of appropriate capacity. The field `costPerTimeUnit` keeps track of the amount

```

class DynamicBalancer(CloudProvider provider) implements Balancer {
  Map<Int, Set<Server>> sleepingMachines = EmptyMap;
  Int costPerTimeUnit = 0; Int machineStartTime = 0;

  Server getMachine(Int size) {
    Server server = null; Time t = now();
    costPerTimeUnit = costPerTimeUnit + size;
    if (hasMachine(sleepingMachines, size)) {
      server = take(lookup(sleepingMachines, size));
      sleepingMachines= removeMachine(sleepingMachines, size, server);
      Fut<DC> fdc = server!getDC(); await fdc?; DC dc = fdc.get;
      Fut<Unit> fa = provider!acquireMachine(dc); await fa?;
    } else {
      Fut<DC> fdc = provider!createMachine(size);
      await fdc?; DC dc = fdc.get;
      [DC: dc] server = new cog Server();
    }
    machineStartTime = timeDifference(t, now()); return server;
  }

  Unit dropMachine(Server server) {
    Fut<DC> fdc = server!getDC(); await fdc?; DC dc = fdc.get;
    Fut<Unit> fr = provider!releaseMachine(dc); await fr?;
    Fut<Int> fs = dc!total("CPU"); await fs?; Int size = fs.get;
    costPerTimeUnit = costPerTimeUnit - size;
    sleepingMachines = addMachine(sleepingMachines, size, server);
  }

  Bool request (Int cost) {
    Int resources = (cost / durationValue(deadline())) + 1
      + machineStartTime;
    Server server = this.getMachine(resources);
    Fut<Unit> r = server!process(cost); await r?;
    this.dropMachine(server); return durationValue(deadline()) > 0;
  }
}

```

Fig. 7. The Real-Time ABS model of the as-needed balancer.

of resources *currently leased from the cloud provider*, and is updated by both methods `getMachine` and `releaseMachine`. This is the amount of resources for which the application is currently charged.

The Budget-Aware Balancer is a resource management strategy in which the balancer has a certain *budget per time interval*, and may save resources for later. This balancer is modeled by the class `BudgetBalancer` in Figure 8, with a class parameter `budgetPerTimeUnit` which determines this budget, and a field `availableBudget` which keeps track of the accumulated (saved) resources. The fields `sleepingMachines`, `costPerTimeUnit`, and `machineStartTime` and the methods `getMachine` and `dropMachine` are as in the `DynamicBalancer` class. When the budget-aware balancer gets a request, it calculates the resources needed to fulfill the request in the variable `wantedResources` and the resources it has available on the budget in

```

class BudgetBalancer(CloudProvider provider,Int budgetPerTimeUnit)
implements Balancer {
  Map<Int, Set<Server>> sleepingMachines = EmptyMap;
  Int costPerTimeUnit = 0; Int machineStartTime = 0;

  Int availableBudget = 1;
  List<Int> budgetHistory = Nil;

  Unit run() {
    while (True) {
      availableBudget = availableBudget + budgetPerTimeUnit
        - costPerTimeUnit;
      budgetHistory = Cons(availableBudget, budgetHistory);
      await duration(1, 1);
    }
  }

  Bool request(Int cost) {
    Bool result = False;
    Int wantedResources = (cost / durationValue(deadline())) + 1
      + machineStartTime;
    Int maxResources = (budgetPerTimeUnit - costPerTimeUnit)
      + (max(availableBudget, 0) / durationValue(deadline()));
    if (maxResources > 0) {
      Server server= this.getMachine(min(wantedResources,maxResources));
      Fut<Unit> r = server!process(cost);
      await r?;
      this.dropMachine(server);
      result = (durationValue(deadline()) > 0);
    }
    return result;
  }

  Server getMachine(Int size) { ... } // as in the DynamicBalancer
  Unit dropMachine(Server server) { ... } // as in the DynamicBalancer
}

```

Fig. 8. The Real-Time ABS model of the budget-aware balancer.

maxResources. If there are resources available on the budget, the budget-aware balancer calls `getMachine` to get the best server the request according to the budget. The budget-aware balancer has an active `run` method which monitors the resource usage and updates the available budget for every time interval. It also maintains a log `budgetHistory` of the available resources over time.

