Integrating Deployment Architectures and Resource Consumption in Timed Object-Oriented Models ☆

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

Department of Informatics, University of Oslo, Norway

Abstract

Software today is often developed for many deployment scenarios; the software may be adapted to sequential, concurrent, distributed, and even virtualized architectures. Since software performance can vary significantly depending on the target architecture, design decisions need to address which features to include and what performance to expect for different architectures. To make use of formal methods for these design decisions, system models need to range over deployment scenarios. For this purpose, it is desirable to lift aspects of low-level deployment to the abstraction level of the modeling language. This paper proposes an integration of deployment architectures in the Real-Time ABS language, with restrictions on processing resources. Real-Time ABS is a timed, abstract and behavioral specification language with a formal semantics and a Java-like syntax, that targets concurrent, distributed and object-oriented systems. A separation of concerns between execution cost at the object level and execution capacity at the deployment level makes it easy to compare the timing and performance of different deployment scenarios already during modeling. The language and associated simulation tool is demonstrated on examples and its semantics is formalized.

Keywords: Deployment architecture; resource management; object orientation; formal methods; performance; Real-Time ABS

☆This work was done in the context of the EU project FP7-610582 *ENVISAGE: Engineering Virtualized Services* (`http://www.envisage-project.eu`).

Email addresses: `einarj@ifi.uio.no` (Einar Broch Johnsen), `rudi@ifi.uio.no` (Rudolf Schlatte), `sltarifa@ifi.uio.no` (S. Lizeth Tapia Tarifa)

Preprint, article to appear in the
Journal of Logical and Algebraic Methods in Programming
http://dx.doi.org/10.1016/j.jlamp.2014.07.001

1. Introduction

Software is increasingly often developed as a range of systems. Different versions of a software may provide different functionality and advanced features, depending on the target users. A development method which attempts to systematize this software variability is product line engineering [1]; in a product line, different versions of a software (i.e., the products) may be instantiated with different features. An example is software for cell phones. Products for different cell phones and service subscriptions are produced by selecting among functional features such as call forwarding, answering machine, text messaging, etc. However, the selection of features in a product may be restricted by the hardware capacity of the different targeted cell phones. In addition to their functional variability, software systems need to adapt to different *deployment architectures*. For example, *operating systems* adapt to specific hardware and even to different numbers of available cores; *virtualized applications* are deployed on a varying number of (virtual) servers; and *services on the cloud* may need to dynamically adapt to the underlying cloud infrastructure and to changing load scenarios. This kind of adaptability raises new challenges for the modeling and analysis of component-based applications [2]. To apply formal methods to the design of such systems, it is interesting to lift aspects of low-level deployment concerns to the abstraction level of the modeling language.

The motivation for the work presented in this paper is to apply performance analysis to formal object-oriented models in which objects are deployed on resource-constrained deployment architectures. The idea underlying our approach is to make a separation of concerns between the *cost* of performing a computation and the available resource *capacity* of the deployment architecture, rather than to assume that this relationship is fixed in terms of, e.g., specified execution times. Although a range of resources could be considered, this paper focuses on processing capacity. In our approach, the underlying deployment architecture of the targeted system forms an integral part of the system model, but defaults are provided which allow the modeler to ignore architectural design decisions when desirable. The separation of concerns between cost and capacity allows the performance of a model to be compared for a range of deployment choices. By comparing deployment choices many interesting questions concerning performance can be addressed during the system design phase, for example:

- How will the response time of my system improve if I double the number of servers?

- How do fluctuations in client traffic influence the performance of my system on a given deployment architecture?
- Can I better control the performance of my system by means of application-specific load balancing?

Our approach is based on ABS [3], a modeling language for distributed concurrent object groups akin to concurrent objects (e.g., [4, 5, 6]), Actors (e.g., [7, 8]), and Erlang processes [9]. Concurrent object groups communicate by asynchronous method calls and futures [6]. ABS is an executable imperative language which allows modeling abstractions; for example, functions and algebraic data types can be used to abstract from imperative data structures while retaining an overall object-oriented design. ABS has a formal semantics in an SOS style [10] as well as a tool suite to support the development and analysis of models [11]. The core of this tool suite is an editor in Eclipse with a compiler and a language interpreter executing on Maude [12], a platform for programs written in rewriting logic [13]. The syntax of ABS and its semantics for sequential object-oriented programs are sufficiently similar to industrial programming languages (specifically, Java) that a moderately experienced software engineer can start using it with a reasonably small learning effort.

To model object-oriented applications in resource-constrained deployment architectures, we extend ABS with *deployment components*. Deployment components were originally proposed by the authors in [14]. Deployment components capture the execution capacity of a location in the deployment architecture, on which a number of concurrent objects are deployed. Deployment components are parametric in the amount of concurrent execution capacity they allow within a time interval. This allows us to analyze how the execution capacity of a deployment component influences the performance of objects executing on the deployment component. The authors also extended this approach to support dynamic resource reallocation [15] and object mobility [16]. This paper improves and combines results from [14, 15, 16] by, first, giving a *unified* presentation of this work; second, adapting our approach to a dense *real-time* model whereas the previous papers used discrete time; third, refining the cost model of the previous papers [14, 15] from fixed costs to the *flexible user-defined cost expressions* introduced in [16]; and fourth, refining the semantics to directly handle slow computations which require several time intervals. To validate and compare the concurrent behavior of models under restricted concurrency assumptions, we use the tool suite for Real-Time ABS.

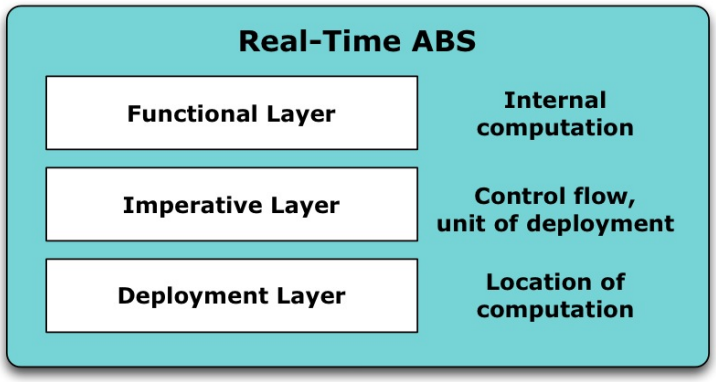

Figure 1: The Layers of the Real-Time ABS Modeling Language

Paper overview. Section 2 presents the Real-Time ABS modeling language and Section 3 extends Real-Time ABS with a deployment layer to capture deployment architectures and resource consumption. Sections 4, 5, and 6 highlight different aspects of the modeling language through examples and show how the Real-Time ABS tool can be used to obtain insights into the deployment aspects of the models. Section 7 formalizes the modeling language in terms of an operational semantics. Section 8 discusses related and future work, and Section 9 concludes the paper.

2. Modeling Timed Behavior in Real-Time ABS

ABS is an executable object-oriented modeling language which combines functional and imperative programming styles to develop high-level executable models. ABS targets the modeling of distributed systems by means of concurrent object groups that internally support interleaved concurrency. Concurrent object groups execute in parallel and communicate through asynchronous method calls. A concurrent object group has at most one active process at any time and a queue of suspended processes waiting to execute on an object in the group. This makes it very easy to combine active and reactive behavior in the concurrent object groups, based on a cooperative scheduling [3] of processes which stem from method activations. Objects in ABS are dynamically created from classes typed by interface; i.e., there is no explicit notion of hiding as the object state is always encapsulated behind interfaces which offer methods to the environment.

Inside an object, internal computation is captured in a simple functional language based on user-defined algebraic data types and functions. Thus, the

modeler may abstract from many details of the low-level imperative implementations of data structures, and still maintain an overall object-oriented design which is close to the target system. A schematic view of the modeling layers of ABS is given in Figure 1; this section presents the functional and imperative layers, the deployment layer is discussed in Section 3.

At a high level of abstraction, concurrent object groups typically consist of a single concurrent object; other objects may be introduced into a group as required to give some of the algebraic data structures an explicit imperative representation when this is natural in a model. To simplify the presentation in this paper, we aim at high-level models and only consider concurrent objects (i.e., the groups will always consist of single concurrent objects). We make use of ABS *annotations* (a general mechanism to add meta-data to statements) to express timing and deployment aspects in our models. This paper assumes that all ABS programs are well-typed. In particular, the presented syntax definition and operational semantics assume that annotations only occur as discussed in the sequel. (A type system for the core ABS language is given in [3].)

Real-Time ABS [17] is an extension of ABS to model the timed behavior of concurrent objects in ABS. The object-oriented perspective on timed behavior is captured by *deadlines* on method calls. Every method activation in Real-Time ABS has an associated deadline; this deadline captures the remaining execution time, so it decreases with the passage of time. Deadlines are *soft*; i.e., the execution of the method does not stop because the deadline is missed. By default the deadline associated with a method activation is infinite, so in an untimed ABS model deadlines will never be missed. We use the annotation mechanism of ABS to override the default deadline for specific method calls.

2.1. The Functional Layer of Real-Time ABS

The functional layer of Real-Time ABS consists of a library of algebraic data types such as the empty type Unit, booleans Bool, integers Int, rational numbers Rat, and strings String; parametric data types such as sets Set<A> and maps Map<A, B> (given values for the type variables A and B); and (parametric) functions over values of these data types. For simplicity, Real-Time ABS does not support operator overloading.

Example 1. (Polymorphic sets in Real-Time ABS.) Polymorphic sets can be defined using a type variable A and two constructors EmptySet and Insert. We define a function contains which recursively checks whether an element el is in a set ss by pattern matching over ss.

Syntactic categories.		*Definitions.*
T	in GroundType	$T ::= B \mid I \mid D \mid D\langle\overline{T}\rangle$
A	in Type	$A ::= N \mid T \mid N\langle\overline{A}\rangle$
x	in Variable	$Dd ::= \textbf{data}\ D[\langle\overline{A}\rangle] = [\overline{Cons}];$
e	in Expression	$Cons ::= Co[(\overline{A})]$
v	in Value	$F ::= \textbf{def}\ A\ fn[\langle\overline{A}\rangle](\overline{A}\,\overline{x}) = e;$
br	in Branch	$e ::= x \mid v \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \textbf{case}\ e\ \{\overline{br}\}$
p	in Pattern	$\mid \textbf{this} \mid \textbf{now}(\,) \mid \textbf{deadline}(\,) \mid \textbf{destiny}(\,)$
		$v ::= Co[(\overline{v})] \mid \textbf{null}$
		$br ::= p \Rightarrow e;$
		$p ::= \_ \mid x \mid v \mid Co[(\overline{p})]$

Figure 2: 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.

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

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

The underscore _ matches any element in a constructor pattern without introducing a variable binding, the new binding xs matches the rest of the set in the last **case** clause.

The following statement defines a variable b and sets its value to True.

```
Bool b = contains(1, Insert(1, EmptySet));
```

End of example.

To express time, we consider a dense time model represented by two types Time and Duration. Time values capture points in time as reflected on a global clock during execution. In contrast, finite durations reflect the execution time (i.e., the difference between two time values). However, durations may be infinite (or unbounded). Infinite durations are captured by the term InfDuration, which is such that for all other durations d_1, d_2, the sum $d_1 + d_2$ is smaller than InfDuration.

