SOCA (2014) 8:323–339
DOI 10.1007/s11761-013-0148-0

SPECIAL ISSUE PAPER

# Formal modeling and analysis of resource management for cloud architectures: an industrial case study using Real-Time ABS

**Elvira Albert · Frank S. de Boer · Reiner Hähnle · Einar Broch Johnsen · Rudolf Schlatte · S. Lizeth Tapia Tarifa · Peter Y. H. Wong**

Received: 7 January 2013 / Revised: 14 June 2013 / Accepted: 5 November 2013 / Published online: 1 December 2013

**Abstract** We demonstrate by a case study of an industrial distributed system how performance, resource consumption, and deployment on the cloud can be formally modeled and analyzed using the abstract behavioral specification language Real-Time ABS. These non-functional aspects of the system are integrated with an existing formal model of the functional system behavior, achieving a separation of concerns between the functional and non-functional aspects in the integrated model. The resource costs associated with execution in the system depend on the size of local data structures, which evolve over time; we derive corresponding worst-case cost estimations by static analysis techniques and integrate them into our resource-sensitive model. The model is further 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, analyze, and compare deployment scenarios on the cloud.

**Keywords** Virtualization · Cloud computing · Formal methods · Abstract behavioral specification · Executable modeling · Worst-case cost analysis · Static analysis · Hybrid validation

## 1 Introduction

Virtualization is a key technology enabler for cloud computing. Although the added value and compelling business drivers of cloud computing are undeniable [19], this new paradigm also poses considerable new challenges that have to be addressed to render its usage effective for industry. Virtualization gives services at the application-level access to elastic amounts of resources; for example, the processing capacity allocated to a service may be changed according to demand.

Current software development methods, however, do not support the development of virtualized software in a satisfactory way; in particular, they do not help developers model and validate application-level resource management strategies for virtualized resources early in the development process. For example, the concrete and precise cost and resource requirements needed to deploy a service in the cloud needs to be estimated manually on a case by case basis depending on each customer's requirements and expectations. Whereas some cloud providers already offer functionality to dynam-

Partly funded by the EU projects FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu) and FP7-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu).

E. Albert
DSIC, Complutense University of Madrid, Madrid, Spain
e-mail: elvira@fdi.ucm.es

F. S. de Boer
CWI, Amsterdam, The Netherlands
e-mail: f.s.de.boer@cwi.nl

R. Hähnle
Technical University of Darmstadt, Darmstadt, Germany
e-mail: haehnle@cs.tu-darmstadt.de

E. B. Johnsen (✉) · R. Schlatte · S. L. Tapia Tarifa
Department of Informatics, University of Oslo, Oslo, Norway
e-mail: einarj@ifi.uio.no

R. Schlatte
e-mail: rudi@ifi.uio.no

S. L. Tapia Tarifa
e-mail: sltarifa@ifi.uio.no

P. Y. H. Wong
SDL Fredhopper, Amsterdam, The Netherlands
e-mail: peter.wong@fredhopper.com

ically manage resource allocation, this functionality is not reflected in current design processes. These issues seriously restrict the potential for fine-tuning a service to the available virtualized resources during the design phase and force the developer to introduce and validate resource management for the service after the development of the overall service logic and control flow. In the worst case, faulty provisioning can make it necessary to redesign an application, which is extremely expensive when discovered after deployment. This article shows how to overcome these limitations by integrating resource management into a formal, yet realistic, model that permits simulation and analysis during service *design time*.

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 abstract behavioral specification language ABS [30]. ABS is *object-oriented* to stay close to high-level programming languages and to be easily usable as well as accessible to software developers, it is *executable* to support full code generation and (timed) validation of models, and it has an operational *formal semantics*. The resource analysis tool used in our work is based on that formal semantics. This guarantees that the static analysis of the worst-case resource consumption of ABS models is sound for any input. In consequence, it is actually possible to (automatically) generate formal correctness proofs for the obtained resource bounds [6].

Real-Time ABS extends ABS with time and with primitives for leveraging resources and their dynamic management to the abstraction level of software models [15,33]. 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 and analysis 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.

### 1.1 Summary of contributions

The overall contribution of this article is a large industrial case study, which demonstrates that it is possible to model aspects of performance, resource consumption, and deployment on the cloud *at design time* based on Real-Time ABS. This article refines and substantially extends a paper published at ESOCC 2012 [24], which focused on the modeling and validation of the case study for average costs by means of simulation; in the present article, we take worst-case costs into account and apply *static* resource analysis and *hybrid* validation to the case study. The main contributions of the presented work are as follows. The last two points are original contributions of this article.

- *Modeling* 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 of the case study. The ABS model is parameterized over deployment scenarios, which capture different application-level management policies for virtualized resources.
- *Validation* 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 [34] details the modeling of the cloud provider and compares our approach to results obtained by simulation tools.
- *Analysis* For the first time, resource analysis [3] has been applied to a case study of industrial size. In particular, we have been able to analyze statically a large fragment of the model whose Java implementation included an *undetected hot spot* in the program. We succeeded to infer the worst-case resource consumption of this part of the code, and the formal analysis explains precisely why its resource consumption is high.
- *Hybrid validation* We have *combined* dynamic and static analysis to validate the model in a novel way. In particular, we have used the resource bound inferred by static resource analysis during the *timed* simulation of deployed, concurrent models. In previous work [7], the results of resource analysis had merely been used combined with dynamic analysis to validate the untimed behavior of sequential fragments of code for small examples.

### 1.2 Organization of the article

The article is organized as follows: In Sect. 2, we motivate and describe the case study. Section 3 presents the modeling layers of Real-Time ABS, starting with the functional and imperative layers before we explain how deployment scenarios are modeled in Real-Time ABS. Section 4 describes the modeling of the case study and proceeds with resource analysis and calibration of the model, before we combine the obtained worst-case bounds with simulation techniques in a hybrid validation approach to the case study. Section 5 discusses related work, and Sect. 6 concludes the article.

## 2 The case study: background and motivation

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 [48] for a detailed presentation of the individual components of FAS and its deployment model.

For the purposes of this case study, 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. The *staging environment* is responsible for receiving data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments. The distribution of indexed data is done according to a *Replication Protocol*, which is implemented by the *Replication System*. The Replication System consists of a SyncServer at the staging environment and one SyncClient for each live environment. The SyncClient initially performs a Boot job to identify Replication jobs that are relevant for its live environment and creates listeners for these Replication jobs. Replication jobs may relate to the *search* index, *business rules* (e.g., presentation and promotions), and *data* (e.g., navigation). The SyncServer determines the schedule and content of different Replication jobs, while a SyncClient receives data and configuration updates.

The SyncServer communicates to a SyncClient by creating ConnectionThread objects. These objects serve as the interface to the server side of the Replication Protocol. A ConnectionThread object delegates actual data replication to its Handler via the Acceptor. A SyncClient, on the other hand, schedules and creates ClientJob objects to handle communications to the client side of the Replication Protocol; thus, ClientJob objects on a client-side live environment recursively realize the replication schedules computed by the ConnectionThread objects on the server side staging environment. Figure 1 shows a UML sequence diagram which illustrates the replication protocol between a ClientJob, a ConnectionThread, and an Acceptor. In this article, 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 client 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 grow-

