Formal Modeling of Resource Management for Cloud Architectures: An Industrial Case Study *

Frank S. de Boer[1], Reiner Hähnle[2], Einar Broch Johnsen[3], Rudolf Schlatte[3], and Peter Y. H. Wong[4]

[1] CWI, Amsterdam, The Netherlands, f.s.de.Boer@cwi.nl
[2] Technical University of Darmstadt, Germany, haehnle@cs.tu-darmstadt.de
[3] Dept. of Informatics, University of Oslo, Norway, einarj|rudi@ifi.uio.no
[4] Fredhopper B.V., Amsterdam, The Netherlands, peter.wong@fredhopper.com

Abstract. We show how aspects of performance, resource consumption, and deployment on the cloud can be formally modeled for an industrial case study of a distributed system, using the abstract behavioral specification language ABS. These non-functional aspects are integrated with an existing formal model of the functional system behavior, supporting a separation of concerns between the functional and non-functional aspects in the integrated model. The ABS model is parameterized with respect to deployment scenarios which capture different application-level management policies for virtualized resources. The model is validated against the existing system's performance characteristics and used to simulate and compare deployment scenarios on the cloud.

1 Introduction

Virtualization is a key technology enabler for cloud computing. Although the added value and compelling business drivers of cloud computing are undeniable [12], this new paradigm also poses considerable new challenges that have to be addressed to render its usage effective for industry. Virtualization makes elastic amounts of resources available to application-level services; for example, the processing capacity allocated to a service may be changed according to demand. Current software development methods, however, do not support the modeling and validation of application-level resource management strategies for virtualized resources in a satisfactory way. This seriously limits the potential for fine-tuning a service to the available virtualized resources. This paper demonstrates how to overcome this limitation by integration of resource management into a formal, yet realistic, model that yields to simulation and analysis.

Our long-term goal is the integration of virtualization into the development process of general purpose software services, by leveraging resources and resource management to the modeling of software. Our starting point in addressing this challenge is the recently developed **a**bstract **b**ehavioral **s**pecification language

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

ABS [19]. ABS is *object-oriented* to stay close to high-level programming languages and to be easily usable as well as accessible to software developers, and it is *executable* to support full code generation and (timed) validation of models.

ABS has been extended with time and with primitives for leveraging resources and their dynamic management to the abstraction level of software models [8,23]. The extension achieves a separation of concerns between the *application model*, which requires resources, and the *deployment scenario*, which reflects the virtualized computing environment and provides elastic resources. For example, an application model may be analyzed with respect to deployments on virtual machines with varying features: the amount of allocated computing or memory resources, the choice of application-level scheduling policies for client requests, or the distribution over different virtual machines with fixed bandwidth constraints. The simulation tools developed for ABS may then be used to compare the performance of a service ranging over different deployment scenarios already at the modeling level.

The main contribution of this paper is a large industrial case study, which demonstrates that it is possible to model aspects of performance, resource consumption, and deployment on the cloud based on this extension of ABS. The non-functional aspects are integrated with a model of the functional system behavior, achieving a separation of concerns between the functional and non-functional aspects. The ABS model is parameterized over deployment scenarios which capture different application-level management policies for virtualized resources. The model is validated against the existing system's performance characteristics and used to simulate and compare deployment scenarios on the cloud. A companion paper [22] details the modeling of the cloud provider and compares our approach to results obtained by simulation tools.

The paper is organized as follows: In Sect. 2 we describe the case study used in this paper; in Sect. 3 we explain how deployment scenarios are modeled in the extended ABS; in Sect. 4 we describe the modeling of the case study in ABS, before we discuss related work in Sect. 5 and conclude in Sect. 6.

2 The Case Study: Background

The Fredhopper Access Server (FAS) is a distributed, concurrent object-oriented system that provides search and merchandising services to e-Commerce companies. Briefly, FAS provides to its clients structured search capabilities within the client's data. Each FAS installation is deployed to a customer according to a deployment architecture; see [34] for a detailed presentation of the individual components of FAS and its deployment model. FAS consists of a set of live environments and a single staging environment. A *live environment* processes queries from client web applications via web services. FAS aims at providing a constant query capacity to client-side web applications. A *staging environment* is responsible for receiving data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments according to a *Replication Protocol* which is implemented by the *Replication System*. The

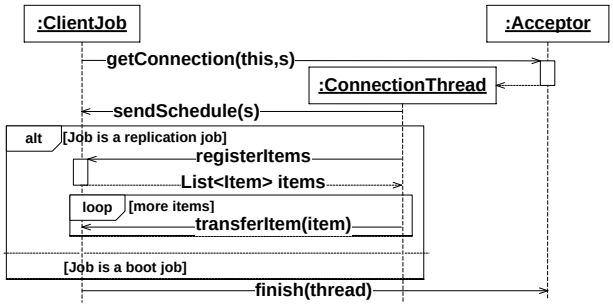

Fig. 1. Interaction between ClientJob, Acceptor, and ConnectionThread.