Example 2. (Dense Time in Real-Time ABS.) The datatypes Time and Duration are defined as follows:

```
data Time = Time(Rat timeValue);

data Duration = Duration(Rat durationValue) | InfDuration;
```

Here, timeValue and durationValue are partially defined *accessor* functions on the types Time and Duration. For example, given a rational number r, the value Time(r) is of type Time, Duration(r) is of type Duration, and the expressions timeValue(Time(r)) and durationValue(Duration(r)) both evaluate to r. Functions are defined in a standard way, shown here are a subtraction function timeDifference on Time values, a unary predicate `isInfinite` on Duration values, and the binary less-than relation `lt` on Duration values:

```
def Duration timeDifference(Time t1, Time t2) =
  Duration(timeValue(t1)−timeValue(t2));

def Bool isInfinite(Duration d) = d == InfDuration;

def Bool lt(Duration d1, Duration d2) =
  case d1 { InfDuration => False;
    Duration(v1) => case d2 {
      InfDuration => True;
      Duration(v2) => v1 < v2;};};
```

Two Duration values can be added. Since there is no operator overloading in Real-Time ABS, we define addition of durations as a function `add`:

```
def Duration add(Duration d1, Duration d2) =
  case d1 { InfDuration => InfDuration;
    Duration(v1) => case d2 {
      InfDuration => InfDuration;
      Duration(v2) => Duration(v1 + v2);};};
```

Here, the operators $<$ and $+$ are used for comparison and addition in the underlying datatype of rational numbers. *End of example.*

The formal syntax of the functional language is given in Figure 2. The ground types T consist of basic types B such as Bool and Int, as well as names D for datatypes and I for interfaces. In general, a type A may also contain type variables N (i.e., uninterpreted type names [18]). In *datatype declarations* Dd, a datatype D has a set of constructors $Cons$, each of which has a name Co and a list of types $\overline{A}$ for their arguments. *Function declarations* F have a return type A, a function name fn, a list of parameters $\overline{x}$ of types $\overline{A}$, and a function body e. Both datatypes and functions may be polymorphic and have a bracketed list of type parameters (e.g., Set<Bool>).

The layered type system allows functions in the functional layer to be defined over types A which are parametrized by type variables but only applied to ground types T in the imperative layer; e.g., the head of a list is defined for List<A> but applied to ground types such as List<Int>.

Expressions e include variables x, values v, constructor expressions $Co(\overline{e})$, function expressions $fn(\overline{e})$, case expressions **case** e $\{\overline{br}\}$, the self-identifier **this**, **now**() of type Time, which evaluates to the current value of the global system clock, **deadline**() of type Duration, which evaluates to the remaining execution time before the reply from the current process is due, and **destiny**() which refers to the future variable where the return value from the current process is stored after finishing the execution (see the next section). *Values* v are expressions which have reached a normal form; i.e., constructors applied to values $Co(\overline{v})$ or **null** (omitted from Figure 2 are values of the basic types String, Rat and Int, which are standard).

Case expressions match a value against a list of branches $p \Rightarrow e$, where p is a pattern. Patterns are composed of the following elements:

- wild cards _ which match anything;
- variables x match anything if they are free or match against the existing value of x if they are bound;
- values v which are compared literally;
- constructor patterns $Co(\overline{p})$ which match Co and then recursively match the elements $\overline{p}$.

The branches are evaluated in the listed order, free variables in p are bound in the expression e.

2.2. The Imperative Layer of Real-Time ABS

The imperative layer of Real-Time ABS addresses concurrency, communication, and synchronization in the system design, and defines interfaces, classes, and methods in an object-oriented language with a Java-like syntax. In Real-Time ABS, concurrent objects are *active* in the sense that their run method, if defined, gets called upon creation.

Statements are standard for sequential composition $s_1; s_2$, and for **skip**, **if**, **while**, and **return** constructs. The statement **duration**(e_1, e_2) causes time to advance between a best case e_1 and a worst case e_2 execution time, where e_1 and e_2 are rational numbers. Cooperative scheduling in ABS is achieved by explicitly suspending the execution of the active process. The statement **suspend** unconditionally suspends the execution of the active process

and moves this process to the queue. The statement **await** g conditionally suspends execution; the guard g controls processor release and consists of Boolean conditions b and return tests $x?$ (explained in the next paragraph). Just like expressions e, the evaluation of guards g is side-effect free. However, if g evaluates to false, the processor is released and the process *suspended*. When the execution thread is idle, an enabled task may be selected from the pool of suspended tasks by means of a default scheduling policy. In addition to expressions e, the right hand side of an assignment $x{=}rhs$ includes object group creation **new cog** $C(\overline{e})$, method calls $o!m(\overline{e})$, and future dereferencing $x.$**get**. Method calls and future dereferencing are explained in the next paragraph.

Communication and *synchronization* are decoupled in Real-Time ABS. Communication is based on asynchronous method calls, denoted by assignments $f{=}o!m(\overline{e})$ to future variables f of type $\mathbf{Fut}\langle T\rangle$, where T corresponds to the return type of the called method m. Here, o is an object expression, m a method name, and $\overline{e}$ are expressions providing actual parameter values for the method invocation. (Local calls are written $\mathbf{this}!m(\overline{e})$.) After calling $f{=}o!m(\overline{e})$, the future variable f refers to the return value of the call, and the caller may proceed with its execution *without blocking*. Two operations on future variables control synchronization in Real-Time ABS. First, the guard **await** $f?$ *suspends the active process* unless a return to the call associated with f has arrived, allowing other processes in the object 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. Futures are first-class citizens of Real-Time ABS; the expression **destiny**() refers to the future associated with the current process [6]. The statement sequence $x{=}o!m(e)\,;\, v{=}x\,.\,$**get** encodes commonly used *blocking calls*, abbreviated $v{=}o.m(e)$ (often referred to as synchronous calls). If the return value of a call is without interest, the call may occur directly as a statement $o!m(e)$ with no associated future variable. This corresponds to asynchronous message passing. The default deadline of a method activation is InfDuration. However, this default may be overridden by an optional *deadline annotation* to the method call statement, which takes as its argument a duration value. Note that deadline annotations can only occur associated with method calls.

Example 3. (Deadlines.) Assume that an object o implements a method m which takes a formal parameter of type T. We define a wrapper method n which calls m on o and specify a deadline for this synchronized call, given as an annotation and expressed in terms of its own remaining deadline. The method n succeeds if it can return within its given deadline. Note that if its

Syntactic categories.	*Definitions.*
s in Stmt	$P ::= \overline{IF}\ \overline{CL}\ \{[\overline{T}\ \overline{x};]\ s\}$
e in Expr	$IF ::= \textbf{interface}\ I\ \{\ [\overline{Sg}]\ \}$
b in BoolExpr	$CL ::= \textbf{class}\ C\ [(\overline{T}\ \overline{x})]\ [\textbf{implements}\ \overline{I}]\ \{\ [\overline{T}\ \overline{x};]\ \overline{M}\}$
a in Annotation	$Sg ::= T\ m\ ([\overline{T}\ \overline{x}])$
g in Guard	$M ::= Sg\ \{[\overline{T}\ \overline{x};]\ s\}$
	$s ::= s; s \mid [\,\texttt{[}a\texttt{]}\,]\,s \mid \textbf{skip} \mid x = rhs \mid \textbf{if}\ b\ \{\ s\ \}\ [\textbf{else}\ \{\ s\ \}]$
	$\mid \textbf{while}\ b\ \{\ s\ \} \mid \textbf{duration}(e, e) \mid \textbf{suspend}$
	$\mid \textbf{await}\ g \mid \textbf{return}\ e$
	$a ::= \texttt{Deadline:}\ e$
	$rhs ::= e \mid cm \mid \textbf{new cog}\ C\ (\overline{e})$
	$cm ::= e!m(\overline{e}) \mid x.\textbf{get}$
	$g ::= b \mid x? \mid g \wedge g$

Figure 3: Syntax for the imperative layer of Real-Time ABS. Terms like $\overline{e}$ and $\overline{x}$ denote (possibly empty) lists over the corresponding syntactic categories, square brackets `[ ]` denote optional elements.

own deadline is InfDuration, then the deadline to m will also be unlimited. The function scale(d,r) multiplies the value of a duration d by a rational number r (we omit the definition of scale, which is straightforward):

```
Bool n (T x){
  [Deadline: scale(deadline(), 9/10)] f=o.m(x);
  return deadline() > 0;
}
```

End of example.

The formal syntax of the imperative layer of Real-Time ABS is given in Figure 3. A program P consists of lists of interface and class declarations followed by a main block $\{\overline{T}\ \overline{x}; s\}$, which is similar to a method body. An interface IF has a name I and method signatures Sg. A class CL has a name C, interfaces $\overline{I}$ (specifying types for its instances), class parameters and state variables x of type T, and methods M (The *attributes* of the class are both its parameters and state variables). A method signature Sg declares the return type T of a method with name m and formal parameters $\overline{x}$ of types $\overline{T}$. M defines a method with signature Sg, local variable declarations $\overline{x}$ of types $\overline{T}$, and a statement s. Statements may access attributes, locally defined variables, and the method's formal parameters. There are no type variables at the imperative layer of Real-Time ABS.

2.3. Explicit and Implicit Time in Real-Time ABS

In Real-Time ABS, the local passage of time can be modeled both *explicitly* and *implicitly*. With explicit time, the modeler inserts duration state-

ments with best-case and worst-case execution times into the model. This is the standard approach to modeling timed behavior, well-known from, e.g., timed automata in UPPAAL [19]. Duration statements specify explicit execution times when the model abstracts from the system's deployment architecture (e.g., the deployment architecture is assumed to be fixed and the load captured by worst- and best-case execution times).

Example 4. (Explicit time.) Let f be a function defined in the functional layer of Real-Time ABS, which recurses through some data structure x of type T, and let the function size measure this data structure. Consider a method m which takes as input such a value x and returns the result of applying f to x. Let us assume that the time needed for this computation depends on the size of x; e.g., the execution time is between a duration size(x)/2 and a duration 4∗size(x). An interface I which provides the method m and a class C which implements I, including the execution time for m using the explicit time model, are specified as follows:

```
interface I { Int m(T x) }
class C implements I {
  Int m (T x){ duration(size(x)/2, 4∗size(x)); return f(x);
  }
}
```

End of example.

With implicit time the execution time is not specified explicitly in terms of durations, but rather *observed* on the executing model. This is done by comparing clock values from the global clock during model execution.

Example 5. (Implicit time.) We specify an interface J with a method p which, given a value of type T, returns a value of type Duration, and we implement p in a class D such that p measures the time needed to call the method m of Example 4 above, as follows:

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

End of example.

Observe that with implicit time, no assumptions about execution times are given in the model. The execution time depends on how quickly the

method call is effectuated by the other object. In Example 5 the execution time is simply observed by comparing the time before and after making the call. As a consequence, the time needed to execute a statement depends on the *capacity* of the chosen deployment architecture and on *synchronization* with (slower) objects. In many cases it is natural to use both explicit and implicit time in a model, so both are supported in Real-Time ABS.