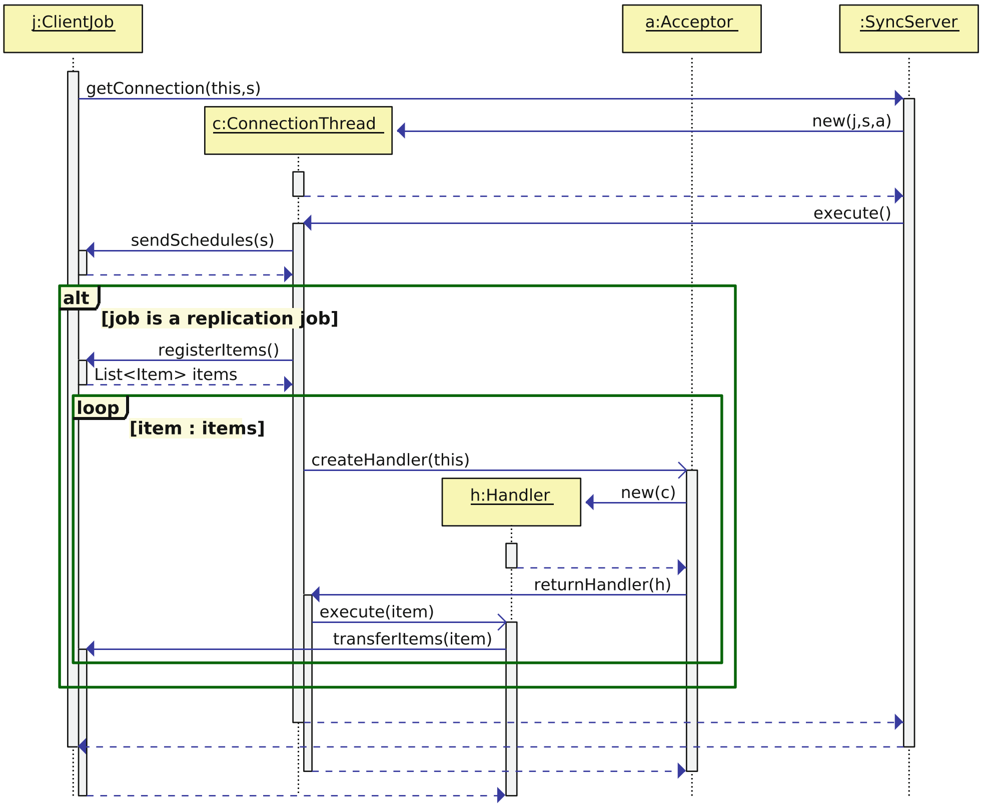

**Fig. 1** The interaction between the ClientJob, the Acceptor, the Handler, and the ConnectionThread

1. SyncServer starts by listening for connections from SyncClient objects.
2. SyncClient creates a ClientJob object with a Boot job, which connects to the SyncServer. This corresponds to the message getConnection(**this**,s) in Figure 1, where s is a schedule.
3. SyncServer creates a ConnectionThread to communicate with this ClientJob.
4. The ClientJob asks ConnectionThread for *all replication schedules*. 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 SyncClient objects. 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, denoted by the message sendSchedule(s) in Figure 1, and creates new ClientJob objects representing the different 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 immediately connects to the SyncServer.
7. SyncServer creates a ConnectionThread to communicate with each ClientJob.
8. A ClientJob receives its replication schedule from the ConnectionThread 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 terminates.
9. The ConnectionThread first sends a replication schedule to the ClientJob according to the ClientJob's replication type, the ConnectionThread then sends the message registerItems to the ClientJob to acquire which sets of files to update. For each set of files to be updated, the ConnectionThread creates a Handler via the Acceptor to send the message(s) transferItem(item) (See Figure 1). After sending the file updates, the ConnectionThread terminates.

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

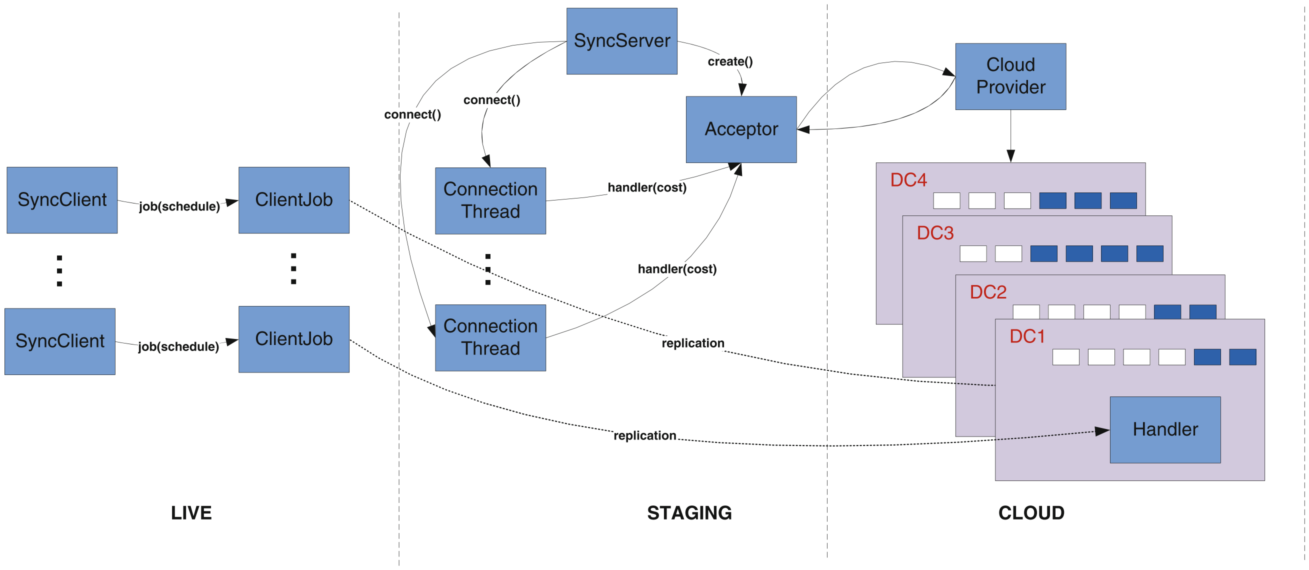

**Fig. 3** An on-demand deployment architecture for the Replication System using Cloud resources

ing demand on throughput and update frequency. This is visible in particular when customers experience drastic increases on throughput and data updates at certain time periods. For example, retail customers typically expect large throughput during seasonal sales. In these periods, considerably larger amounts of purchases are made and stock units need to be updated very frequently, in order to avoid that customers receive incorrect information. Higher throughput requires a larger number of live environments and, in turn, a larger number of SyncClient objects. On-premise deployment cannot cope with varying, on-demand requirements without large up-front investments in hardware which remains unused most of the time.

To cater for these requirements, FAS is today 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. Figure 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 allocate resources to each replication job based on the *cost* and the *deadline* of the replication to be conducted by the corresponding ClientJob object.

## 3 Modeling virtualized resources and deployment in Real-Time ABS

ABS is an abstract, executable, object-oriented modeling language with a formal semantics [30], targeting distributed systems. ABS is based on concurrent object groups (COGs), akin to concurrent objects [22,31], Actors [1], and Erlang processes [10]. 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 time [15]; the execution time can be specified directly in terms of durations (as in, e.g., UPPAAL [37]), but it can also be *implicit* and just *observed* by measurements of the executing model. With implicit time, no assumptions about execution times are hard-coded into the models. Instead, the execution time of a method call depends on how quickly the call is effectuated by the server object. In fact, the execution time of a statement then varies with the *capacity* of the chosen deployment architecture and on *synchronization* with (slower) objects; *similar calls to the same method do not always take the same amount of time.*

Below, we first briefly introduce behavioral modeling in ABS, which combines a functional and an imperative layer, and then explain the artefacts introduced for deployment modeling. The explanations are kept at an intuitive and informal level, with a focus on intuitions rather than the underlying formal semantics.

### 3.1 Behavioral modeling in ABS

ABS combines functional and imperative programming styles with a Java-like syntax. COGs execute in parallel and communicate through asynchronous method calls. However, the 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.

### *3.1.1 The functional layer*

The functional layer of ABS consists of algebraic data types such as the empty type Unit, booleans Bool, and integers Int; parametric data types such as sets Set<X> and maps Map<X> (for a type parameter X); and functions over values of these data types, with support for pattern matching.