Replication System consists of a *SyncServer* at the staging environment and one *SyncClient* for each live environment. The SyncServer determines the schedule and content of replication, while a SyncClient receives data and configuration updates.

The SyncServer communicates to SyncClients by creating *ConnectionThread* objects via its *Acceptor*. ConnectionThreads serve as the interface to the server side of the Replication Protocol. SyncClients schedule and create *ClientJob* objects to handle communications to the client-side of the Replication Protocol. Fig. 1 is a UML sequence diagram that depicts the replication protocol between a ClientJob, a ConnectionThread, and an Acceptor. In this paper we detail a part of the Replication Protocol that is informally described in Fig. 2.

Relevance to Cloud Computing. FAS provides structured search and navigation capabilities within a client's data. For the last decade, FAS installations were deployed as server-based products on client premises with fixed hardware resources. However, on-premise deployment does not scale with increasing demand on throughput and update frequency. This is especially visible when customers experience drastic increase on throughput and data updates at certain time periods. For example, retail customers might expect large throughput during seasonal sales. In these periods, much larger amounts of purchases will be made and stock units would need to be updated very frequently so that customers do not receive incorrect information. Higher throughput requires a larger number of live environments and, in turn, larger number of SyncClients. On-premise deployment cannot cope with on-demand, varying requirements without large up-front investments in hardware which remains unused most of the time.

To cater for these requirements, FAS is deployed as a service (SaaS) over virtualized resources that provide the necessary elasticity. To this end, virtualization makes elastic amounts of resources available to application-level services; for example, the processing capacity allocated to a service can be changed on demand. Fig. 3 shows how an on-demand deployment architecture for the Replication System on virtual environments is implemented using cloud resources. Virtualized resources allow the SyncServer (via the Acceptor) to elastically allo-

1. SyncServer starts the Acceptor, which listens for connections.
2. SyncClient schedules a *Boot* job, and an associated ClientJob object is created and connected to the Acceptor. This corresponds to the message `getConnection(this,s)` in Fig. 1, where `s` is a schedule.
3. SyncServer creates a ConnectionThread to communicate with this ClientJob.
4. The ClientJob asks the ConnectionThread for *all replication schedules*, denoted by the message `sendSchedule(s)` in Fig. 1. Replication schedules dictate when and where the SyncServer monitors for changes in the staging environment. These changes are replicated to the live environments through their SyncClients. Each schedule specifies a *replication type*; i.e., the number of locations and type of data to be replicated. The schedule also specifies the amount of time until the replication must commence and the deadline of each replication.
5. The ClientJob receives the replication schedules and creates new ClientJob objects representing these schedules. If the old ClientJob object represented a Boot job, it releases the ConnectionThread and terminates.
6. When a *Replication* job is triggered, its associated ClientJob object connects immediately to the Acceptor.
7. SyncServer creates a ConnectionThread to communicate with each ClientJob.
8. A ClientJob asks the ConnectionThread for its replication schedule and recursively creates a new ClientJob object to deal with the next schedule. The ClientJob then receives a sequence of file updates according to its replication type, after which it releases the ConnectionThread and terminates. This is denoted by the message `registerItems` followed by zero or more `transferItem(item)` messages in Fig. 1.
9. The ConnectionThread first sends a replication schedule to the ClientJob according to the ClientJob's replication type, then a sequence of file updates according to this replication type, and then it terminates.

Fig. 2. Informal description of the interactions in the Replication Protocol.

cate resources to each replication job based on the *cost* and the *deadline* of the replication to be conducted by the corresponding ClientJob object.

3 ABS Deployment Architecture

ABS is an abstract, executable, object-oriented modeling language with a formal semantics [19], targeting distributed systems. ABS is based on concurrent object groups (COGs), akin to concurrent objects [15,20], Actors [1], and Erlang processes [4]. COGs in ABS support interleaved concurrency based on guarded commands. This allows active and reactive behavior to be easily combined, by means of a cooperative scheduling of processes which stem from method calls. Real-Time ABS extends ABS models with *implicit* time [8]; execution time is not specified directly in terms of durations (as in, e.g., UPPAAL [25]), but rather *observed* by measurements of the executing model. 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