3. Modeling Deployment Architectures

The execution time in a distributed system depends on the amount of computation which takes place at different locations, and on the execution capacity of those locations. Deployment architectures express how different software units are deployed on physical or virtual hardware. We can think of a deployment architecture as a collection of locations with resource constraints, on which the software units are deployed. Real-Time ABS makes a separation of concerns between the resource *cost* of performing a computation and the resource *capacity* of a given location.

In this paper we focus on CPU resources. Deployment components are used to model locations which are restricted in their execution capacity. Resource cost annotations are used to express resource consumption during computation. Other resource types like bandwidth, memory, and power consumption can be handled with similar techniques as the ones presented here; work in this area is ongoing.

3.1. Deployment Components

A *deployment component* captures the execution capacity of a location on which a number of concurrent objects are deployed. The capacity is specified as an amount of resources which is available during a time interval; the time interval corresponds to the time between integer values in the dense time domain of Real-Time ABS. The available resources may be used to perform computation within the time interval, and they are renewed for the next time interval. The renewal of resources for different deployment components is synchronized.

The main block of the model executes in a root object located on a default deployment component, which we call environment, with unrestricted processing capacity. A model may be extended with other deployment components with different capacities. When objects are created, they are by default allocated to the same deployment component as their creator, but they may also be allocated to a different deployment component. Thus, in a model

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

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

Figure 4: The specification of resources and the interface of deployment components.

without explicit deployment components all objects run in 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, they support a number of methods, and they may be created dynamically, depending on control flow, or statically in the main block of the model. 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 a class DeploymentComponent, which implements DC. The interface DC is defined as in Figure 4.

The DC interface 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 during the last n time intervals in a percentage scaled from 0 to 100 (i.e., a load of 90 means that 90% of the total resources have been used in the last n time intervals in average), and **transfer**(target,r) reallocates r resources from the deployment component to the target deployment component. If the deployment component has less than r resources available, the available amount is transferred.

The type DCData, given in Figure 4, reflects processing capacity. It has constructors InfCPU for infinite resources and CPU(r), where r represents the amount of processing resources available in a time interval. The observer function capacity is defined for the constructor CPU(r) and returns the amount r. Deployment components are created by an assignment with the right hand side **new cog** DeploymentComponent(descriptor,capacity). The parameter capacity of type DCData specifies the initial CPU capacity of the deployment component. The parameter descriptor of type String is a descriptor mainly used for monitoring purposes; i.e., it defines a user-defined name for the deployment component which facilitates querying the run-time state

but that has no semantic effect. The use of descriptors is further illustrated in the examples of Sections 4 and 6. Objects are deployed on deployment components when the objects are created. By default an object is deployed on the same deployment component as its creator. However, a different deployment component may be selected by means of 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 concurrent object groups.

Example 6. (Static Deployment Architecture.) Given the interfaces I and J and classes C and D of Examples 4 and 5, we can specify a static deployment architecture in which two C objects, deployed on different deployment components Server1 and Server2, interact with D objects deployed on a deployment component ClientServer, as follows. We create three deployment components with descriptors Server1, Server2, and ClientServer and processing capacities 6, 3, and InfCPU (i.e., the ClientServer has no resource restrictions). The local variables dc1, dc2, and dc3 refer to these three deployment components in the scope of the main block of the model. Objects are explicitly allocated to the servers by deployment annotations; below, object1 is allocated to Server1, etc.

```
{ // This main block initializes a static deployment architecture:
  DC dc1 = new cog DeploymentComponent("Server1",CPU(6));
  DC dc2 = new cog DeploymentComponent("Server2",CPU(3));
  DC dc3 = new cog DeploymentComponent("ClientServer", InfCPU);
  [DC: dc1] I object1 = new cog C;
  [DC: dc2] I object2 = new cog C;
  [DC: dc3] J client1monitor = new cog D(object1);
  [DC: dc3] J client2monitor = new cog D(object2);
}
```

Figure 5 depicts this deployment architecture and the artefacts introduced into the modeling language. *End of example.*

3.2. Resource Consumption

The available resource capacity of a deployment component determines how much computation 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. In the case of CPU resources, the resources of the deployment component define its capacity inside a time interval, after which the resources are renewed.

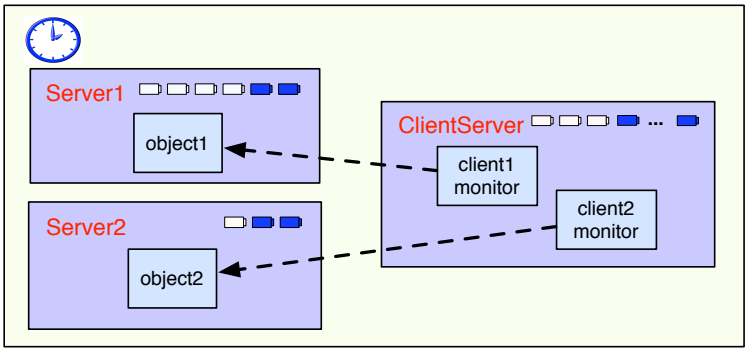

Figure 5: A deployment architecture in Real-Time ABS, with the three deployment components Server1, Server2, and ClientServer described in Section 3.1. In each deployment component, we see its allocated objects and the "battery" of allocated and available processing resources (top right).

The resource consumption of executing statements in the 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]. Note that cost annotations can be associated with any statement, and that a statement may have several annotations (for example, a **new** statement may have both a cost and a DC annotation; see Figures 3 and 6).

Example 7. (Annotation with concrete cost.) Reconsider the class C of Example 4 and assume that the exact cost of computing the function f(x) may be given as a function g which depends on the size of the input value x. In the context of deployment components, a *resource-sensitive* implementation of interface I may be modeled, which does not have a predefined duration as in the explicit time model of class C. The resulting class C2 can be defined as follows:

```
class C2 implements I {
  Int m (T x){
    [Cost: g(size(x))] return f(x);
  }
}
```

Definitions.

$a ::= \texttt{DC:}\ e \mid \texttt{Cost:}\ e \mid a, a \mid \ldots$

$e ::= \textbf{thisDC}(\,) \mid \ldots$

$rhs ::= \textbf{new cog}\ \texttt{DeploymentComponent}\,(e, e) \mid \ldots$

$cm ::= e!\textbf{load}(e) \mid e!\textbf{total}(\,) \mid e!\textbf{transfer}(e, e) \mid \ldots$

$s ::= \textbf{movecogto}(e) \mid \ldots$

Figure 6: Syntax extension for the deployment layer.

End of example.

It is the responsibility of the modeler to specify appropriate resource costs. A behavioral model with default costs may be gradually refined to provide more realistic resource-sensitive behavior. For the computation of the cost functions such as g in Example 7, the modeler may be assisted by the COSTABS tool [20], which can compute a worst-case approximation of the cost of f in terms of abstract execution steps for an input value x based on static analysis techniques, when given the ABS definition of the expression f. However, the modeler may also want to capture resource consumption at a more abstract level; for example, resource limitations can be made explicit in the model during the early stages of system design. Therefore, cost annotations may be used by the modeler to abstractly represent the cost of some computation which is not fully specified.

Example 8. (Annotation with abstract cost.) The class C3 below may represent a draft version of our method m from Example 7, in which the cost of the computation is specified although the function f has yet to be introduced:

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

End of example.

3.3. The Deployment Layer of Real-Time ABS

Figure 6 summarizes the extensions to the syntax (as presented in Figures 2 and 3) for modeling deployment in Real-Time ABS. Annotations a are extended with deployment component annotations [DC: e] as explained in Section 3.1) and cost annotations [Cost: e] as explained in Section 3.2.

Expressions e are extended with **thisDC**(); 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 self reference **this**. The right hand side rhs of assignments is extended with deployment component creation **new cog** DeploymentComponent(descriptor,capacity) explained in Section 3.1. Method invocation cm is extended with methods **load**(n), **total**(), and **transfer**(target,amount), also explained in Section 3.1. Statements s are extended with a primitive **movecogto**(e) for object group reallocation, an object may relocate its concurrent object group to a deployment component e by executing this statement.

4. Example: A Client-Server System

This section presents the first of three larger examples. We illustrate the modeling of deployment architecture and resource consumption through a client-server system and its behavior under various constant load scenarios. To focus on the mechanisms of the modeling of deployment architecture and resource consumption, we consider a simple model of a client-server system that models the general architecture and control flow of, e.g., a website or computation service, while mostly abstracting from the internal software architecture of the concrete system.

On the server, an agent distributes sessions to clients from a pool of session objects and dynamically creates new session objects as required (at a somewhat higher cost than re-using existing sessions). A client obtains a session through the getSession method of the Agent object; the session objects return themselves to the agent when the session is completed. Clients submit work to the server by calling the order method of a Session object, with a cost parameter that allows the model to specify the execution costs of the invoked service while abstracting from the concrete implementation of the service. Each session stays valid for one order, after which the client can ask for a new session. The Real-Time ABS model of the server is given in Figure 7.

In the implementation of the Session class the completion of an order requires a specific amount of resources, specified via its cost parameter. The **skip** statement in the order method consumes the given cost. An order is successful if it is completed within its deadline; success is calculated by checking that the **deadline**() expression is larger than Duration(0). Note that when sessions run on a deployment component with unlimited resources InfCPU, all orders will be completed immediately, as expected from an infinitely fast server. In the Agent class, the attribute sessions stores a set of Session objects

```
interface Agent {
  Session getSession();
  Unit free(Session session, Bool success); }

interface Session { Bool order(Int cost); }

class Session(Agent agent) implements Session {
  Bool order(Int cost) {
    [Cost: cost] skip;
    Bool success = durationValue(deadline()) > 0;
    agent!free(this, success);
    return success; }
}

class Agent implements Agent {
  Set<Session> sessions = EmptySet;
  Int requestcount = 0;
  Int successcount = 0;

  Session getSession() {
    Session session;
    if (emptySet(sessions)) {
      [Cost: 2]session = new cog Session(this);
    } else {
      [Cost: 1]session = take(sessions);
      sessions = remove(sessions, session);
    }
    requestcount = requestcount + 1;
    return session; }

  Unit free(Session session, Bool success) {
    if (success) { successcount = successcount + 1; }
    sessions = Insert(session, sessions); }
}
```

Figure 7: A session-oriented server model in Real-Time ABS. An Agent object hands out Session objects, reusing them if possible. The behavior of the order method itself is left abstract.

which are currently not in use by any client (the ABS datatype for sets has two constructors EmptySet and Insert, and operations such as, emptySet to check for the empty set, take to select some element of a non-empty set, and remove to remove an element from a set). When a client requests a Session, the Agent takes a session from the set of available sessions if possible, otherwise it creates a new session. Both re-using a session object and creating a new session have associated costs, to accurately model behavior under heavy load or denial-of-service attacks from the environment. The method free inserts a session in the available sessions of the Agent, and is called by the

```
interface Client {}

class Client (Agent agent, Int cycle, Int cost, Int deadline)
implements Client {
  Int ordercount = 0;
  Int successcount = 0;

Unit run() {
  await duration(cycle, cycle);
  Session session = agent.getSession();
  [Deadline: Duration(deadline)] Fut<Bool> f = session!order(cost);
  ordercount = ordercount + 1;
  this!run();
  await f?;
  Bool result = f.get;
  if (result) { successcount = successcount + 1; } }
}

{//Main block
  DC shop = new cog DeploymentComponent("Shop", CPU(20));
  [DC: shop] Agent agent = new cog Agent();
  Client client1 = new cog Client(agent, 2, 5, 5);
  ...
}
```

Figure 8: Deployment environment and client model of the web shop example.

session itself upon completion of an order.