The syntax of the functional layer of ABS is given in Fig. 4. The ground types $T$ are interfaces $I$, type names $D$, and instantiated parametric data types $D\langle\overline{A}\rangle$. Parametric data types $A$ allow type names to be parameterized by type variables $X$. User-defined data types definitions $Dd$ introduce a name $D$ for a new data type, parameters $\overline{A}$, and a list of constructors $Cons$. User-defined function definitions $F$ have a return type $A$, a name $fn$, possible type parameters, a list of typed input variables $x$, and an expression $e$. Expressions $e$ are variables $x$, values $v$, constructor, functional, and case expressions, or the expressions **this, now(), destiny()**, and **deadline()**. Values $v$ are constructors applied to values, or **null**. Case expressions match an expression $e$ to a list of case branches $br$ on the form $p \Rightarrow e$ which associate a pattern $p$ with an expression $e$. Branches are evaluated in the listed order, the (possibly nested) pattern $p$ includes an underscore which works as a wild card during pattern matching; variables in p are bound during pattern matching and are in the scope of the branch expression $e$. ABS provides a library with standard data types such as booleans, integers, sets, and maps, and functions over these data types.

$$
\begin{aligned}
T &::= I \mid D \mid D\langle\overline{T}\rangle\\
A &::= X \mid T \mid D\langle\overline{A}\rangle\\
Dd &::= \textbf{data}\ D[\langle\overline{A}\rangle] = [\overline{Cons}];\\
Cons &::= Co[(\overline{A})]\\
F &::= \textbf{def}\ A\ fn[\langle\overline{A}\rangle](\overline{A\ x}) = e;\\
e &::= x \mid v \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \textbf{case}\ e\ \{\overline{br}\}\\
&\quad\mid \textbf{this} \mid \textbf{now}(\,) \mid \textbf{destiny}() \mid \textbf{deadline}(\,)\\
v &::= Co[(\overline{v})] \mid \textbf{null}\\
br &::= p \Rightarrow e;\\
p &::= \_ \mid x \mid v \mid Co[(\overline{p})]
\end{aligned}
$$

**Fig. 4** Syntax for the functional layer of Real-Time ABS. Terms $\overline{e}$ and $\overline{x}$ denote possibly empty lists over the corresponding syntactic categories, and square brackets [ ] optional elements

The functional layer of ABS can be illustrated by considering *polymorphic sets* defined using a type variable X and two constructors EmptySet and Insert:

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

Two functions contains, which checks whether an item el is an element in a set set, and take, which selects an element from a non-empty set set, can be defined by pattern matching over set:

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

def X take<X>(Set<X> set) =
 case set {
   Insert(e, _) => e; };
```

Similarly, a data type Resource can be defined to model CPU capacity, which may be either unrestricted, expressed by the constructor InfCPU, or restricted to r resources, expressed by the constructor CPU(r) where r represents the amount of available resources.

```
data Resource = InfCPU | CPU(Int capacity);
```

Here, an *observer function* capacity is automatically defined for the constructor CPU, such that capacity(CPU(r)) returns r. It is not defined for InfCPU.

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. Using the built-in function timeDifference to compare two values of type Time, we can for example define a function to give us the elapsed time since a given starting time start, as follows:

```
def Rat elapsed(Time start) = timeDifference(now(),start);
```

### 3.1.2 The imperative layer

The imperative layer of ABS addresses concurrency, communication, and synchronization at the level of objects, and defines interfaces, classes, and methods. In contrast to mainstream object-oriented languages, ABS does not have an explicit concept of thread. Instead the thread of execution is unified with the COGs as the units of concurrency and distribution, which eliminates race conditions in the models. Objects in a COG are *active* in the sense that their run method, if defined, gets called upon creation.

The syntax of the imperative layer of ABS is given in Fig. 5. A program $P$ lists interface definitions $IF$ and class definitions $CL$, and has a main block $\{[\overline{T}\ \overline{x};]\ s\}$ where the variables $x$ of types $T$ are in the scope of the statement $s$. Interface and class definitions, as well as signatures $Sg$ and method definitions $M$, are as in Java. Below we focus on explaining the asynchronous communication and suspension mechanisms of ABS.

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).

$$
\begin{array}{rcl}
P & ::= & \overline{IF}\ \overline{CL}\ \{[\overline{T}\ \overline{x};]\ s\} \\
IF & ::= & \textbf{interface}\ I\ \{\ [\overline{Sg}]\ \} \\
CL & ::= & \textbf{class}\ C\ [(\overline{T}\ \overline{x})]\ [\textbf{implements}\ \overline{I}]\ \{\ [\overline{T}\ \overline{x};]\ \overline{M}\} \\
Sg & ::= & T\ m\ ([\overline{T}\ \overline{x}]) \\
M & ::= & Sg\ \{[\overline{T}\ \overline{x};]\ s\} \\
g & ::= & b \mid x? \mid \textbf{duration}(e,e) \mid g \wedge g \\
s & ::= & s; s \mid \textbf{skip} \mid \textbf{if}\ b\ \{\ s\ \}\ [\textbf{else}\ \{\ s\ \}] \mid \textbf{while}\ b\ \{\ s\ \} \mid \textbf{return}\ e \\
 & & \mid \textbf{duration}(e,e) \mid \textbf{suspend} \mid \textbf{await}\ g \mid x = rhs \\
rhs & ::= & e \mid cm \mid \textbf{new}\ [\textbf{cog}]\ C\ (\overline{e}) \\
cm & ::= & [e]!m(\overline{e}) \mid x.\textbf{get}
\end{array}
$$

**Fig. 5** Syntax for the imperative layer of Real-Time ABS

Concurrent object groups (COGs) locally sequentialize execution, resembling a monitor with release points but without explicit signaling. A COG can have at most one active process, executing in one of the objects of the COG. This active process can be unconditionally suspended by the statement **suspend**, adding this process to the queue of the COG, 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 $b$ conjoined with return tests **f?** on future variables f and with time-bounded suspensions **duration(e1,e2)** which become enabled between a best-case e1 and a worst-case e2 amount of time. Just like functional expressions, guards g are side effect free. Instead of suspending, the active process may *block* while waiting for a reply as discussed above, or it may block for some amount of time between a best-case e1 and a worst-case e2, using the syntax **duration(e1,e2)**. The remaining statements of Real-Time ABS are standard; e.g., sequential composition $s_1; s_2$, assignment x=rhs, and **skip, if, while**, and **return** constructs. Right-hand side 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, and future dereferencing **f.get**, in addition to the functional expressions e.

To illustrate how the two layers of Real-Time ABS can be combined, let us assume that an interface I implements a method m. We define a method timer which takes as parameter an object of interface I and returns the elapsed time for the method call to o, as follows:

```
Rat timer(I o){ Time start, Bool b;
  start = now(); b = o.m(); return elapsed(start);
}
```

## 3.2 Deployment modeling in ABS

The response time to a request in a distributed system depends not only on the size of the job requested, but also on the amount of available resources and on the usage policy for these resources, which are scattered around the deployment architecture of the distributed system. Deployment architectures express how distributed systems are mapped on physical and/or virtual media with many locations; the planning and validation of a deployment architecture to optimize performance imply determining the amount of necessary resources at the different locations as well as an optimal usage of these resources, such that the system fulfills its performance requirements.

Real-Time ABS lifts deployment architectures to the abstraction level of the modeling language, where the physical or virtual media are represented as *deployment compo-*

*nents*. In a Real-Time ABS model, different deployment components may have different bounds on the locally available resources.

Real-Time ABS introduces a separation of concerns between the *resource cost* of performing a computation and the *resource capacity* of a given deployment component. This separation of concerns between resource cost and resource capacity aids to model and validate different deployment scenarios at an early stage during the software development process. The focus in this article is on CPU resources in virtualized media; we use resource cost annotations to express the resource consumption during computation. Deployment components are discussed in Sect. 3.2.1 and resource consumption in Sect. 3.2.2.

### 3.2.1 Deployment components

A *deployment component* in Real-Time ABS captures the execution capacity of a location in the deployment architecture, on which a number of COGs can be deployed. The execution capacity is specified as the amount of resources which are available per accounting period; for simplicity, this accounting period is fixed in the semantics of Real-Time ABS and corresponds to the time intervals between integer values in the dense time domain of the language. The main block of a model executes in a root COG located on a default deployment component environment, with unrestricted processing capacity. To capture different deployment architectures, a model may be extended with other deployment components with different resource capacities. When COGs are created, they are by default allocated to the same deployment component as their creator, but they may also be allocated to a different deployment component. In contrast, an object which belongs to a COG will always be located on the same deployment components as its group. Thus, in a model without explicit deployment components, all objects execute in the default environment, which places no restrictions on the processing capacity of the model.

Deployment components are first-class citizens of Real-Time ABS. They may be passed around as arguments to method calls, and they support a number of methods. Deployment components may be created dynamically, depending on control flow, or statically in the main block of the model. This means that Real-Time ABS is expressive enough to model, e.g., that new deployment components are created by a provider, or that deployment components are requested from a provider by a resource-aware and scalable application, put to use, and later released. This is illustrated in detail by our case study in Sect. 2. Syntactically, deployment components in Real-Time ABS are manipulated in a way similar to objects. Variables which refer to deployment components are typed by an interface DC, and new deployment components are dynamically created as instances of class DeploymentComponent, which implements DC.