4.4 Comparing Balancing Strategies

Real-Time Maude has a formally defined semantics [7] which is used to implement a model simulator in the Maude system [11]. In order to compare the three balancing strategies of our case study, we simulate their behavior for the two user scenarios described in Section 4.2, in each case with a single “user” object generating requests. For simplicity, we here set the budget of the budget-aware

Strategy	User scenario			
	Steady load		Load spike	
	QoS	Cost	QoS	Cost
Constant balancer	100%	200	53%	200
As-needed balancer	100%	80	100%	128
Budget-aware balancer	100%	80	68%	97

Table 1. Simulation results.

balancer to 1. All simulations were run for 100 units of simulated time. The following measurements were extracted from the simulation traces:

- *quality of service* measured as the number of successful requests (i.e., requests completed within the deadline) divided by the total number of requests; and
- *accumulated cost* of running the machines, measured as the total sum of CPU resources made available by the cloud provider.

Table 1 summarizes the results. Not surprisingly, the as-needed balancer exhibits the best QoS numbers, but at potentially unbounded runtime cost. The constant balancer with a single running server exhibited both the highest runtime cost and the worst QoS under unexpected load with the chosen scenarios.

The budget-aware strategy exhibits only slightly better QoS characteristics under load than the constant balancer approach, which reflects how the budget was chosen. Figure 9 shows the available and used budget over time. It can be seen that the available budget is mostly used during normal load, so there are not many saved resources which can be used to deal with the load spike between time 60 and 80. A more realistic system would have a monitoring component to alert an operator, who would be able to manually add budget or switch to other balancing strategies, but this functionality was not considered in our case study.

5 Related Work

The concurrency model of ABS is akin to concurrent objects and Actor-based computation, in which software units with encapsulated processors communicate asynchronously [5, 19, 22, 30]. 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 [4, 23, 24], the authors have introduced *deployment components* as a modeling concept for deployment architectures, which captures restricted resources shared between a group of concurrent objects, and shown how components with parametric resources may be used to capture a model’s behavior for different assumptions about the available resources. The formal details of this approach are given in [24]. In previous work, the cost of execution was fixed in the language semantics. In this paper, we generalize that approach by proposing the specification of resource costs as part of the software development process. This

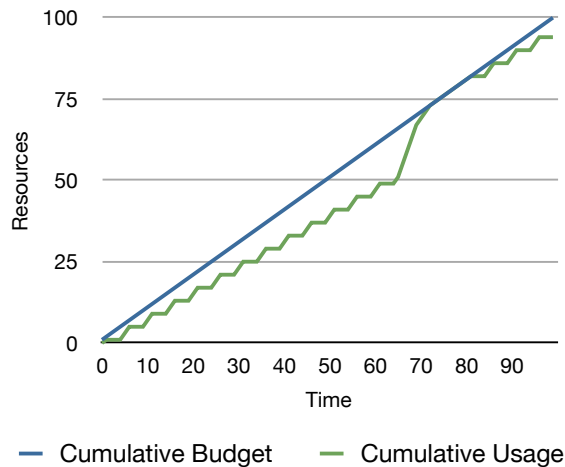


Fig. 9. Budget use over time for the budget-aware balancer. The load spike between time 60 and 80 quickly consumes the saved-up funds.

is supported by letting default costs be overridden by annotations with user-defined cost expressed in terms of the local state and the input parameters to methods. This way, the cost of execution in the model may be adapted by the modeler to a specific cost scenario. This allows us to abstractly model the effect of deploying concurrent objects on deployment components with different amounts of allocated resources at an early stage in the software development process, before modeling the detailed control flow of the targeted system. In two larger case studies addressing resource management in the cloud [13, 26], the presented approach is compared to specialized simulation tools and to measurements on deployed code.

Complementing the balancing strategies considered in this paper, the authors have studied extensions to the deployment component framework which support more advanced (or fine-grained) load-balancing. We have considered two such extensions, based on adding an expression `load(n)` which returns the average load of the current deployment component over the last n time intervals. First, by including resources as first-class citizens of ABS and allowing (virtual) resources to be reallocated between deployment components [23]. Second, by allowing objects to be marshaled and reallocated between deployment components [25].

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 [16]. A survey of model-based performance analysis techniques is given in [6]. Formal systems using process algebra, Petri Nets, game theory, and timed automata have been

used in the embedded software domain (e.g., [10,17]), but also to the schedulability of processes in concurrent objects [20]. The latter work complements ours as it does not consider restrictions on shared deployment resources, but associates deadlines with method calls with abstract duration statements.