Simulating and Testing the Server. The behavior of the server can be analyzed by extending the model with a deployment scenario and an environment to simulate a workload. The operational semantics of Real-Time ABS with deployment components and resource consumption, presented in Section 7, has been specified in rewriting logic [13], which allows models to be analyzed using the rewriting tool Maude [12]. Given an initial configuration, Maude supports simulation and breadth-first search through reachable states to check safety properties and model checking of finite reachable states for LTL properties. In this paper, Maude is used as an interpreter for the semantics of Real-Time ABS to simulate and test Real-Time ABS models with deployment components and resource consumption.

The environment is modeled by creating one or more instances of the class Client, given in Figure 8. An instance of Client periodically calls order every c time intervals, corresponding to *periodic requests*. (The work in [14] showed the effects of clients with varying co-operative vs. flooding behavior on a similar model.) In the main block of the model, shown in Figure 8,

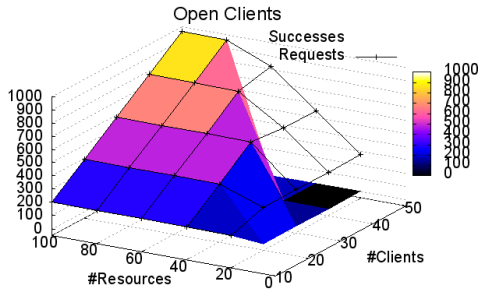

Figure 9: Number of total requests and successful orders, depending on the number of clients and resources. Once the load increases over a certain threshold, no deadlines are met in the simulated system.

a deployment component shop is created with a processing capacity of 20 resources available for the objects allocated on the shop. An instance of Agent is created in that deployment component, which in turn creates Session objects when required by clients.

Figure 9 shows the number of total requests and successful orders for a set of simulation runs, where each run lasted for a duration of 100 time intervals. The scenarios range from 10 to 50 clients and from 20 to 100 resources on the shop deployment component. The scenario shows the effect of flooding the server with requests: After a certain threshold of incoming requests, server response QoS (expressed in responses within deadline vs. requests) collapses and no requests are successfully processed within the specified deadline.

5. Example: Implementing Object Migration to Mitigate Overload

In this section, the example of Section 4 is extended to dynamic deployment scenarios based on the migration of concurrent object groups. Figure 9 showed how heavy client traffic may lead to congestion on the server, which in turn can cause serious degradation of the server's quality of service. In order to investigate the effects of more dynamic deployment scenarios on the quality of service of timed software models, we compare the behavior of Real-Time ABS models with the same functional behavior and workload when the models are run on two different dynamic deployment scenarios.

Real-Time ABS models can include *load balancing strategies*, which aim to decrease congestion and thus improve the overall quality of service compared to models with static deployment scenarios. Load balancing strategies are typically expressed in Real-Time ABS using the resource-related language constructs **total** and **load** to inspect the state of the deployment architec-

```
interface Session { ...
  Unit moveTo(DC dc);
}

class SessionImp(Agent agent) implements Session {
  ...
  Unit moveTo(DC dc) {
    if (dc != thisDC()) {
      [Cost: 1] movecogto(dc);
      [Cost: 1] skip;
    }
  }
}

class SmartAgent(DC backupserver) implements Agent {
  ...
  Unit free(Session session) {
    ...
    session!moveTo(thisDC());
  }
  Session getsession() {
    ...
    Rat load = thisDC().load(1);
    DCData total = thisDC().total();
    if (total != InfCPU && load > 50) {
      session!moveTo(backupserver);}
    return session; }
}
```

Figure 10: An agent which performs load balancing. If the main server load is more than 50%, sessions are started on the backup server. (Code which is identical to that of Figure 8 has been elided for brevity.)

ture. For our server example, two sensible load balancing strategies might be to start requests on a *backup server* once the main server's load exceeds a certain threshold, or to migrate long-running requests to the backup server in order to free resources on the main server. This can be done by moving concurrent object groups between two deployment components, using the **movecogto** primitive.

In this section we model and simulate these two different load balancing strategies: (1) a *load balancing agent* which starts sessions on a backup server when the load on the main server is above a given threshold and (2) *self-monitoring sessions* which move themselves to the backup server once the processing of their current request exceeds a given time limit. Both of these dynamic deployment scenarios are analyzed using an open workload scenario (in which the clients send periodic requests without synchronizing).

```
class SmartSession(Agent agent, Duration limit, DC backupserver)
implements Session {
  Bool mightNeedToMove = False;
  Time timeToMove = Time(0);
  DC origserver = thisDC();

  Unit moveTo(DC dc) { ... } // As before

  Bool order(Int cost) {
    timeToMove = addDuration(now(), limit);
    while (cost > 0) {
      [Cost: 1] cost = cost − 1;
      if (timeValue(now()) > timeValue(timeToMove) && thisDC() != backupserver) {
        this.moveTo(backupserver);
      }
    }
    Bool success = durationValue(deadline()) > 0;
    agent!free(this, success);
    this.moveTo(origserver);
    return success;
  }
}
```

Figure 11: Self-monitoring session objects. The session moves to the backupserver if the request runs longer than limit.

Figure 10 shows the Real-Time ABS class SmartAgent which models a load balancing agent which moves sessions to a backup server when the load on the main server increases beyond a certain threshold, namely that the average load of the main server in the last past four time intervals exceeds 50%. This load balancing strategy tries to minimize the amount of work done on the backup server, while maintaining an acceptable quality of service. When the load threshold is reached, the getsession method calls the moveTo method of the session object before the session is returned. When the session is finished, the method free similarly returns the session to the main server.

Figure 11 shows the Real-Time ABS class SmartSession which models self-monitoring session objects which move themselves to the backup server if the execution of the current request exceeds a given time limit (which is set at creation time in the example through the constructor parameter limit). Here, the order method initially calculates the threshold execution time timeToMove and moves the session to the backup server once execution time passes the threshold.

Simulations of Load Balancing Deployment Scenarios. For the simulations of the server example augmented with load balancing strategies, we added

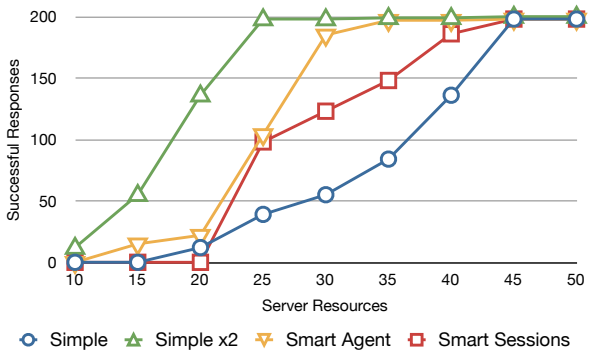

Figure 12: Simulation results for the load balancing strategies using a balancing agent (smart agent) and self-monitoring sessions (smart sessions) and for single servers.

a second deployment component with the same capacity as the primary deployment component. The simulated client job size was chosen so that the capacity of the servers range from complete overloaded to successful completion of all requests.

Figure 12 summarizes all three scenarios (single server, load balancing agent, and self-balancing sessions) when the capacity of the deployment components ranges from 10 to 50 resources, as well as a single server with twice the resources (i.e., ranging from 20 to 100). This second single server has the same capacity as the two balanced servers combined and illustrates the efficiency of the different balancing scenarios under the chosen workload.

6. Example: Load Balancing via Resource Transfer

At midnight on New Year's Eve the behavior of cellphone users briefly changes from normal usage (i.e., a fairly low number of calls and messages) to sending large numbers of SMS messages. We use this phenomenon to motivate and illustrate a complementary approach to load balancing based on the reallocation of (virtualized) resources between deployment components.

The model consists of two cooperating services, TelephoneService and SMSService, and a number of handset clients interacting with these services. The interfaces and implementations of the two services are given in Figure 13. The method call will be invoked *synchronously*; as a parameter the client provides a duration for the call. The method sendSMS will be called *asynchronously*. Note that this model abstracts from many further details which can be added as needed (e.g., a data model, bandwidth, server internals).

```
interface TelephoneServer { Unit call(Int calltime); }
interface SMSServer { Unit sendSMS(); }

class TelephoneServer implements TelephoneServer {
  Int callcount = 0;
  Unit call(Int calltime){
    while (calltime > 0) {
      [Cost: 1] calltime = calltime − 1;
      await duration(1, 1);
    }
    callcount = callcount + 1;
  }
}
class SMSServer implements SMSServer {
  Int smscount = 0;
  Unit sendSMS() {
    [Cost: 1] smscount = smscount + 1;
  }
}
```

Figure 13: The telephony and SMS services.

The model of the handset clients interoperating with the services is given in Figure 14. Client behavior is regulated by a parameter cycle, which determines the frequency of phone calls and messages sent from the handset. Between time $t = 50$ and 70, Handset objects (modeling the behavior of their clients) change to "midnight" behavior and send SMS messages in a rapid pace, otherwise they have "normal" behavior and alternate between sending SMS and making calls.

Simulating this model in a scenario with infinite resources leads to a *purely behavioral model*, where each object acts according to its specification (as in normal Real-Time ABS). Placing the SMS service in an environment with restricted resources leads to observable overload during the midnight window, given a sufficient number of clients to consume all its resources.

In Figure 15 the main block defines a scenario where each service runs in its own deployment component with a capacity of 20 resources, and four clients run in the unrestricted root deployment component environment. Dynamic load balancing is implemented by the Balancer class, an instance of which runs in parallel with the service in each component. This class implements a simple load balancing strategy, transferring resources to its partner deployment component when receiving a request message, and monitoring its own load and requesting assistance when needed. More involved or hierarchical schemes for distributing resources among deployment components can be defined similarly.

```
class Handset (Int cyclelength, TelephoneServer ts, SMSServer smss) {
  Bool call = False;

  Unit normalBehavior() {
    if (timeValue(now()) > 50 && timeValue(now()) < 70) {
      this!midnightWindow();
    } else {
      if (call) { ts.call(1);}
      else { smss!sendSMS(); }
      call = ~ call;
      await duration(cyclelength,cyclelength);
      this!normalBehavior();
    }
  }
  Unit midnightWindow() {
    if (timeValue(now()) >= 70) {
      this!normalBehavior();
    } else {
      Int i = 0;
      while (i < 10) {
        smss!sendSMS();
        i = i + 1;
      }
      await duration(1,1);
      this!midnightWindow();
    }
  }
  Unit run(){
    this!normalBehavior();
  }
}
```

Figure 14: The Handset class, implementing "New Year's Eve" behavior. Before and after midnight, clients alternate between short calls and sending single messages. During the midnight window ($50 \leq t \leq 70$), ten SMS are sent per cycle.