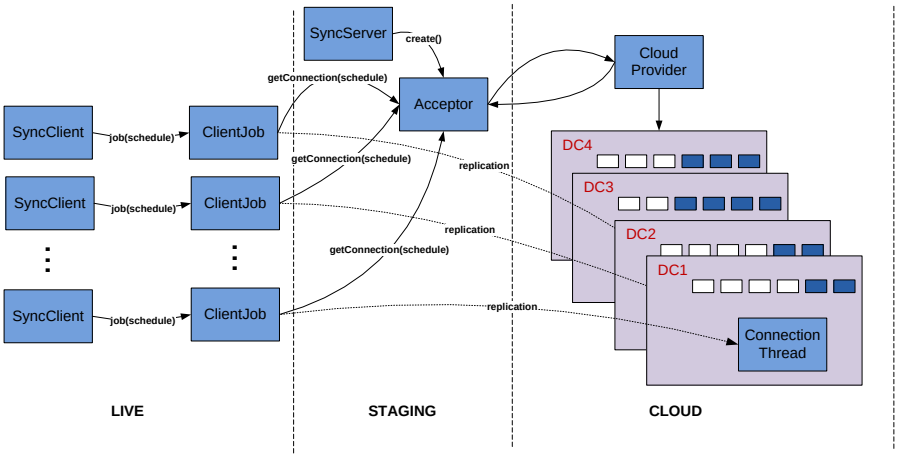

Fig. 3. An on-demand deployment architecture for the Replication System using Cloud resources.

object. In fact, the execution time of a statement varies with the *capacity* of the chosen deployment architecture and on *synchronization* with (slower) objects.

3.1 Behavioral Modeling in ABS

ABS combines functional and imperative programming styles with a Java-like syntax [19]. COGs execute in parallel and communicate through asynchronous method calls. Data manipulation inside methods is modeled using 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 while maintaining an overall object-oriented design close to the target system. The functional part of ABS consists 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>` (for a type parameter `A`); and functions over values of these data types, with support for pattern matching. In Real-Time ABS, measurements are additionally obtained by comparing values from a global clock, which can be read by an expression **`now`**`()` of type `Time`.

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. Communication and synchronization are decoupled in ABS. 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 caller may proceed with its execution *without blocking* on the method reply. Two operations on future variables 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 same COG 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. Inside a COG, ABS also supports standard synchronous method calls `o.m(e)`.

The active process of an object can be unconditionally suspended by the statement **suspend**, adding this process to the queue, from which an enabled process is then selected for execution. The guards `g` in **await** `g` control suspension of the active process and consist of Boolean conditions conjoined with return tests `f?` on future variables `f`. Just like functional expressions, guards `g` are side-effect free. The remaining statements are standard; e.g., sequential composition $s_1; s_2$, assignment `x=rhs`, and **skip**, **if**, **while**, and **return** constructs. 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, future dereferencing `f.get`, and functional expressions `e`.

3.2 Deployment Modeling in ABS

The execution capacity of (virtualized) locations can be abstractly captured by *deployment components* (for formal definitions, see [21]). Deployment components share resources between their allocated objects. A deployment component `environment` with unlimited resources is used for the model's root object and, e.g., clients. Deployment components with different capacities may be dynamically created depending on the control flow of the ABS model or statically created in the main block. Objects are by default allocated to the deployment component of their creator, but they may also be allocated differently.

Deployment components in ABS have the type `DC` and are instances of the class `DeploymentComponent`, which takes as parameters a location name and a set of resource bounds. Our focus is on resources reflecting *processing capacity*.

The *resource capacity* of a deployment component is specified by the constructor `CPUCapacity(r)`, where `r` of type `Resource` represents the amount of abstract processing resources available between observable (discrete) points in time, after which the resources are renewed. Objects are explicitly allocated to specific deployment components via annotations. They compete for the shared resources to execute methods and may execute until the component runs out of resources or until they are otherwise blocked. The method `total("CPU")` of a deployment component returns its total amount of allocated CPU resources.

The *cost* of executing statements in ABS is specified by the modeler. A default cost for statements can be set as a compiler option (e.g., `defaultcost=10`). However, this default cost does not discriminate between different statements. More precise cost expressions are often desirable; for example, if `f(x)` is a complex expression, then the statement `result=f(x)` should have a significantly higher cost than **skip**. Fine-grained costs can be introduced via statement annotations; e.g., `[Cost: g(size(x))] result=f(x)` where the cost is given by a function `g` in terms of the *size* of the input values `x` to the function `f`.

It is the responsibility of the modeler to specify realistic costs. A behavioral model with default costs may be gradually refined to provide fine-grained

```
data Schedule =
  Schedule(String sn, Int il, Int dl, Int ct, List<File> files);
interface ConnectionThread { }
interface ClientJob { Unit executeJob(); }
interface SyncClient { Unit scheduleJob(Schedule s); }
interface Acceptor {
  ConnectionThread getConnection(ClientJob job, Schedule s);
  Unit finish(ConnectionThread thread);
  Unit end();
}
interface CloudProvider {
  DC createMachine(Int capacity);
  Unit acquireMachine(DC vm);
  Unit releaseMachine(DC vm);
  Int getAccumulatedCost();
}
```

Fig. 4. Data types and interfaces of the Replication System.

resource-sensitive behavior. On the other hand, the modeler may want to capture resource consumption at an *abstract level* without a fully developed model. Cost annotations can be used to abstractly represent the cost of a computation which is not be fully specified; e.g., `[Cost: g(size(x))] skip`.