Work on modeling object-oriented systems with resource constraints is more scarce. Eckhardt et al. [15] use statistical modeling of meta-objects and virtual server replication to maintain service availability under denial of service attacks. Using the UML SPT profile for schedulability, performance, and time, Petriu and Woodside [28] 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 [6]. Closer to our work is M. Verhoef’s extension of VDM++ for embedded real-time systems [29], 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.

Others interesting lines of research are static cost analysis (e.g., [3,18]) and symbolic execution for object-oriented programs. 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 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 to memory analysis for ABS models [4]. However, the generalization of that work for general, user-defined cost models and its integration into the software development process remains future work. A future extension of our approach with symbolic execution would allow us to calculate best- and worst-case response time for the different balancing strategies depending on the available resources and the user load.

6 Conclusion

This paper gives an overview of how deployment architectures can be modeled by means of deployment components in Real-Time ABS. We show how this approach may be used to model virtualized systems by developing a case study of application-level management of virtualized resources in a cloud computing context. In our case study, an abstract cloud provider leases virtual machines with given amounts of CPU processing capacities to client services. The case

study takes the *client perspective* on virtualized resource management, and models a client application for which three different proposals for a balancer class are compared in order to gain insights into their resource needs. The resulting models in Real-Time ABS are simulated for different user scenarios. In these scenarios, the cost of leasing resources from the cloud provider with the different resource management strategies are compared with respect to the QoS of the service for user requests. We are not aware of similar work addressing the formal modeling of application-level management of virtualized resources for cloud computing. However, we believe this problem is of increasing importance in a world of cloud-enabled applications.

References

1. G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems*. The MIT Press, 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*, volume 4421 of *Lecture Notes in Computer Science*, 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)*, volume 6664 of *Lecture Notes in Computer Science*, pages 353–368. Springer, June 2011.
5. J. Armstrong. *Programming Erlang*. Pragmatic Bookshelf, 2007.
6. 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.
7. J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. User-defined schedulers for real-time concurrent objects. To appear in *Innovations in Systems and Software Engineering*, 2012.
8. 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.
9. D. Caromel and L. Henrio. *A Theory of Distributed Object*. Springer, 2005.
10. A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Resource interfaces. In *Proc. Third Intl. Conf. on Embedded Software (EMSOFT'03)*, volume 2855 of *Lecture Notes in Computer Science*, pages 117–133. Springer, 2003.
11. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
12. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, Mar. 2007.
13. 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 Conf. on Service-Oriented and Cloud Computing (ESOCC)*, Lecture Notes in Computer Science. Springer, Sept. 2012. To appear.

14. F. S. de Boer, M. M. Jaghoori, and E. B. Johnsen. Dating concurrent objects: Real-time modeling and schedulability analysis. In *Proc. CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 1–18. Springer, Sept. 2010.
15. J. Eckhardt, T. Mühlbauer, M. AlTurki, J. Meseguer, and M. Wirsing. Stable availability under denial of service attacks through formal patterns. In *Proc. FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 78–93. Springer, 2012.
16. I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by runtime parameter adaptation. In *Proc. ICSE*, pages 111–121. IEEE, 2009.
17. E. Fersman, P. Krcál, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.
18. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *POPL*, pages 127–139. ACM, 2009.
19. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.
20. M. M. Jaghoori, F. S. de Boer, T. Chothia, and M. Sirjani. Schedulability of asynchronous real-time concurrent objects. *Journal of Logic and Algebraic Programming*, 78(5):402–416, 2009.
21. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proc. 9th Intl. Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.
22. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
23. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Dynamic resource re-allocation between deployment components. In *Proc. Formal Engineering Methods (ICFEM’10)*, volume 6447 of *Lecture Notes in Computer Science*, pages 646–661. Springer, Nov. 2010.
24. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Validating timed models of deployment components with parametric concurrency. In *Proc. Intl. Conf. on Formal Verification of Object-Oriented Software (FoVeOOS’10)*, volume 6528 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2011.
25. E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. A formal model of object mobility in resource-restricted deployment scenarios. In *Proc. Formal Aspects of Component Software (FACS 2011)*, volume 7253 of *Lecture Notes in Computer Science*. Springer, 2012.
26. E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Modeling resource-aware virtualized applications for the cloud in Real-Time ABS. In *Proc. Formal Engineering Methods (ICFEM’12)*, *Lecture Notes in Computer Science*. Springer, Nov. 2012. To appear.
27. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.
28. 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.
29. M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In *Proc. Formal Methods (FM’06)*, volume 4085 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2006.
30. A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. OOPSLA*, pages 439–453, ACM Press, 2005.