Figure 16 presents simulation results for this scenario. Results for a scenario without any load balancing is also presented, which shows that the allocated resources are more than sufficient for servicing the normal client behavior, but the SMS service is overloaded during the whole load peak and for another 20 time intervals while catching up with the backlog of delayed messages. In the load balancing scenario, the SMS service is working at capacity during the midnight window but finishes the work backlog two time intervals after the demand spike subsides. After the load on the SMS service returns to normal, the capacity between the two balancers is rebalanced. Note that both scenarios use the identical functional model. The balancing functionality is implemented by two active objects, more elaborate load

```
interface Balancer {
  Unit requestdc(DC comp);
  Unit setPartner(Balancer p);
}

class Balancer implements Balancer {
  Balancer partner = null;

 Unit run() {
    await partner != null;
    while (True) {
      await duration(1, 1);
      Rat ld = thisDC().load(1);
      if (ld > 90) {
        Fut<Unit> r = partner!requestdc(thisDC());
        await r?;
      }
    }
  }

  Unit requestdc(DC comp) {
    DCData total = thisDC().total();
    Rat ld = thisDC().load(1);
    if (ld < 50) {
      thisDC()!transfer(comp, capacity(total) / 3);
    }
  }
  Unit setPartner(Balancer p) { partner = p; }
}
{ // Main block
  DC smscomp = new cog DeploymentComponent("smscomp", CPU(50));
  DC telcomp = new cog DeploymentComponent("telcomp", CPU(50));

  [DC: smscomp] SMSServer sms = new cog SMSServer();
  [DC: telcomp] TelephoneServer tel = new cog TelephoneServer();
  [DC: smscomp] Balancer smsb = new cog Balancer();
  [DC: telcomp] Balancer telb = new cog Balancer();
  smsb!setPartner(telb);
  telb!setPartner(smsb);
  new cog Handset(1,tel,sms); new cog Handset(1,tel,sms);
  await duration(1, 1);
  new cog Handset(1,tel,sms); new cog Handset(1,tel,sms);
}
```

Figure 15: A resource reallocation strategy and deployment configuration. Without the Balancer objects, the model runs with no functional changes but with a different timing behavior due to overload in the SMS deployment component.

balancing strategies can be added in similar ways.

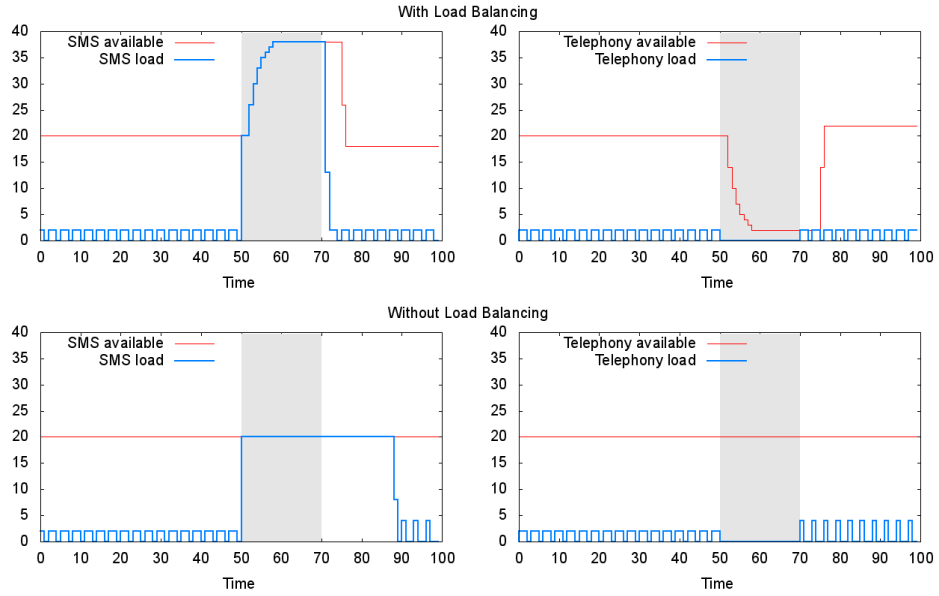

Figure 16: Simulation of "New Year's Eve" behavior (SMS load spike between t=50 and t=70), with (top) and without resource balancing (bottom).

7. Semantics

The operational semantics of Real-Time ABS extended with deployment components and resource consumption is presented as a transition system in an SOS style [10].

7.1. Runtime Configurations

The runtime syntax is given in Figure 17. A *timed configuration tcn* adds a global clock $cl(t)$ to a configuration (where t is a value of type Time). A *configuration cn* is a multiset of objects, invocation messages, futures, and deployment components. The associative and commutative union operator on (timed) configurations is denoted by whitespace and the empty configuration by ε. Note the use of brackets on timed configurations $\{tcn\}$ which will be used when we consider the *whole* configuration and not just some of its terms; i.e., a bracketed configuration will only give a top-level match in the transition system (detailed in Section 7.3).

An *object obj* is a term $o(\sigma, p, q)$ where o is the object's identifier, σ is a substitution representing the binding of the object's fields, p is an (active) process, and q a *pool of processes*. For substitutions σ and process pools

$$
\begin{array}{rcl@{\qquad}rcl}
tcn & ::= & cn\ \ cl(t) \mid \{cn\ \ cl(t)\} & v & ::= & o \mid f \mid dc \mid \ldots \\
cn & ::= & \varepsilon \mid obj \mid msg \mid fut \mid cmp \mid cn\ \ cn & \sigma & ::= & x \mapsto v \mid \sigma \circ \sigma \\
fut & ::= & f \mid f(v) & p & ::= & \{\sigma|s\} \mid idle \\
obj & ::= & o(\sigma, p, q) & q & ::= & \varepsilon \mid p \mid q \circ q \\
msg & ::= & m(o, \overline{v}, f, d) & s & ::= & \mathsf{duration2}(v, v) \mid \ldots \\
cmp & ::= & dc(n, u, k, \overline{h}, \overline{z}) & e & ::= & \mathsf{case2}\ v\ \{\overline{br}\} \mid \ldots
\end{array}
$$

Figure 17: Runtime syntax; here, o, f, and dc are identifiers for objects, futures, and deployment components, x is the name of a variable, and d is the deadline annotation.

q, concatenation is denoted by $\sigma_1 \circ \sigma_2$ and $q_1 \circ q_2$, respectively. A *process* $\{\sigma|s\}$ consists of a substitution σ of local variable bindings and a list s of statements, or it is *idle*. (We identify any process with an empty statement list with the *idle* process.) We let the fields of an object include *this* and *thisDC*, and the local variables of a process include *deadline* and *destiny* (assuming no name capture). The value of *this* is the identifier of the object and the value of *thisDC* is bound to the object's current deployment component. The value of *deadline* is the remaining duration of the deadline of the process and the value of *destiny* is the address for the return of the process.

In an *invocation message* $m(o, \overline{v}, f, d)$, m is the method name, o the callee, $\overline{v}$ the call's actual parameter values, f the future to which the call's result is returned and d is the provided deadline. A *future* is either an identifier f or a term $f(v)$ with an identifier f and a reply value v. For simplicity, classes are not represented explicitly in the semantics, but may be seen as static tables of object layout and method definitions. In a *deployment component* $dc(n, u, k, \overline{h}, \overline{z})$, dc is its identity, n is the total number of available processing resources allocated for the current time interval, u the used resources in the current time interval, k is the number of resources to be allocated for the next time interval, $\overline{h}$ the (possibly empty) sequence of resource usage over time intervals and $\overline{z}$ the (possibly empty) sequence of total allocated resources over time intervals.

The values v are extended with identifiers for the dynamically created objects, futures, and deployment components, statements with $\mathsf{duration2}(v, v)$ (where the best and worst case expressions must similarly be values instead of expressions), and expressions e with $\mathsf{case2}\ v\ \{\overline{br}\}$ (where the condition must be a value). In the statements s, we further assume for simplicity that all method call assignments have deadline annotations as explained in Section 2.2 and we let *default*(T) denote a default value of type T; e.g., **null** for interface types.

$$
\begin{array}{l}
[\![x]\!]^t_\sigma = \sigma(x) \\
[\![v]\!]^t_\sigma = v \\
[\![\textsf{now}()]\!]^t_\sigma = t \\
[\![Co(\overline{e})]\!]^t_\sigma = Co([\![\overline{e}]\!]^t_\sigma) \\
[\![\textsf{destiny}()]\!]^t_\sigma = \sigma(\mathit{destiny}) \\
[\![\textsf{deadline}()]\!]^t_\sigma = \sigma(\mathit{deadline}) \\
[\![\textsf{this}]\!]^t_\sigma = \sigma(\mathit{this}) \\
[\![\textsf{thisDC}()]\!]^t_\sigma = \sigma(\mathit{thisDC}) \\
[\![fn(\overline{e})]\!]^t_\sigma = \begin{cases} [\![e_{fn}]\!]^t_{\overline{x}\mapsto\overline{v}} & \text{if } \overline{e} = \overline{v} \\ [\![fn([\![\overline{e}]\!]^t_\sigma)]\!]^t_\sigma & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{l}
[\![\textsf{case } e\ \{\overline{br}\}]\!]^t_\sigma = [\![\textsf{case2 } [\![e]\!]^t_\sigma\ \{\overline{br}\}]\!]^t_\sigma \\
[\![\textsf{case2 } v\ \{p \Rightarrow e; \overline{br}\}]\!]^t_\sigma = \begin{cases} [\![e]\!]^t_{\sigma \circ match(p,v)} & \text{if } match(p,v) \neq \bot \\ [\![\textsf{case2 } v\ \{\overline{br}\}]\!]^t_\sigma & \text{otherwise} \end{cases}
\end{array}
$$

Figure 18: The evaluation of functional expressions.

Initial configuration. The initial configuration of a program reflects its main block; for a program with main block $\{\overline{T}\ \overline{x}; s\}$ the initial configuration has the form

$$main(a, \{l|s\}, \varepsilon)\ environment(InfCPU, 0, InfCPU, \varepsilon, \varepsilon)\ cl(0)$$

where *main* is an object, *environment* is the default deployment component with unlimited allocated resources in both the current and next time intervals, and $cl(0)$ is the system clock at time 0. In the *main* object, let a be the substitution $\varepsilon[\mathit{this} \mapsto \mathit{main}, \mathit{thisDC} \mapsto \mathit{environment}]$ and l be the substitution $\varepsilon[\mathit{destiny} \mapsto \mathit{default}(\textsf{Fut<Unit>}), \mathit{deadline} \mapsto \mathit{InfDuration}], \overline{x} \mapsto \mathit{default}(\overline{T})$. (We assume that for a well-typed program, the main block does not refer to the expressions **this**, **destiny()**, and **deadline()**.)