4 Case Study: The ABS Model

The Replication System, introduced in Sect. 2, is part of the Fredhopper Access Server (FAS). The current Java implementation of FAS has over 150,000 lines of code, of which 6,500 are part of the Replication System. The functional aspects of the Replication System have previously been modeled in detail in ABS [34]. This section describes how the model was extended to capture non-functional and resource aspects of the Replication System. The extended model consists of 40 classes, 17 data types, and 80 user-defined functions (in total 5,000 lines of ABS code, 25% of which capture scheduling information as well as file systems and data bases from third party libraries not included in the Java implementation).

Fig. 4 shows the main data type and interfaces. Data type `Schedule` records interval, deadline, cost, and file locations to receive updates for each type of replication schedule. The interface `ConnectionThread` models its objects as active objects (without methods). The interface `ClientJob` defines the method `executeJob()` for executing replication schedules, and `SyncClient` defines the method `scheduleJob(Schedule s)` for scheduling the given replication schedule `s`. The `getConnection(job, s)` method of the `Acceptor` interface is called from the `ClientJob`. The acceptor creates `ConnectionThread` objects on virtual machines acquired from the `CloudProvider` via the method `createMachine`. The methods `acquireMachine` and `releaseMachine` are used to start and stop virtual machines (modeled by deployment components) to let replication schedules be conducted by `ConnectThread` objects. For presen-

tation purposes, we focus on the interface implementations given in the classes `CloudProvider`, `ConnectionThread`, and `Acceptor`.

The `CloudProvider` interface (shown in Fig. 4) is implemented by a class of the same name. Virtual machines are modeled by deployment components in ABS, on which the client application can deploy objects. In addition, the cloud provider keeps track of the *accumulated cost* incurred by the client application. This accumulated cost is retrievable by the method `getAccumulatedCost()` during execution. Accumulated cost is calculated in terms of the sum of the processing capacities of the *active* virtual machines over time; i.e., a call to `acquireMachine(vm)` starts the accounting for machine `vm` and a call to `releaseMachine(vm)` stops the accounting. Inside the cloud provider, an active `run()` method does the accounting for every time interval. Since we focus on the application-level management of virtualized resources, as implemented by the balancer, and not on a specific strategy for cloud provisioning, we do not detail the cloud provider further.

We model and compare three potential balancing strategies offered by the `Acceptor` for the application-level management of virtualized resources. An `Acceptor` gets requests for replication sessions from `ClientJob` objects. It deploys `ConnectionThread` objects on cloud instances to conduct replications with the `ClientJob` objects. The implementations of `Acceptor` reflect different strategies for interacting with the cloud provider to achieve resource management:

Constant balancing deploys `ConnectionThread` objects to a single virtual machine sufficient for the expected load, and keeps this machine running;

As-needed balancing calculates the CPU capacity of the virtual machine needed for a specific replication schedule with a given deadline, and deploys `ConnectionThread` objects to a machine supplying the needed resources disregarding the cost; and

Budget-aware balancing calculates the CPU capacity of the cloud instance for a given budget. Unused funds can be "saved up" to cope with load spikes, but the cost of running the system is still bounded by the overall budget.

The Cloud User Account. Each acceptor encapsulates an `Account` object that realizes book-keeping for a cloud user account (see Fig. 5). The implementation maintains a data structure in the field `is` which sorts available machines by CPU processing capacity, the current cost per time unit `costPerTimeUnit` for the cloud user account, and the time `instanceStartUpTime` it took the most recent machine to start up. Method `getInstance(size)` either requests a new virtual machine from the cloud provider or brings online an existing offline machine. The method `dropInstance(d)` takes a machine offline when it is no longer active.