```
interface DC {
  Int total();
  Int load(Int n);
  Int transfer(DC target, Int amount);
}
```

**Fig. 6** The interface of deployment components

$$\begin{aligned} a &::= \text{Cost: } e \mid \text{DC: } e \mid a, a \mid \ldots \\ e &::= \textbf{thisDC}(\,) \mid \ldots \\ rhs &::= \textbf{new cog } \text{DeploymentComponent}\,(e, e) \mid \ldots \\ s &::= \textbf{movecogto}(e) \mid [a]\; s \mid \ldots \end{aligned}$$

**Fig. 7** Syntax for deployment modeling in Real-Time ABS

The interface DC, shown in Fig. 6, provides the following methods for resource management: **total()** returns the number of resources currently allocated to the deployment component, **load(n)** returns the deployment component's average load in percentage during the last n accounting periods (i.e., the used compared to the total number of resources for each accounting period in the window), and **transfer(target,r)** reallocates r resources from the current deployment component to a target deployment component. If the former has less than r resources available, the available amount is transferred (and provided as the return value of the call).

Deployment can be expressed with a small syntactic extension to Real-Time ABS, shown in Fig. 7. Deployment components are created by the expression **new cog** DeploymentComponent(d,c). Here, the parameter c of type Resource specifies the initial CPU capacity of the deployment component. The parameter d of type String is a descriptor mainly used for monitoring purposes; i.e., it provides a user-defined name for the deployment component which facilitates querying the run-time state but that has no semantic effect. Statements are extended to support two kinds of *annotations*: deployment and cost annotations. Whereas cost annotations are explained in Sect. 3.2.2, deployment annotations relate to where new objects are located. By default, an object is deployed on the same deployment component as its creator. However, a different deployment component may be selected associating an optional deployment annotation [DC: e] to the object creation statement, where e is an expression of type DC. Note that deployment annotations can only occur associated with the creation of COGs.

An object may relocate the COG and it belongs to a different deployment component e by executing the statement **movecogto(e)**. This is well defined, because at most one object can be active in a COG at any given time. Since all objects are deployed on some deployment component, we let the expression **thisDC()** refer to the deployment component where the object is currently deployed, similar to the standard self-reference this in object-oriented languages.

### 3.2.2 *Resource consumption*

The available resource capacity of a deployment component determines the amount of computation which may occur in the objects deployed on that deployment component. Objects allocated to the deployment component compete for the shared resources in order to execute, and they may execute until the deployment component runs out of resources or they are otherwise blocked. For the case of CPU resources, the resources of the deployment component define its capacity inside an accounting period, after which the resources are renewed.

The resource consumption of executing statements in the Real-Time ABS model is determined by a default cost value which can be set as a compiler option (e.g., -defaultcost=10). However, the default cost does not discriminate between the statements, so a more refined cost model will often be desirable. For example, in a realistic model, the assignment x=e should have a significantly higher cost for a complex expression e than for a constant. For this reason, more fine-grained costs can be inserted into Real-Time ABS models by means of *cost annotations* [Cost: e] (see Fig. 7). Note that cost annotations can be associated with any statement and that a statement may have several annotations.

It is the responsibility of the modeler to specify appropriate resource costs. A behavioral model with default costs may be gradually transformed to provide more realistic resource-sensitive behavior by inserting such cost annotations. The manual estimation of resource cost is time consuming and error-prone. Therefore, it is desirable to have tool support for this activity. COSTABS [3] is an automated static analysis tool that is able to compute a worst-case approximation of the resource consumption of the non-virtualized programs, based on static analysis techniques. In the sequel, we apply COSTABS to obtain such worst-case cost expressions for the parts of the model that is deployed on virtual machines and use these expressions in our cost annotations (see Sect. 4.1).

However, the modeler may also want to capture *normative* constraints on resource consumption, such as resource limitations, at an abstract level; these can be made explicit in the model during the very early stages of the system design. To this end, cost annotations may be used by the modeler to abstractly represent the cost of some computation which is not fully specified.

## 4 Case study: The ABS model

The Replication System, introduced in Sect. 2, is part of the 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 [48]. The model was manually constructed. The automatic extraction of models from Java implementations has been studied for certain restricted kinds of distributed Java applications in [8]. However, the generalization of such automatic extraction techniques to standard Java applications would be much more difficult, mainly due to the differences in the concurrency models of Java and ABS.

In our case study, the model was re-engineered from the existing Java code, i.e., in hindsight. This is useful to document and to analyze legacy code, as done here. But ABS models can also be created and profitably used during the design stage to find out the consequences of design decisions in the concurrency architecture of an implementation at an early stage. ABS models can be created by anyone who has some familiarity with programming and the basics of concurrency. In particular, it is not necessary to know low-level concurrency aspects such as when programming in Java or C++.

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).

Figure 8 shows the main interfaces. The interface ClientJob declares two methods. The task of method executeJob() is to execute replication schedules, and method size() returns the size of the client job's underlying file structure. Schedul-

```
interface ClientJob {
  Unit executeJob();
  Int size();
}

interface SyncClient {
  Unit scheduleJob(Schedule s);
}

interface ConnectionThread { }
interface Handler {
  Unit transfer(Set<File> files);
}

interface Acceptor {
  Handler createHandler(ClientJob job, Int cost);
  Unit finish(Handler h);
  Unit end();
}

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

**Fig. 8** Interfaces of the Replication System

ing of a given replication schedule s is done with method scheduleJob(Schedule s) declared in interface SyncClient.

Objects of type ConnectionThread do not receive method calls, so the interface ConnectionThread models its objects as active objects (without methods). The interface Handler declares the method files that is responsible for file updating. The Handler objects are created by objects of class ConnectionThread by invoking the createHandler(job,cost) method of the Acceptor. On method invocation, the acceptor creates handler objects on virtual machines which are acquired from a CloudProvider by the method createMachine(capacity). The methods acquireMachine(vm), releaseMachine(vm) are used to start and stop virtual machines (modeled by deployment components) to let replication schedules be conducted by ConnectionThread objects. For presentation purposes, we focus here on the interface implementations given in the classes CloudProvider, Acceptor, and ConnectionThread.

The CloudProvider interface (shown in Fig. 8) 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 can be retrieved with the help of the method getAccumulatedCost() at any time during execution. The 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 the method acquireMachine(vm) starts the accounting for machine vm and a call to the method releaseMachine(vm) stops the accounting for vm. Inside the cloud provider, an active run() method ensures that the accounting is done for every accounting period. To focus on 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 (one possible implementation of the CloudProvider interface in Real-Time ABS is given in [34]).

We model and compare three potential load balancing strategies offered by different implementations of the Acceptor interface, for the application-level management of virtualized resources. When an Acceptor receives requests for file updates from ConnectionThread objects, it deploys Handler objects on cloud instances to conduct file updating with the ClientJob objects. The implementations of Acceptor reflect different strategies for interacting with the cloud provider to achieve application-level resource management:

- *Constant balancing* simply deploys all the objects of type ConnectionThread on a single virtual machine sufficient for the expected load and keeps this machine running;
- *As-needed balancing* calculates the needed CPU capacity of the virtual machine for a specific replication schedule with a given deadline and deploys Handler objects to a machine supplying the required 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 the book-keeping for a cloud user account (see Fig. 9). The implementation AccountImpl maintains a data structure in the field instances, which sorts the available virtual machines by CPU processing capacity. Furthermore, the class keeps track of the current cost per time unit costPerTimeUnit for the cloud user account, and the observed start-up time instanceStartUpTime of the most recent virtual machine to start up. The method getInstance(size) either requests a new virtual machine from the cloud provider or brings online an existing offline machine of the appropriate size. The method dropInstance(d) takes the machine d offline when it is no longer active.