7.2. The Timed Evaluation of Expressions

Let σ be a substitution which binds the name *destiny* to a future identifier, *deadline* to a duration value, *this* to an object identifier, and *thisDC* to the identifier of a deployment component. The evaluation function for expressions e given a substitution σ at a time t is defined inductively over the data types of the functional language (see Figure 18) and is mostly standard, hence this subsection only contains brief remarks about some of the expressions. For every (user defined) function definition

$$\textsf{def } T\ fn(\overline{T\ x}) = e_{fn},$$

the evaluation of a function call $[\![fn(\overline{e})]\!]^t_\sigma$ reduces to the evaluation of the corresponding expression $[\![e_{fn}]\!]^t_{\overline{x}\mapsto\overline{v}}$ when the arguments $\overline{e}$ have already been reduced to ground terms $\overline{v}$. (Note the change in scope. Since functions are defined independently of the context where they are used, we here assume that the expression e does not contain free variables and the substitution σ does not apply in the evaluation of e.) In the case of pattern matching, variables in the pattern p may be bound to argument values in v. Thus the substitution context for evaluating the right hand side e of the branch $p \to e$ extends the current substitution σ with bindings that occurred during the pattern matching. Let the function $match(p, v)$ return a substitution such that $match(p, v)(p) = v$ (if there is no match, $match(p, v) = \bot$). For simplicity, we here assume that the evaluation of functional expressions is terminating.

7.3. A Transition System for Timed Configurations

Let the transition relation $\to_t$ capture transitions between timed configurations let $\to$ represent untimed execution. A timed run is a non-terminating sequence of timed configurations $\{tcn_0\}, \{tcn_1\}, \ldots$ such that $\{tcn_i\} \to_t \{tcn_{i+1}\}$. Similarly, an untimed run is a possibly terminating sequence of (timed) configurations $tcn_0, tcn_1, \ldots$ such that $tcn_i \to tcn_{i+1}$. Let $tcn \xrightarrow{!} tcn'$ denote that tcn' is a normal form resulting from a terminating run from the initial configuration tcn; i.e., there is no configuration tcn'' such that $tcn' \to tcn''$.

We define a *maximal progress semantics* in which the rules for time advance only apply when untimed execution is blocked; i.e., for a timed configuration tcn_i, the relation $\{tcn_i\} \to_t \{tcn_{i+1}\}$ is defined by $tcn_i \xrightarrow{!} tcn'_i$ and $tcn_{i+1} = \phi(tcn'_i)$, where ϕ is one of the auxiliary functions which express the effect of advancing time on the terms of the configuration tcn'_i. When auxiliary functions such as ϕ are used in the semantics, these are evaluated in between the application of transition rules in a run. Rules apply to subsets of configurations (the standard context rules are not listed). For simplicity we assume that configurations can be reordered to match the left hand side of the rules, i.e., matching is modulo associativity and commutativity as in rewriting logic [13]. Real-Time ABS does not assume that time will always advance; time will never advance in models where execution never runs out of resources and never gets blocked. This corresponds to resource-unaware or infinitely fast models.

Evaluating guards. Given a substitution σ, a time t and a configuration cn, we lift the evaluation function for functional expressions to guards and

denote by $[\![g]\!]^{t,cn}_{\sigma}$ an evaluation function which reduces guards g to data values (here the configuration cn is needed to evaluate future variables). Let $[\![g_1 \wedge g_2]\!]^{t,cn}_{\sigma} = [\![g_1]\!]^{t,cn}_{\sigma} \wedge [\![g_2]\!]^{t,cn}_{\sigma}$, $[\![x?]\!]^{t,cn}_{\sigma} = \mathsf{True}$ if $[\![x]\!]^{t,cn}_{\sigma} = f$ and $f(v) \in cn$ for some value v (i.e., the future already has a value), otherwise $f \in cn$ and we let $[\![x?]\!]^{t,cn}_{\sigma} = \mathsf{False}$. Guards that are Boolean expressions reduce as expected: $[\![e]\!]^{t,cn}_{\sigma} = [\![e]\!]^{t}_{\sigma}$ (note that such guards can change their value only if they refer to the object state).

Auxiliary functions. If the class of an object o has a method m, we let $\mathit{bind}(m, o, \overline{v}, f, d)$ return a process resulting from the activation of m on o with actual parameters $\overline{v}$, an associated future f, and a deadline d. If the binding succeeds, the local variable *destiny* in the new process is bound to f, *deadline* is bound to d, and the method's formal parameters are bound to $\overline{v}$. The function $\mathit{select}(q, \sigma, cn)$ schedules a process which is ready to execute from the process queue q of an object $o(\sigma, \mathit{idle}, q)$ in a configuration cn. The function $\mathit{atts}(C, \overline{v}, o, dc)$ returns the initial substitution σ for the fields of a new instance o of class C, in which the formal parameters are bound to $\overline{v}$, the field *this* is bound to the object identity o and the field *thisDC* to the deployment component dc. The function $\mathit{init}(C)$ returns an activation (process) of the *init* method of C, if defined. Otherwise it returns the *idle* process. The predicate $\mathit{fresh}(n)$ asserts that a name n is globally unique (where n may be an identifier for an object, a future, or a deployment component). The definition of these functions is straightforward but requires that the class table is explicit in the semantics, which we have omitted for simplicity.

Transition rules. Transition rules transform configurations into new configurations, and are given in Figures 19 and 20. In the semantics, different assignment rules are defined for side effect free expressions (Assign1 and Assign2), object creation (New-Object1 and New-Object2), method calls (Async-call), and future dereferencing (Read-Fut). We conventionally write a to denote the substitution which maps fields to values in an object and l to denote the substitution which maps local variables to values in a process. Annotations are used to provide a deadline, a cost, and to associate objects with deployment components. (In the implementation, these annotations are generated with default values by the compiler if they are not explicitly given in the source code.)

Rule Skip consumes a **skip** in the active process. Here and in the sequel, the variable s will match any (possibly empty) statement list. Rules Assign1 and Assign2 assign the value of expression e to a variable x in the local

$$\text{(Skip)}\quad o(a, \{l \mid \textbf{skip}; s\}, q) \to o(a, \{l \mid s\}, q)$$

$$\text{(Activate)}\quad \frac{p = select(q, a, cn)}{\{o(a, idle, q)\ \ cn\ \ cl(t)\} \to \{o(a, p, (q \setminus p))\ \ cn\ \ cl(t)\}}$$

$$\text{(Suspend)}\quad o(a, \{l \mid \textbf{suspend}; s\}, q) \to o(a, idle, \{l \mid s\} \circ q)$$

$$\text{(Assign1)}\quad \frac{x \in\ dom(l)}{o(a, \{l \mid x = e; s\}, q)\ \ cl(t) \to o(a, \{l[x \mapsto [\![e]\!]^t_{a \circ l}] \mid s\}, q)\ \ cl(t)}$$

$$\text{(Assign2)}\quad \frac{x \in\ dom(a)}{o(a, \{l \mid x = e; s\}, q)\ \ cl(t) \to o(a[x \mapsto [\![e]\!]^t_{a \circ l}], \{l \mid s\}, q)\ \ cl(t)}$$

$$\text{(Cond1)}\quad \frac{[\![e]\!]^t_{a \circ l}}{o(a, \{l \mid \textbf{if}\ e\ \{s_1\}\ \textbf{else}\ \{s_2\}; s\}, q)\ \ cl(t) \to o(a, \{l \mid s_1; s\}, q)\ \ cl(t)}$$

$$\text{(Cond2)}\quad \frac{\neg [\![e]\!]^t_{a \circ l}}{o(a, \{l \mid \textbf{if}\ e\ \{s_1\}\ \textbf{else}\ \{s_2\}; s\}, q)\ \ cl(t) \to o(a, \{l \mid s_2; s\}, q)\ \ cl(t)}$$

$$\text{(Await1)}\quad \frac{[\![e]\!]^{t,cn}_{a \circ l}}{\{o(a, \{l \mid \textbf{await}\ e; s\}, q)\ \ cl(t)\ \ cn\} \to \{o(a, \{l \mid s\}, q)\ \ cl(t)\ \ cn\}}$$

$$\text{(Await2)}\quad \frac{\neg [\![e]\!]^{t,cn}_{a \circ l}}{\{o(a, \{l \mid \textbf{await}\ e; s\}, q)\ \ cl(t)\ \ cn\} \to \{o(a, \{l \mid \textbf{suspend}; \textbf{await}\ e; s\}, q)\ \ cl(t)\ \ cn\}}$$

$$\text{(Async-Call)}\quad \frac{\begin{array}{c} an = \texttt{Deadline}: e', an' \\ [\![e]\!]^t_{a \circ l} = o' \quad [\![e']\!]^t_{a \circ l} = d \quad fresh(f) \end{array}}{\begin{array}{c} o(a, \{l \mid [an]\ x = e!m(\overline{e}); s\}, q)\ \ cl(t) \\ \to o(a, \{l \mid [an']\ x = f; s\}, q)\ \ m(o', [\![\overline{e}]\!]^t_{a \circ l}, f, d)\ \ f\ \ cl(t) \end{array}}$$

$$\text{(Bind-Mtd)}\quad \frac{q' = bind(m, o, \bar{v}, f, d) \circ q}{o(a, \{l \mid s\}, q)\ \ m(o, \bar{v}, f, d) \to o(a, \{l \mid s\}, q')}$$

$$\text{(Return)}\quad \frac{f = l(destiny)}{o(a, \{l \mid \textbf{return}(e); s\}, q)\ \ f\ \ cl(t) \to o(a, idle, q)\ \ f([\![e]\!]^t_{a \circ l})\ \ cl(t)}$$

$$\text{(Read-Fut)}\quad \frac{f = [\![e]\!]^t_{a \circ l}}{o(a, \{l \mid x = e.\textbf{get}; s\}, q)\ \ f(v)\ \ cl(t) \to o(a, \{l \mid x = v; s\}, q)\ \ f(v)\ \ cl(t)}$$

$$\text{(Duration1)}\quad \frac{v_1 = [\![e_1]\!]^t_{a \circ l} \quad v_2 = [\![e_2]\!]^t_{a \circ l}}{o(a, \{l \mid \textbf{duration}(e_1, e_2); s\}, q)\ \ cl(t) \to o(a, \{l \mid \textbf{duration2}(v_1, v_2); s\}, q)\ \ cl(t)}$$

$$\text{(Duration2)}\quad \frac{v_1 \leq 0}{o(a, \{l \mid \textbf{duration2}(v_1, v_2); s\}, q) \to o(a, \{l \mid s\}, q)}$$

Figure 19: Semantics for Real-Time ABS with deployment components and resource consumption (1).