Constant balancing over-provisions by processing all replication schedules on a single virtual machine with sufficient capacity, and is captured by the class `ConstantAcceptor` in Fig. 6. The acceptor initially requests a single machine

```
interface Account {
  DC   getInstance(Int size);
  Unit dropInstance(DC d);
  Int  getCostPerTimeUnit();
  Int  getLastInstanceStartUpTime();
}

class AccountImpl implements Account {
  Map<Int, Set<DC>> is = EmptyMap;
  Int costPerTimeUnit = 0; Int instanceStartUpTime = 0;

  DC getInstance(Int size) {
    DC d = null;
    Time t = now();
    costPerTimeUnit = costPerTimeUnit + size;
    if (hasSetFor(is, size)) {
      d = takeOne(lookup(is, size));
      is = removeFrom(is, size, d);
      Fut<Unit> fa = provider!acquireMachine(d); await fa?;
    } else {
      Fut<DC> fdc = provider!createMachine(size);
      await fdc?; d = fdc.get;
    }
    instanceStartTime = timeDifference(t, now());
    return d;
  }
  Unit dropInstance(DC d) {
    Fut<Unit> fr = provider!releaseMachine(d); await fr?;
    Fut<Int> fs = d!total("CPU"); await fs?; Int size = fs.get;
    costPerTimeUnit = costPerTimeUnit - size;
    is = addToSet(is, size, d);
  }
}
```

Fig. 5. The Cloud User Account.

through its cloud user account and deploys all `ConnectionThread` objects to this machine after initialization, to process the replication schedules.

As-needed balancing receives a `request` for a connection from a `ClientJob` object, calculates the `resources` needed by the virtual machine to fulfill the replication schedule, and requests a machine of appropriate size through the user account. Implementation details are omitted for brevity.

Budget-aware balancing (class `BudgetAcceptor` in Fig. 6) is a strategy where the acceptor has a certain *budget per time interval* and may "save resources" for later. The class parameter `budgetPerTimeUnit` determines this budget and the field `availableBudget` keeps track of the accumulated (saved) resources. When the acceptor gets a request from a `ClientJob` it calculates the resources needed to fulfill the replication schedule in `wantedResources` and the resources it has available on the budget in `maxResources`. If resources are available on the budget, the acceptor calls `getInstance(size)` on the user account to get the best machine according to the budget. The `run()` method monitors the resource usage and updates the available budget for every time

```
class ConstantAcceptor(SyncServer server,
  Int instanceSize, Account acc) implements Acceptor {
  DC dc = null;
  Unit run() { dc = acc.getInstance(instanceSize); }
  ConnectionThread getConnection(ClientJob job, Schedule schedule) {
    Int cost = scheduleCost(schedule);
    await dc != null; ConnectionThread th = null;
    [DC: dc] th = new cog ConnectionThreadImpl(job,server,cost);
    return th; }
  Unit finish(ConnectionThread t) { }
  Unit end() { acc.dropInstance(dc); }
}

class BudgetAcceptor(SyncServer server, Account acc,
  Int budgetPerTimeUnit) implements Acceptor {
  Int availableBudget = 1; List<Int> budgetHistory = Nil;
  Unit run() {
    while (True) {
      Int cu = acc.getCostPerTimeUnit();
      availableBudget = availableBudget + budgetPerTimeUnit - cu;
      budgetHistory = Cons(availableBudget, budgetHistory);
      await duration(1, 1);} }
  ConnectionThread getConnection(ClientJob job, Schedule schedule) {
    Int cost = scheduleCost(schedule);
    Int dur = durationValue(deadline());
    Int startUp = acc.getLastInstanceStartUpTime();
    Int wantedResources = (cost / dur) + 1 + startUp;
    Int maxResources = (budgetPerTimeUnit - costPerTimeUnit) +
      (max(availableBudget, 0) / dur);
    ConnectionThread th = null;
    if (maxResources > 0) {
      DC dc = acc.getInstance(min(wantedResources, maxResources));
      [DC: dc] th = new cog ConnectionThreadImpl(job,server,cost);}
    return th; }
  Unit finish(ConnectionThread thread) {
    Fut<DC> fdc = thread!release();
    await fdc?; DC instance = fdc.get;
    acc.dropInstance(instance);}
  Unit end() { }
}
```

Fig. 6. The classes `ConstantAcceptor` and `BudgetAcceptor`.

interval. It also maintains a log `budgetHistory` of the available resources over time.

4.1 Calibration

To obtain a realistic cost model of the Replication System in ABS, we measured the execution time of replication sessions for the different schedules on the Java implementation. We were interested in three types of replication schedules:

Search: A session replicates changes from the search index, i.e., the underlying data structure providing search capability on a customer's product items.

Business rule: A session replicates changes to the business configuration; i.e., the presentation of search results such as the sorting of items and promotions.

Schedule	Execution Time	Cost	Interval	Deadline
Search	34.0s	14	11	3
Business rules	2.5s	1	11	2
Data	274.9s	110	11	11

Table 1. Measurements on the Java implementation of the Replication System and derived simulation parameters.

Data: A session replicates changes concerning the item and navigation indices, i.e., the core index structures and data model for providing faceted navigation on a customer's product items.