The load balancing strategies are defined in classes implementing the Acceptor interface (two of these implementations are detailed in Fig. 10), as follows:

*Constant balancing* over-provisions by processing all replication schedules on a single virtual machine with sufficient capacity. Its implementation is shown in Fig. 10 (class ConstantAcceptor). The acceptor initially requests a single

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

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

  DC getInstance(Int size) {
    DC d = null;
    Time t = now();
    costPerTimeUnit = costPerTimeUnit + size;
    if (hasSetFor(instances, size)) {
      d = takeOne(lookup(instances, size));
      instances = removeFrom(instances, 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;
    instances = addToSet(instances, size, d);
  }
}
```

**Fig. 9** The cloud user account

```
class ConstantAcceptor(Int instanceSize, Account acc)
implements Acceptor {
  DC dc = null;
  Unit run() { dc = acc.getInstance(instanceSize); }
  Handler createHandler(ClientJob job, Int cost) {
    await dc != null;
    [DC: dc] Handler h = new cog HandlerImpl(job, cost);
    return h;
  }
  Unit finish(Handler h) { }
  Unit end() { acc.dropInstance(dc);
  }
}

class BudgetAcceptor(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);}
  }
  Handler createHandler(ClientJob job, Int cost) {
    Int dur = durationValue(deadline());
    Int startUp = acc.getLastInstanceStartUpTime();
    Int wanted = (cost / dur) + 1 + startUp;
    Int maxresource =
      (budgetPerTimeUnit − costPerTimeUnit)
      + (max(availableBudget, 0) / dur);
    Handler handler = null;
    if (maxresource > 0) {
      DC dc = acc.getInstance(min(wanted, maxresource));
      [DC: dc] Handler h = new cog HandlerImpl(job, cost);}
    return h;
  }
  Unit finish(Handler handler) {
    Fut<DC> fdc = handler!release();
    await fdc?; DC instance = fdc.get;
    acc.dropInstance(instance);
  }
  Unit end() { }
}
```

**Fig. 10** The classes ConstantAcceptor and BudgetAcceptor, which implement the *constant* and *budget-aware* strategies

machine 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* is a strategy where the acceptor has a given *budget per accounting period* and may "save resources" for later (class BudgetAcceptor in Fig. 10). The class parameter budgetPerTimeUnit provides this budget, and the field availableBudget keeps track of the accumulated (saved) unused resources, according to the budget. When the acceptor receives a request from a ClientJob object, it calculates the resources needed to fulfill the schedule in wanted 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 accounting period. It also maintains a log budgetHistory of the available resources over time.

## 4.1 Resource analysis and calibration

Our initial approach to providing cost annotations for the ABS model of the Replication System was to average measurements of the execution time of client jobs for the different replication schedules on the Java implementation (reported in [24]). In particular, we were interested in jobs specified by the following two types of replication schedules:

- *Search* A replication job originating from the search schedule replicates changes from the search index, i.e., the underlying data structure providing search capability on a customer's product items; and
- *Data* A replication job initiated by the data schedule replicates changes concerning the item and navigation indices, i.e., the core index structures and data model for providing navigation on a customer's product items.

During the calibration of the simulation constants, we identified a *hot spot* in the Java implementation of the Replication System and located it to be in the method transferItem(fileset) of ConnectionThread. In ABS this corresponds to the method transferItem of the Handler interface, implemented by the class HandlerImpl shown in Fig. 11. A hot spot denotes a region of a program where a high proportion of executed instructions occur or where most time is spent during the program's execution.

In our measurements, this method accounts for 99 % of the execution time. The hot spot justifies adding a single cost annotation to the corresponding method in the ABS model. As we aim to provide accurate simulation results at the modeling level, we make an in-depth analysis of the hot spot in our model and use the COSTABS tool [3] to *statically analyse* the resource consumption of transferItem(fileset).

COSTABS is a cost analysis tool that, given an input method, is able to automatically infer a sound *upper bound* on its resource consumption. In other words, the analysis guarantees that the execution of the method will never exceed the inferred amount of resources for any input data.

```
class HandlerImpl(ClientJob job, Int cost) implements Handler {
  Unit transfer(Set<File> fileset) {
     [Cost: cost] this.transferItem(fileset);
  }
  Unit transferItem(Set<File> fileset) {...}
}
```

**Fig. 11** Class HandlerImpl

In Sect. 4.1.1–4.1.4, we describe in detail how we have used COSTABS to analyze the resource consumption of the transferItem(fileset) method and explain the different parameters that occur in the analysis. Using the derived cost expression, in Sect. 4.1.5, we describe how we synthesize other simulation parameters.

It is important to note that the derived cost expression is for a single request. Results for dynamic behavior of the environment, e.g., fluctuating demand or a load spike at a certain time, can be obtained by implementing these scenarios as part of the model and measuring the load on the deployment components. An example of implementing dynamic client scenarios in such a way can be found in [32].

### 4.1.1 Cost metrics

The first option to be selected in COSTABS is the *cost metric* (a.k.a. cost model) which specifies the type of resource we want to measure. COSTABS offers a wide range of cost models [2], including traditional cost models for measuring the number of execution steps and memory allocation, and also cost models specific to concurrent and distributed applications like the task-level of the program which estimates the peak of tasks that can be simultaneously spawned in the execution.

In our case study, we are interested in justifying the worst-case execution time of transferItem(fileset). For this purpose, we have selected the cost model that counts the number of execution steps. This includes both steps performed in the functional and steps in the imperative part of the ABS model. On the one hand, execution time is often directly related to the number of execution steps performed, although even for static deployment some other factors also clearly influence the execution time (see work on WCET [35]). On the other hand, we are also interested in understanding the computational complexity of the method transferItem. Thus, we need a cost model which assigns cost to all instructions of the program and does not ignore certain parts. This would not be achieved for instance with a cost model that infers memory consumption, since a loop that does not allocate memory has an associated resource consumption zero, while its computational complexity is not zero.

### 4.1.2 Size abstraction

The goal of the cost analysis is to infer closed-form upper bounds provided as functions on the data input *sizes*. When a program manipulates terms, its cost usually depends on the *size* of the terms. For instance, if a loop traverses a list, the cost of the loop often depends on its length. COSTABS relies on the notion of *norms* [14] to define the *size of a term*. Norms are functions that map terms to their sizes. Any norm can be used in the analysis, depending on the nature of the data structures used in the program. They can also be synthesized automatically from the program's type definitions. In what follows, we use the *term-size* norm, which counts the number of type constructors in a given term, defined as: $size(Co(t_1, \ldots, t_n)) = 1 + \Sigma_{i=1}^{n} size(t_i)$ and $size(x) = x$. Note that the size of a program variable $x$ is defined as $x$. In this way, we account for the size of the term to which $x$ is bound at runtime.

```
def Int sizeofFiles(Set<File> t) =
  case t {
    EmptySet => 1;
    Insert(f, fs) => 1 + sizeofFile(f) + sizeofFiles(fs);
  };

def Int sizeofFile(File t) =
  case t {
    Pair(fId,fContent) =>
      1 + sizeofFileId(fId) + sizeofFileContent(fContent);
  };

def Int sizeofFileId(FileId t) = strlen(t);
```

**Fig. 12** Selected size abstractions for Set of Files

Our target method transferItem(fileset) has a parameter of type Set<File>. Set is a predefined ABS type that can be EmptySet or Insert(f,fs), where f is of type File and fs of type Set<File>. The type File is defined as typeFile = Pair<FileId,FileContent> in ABS (we omit the definition of FileContent since it is not relevant). The functions provided in Fig. 12 define (part of) the size abstraction used by the analyzer for Set<File>. These functions will be used later by the simulator to evaluate the cost functions on specific input data.

### 4.1.3 Class invariants

When the cost depends on the size of data stored in fields, inferring cost often requires *class invariants* that provide guarantees on such fields. In general, such class invariants are a way to incorporate guarantees on the global states when the considered process is resumed. In our case study, the cost of executing the method transferItem(fileset) depends on a field rdir that has type Directory and is declared in class ClientDataBaseImp. It represents the directory which keeps all files whose content is to be transferred. Some operations carried out to transfer items traverse the directory rdir. To be able to infer an upper bound for the method, we need to specify that the field rdir is finite, i.e., bounded from above. This can be specified by means of the following class invariant [rdir <= max(rdir)] which states that field rdir has a maximum value that is denoted by max(rdir) in what follows. This invariant is the only manual input required to infer an upper bound for transferItem(fileset): the analysis is fully automatic otherwise.

### *4.1.4 Upper bounds*

After having selected the cost model and the size abstraction, and having provided the class invariant, we automatically infer the following (asymptotic) upper bound using COSTABS:

$$\mathsf{nat}(\mathsf{fileset})^2 * \mathsf{nat}(\mathsf{max}(\mathsf{rdir})) + \mathsf{nat}(\mathsf{fileset})^3$$

This upper bound is a polynomial of degree 3 on the argument fileset and the class field rdir. Function nat is defined as: "**def** Int nat(Int a) = **if** a > 0 then a **else** 0;" and used to avoid negative values of the upper bound expression.

Method transferItems has 28 lines of code. All such code is included in a loop that traverses the data structure fileSet (the method parameter) using an iterator. Hence, its cost is an expression of the form nat(fileSet)*(max_cost_iteration), where nat(fileSet) corresponds to the number of iterations of the loop (i.e., the size of the traversed data structure). As regards the worst-case cost of each iteration, denoted as max_cost_iteration, we have that 8 functions and 7 methods are invoked directly within such loop. Besides, such functions and methods invoke transitively others. Altogether, in order to obtain this upper bound, COSTABS has analyzed 37 methods and functions that were transitively invoked from transferItem(fileset). The worst-case cost of all such 15 methods and functions is accumulated together in order to obtain max_cost_iteration. In particular, one of the invoked methods has already an asymptotic complexity of nat(fileSet)*nat(max(rdir))+nat(fileSet)$^{2*}$. The other methods that are invoked have the same order or smaller complexity. Thus, when adding (asymptotically) all complexities of the invoked methods we have that max_cost_iteration=nat (fileSet)*nat(max(rdir))+nat(fileSet)$^2$, which is multiplied by nat(fileSet), resulting in the above upper bound.

The analysis has revealed that the execution time of the method is asymptotically larger than any other bound obtained for any of the remaining functions and methods. This explains the hot spot and gives clear directives for potential optimizations.

### *4.1.5 Model calibration*

The implementation of the ConnectionThread interface is sketched Fig. 13 along with the type signature of the cost function that is generated by COSTABS and that returns the asymptotic upper bound computed in the previous section of the number of execution steps inside transferItem(fileset).