(New-Object1)

$$\frac{\mathit{fresh}(o') \quad an = \texttt{DC}\colon e', an' \quad [\![e']\!]^t_{a\circ l} = dc \quad p' = \mathit{init}(C) \quad a' = \mathit{atts}(C, [\![\overline{e}]\!]^t_{a\circ l}, o', dc)}{o(a, \{l \mid [an]\ x = \textbf{new}\ C(\overline{e}); s\}, q)\ \ cl(t) \to o(a, \{l \mid [an']\ x = o'; s\}, q)\ \ o'(a', p', \emptyset)\ \ cl(t)}$$

(New-Object2)

$$\frac{\mathit{fresh}(o') \quad a(\mathit{thisDC}) = dc \quad p' = \mathit{init}(C) \quad a' = \mathit{atts}(C, [\![\overline{e}]\!]^t_{a\circ l}, o', dc)}{o(a, \{l \mid x = \textbf{new}\ C(\overline{e}); s\}, q)\ \ cl(t) \to o(a, \{l \mid x = o'; s\}, q)\ \ o'(a', p', \emptyset)\ \ cl(t)}$$

(New-DC)

$$\frac{\mathit{fresh}(dc) \quad [\![e]\!]^t_{a\circ l} = n}{o(a, \{l \mid x = \textbf{new}\ \texttt{DeploymentComponent}(e_0, e); s\}, q)\ \ cl(t) \to o(a, \{l \mid x = dc; s\}, q)\ \ dc(n, 0, n, \varepsilon, \varepsilon)\ \ cl(t)}$$

(Emp-Annotation)

$$o(a, \{l \mid [\,\varepsilon\,]\ s\}, q) \to o(a, \{l \mid s\}, q)$$

(Cost1)

$$\frac{a(\mathit{thisDC}) = dc \quad an = \texttt{Cost}\colon e, an' \quad [\![e]\!]^t_{a\circ l} = c \quad c \le n - u \quad o(a, \{l \mid [an']\ s\}, q)\ \ cl(t)\ \ cn \to o(a', p', q')\ \ cl(t)\ \ cn'}{o(a, \{l \mid [an]\ s\}, q)\ \ dc(n, u, k, \overline{h}, \overline{z})\ \ cl(t)\ \ cn \to o(a', p', q')\ \ dc(n, u + c, k, \overline{h}, \overline{z})\ \ cl(t)\ \ cn'}$$

(Cost2)

$$\frac{a(\mathit{thisDC}) = dc \quad an = \texttt{Cost}\colon e', an' \quad [\![e']\!]^t_{a\circ l} = c \quad c > n - u \quad n \neq u \quad c' = c - (n - u) \quad an'' = \texttt{Cost}\colon c', an'}{o(a, \{l \mid [an]\ s\}, q)\ \ dc(n, u, k, \overline{h}, \overline{z})\ \ cl(t)\ \ cn \to o(a, \{l \mid [an'']\ s\}, q)\ \ dc(n, n, k, \overline{h}, \overline{z})\ \ cl(t)\ \ cn}$$

(Total)

$$\frac{\mathit{fresh}(f) \quad an = \texttt{Deadline}\colon e', an' \quad [\![e]\!]^t_{a\circ l} = dc}{o(a, \{l \mid [an]\ x = e!\textbf{total}(); s\}, q)\ \ dc(n, u, k, \overline{h}, \overline{z})\ \ cl(t) \to o(a, \{l \mid [an']\ x = f; s\}, q)\ \ dc(n, u, k, \overline{h}, \overline{z})\ \ f(n)\ \ cl(t)}$$

(Move)

$$\frac{[\![e]\!]^t_{a\circ l} = dc}{o(a, \{l \mid \textbf{moveto}(e); s\}, q)\ \ cl(t) \to o(a[\mathit{thisDC} \mapsto dc], \{l \mid s\}, q)\ \ cl(t)}$$

(Transfer)

$$\frac{\mathit{fresh}(f) \quad an = \texttt{Deadline}\colon e', an' \quad [\![e]\!]^t_{a\circ l} = dc \quad [\![e']\!]^t_{a\circ l} = dc' \quad [\![e'']\!]^t_{a\circ l} = i \quad i' = \mathit{min}(i, k)}{o(a, \{l \mid [an]\ x = e!\textbf{transfer}(e', e''); s\}, q)\ \ dc(n, u, k, \overline{h}, \overline{z})\ \ dc'(n', u', k', \overline{h'}, \overline{z'})\ \ cl(t) \to o(a, \{l \mid [an']\ x = f; s\}, q)\ \ dc(n, u, k - i', \overline{h}, \overline{z})\ \ dc'(n', u', k' + i', \overline{h'}, \overline{z'})\ \ f(i')\ \ cl(t)}$$

(Load)

$$\frac{an = \texttt{Deadline}\colon e', an' \quad [\![e]\!]^t_{a\circ l} = dc \quad \mathit{fresh}(f) \quad v = \mathit{avg}(\overline{h}, \overline{z}, i) \quad [\![e']\!]^t_{a\circ l} = i}{o(a, \{l \mid [an]\ x = e!\textbf{load}(e'); s\}, q)\ \ dc(n, u, k, \overline{h}, \overline{z})\ \ cl(t) \to o(a, \{l \mid [an']\ x = f; s\}, q)\ \ dc(n, u, k, \overline{h}, \overline{z})\ \ f(v)\ \ cl(t)}$$

(Run-Inside-Interval)

$$\frac{cn\ \ cl(t) \xrightarrow{!} cn'\ \ cl(t) \quad 0 < d \le \mathit{mte}(cn', t) \quad \lfloor t \rfloor = \lfloor t + d \rfloor}{\{cn\ \ cl(t)\} \to_t \{\mathit{timeAdv}(cn', d)\ \ cl(t + d)\}}$$

(Run-To-New-Interval)

$$\frac{cn\ \ cl(t) \xrightarrow{!} cn'\ \ cl(t) \quad 0 < d \le \mathit{mte}(cn', t) \quad \lceil t \rceil = t + d}{\{cn\ \ cl(t)\} \to_t \{\mathit{timeAdv}(\mathit{rscRefill}(cn'), d)\ \ cl(t + d)\}}$$

Figure 20: Semantics for Real-Time ABS with deployment components and resource consumption (2).

variables l or in the fields a, respectively. Rules Cond1 and Cond2 cover the two cases of conditional statements. (We omit the standard rule which unfolds while-loops into the conditional.)

Note that in the Activate rule, in order to evaluate guards on futures, the entire configuration cn is passed to the *select* function. This explains the use of brackets in this rule, which ensures that cn is bound to the full configuration and not just a part of the configuration. The same approach is used to evaluate guards in the rules Await1 and Await2 below.

Rule Suspend enables cooperative scheduling and suspends the active process to the process pool, leaving the active process *idle*. Rule Await1 consumes the **await** g statement if g evaluates to true in the current state of the object, rule Await2 adds a **suspend** statement to the process if the guard evaluates to false.

In rule Bind-Mtd the function $bind(m, o, \overline{v}, f, d)$ binds a method call in the class of the callee o. This results in a new process $\{l \mid s\}$ which is placed in the queue, where $l(destiny) = f$, $l(deadline) = d$, and where the formal parameters of m are bound to $\overline{v}$ in l.

Method calls. Rule Async-Call sends an invocation message to $[\![e]\!]_{a \circ l}$ with the unique identity f of a new future (since $fresh(f)$), the method name m, actual parameters $\overline{v}$, and deadline d. The identifier of the new future is placed in the configuration, and is bound to a return value in Return. Rule Return places the evaluated return expression in the future associated with the executing process, and stops the execution. Rule Read-Fut dereferences a future on the form $f(v)$. Note that if the future lacks a return value, it is of the form f and the reduction in this object is *blocked*.

Durations. In rule Duration1, the statement **duration**(e_1, e_2) is transformed to **duration2**(v_1, v_2) by reducing the expressions e_1 and e_2 to their values. In rule Duration2, this statement *blocks execution on the object* until the best case execution time v_1 has passed. This depends on the *time advance function*; the effect of advancing the time by a duration d is that **duration2**(v_1, v_2) is reduced to **duration2**$(v_1 - d, v_2 - d)$. The *maximal time elapse function* similarly ensures that time cannot pass beyond duration v_2 before the statement has been executed. These two functions, which control time advance in the semantics, are discussed in detail below.

Object creation. Rules New-Object1 and New-Object2 create a new object with a unique identifier o'. The object's fields are given default values by $atts(C, [\![\overline{e}]\!]^t_{a \circ l}, o', dc)$, extended with the actual values $[\![\overline{e}]\!]^t_{a \circ l}$ for the class parameters (evaluated in the context of the creating process), o' for *this* and dc for *thisDC*. In order to instantiate the remaining attributes, the process $init(C)$ will be active (this function returns *idle* if the *init* method is unspec-

ified in the class C, and it asynchronously calls `run` if the latter is specified). New-Object1 deals with deployment component annotations.

Deployment components. Rule New-DC creates a new deployment component $dc(n, 0, n, \varepsilon, \varepsilon)$ where dc is a unique identifier, n the capacity of the deployment component, and ε an empty list. Rule Emp-Annotation removes an empty list of annotations. The rules Cost1 and Cost2 capture the reduction of an object o in which the head of the statement list in the active process has a cost annotation with expression e. Rule Cost1 covers the case in which the deployment component has enough resources to execute the statement inside the time interval. Rule Cost2 covers the case in which the deployment component does not have enough resources to execute the statement inside the time interval; i.e., the required resources c are larger than the available resources $n - u$. Since we work with processing resources (as opposed to, e.g., memory resources which must be obtained atomically), we allow execution to take several time intervals, and let the cost expression be gradually reduced. In both rules, the consumed resources are added to u in dc. Rule Total assigns to the variable x the total amount of resources in dc in the current time interval. Observe that the deadline annotation is ignored, since the result is obtained in the same execution step. This also applies for **load** and **transfer**. Rule Move changes the deployment component associated with the object o to dc. The rule Transfer reallocates i' resources from dc to dc' to be effective in the next time interval. Note that if the deployment component does not have i resources available for the next time interval, only the available amount k will be reallocated. The rule Load calculates and assigns to the variable x the average percent of used resources in dc during the last i time intervals. Let $nth(\overline{h}, n)$ select the n'th element of a sequence $\overline{h}$, and $length(\overline{h})$ the number of elements in $\overline{h}$. It may be the case that $length(\overline{h}) < i$, in which case we can only calculate $\mathsf{load}(length(\overline{h}))$. Therefore, we define the average resource load in percentage (scaled from 0 to 100) as follows:

$$avg(\overline{h}, \overline{z}, i) = \sum_{j=1}^{min(i, length(\overline{h}))} \frac{nth(\overline{h}, j)}{nth(\overline{z}, j)} \times \frac{100}{min(i, length(\overline{h}))}.$$

Time advance. Time advance in the system is specified by the two rules Run-Inside-Interval and Run-To-New-Interval. Our model of time is based on maximal progress, so time will only advance when execution is otherwise blocked. The rule Run-Inside-Interval captures time advance which does not influence the resource availability in the deployment components of the system, and the rule Run-To-New-Interval captures the case when the resources

in the deployment components should be "refilled" for the next time interval.

Following the approach of Real-Time Maude [21], we define an auxiliary function $mte(cn, t)$ which computes the *maximum time elapse* of a configuration cn at time t, and an auxiliary function $timeAdv(cn, d)$ which captures the effect on a configuration cn of advancing time by a duration d. For any configuration cn and time t, the rules Run-Inside-Interval and Run-To-New-Interval allow time to advance by a duration $d \leq mte(cn, t)$. However, we are not interested in advancing time by a duration 0, which would leave the system in the same configuration. The definition of *mte* ensures that the time for renewing resources in the deployment components is never bypassed. When time advances to the next time interval, the auxiliary function $rscRefill(cn)$ is used to capture the effect of time advance on the deployment components in cn.