Table 1 shows the average execution time of a single ClientJob for each schedule type for one SyncClient on a reference machine (4 core CPU 2.5GHz, 8GB memory), as well as the cost value subsequently used in the ABS model, which is directly proportional to execution time. Based on these schedule execution times and costs, we extend the functional ABS model of the Replication System with resource and timing information. To determine suitable deadlines for individual schedules and intervals between each ClientJob, we iteratively simulated a Replication System consisting of one SyncClient and a fixed number of replication jobs on a single cloud instance with CPU capacity set to 30 and interval to 11. With these parameters fixed, the lowest deadlines for schedules could be determined for 100% quality of service (QoS). Table 1 also shows the results of this initial simulation.

We also identified a *hot spot* of the `ConnectionThread` in the Java implementation of the method `transferItem(fileset)`, which accounts for 99% of the execution time. The hot spot justifies adding a single cost annotation to the `ConnectionThread` class of the model so that resources are consumed upon invocation of that method. Fig. 7 sketches the implementation of the `ConnectionThread` interface.

4.2 The Results

Fig. 8 shows simulation results of the different balancing strategies over the number of live environments. We simulated from 1 to 20 environments over 100 time units; 20 environments are typically required to handle large query throughputs over a large number of product items. For each number of environments and scheduling strategy, the graph shows the quality of service (QoS) as a percentage of deadlines that have been met and the total amount of CPU resources made available by the cloud provider during the simulation. The *constant* balancing strategy models machine over-provisioning as mentioned in Sect. 2, with good QoS but the highest cost, independently of the load (QoS degrades once more environments are added than provisioned for). As expected, the *as-needed* balancing strategy exhibits 100% QoS, albeit with rising cost. The *budget-aware* balancing strategy exhibits rising cost up to the chosen budget and degrading QoS thereafter.

```
class ConnectionThreadImpl(ClientJob job,
  SyncServer server, Int cost) implements ConnectionThread {
  Maybe<Command> cmd = Nothing;
  Set<Schedule> schedules = EmptySet;

  Unit consumeResource(Int amount) {
    Int c = 0; while (c <= amount) { [Cost: 1] c = c + 1; }}
  Unit run() {
    await cmd != Nothing;
    schedules = this.sendSchedule();
    if (cmd != Just(ListSchedule)) {
      ...
      Int size = size(filesets);
      while (hasNext(filesets)) {
        Pair<Set<Set<File>>,Set<File>> nfs = next(filesets);
        filesets = fst(nfs); Set<File> fileset = snd(nfs);
        this.transferItem(fileset);
        this.consumeResource(cost/size);}
      ... } }
}
```

Fig. 7. The class `ConnectionThreadImpl`.

To compare the simulation results against the running system, we executed the simulation task loads on the Java implementation of the Replication System and measured the execution time. From Fig. 9, it can be seen that there is a direct correlation between simulated cost and measured execution time, except for a constant factor resulting from system start-up time. Upon reflection, we decided not to model start-up time, since this one-time cost is amortized among all requests made during the server lifetime and hence is a negligible factor in a long-running system such as the FAS Replication System.

5 Related Work

To reduce complexity, general-purpose modeling languages strive for *abstraction* [24]: descriptions primarily focus on the functional behavior and logical composition of software, largely overlooking how the software's deployment influences its behavior. However, by using virtualization technologies an application can *modify resources of its deployment scenario during execution*; e.g., to dynamically create virtual processors. For cyber-physical and embedded systems, it is today accepted that modeling and programming languages need a timed semantics [26]. The Java Real-Time Specification (RTSJ) [11] extends Java with high-resolution time, including absolute time and relative time, and new thread concepts to solve time-critical problems: threads in RTSJ offer more precise scheduling than standard threads, with 28 strictly enforced priority levels. The modeling and analysis of single resources is discussed in, e.g., [2, 16, 33]. *Resource-aware programming* allows users to monitor the resources consumed by their programs, to manage such resources in the programs, and to transfer (i.e., add or remove) resources dynamically between distributed computations [27].

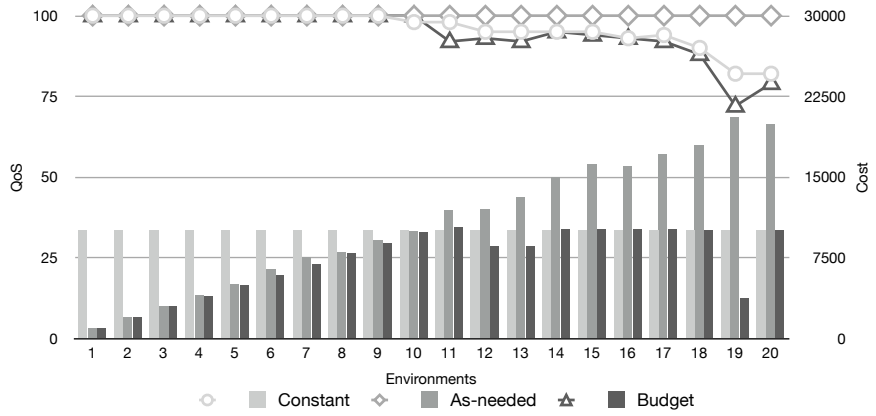

Fig. 8. Simulation results: QoS as percentage of successful sessions (left scale) and accumulated cost (right scale).