```
def Int cost(Int sizeOfFileSet, Int sizeOfFileStructure) = ...;

class ConnectionThreadImpl(ClientJob job, Acceptor accpt,
  SyncServer server) implements ConnectionThread {
  Maybe<Command> cmd = Nothing;
  Set<Schedule> schedules = EmptySet;
  Unit run() {
    await cmd != Nothing;
    schedules = this.sendSchedule();
    if (cmd != Just(ListSchedule)) {
      ...
      while (hasNext(filesets)) {
        Pair<Set<Set<File>>,Set<File>> nfs = next(filesets);
        filesets = fst(nfs); Set<File> fileset = snd(nfs);
        Fut<Int> sf = job!size(); Int size = sf.get;
        accpt.createHandler(job, cost(sizeofFiles(fileset), size));
      }
      ...
    } } }
```

**Fig. 13** Class ConnectionThreadImpl

**Table 1** Derived interval and deadline parameters for simulation

| Schedule | Interval | Deadline |
|---|---|---|
| Search | 11 | 3 |
| Data | 11 | 11 |

Using the inferred cost expression, we extend the functional ABS model of the Replication System with annotations for resource and timing information. To determine suitable deadlines for the individual schedules and the intervals between each ClientJob, we calibrate the model by considering a reference scenario for which we aim to meet all deadlines. Here, we select as our reference scenario the case of one single typical live environment and set the capacity of the cloud instance such that it can successfully handle its client. We iteratively simulated a Replication System consisting of one SyncClient and a fixed number of replication jobs on a single cloud instance that offers 161600 CPU resources per time unit (corresponding to an accounting period in the model) and the interval to 11 time units between replication jobs. With these parameters, we determine the lowest deadlines of each type of schedules that can be met by all replication jobs, that is, 100 % quality of service (QoS). Table 1 also shows the results of this initial simulation.

## 4.2 Simulation results

Table 2 shows a comparison between the average execution time of replication jobs per type of schedule measured on the Java implementation of the Replication System and the average cost of simulated jobs in the ABS model using the inferred upper bound cost expression from COSTABS. We observe that jobs specified by the Search schedule are ca. 8.1 times cheaper than jobs specified by the Data schedule in the Java implementation while they are ca. 9.76 times cheaper in the ABS model. This is due to the fact that the inferred cost expression defines the *worst* cost instead of the average cost.

The following figures show the simulation results that we obtained for the different balancing strategies, varying over the number of live environments. Each simulation runs for

**Table 2** Comparison between execution time of the Java implementation of the Replication System against simulated cost

| Schedule | Execution time (s) | Simulated cost |
|---|---|---|
| Search | 34.0 | 90,682 |
| Data | 274.9 | 885,315 |

100 time units in different scenarios varying from 1 to 20 live environments (in reality, up to 20 environments are typically required to handle large query throughputs over a large number of product items). For each number of environments and for each balancing strategy, Fig. 14 shows the QoS as a percentage of replication jobs whose deadlines have been met and Fig. 15 shows the total accumulated cost. The results exhibit some of the brittleness of an overloaded system.

As explained in Sect. 4.1.5, we calibrated the ABS model by establishing a deployment component size for the constant and budget balancing strategies such that the Replication System was able to handle a scenario with one live environment without missing deadlines (100 % QoS). We then chose a size and a budget for the balancing strategies that would let us observe behavior both under normal load and overloaded scenarios. The system behavior differed remarkably for the three balancing strategies under higher load. As expected, the *as-needed* balancing strategy exhibits 100 % QoS, albeit with simulated cost rising linearly with the number of clients and ending orders of magnitude above the other two strategies.

On the other hand, to model the budget-aware balancing strategy, the BudgetAcceptor needs to be able to reject tasks (to stay within the budget). When the system is overloaded, this can be observed through a degraded QoS and, somewhat more surprisingly at first sight, by a decreasing overall cost. We observed that under high load, the BudgetAcceptor only managed to fit in jobs from the cheaper Search schedule, while rejecting most jobs of the more expensive Data schedule (see Table 2). It must be said that this scenario is not conducive for showing the advantages of the budget-aware strategy—in a scenario of permanent high load, this strategy cannot ever accumulate the surplus needed to handle momentary load spikes. On the other hand, below the maximum load, the budget-aware strategy performs as well as but much cheaper than the over-provisioning (constant) strategy, while being able to deal with transient higher loads.

Finally, we compared the simulation results to measurements obtained from the real system. We measured the execution time of individual replication schedules on the Java implementation of the Replication System on a reference machine (4 core CPU 2.5 GHz, 8 GB memory). The Replication System was configured to conduct search and data replications at every 11 s interval, while the staging environment was subjected to continuous data update, with the number of live environments ranging from one to twenty. I.e., both simulation and real system were running under identical replication load, so the simulated cost and measured execution time should correlate.

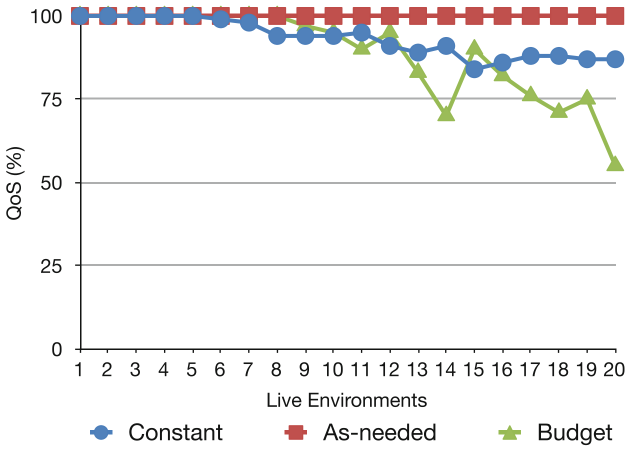

**Fig. 14** Simulation results: QoS, varying over the number of live environments

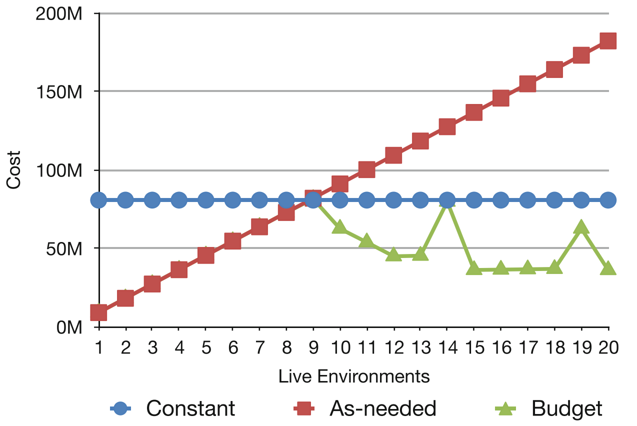

**Fig. 15** Simulation results: total cost, varying over the number of live environments

Figure 16 shows that both simulated cost and measured execution time grow linearly as expected. It can further be seen that the real system's execution time does not start from zero, which can be seen most clearly in the case of few live environments and is a result of the start-up time of the real system (Java initialization, etc.). 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. For a short-running system where this factor becomes important, it can trivially be incorporated into the model using a constant cost annotation at the beginning of execution.

## 4.3 Experience

- *Abstraction* When modeling the existing Java implementation of the Replication System using Real-Time ABS,

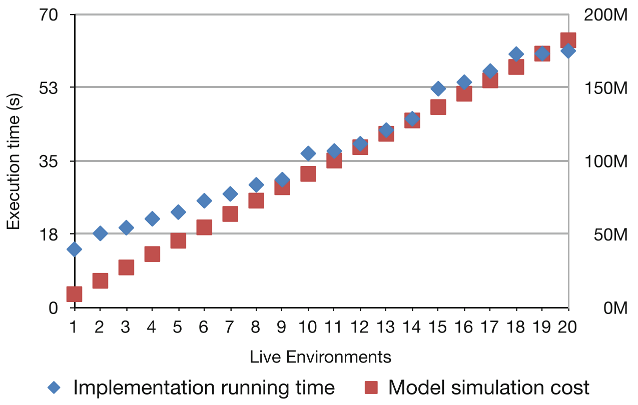

**Fig. 16** Comparison of the measured execution time of the implementation (*left scale*) and the accumulated cost of the simulation for the as-needed policy (*right scale*)

we have employed the abstraction mechanisms offered by Real-Time ABS. For example, we express disk-level and file operations as functions over algebraic data types while hardware constraints as deployment components.

- *Functional and OO* Given some practical experience with functional (e.g., Haskell) and object-oriented programming (e.g., Java), when modeling the Replication System, we were able to very quickly become proficient at both the functional and the object-oriented language layers of Real-Time ABS.
- *Concurrency* When modeling the Replication Protocol using Real-Time ABS, we have to extract the implicit multi-threaded behavior manifest from the combination of reentrant read write lock usage and subclassing to a cooperative scheduling model whereby all scheduling points are made explicit at the syntactic level. This modeling exercise increased our insight into the existing Java implementation.