Resource constraints in the embedded systems domain led to a large body of work on performance analysis using formal models based on, e.g., process algebra [7], Petri Nets [31], and priced [10], timed [3], and probabilistic [6] automata and games (an overview of automata-based approaches is [33]). Related approaches are also applied to web services and business processes with resource constraints [18, 28]. These approaches typically abstract from the data flow and *declare* the cost of transitions in terms of time or in terms of a single resource. The automata-based modeling language MODEST [9] combines functional and non-functional requirements for stochastic systems, using a process algebra with dynamically computed weight expressions in probabilistic choice. Compared to ABS, these approaches do not associate capacities with locations but focus on non-functional aspects of embedded system without resource provisioning and management of dynamically created locations as studied in our paper.

Work on the modeling of object-oriented systems with resource constraints is scarce. The UML profile for scheduling, performance and time (SPT) describes scheduling policies according to the underlying deployment model [17]. Using SPT, the Core Scenario Model (CSM) [30] is informally defined to generate performance models from UML. However, CSM is not executable as it only identifies a subset of the possible system behaviors [30]. Verhoef's extension of VDM++ for embedded real-time systems [32] is based on abstract executable specification and models static deployment of fixed resources targeting the embedded domain, namely CPUs and buses.

Related work on simulation tools for cloud computing is mostly reminiscent of network simulators. Testing techniques and tools for cloud-based software systems are surveyed in [5]. In particular, CloudSim [14] and ICanCloud [29] are simulation tools using virtual machines to simulate cloud environments. CloudSim is a mature tool which has been used for a number of papers, but it is restricted

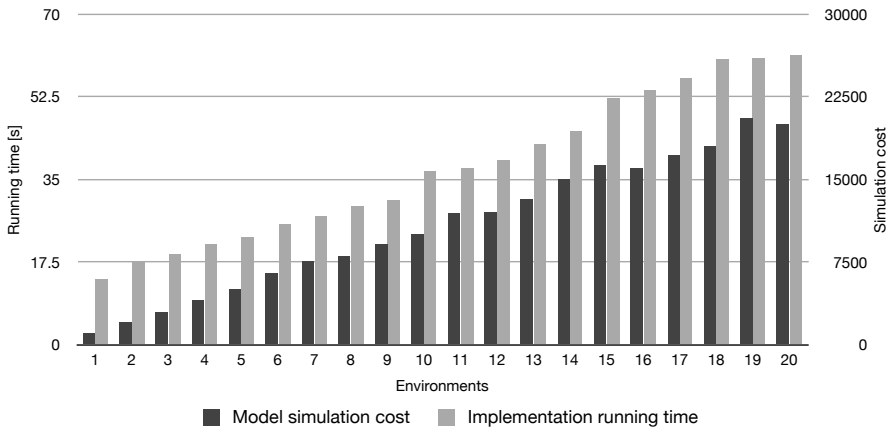

Fig. 9. Comparison of the measured execution time of the implementation (left scale) and the accumulated cost of the simulation for the as-needed policy (right scale).

to simulations on a single computer. In contrast, ICanCloud supports distribution on a cluster. EMUSIM [13] is an integrated tool that uses AEF (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 aims to support the developer of client applications for cloud-based environments at an early phase in the software engineering process and is based on a formal semantics.

In software design, no general, systematic means exists today to model and analyze software in the context of a set of available virtualized resources, nor to analyze resource redistribution in terms of load balancing or reflective operations. None of the cited works directly address the challenges raised by virtualization; in particular, they do not model quantitative resources as data inside the system itself, which is a particular property of virtualized resources.

6 Conclusion and Future Work

In this paper we demonstrated that it is possible to model *low-level* software aspects, including performance, resource consumption, and deployment, in a suitably *abstract* way which is adequate for *cloud computing*. As an immediate benefit this makes it possible to perform comprehensive simulations based on the system *model* that allow to predict and evaluate the consequences of different scheduling, load balancing, or deployment strategies. We demonstrated the feasibility of our approach by modeling part of an industrial e-Commerce product and by comparing the simulated model to the actual code in production.

The modeling in this paper is based on a resource-aware extension of the *abstract behavioral specification* language ABS [8, 23]. ABS has a formal semantics

and was designed such that the models expressed in it are mechanically *analyzable* with respect to correctness, resource consumption, security, etc. Specifically, it is possible to *automatically* compute symbolic worst-case bounds for resource consumption of ABS programs [2]. We plan to generalize this approach to the extension of ABS used here and apply it to cloud scenarios. This would make it possible to automatically analyze the worst-case resource consumption of programs running in the cloud without actually deploying them.

References

1. G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems.* 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. Partial Evaluation and Program Manipulation (PEPM'12)*, pp. 151–154. ACM, 2012.
3. T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: A tool for schedulability analysis and code generation of real-time systems. In *Proc. Formal Modeling and Analysis of Timed Systems*, *LNCS* 2791, pp. 60–72. Springer, 2003.
4. J. Armstrong. *Programming Erlang.* Pragmatic Bookshelf, 2007.
5. X. Bai, M. Li, B. Chen, W.-T. Tsai, and J. Gao. Cloud testing tools. In *Proc. Service Oriented System Engineering (SOSE'11)*, pp. 1–12. IEEE, 2011.
6. C. Baier, B. R. Haverkort, H. Hermanns, and J.-P. Katoen. Performance evaluation and model checking join forces. *Comm. ACM*, 53(9):76–85, 2010.
7. F. Barbanera, M. Bugliesi, M. Dezani-Ciancaglini, and V. Sassone. Space-aware ambients and processes. *TCS*, 373(1–2):41–69, 2007.
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. To appear, available online.
9. H. C. Bohnenkamp, P. R. D'Argenio, H. Hermanns, and J.-P. Katoen. MODEST: A Compositional Modeling Formalism for Hard and Softly Timed Systems. IEEE Trans. Software Eng., 32(10): 812–830, 2006.
10. P. Bouyer, U. Fahrenberg, K. G. Larsen, and N. Markey. Quantitative analysis of real-time systems using priced timed automata. *Comm. ACM*, 54(9):78–87, 2011.
11. E. J. Bruno and G. Bollella. *Real-Time Java Programming: With Java RTS.* Prentice Hall, 2009.
12. 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.
13. 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*, 2012. To appear, available online.
14. 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.
15. D. Caromel and L. Henrio. *A Theory of Distributed Objects.* Springer, 2005.
16. A. Chander, D. Espinosa, N. Islam, P. Lee, and G. C. Necula. Enforcing resource bounds via static verification of dynamic checks. *ACM TOPLAS*, 29(5), 2007.

17. B. P. Douglass. *Real Time UML – Advances in the UML for Real-Time Systems*. Addison-Wesley, 2004.
18. H. Foster, W. Emmerich, J. Kramer, J. Magee, D. S. Rosenblum, and S. Uchitel. Model checking service compositions under resource constraints. In *Proc. European Software Engineering Conf. and Intl. Symp. on Foundations of Software Engineering (ESEC/FSE'07)*, pp. 225–234. ACM, 2007.
19. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proc. Formal Methods for Components and Objects (FMCO 2010)*, *LNCS* 6957, pp. 142–164. Springer, 2011.
20. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
21. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Dynamic resource reallocation between deployment components. In *Proc. Formal Engineering Methods (ICFEM'10)*, *LNCS* 6447, pp. 646–661. Springer, 2010.
22. 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)*. To appear in *LNCS*, Springer 2012.
23. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Validating timed models of deployment components with parametric concurrency. In *Proc. Formal Verification of Object-Oriented Software*, *LNCS* 6528, pp. 46–60. Springer, 2011.
24. J. Kramer. Is abstraction the key to computing? *Comm. ACM*, 50(4):36–42, 2007.
25. 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.
26. E. A. Lee. Computing needs time. *Comm. ACM*, 52(5):70–79, 2009.
27. L. Moreau and C. Queinnec. Resource aware programming. *ACM TOPLAS*, 27(3):441–476, 2005.
28. M. Netjes, W. M. van der Aalst, and H. A. Reijers. Analysis of resource-constrained processes with Colored Petri Nets. In *Proc. Practical Use of Coloured Petri Nets and CPN Tools (CPN 2005)*, *DAIMI* 576. University of Aarhus, 2005.
29. 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. 10.1007/s10723-012-9208-5.
30. 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.
31. M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice Petri nets. In *Proc. Design Automation Conference (DAC'99)*, pp. 805–810. ACM, 1999.
32. 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, pp. 147–162. Springer, 2006.
33. A. Vulgarakis and C. C. Seceleanu. Embedded systems resources: Views on modeling and analysis. In *Proc. Computer Software and Applications Conference (COMPSAC'08)*, pp. 1321–1328. IEEE, 2008.
34. P. Y. H. Wong, N. Diakov, and I. Schaefer. Modelling Distributed Adaptable Object Oriented Systems using HATS Approach: A Fredhopper Case Study. In *2nd Intl. Conf. on Formal Verification of Object-Oriented Software*, *LNCS* 7421, Springer, 2012.