- *Tool support* Real-Time ABS comes with an Eclipse plugin [47] that provides syntax highlighting, on-the-fly type checking, code completion, and implementation and type hierarchy navigation. These facilities have made modeling large systems such as the Replication System using Real-Time ABS practically possible and efficient.

## 5 Related work

To reduce complexity, general-purpose modeling languages strive for *abstraction* [36]: 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 technology, an application can *modify resource parameters of its deployment environment during execution*, e.g., it may dynamically create virtual processors. For cyber-physical and embedded systems, it is today accepted that modeling and programming languages need a timed semantics [38]. The Java Real-Time Specification (RTSJ) [18] 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., [3,23,45]. *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 [39].

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 [13], Petri Nets [43], and priced [17], timed [9], and probabilistic [12] automata and games (an overview of automata-based approaches is [45]). Related approaches are also applied to web services and business processes with resource constraints [27,40]. These approaches typically abstract away 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 [16] 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 article.

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 [25]. Using SPT, the Core Scenario Model (CSM) [42] 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 [42]. Verhoef's extension of VDM++ for embedded real-time systems [44] 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 [11]. In particular, CloudSim [21] and ICanCloud [41] 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 to simulations on a single computer. In contrast, ICanCloud supports distribution on a cluster. EMUSIM [20] 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.

Resource analysis [4,28,29,46] aims at estimating the resource usage of a given program and providing guarantees that the program will not exceed the inferred amount of resources. Typically, such analyses are either based on *monitoring* the program execution, or on *formal methods* that are able to infer this information *statically*, i.e., without executing the program on concrete data. Static approaches have a clear conceptual advantage over monitoring, since they provide guarantees that are valid for *any input*, not only for specific runs. There exist several powerful static resource analyzers (including COSTA [4], SPEED [28], RAML [29]); however, these have mainly dealt with traditional (i.e., non-real-time, non-distributed) applications. A recent extension [3] deals with the theoretical complications that distributed systems pose to static analysis and reports on a prototype implementation. In this article, for the first time, resource analysis has been applied to a case study of industrial size.

In software design, no general, systematic means exist 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 addresses the challenges raised by virtualization; in particular, they do not model quantitative resources as data inside the system itself, which is a key aspect of virtualized computations.

## 6 Conclusion and future work

Industry leaders in cloud computing research recently expressed “the need to describe your infrastructure requirements as executable code” (Thomas Schwindt, IBM Research Labs Zurich in [26]) as well as that “performance modeling is a future key technology” (Bryan Stephenson, HP Labs Palo Alto, also in [26]). These judgments align perfectly with the main result of this article: that it is possible to model and analyze *low-level* software aspects, including performance, resource consumption, and deployment, in an *executable*, *abstract* language. The Real-Time ABS language used for this purpose shows that it can cope with an *industrial* case study.

As an immediate benefit from executable abstract modeling, it is possible to perform comprehensive simulations that allow to predict and to evaluate the consequences of different scheduling, load balancing, and deployment strategies. We also demonstrated that one can calibrate the model in such a way that meaningful comparisons and predictions relative to deployed code are possible. This led to the identification of a hot spot in the actual production code.

The formal semantics of Real-Time ABS makes it feasible to analyze models *statically*. Using the cost analysis tool COSTABS, it is possible to *automatically* compute symbolic worst-case bounds for resource consumption. To the best of our knowledge, static resource analysis is applied here the first time successfully on an industrial strength case study.

The inferred bounds were in turn used to *automatically* provide cost annotations for expressions and statements in the Real-Time ABS model, hence making their manual creation unnecessary. This allows for a *hybrid* deployment validation approach where dynamic simulation and static analysis combine their strengths.

In summary, the results of this paper demonstrate that the combination of abstract, executable modeling together with state-of-the-art static analysis tools is well on the way toward *tool-supported software design of virtualized applications*. This line of research may in a longer perspective pave the way toward a model-based analysis of service-level agreements [5].

We plan to extend this work in several directions. As regards the static analysis tool, it currently infers resource estimates at the level of components (namely objects) and without taking into account deployment configurations. For instance, if two objects are distributed in different machines, their execution can run in parallel and their costs should not be accumulated in order to compute a system-level estimate of the resource consumption. We plan to incorporate such system-level approach in the resource analyzer by taking deployment configurations into account. Besides, further work on industrial size case studies will allow us to both further mature our technology and assess its scope of application.

**Acknowledgments** Although the author list of this article is rather long already, we gratefully thank the many more people who have been involved in the development of the ABS and Real-Time ABS languages, their toolset, as well as the COSTABS system. Without their effort, the research reported here would not have been possible.

## References

1. Agha GA (1986) ACTORS: a model of concurrent computations in distributed systems. MIT Press, Cambridge, MA
2. Albert E, Arenas P, Genaim S, Gómez-Zamalloa M, Puebla G (2011) Cost analysis of concurrent OO programs. In: The 9th Asian symposium on programming languages and systems (APLAS’11), LNCS, vol 7078. Springer, Berlin, pp 238–254
3. Albert E, Arenas P, Genaim S, Gómez-Zamalloa M, Puebla G (2012) COSTABS: a cost and termination analyzer for ABS. In: Kiselyov O, Thompson S (eds) Proceedings of the workshop on partial evaluation and program manipulation (PEPM’12). ACM, pp 151–154
4. Albert E, Arenas P, Genaim S, Puebla G, Zanardini D (2012) Cost analysis of object-oriented bytecode programs. TCS 413(1):142–159

5. Albert E, de Boer F, Hähnle R, Johnsen EB, Laneve C (2013) Engineering virtualized services. In: Babar MA, Dumas M (eds) Proceedings of the 2nd nordic symposium on cloud computing & internet technologies (NordiCloud'13). ACM, pp 59–63
6. Albert E, Bubel R, Genaim S, Hähnle R, Puebla G, Román-Díez G (2011) Verified resource guarantees using COSTA and KeY. In: ACM SIGPLAN workshop on partial evaluation and semantics-based program manipulation (PEPM'11). ACM Press
7. Albert E, Genaim S, Gómez-Zamalloa M, Johnsen EB, Schlatte R, Tapia Tarifa SL (2011) Simulating concurrent behaviors with worst-case cost bounds. In: Butler M, Schulte W (eds) FM 2011, *LNCS*, vol 6664. Springer, Berlin, pp 353–368
8. Albert E, Østvold BM, Rojas JM (2012) Automated extraction of abstract behavioural models from JMS applications. In: Formal methods for industrial critical systems (FMICS'12), *LNCS*, vol 7437. Springer, Berlin, pp 16–31
9. Amnell T, Fersman E, Mokrushin L, Pettersson P, Yi W (2003) TIMES: a tool for schedulability analysis and code generation of real-time systems. In: Larsen KG, Niebert P (eds) Proceeding of the first international workshop on formal modeling and analysis of timed systems (FORMATS 2003), *LNCS*, vol 2791. Springer, Berlin, pp 60–72
10. Armstrong J (2007) Programming Erlang: software for a concurrent world. Pragmatic Bookshelf (2007)
11. Bai X, Li M, Chen B, Tsai WT, Gao J (2011) Cloud testing tools. In: Gao JZ, Lu X, Younas M, Zhu H (eds) Proceedings of the 6th international symposium on service oriented system engineering (SOSE'11). IEEE, pp 1–12
12. Baier C, Haverkort BR, Hermanns H, Katoen JP (2010) Performance evaluation and model checking join forces. Commun ACM 53(9):76–85
13. Barbanera F, Bugliesi M, Dezani-Ciancaglini M, Sassone V (2007) Space-aware ambients and processes. TCS 373(1–2):41–69
14. Benoy F, King A (1997) Inferring argument size relationships with CLP(R). In: Proceedings of the LOPSTR'97, *LNCS*, vol 1207. Springer, Berlin, pp 204–223
15. Bjørk J, de Boer FS, Johnsen EB, Schlatte R, Tapia Tarifa SL (2012) User-defined schedulers for real-time concurrent objects. Innov Syst Softw Eng 9(1):29–43
16. Bohnenkamp HC, D'Argenio PR, Hermanns H, Katoen JP (2006) MODEST: a compositional modeling formalism for hard and softly timed systems. IEEE Trans Softw Eng 32(10):812–830
17. Bouyer P, Fahrenberg U, Larsen KG, Markey N (2011) Quantitative analysis of real-time systems using priced timed automata. Commun ACM 54(9):78–87
18. Bruno EJ, Bollella G (2009) Real-Time Java Programming: with Java RTS. Prentice Hall PTR, Englewood Cliffs, NJ
19. Buyya R, Yeo CS, Venugopal S, Broberg J, Brandic I (2009) Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. Future Gener Comput Syst 25(6):599–616
20. Calheiros RN, Netto MA, Rose CAD, Buyya R (2012) EMUSIM: an integrated emulation and simulation environment for modeling, evaluation, and validation of performance of cloud computing applications. Softw Pract Exp 43(5):595–612
21. Calheiros RN, Ranjan R, Beloglazov A, Rose CAFD, Buyya R (2011) CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw Pract Exp 41(1):23–50
22. Caromel D, Henrio L (2005) A theory of distributed objects. Springer, Berlin
23. Chander A, Espinosa D, Islam N, Lee P, Necula GC (2007) Enforcing resource bounds via static verification of dynamic checks. ACM ToPLaS 29(5), art no 28
24. de Boer FS, Hähnle R, Johnsen EB, Schlatte R, Wong PYH (2012) Formal modeling of resource management for cloud architectures: an industrial case study. In: Paoli FD, Pimentel E, Zavattaro G (eds) Proceedings of the European conference on service-oriented and cloud computing (ESOCC 2012), *LNCS*, vol 7592. Springer, Berlin, pp 91–106
25. Douglass BP (2004) Real Time UML—advances in the UML for Real-Time systems, 3rd edn. Addison-Wesley, Reading, MA
26. ESOCC panel discussion (2012) Global management in service-oriented and cloud computing: challenges and open issues. In: European conference on service-oriented and cloud computing (ESSOC)
27. Foster H, Emmerich W, Kramer J, Magee J, Rosenblum DS, Uchitel S (2007) Model checking service compositions under resource constraints. In: Crnkovic I, Bertolino A (eds) Proceeding of the 6th joint meeting of the European software engineering Conference and the ACM SIGSOFT international symposium on foundations of software engineering (ESEC/FSE'07). ACM, pp 225–234
28. Gulwani S, Mehra KK, Chilimbi TM (2009) Speed: precise and efficient static estimation of program computational complexity. In: POPL. ACM, pp 127–139
29. Hoffmann J, Hofmann M (2010) Amortized resource analysis with polynomial potential. In: The 19th European symposium on programming (ESOP'10), *LNCS*, vol 6012. Springer, Berlin, pp 287–306
30. Johnsen EB, Hähnle R, Schäfer J, Schlatte R, Steffen M (2011) ABS: a core language for abstract behavioral specification. In: Aichernig B, de Boer FS, Bonsangue MM (eds) Proceeding of the 9th international symposium on formal methods for components and objects (FMCO 2010), *LNCS*, vol 6957. Springer, Berlin, pp 142–164
31. Johnsen EB, Owe O (2007) An asynchronous communication model for distributed concurrent objects. Softw Syst Model 6(1):35–58
32. Johnsen EB, Owe O, Schlatte R, Tapia Tarifa SL (2010) Dynamic resource reallocation between deployment components. In: Dong JS, Zhu H (eds) Proceedings of the international conference on formal engineering methods (ICFEM'10), *LNCS*, vol 6447. Springer, Berlin, pp 646–661
33. Johnsen EB, Owe O, Schlatte R, Tapia Tarifa SL (2011) Validating timed models of deployment components with parametric concurrency. In: Beckert, B Marché C (eds) Proceedings of the international conference on formal verification of Object-Oriented software (FoVeOOS'10), *LNCS*, vol 6528. Springer, Berlin, pp 46–60
34. Johnsen EB, Schlatte R, Tapia Tarifa SL (2012) Modeling resource-aware virtualized applications for the cloud in Real-Time ABS. In: Aoki T, Tagushi K (eds) Proceedings of the 14th international conference on formal engineering methods (ICFEM'12), *LNCS*, vol 7635. Springer, Berlin, pp 71–86
35. Kirner R, Puschner PP (2005) Classification of WCET analysis techniques. In: ISORC. IEEE Computer Society, pp 190–199
36. Kramer J (2007) Is abstraction the key to computing? Commun ACM 50(4):36–42
37. Larsen KG, Pettersson P, Yi W (1997) UPPAAL in a nutshell. Int J Softw Tools Technol Transf 1(1–2):134–152
38. Lee EA (2009) Computing needs time. Commun ACM 52(5):70–79
39. Moreau L, Queinnec C (2005) Resource aware programming. ACM ToPLaS 27(3):441–476
40. Netjes M, van der Aalst WM, Reijers HA (2005) Analysis of resource-constrained processes with Colored Petri Nets. In: Jensen K (ed) Proceedings of the sixth workshop on the practical use of coloured petri nets and CPN tools (CPN 2005), *DAIMI*, vol 576. University of Aarhus
41. Nuñez A, Vázquez-Poletti J, Caminero A, Castañé G, Carretero J, Llorente I (2012) iCanCloud: A flexible and scalable cloud infrastructure simulator. J Grid Comput 10:185–209

42. Petriu DB, Woodside CM (2007) An intermediate metamodel with scenarios and resources for generating performance models from UML designs. Softw Syst Model 6(2):163–184
43. Sgroi M, Lavagno L, Watanabe Y, Sangiovanni-Vincentelli A (1999) Synthesis of embedded software using free-choice Petri nets. In: Proceedings of the 36th ACM/IEEE design automation conference (DAC'99). ACM, pp 805–810
44. Verhoef M, Larsen PG, Hooman J (2006) Modeling and validating distributed embedded real-time systems with VDM++. In: Misra J, Nipkow T, Sekerinski E (eds) Proceedings of the 14th international symposium on formal methods (FM'06), *LNCS*, vol 4085. Springer, pp 147–162
45. Vulgarakis A, Seceleanu CC (2008) Embedded systems resources: views on modeling and analysis. In: Proceeding of the 32nd IEEE international computer software and applications conference (COMPSAC'08). IEEE Computer Society, pp 1321–1328
46. Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley DB, Bernat G, Ferdinand C, Heckmann R, Mitra T, Mueller F, Puaut I, Puschner PP, Staschulat J, Stenström P (2008) The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans Embed Comput Syst 7(3)
47. Wong PYH, Albert E, Muschevici R, Proença J, Schäfer J, Schlatte R (2012) The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. Int J Softw Tools Technol Transf (STTT) 14(5):567–588
48. Wong PYH, Diakov N, Schaefer I (2012) Modelling distributed adaptable object oriented systems using HATS approach: a fredhopper case study (invited paper). In: 2nd international conference on formal verification of object-oriented software, *LNCS*, vol 7421. Springer, Berlin (2012)