The auxiliary functions *mte*, *timeAdv*, and *rscRefill* are defined in Figure 21. These functions are recursively defined by cases over the system configuration; the interesting cases are objects and deployment components since these exhibit time-dependent behavior. Additional subscripted functions which apply to elements of the objects are similarly defined by cases for processes, statement, and guards.

The function *mte* calculates the largest amount by which time can advance such that no "interesting" occurrence will be missed in any object or deployment component (e.g., a worst-case duration expires or the deployment components need to be refilled). To ensure maximal progress, the maximum time elapse is 0 for enabled statements which are not time-dependent and infinite if the statement is not enabled, since time may pass when the object is blocked. A statement is not enabled if it has a cost annotation or is otherwise blocked. Thus, for a process which has a cost annotation for its head statement, time must advance before the process can proceed; the maximum time elapse of this process is infinite. Hence, *mte* returns the minimum time increment that makes some object become "unstuck", either by letting its active process continue or enabling one of its suspended processes.

The function *timeAdv* updates the active and suspended processes of all objects, decrementing the values of all deadline variables and **duration2** statements at the head of the statement list in processes. The function *rscRefill* captures the effect of time advance on the deployment components; the available resources n are refilled according to the amount of resources in k, and the histories of resource consumption $\overline{h}$ and of total allocated resources $\overline{z}$ are extended with the used resources u and the current total resources n of the previous time interval, respectively.

$$mte(cn_1\ \ cn_2, t) = min(mte(cn_1, t), mte(cn_2, t))$$
$$mte(o(a, p, q), t) = \begin{cases} mte_p(p, t) & \text{if } p \neq idle \\ mte_p(q, t) & \text{if } p = idle \end{cases}$$
$$mte(dc(n, u, k, \overline{h}, \overline{z}), t) = \lfloor t + 1 \rfloor - t$$
$$mte(cn, t) = \infty \qquad \text{otherwise}$$

$$mte_p(q_1 \circ q_2, t) = min(mte(q_1, t), mte(q_2, t))$$
$$mte_p(\{l|s\}, t) = \begin{cases} w & \text{if } s = \textbf{duration2}(b, w); s_2 \\ mte_g(g, t) & \text{if } s = \textbf{await } g; s_2 \\ 0 & \text{if } s \text{ is enabled} \\ \infty & \text{otherwise} \end{cases}$$
$$mte_p(q) = \infty \qquad \text{otherwise}$$

$$mte_g(g, t) = \begin{cases} max(mte_g(g_1, t), mte_g(g_2, t)) & \text{if } g = g_1 \wedge g_2 \\ 0 & \text{if } g \text{ evaluates to true} \\ \infty & \text{otherwise} \end{cases}$$

$$timeAdv(cn_1\ \ cn_2, d) = timeAdv(cn_1, d)\ \ timeAdv(cn_2, d)$$
$$timeAdv(o(a, p, q), d) = o(a, timeAdv_p(p, d), timeAdv_p(q, d))$$
$$timeAdv(cn, d) = cn \qquad \text{otherwise}$$

$$timeAdv_p((q_1 \circ q_2), d) = timeAdv_p(q_1, d), timeAdv_p(q_2, d)$$
$$timeAdv_p(\{l|s\}, d) = \{l[deadline \mapsto l(deadline) - d]|timeAdv_s(s, d)\}$$
$$timeAdv_p(q, d) = q \qquad \text{otherwise}$$

$$timeAdv_s(s, d) = \begin{cases} \textbf{duration2}(b - d, w - d) & \text{if } s = \textbf{duration2}(b, w) \\ \textbf{await } timeAdv_g(g, d) & \text{if } s = \textbf{await } g \\ timeAdv_s(s_1, d); s_2 & \text{if } s = s_1; s_2 \\ s & \text{otherwise} \end{cases}$$

$$timeAdv_g(g, d) = \begin{cases} timeAdv_g(g_1, d) \wedge timeAdv_g(g_2, d) & \text{if } g = g_1 \wedge g_2 \\ \textbf{duration2}(b - d, w - d) & \text{if } g = \textbf{duration2}(b, d) \\ g & \text{otherwise} \end{cases}$$

$$rscRefill(cn_1\ \ cn_2) = rscRefill(cn_1)\ \ rscRefill(cn_2)$$
$$rscRefill(dc(n, u, k, \overline{h}, \overline{z})) = dc(k, 0, k, u \circ \overline{h}, n \circ \overline{z})$$
$$rscRefill(cn) = cn \qquad \text{otherwise}$$

Figure 21: Functions controlling the advancement of time and its effect on the system configuration.

8. Related and Future Work

The concurrency model of ABS combines concurrent objects from Creol [5, 6] with concurrent object groups [22] and is reminiscent of Actors [7] and Erlang [9] processes: Object groups are inherently concurrent, conceptually each group has a dedicated processor, and there is at most one activity in a group at any time. This concurrency model has attracted attention as an alternative to multi-thread concurrency in object-orientation (e.g., [4]), and been integrated with, e.g., Java [23] and Scala [8]. Concurrent objects support compositional verification of concurrent software [6, 24], in contrast to multi-threaded object systems. Their inherent compositionality allows concurrent objects to be naturally distributed on different locations, because only an object's local state is needed to execute its methods. A particular feature of ABS, inherited from Creol, is its cooperative scheduling of method activations inside the object groups. In order to capture the timing of object-oriented models, Real-Time ABS [17] extends ABS and its tool suite to combine real-time with concurrent object models.

In the authors' early work on deployment components [15, 14], the execution cost was fixed in the language semantics; following an idea proposed in [16], resource consumption is expressed in our paper in terms of optional annotations with user-defined expressions which relate to the local state and the input parameters to methods. This way, the cost of execution in the model may be adapted by the modeler to a specific cost scenario. This allows us to abstractly model the effect of deploying concurrent objects on deployment components with different amounts of allocated resources at an early stage in the software development process, before modeling the detailed control flow of the targeted system.

Whereas this paper has focused on processing resources, initial complementary work addresses deployment components with restricted memory [25] and bandwidth [26]. A more abstract approach to user-defined resource management is discussed in [27], in which the user also specifies when resources are *released* during the execution. Modeling other resource types, as well as the semantics of more than one resource type in a model, is an area of ongoing research for the authors in the scope of the EU FP7 project Envisage, which will extend the approach taken in this paper to cloud computing, service-level agreements, code generation, and monitoring [28]. Preliminary work suggests that annotations can be used in a similar way for other resources, and that the semantics and existing interpreter can be augmented with a generic framework for handling resources.

There is an extensive literature on formal models of locations and mo-

bility based on, e.g., agents, ambient calculi, and process algebras. These models are typically concerned with maintaining correct interactions with respect to, e.g., security, link failure, or location failure. Among non-functional properties, access to shared resources have been studied through type and effect systems (e.g., [29, 30]), QoS-aware processes proposed for negotiating contracts [31], and type-based space control for space-aware processes [32]. Closer to our work, timed synchronous CCS-style processes can be compared for speed using faster-than bisimulation [33], albeit without notions of mobility or location. We are not aware of other formal models connecting reallocatable (virtual) processing capacities to locations.

Techniques for prediction or analysis of non-functional properties are based on either *measurement*, *simulation*, or *modeling* [34]. Measurement-based approaches can only be applied when an implementation already exists (i.e., fairly late in the software life-cycle), using dedicated profiling or tracing tools like JMeter or LoadRunner. Whereas simulations are traditionally done in programming languages (e.g., SIMULA), domain-specific simulation packages and dedicated simulators are very efficient inside their specific application domain but are less flexible [34]. Related work on simulation tools for virtualized resources in cloud computing are typically reminiscent of network simulators. A number of testing techniques and tools for cloud-based software systems are surveyed in [35]. In particular, CloudSim [36] and ICanCloud [37] are simulation tools using virtual machines to simulate cloud environments. CloudSim is a fairly mature tool which has already been used for a number of papers, but it is restricted to simulations on a single computer. In contrast, ICanCloud supports distribution on a cluster. Additionally CloudSim was originally based on GridSim [38], a toolkit for modeling and simulations of heterogeneous Grid resources. EMUSIM [39] is an integrated tool that uses AEF [40] (Automated Emulation Framework) to estimate performance and costs for an application by means of emulations to produce improved input parameters for simulations in CloudSim. Compared to these approaches, our work is based on a formal semantics and aims to support the developer of software applications at an early phase in the development process. The approach of this paper has been encouragingly compared to specialized simulation tools and to measurements on deployed code in two larger case studies addressing resource management in the cloud; an ABS model of the Montage case study [41] is presented in [42] and compared to results from specialized simulation tools and a large ABS model of the Fredhopper Replication Server has been compared to measurements on the deployed system in [43, 44].

Model-based approaches allow abstraction from specific system intrica-

cies, but depend on parameters provided by domain experts [45]. A survey of model-based performance analysis techniques is given in [46]. Formal systems using process algebra, Petri Nets, game theory, and timed automata have been used in the embedded software domain (e.g., [47, 48, 49]), but also to the schedulability of processes in concurrent objects [50, 51]. The latter work complements ours as it does not consider restrictions on shared deployment resources, but studies the schedulability of method activations in the context of concurrent objects as found in Real-Time ABS. User-defined schedulers for concurrent objects were introduced for Real-Time ABS in [17], using optional scheduling annotations and defaults.

Performance evaluation for component-based systems is surveyed in [52]. UML has been extended with a profile for schedulability, performance, and time (SPT) and combined with a methodology for software performance engineering (SPE) [53]. Using the UML SPT profile, Petriu and Woodside [54] informally define the Core Scenario Model (CSM) to solve questions that arise in performance model building. CSM has a notion of resource context, which reflects an operation's set of resources. CSM aims to bridge the gap between UML and techniques to generate performance models [46]. Closer to our work is the extension of VDM++ for embedded real-time systems by M. Verhoef [55] and the Palladio component model by R. Reussner *et al.* [56, 57]. In Verhoef's work, static architectures are explicitly modeled using CPUs and buses. The approach uses fixed resources targeting the embedded domain, namely processor cycles bound to the CPUs. In Palladio, components with explicit resource requirements are deployed on a static architecture where nodes have resources such as CPU, memory, and cache. In contrast to our work, Palladio uses probabilistic finite state machines and abstract from branch conditions and loop iterations. Components in Palladio are assumed to be stateless, but recent work considers an extension to stateful components [58]. Both approaches support simulation-based analysis, which is stochastic for Palladio. Both approaches consider several resources, in contrast to our work which here focuses on CPU. However, these approaches are restricted to static deployment scenarios. Our work goes beyond static scenarios to consider dynamic deployment and load balancing.

Other interesting lines of research relating to our work are techniques for static cost analysis (e.g., [59, 60]) and symbolic execution [61] for object-oriented programs, and statistical model checking [62, 63]. Since Real-Time ABS is fully formalized, it is interesting to see how such formal analysis techniques can be applied to obtain stronger analysis results than simulations. However, most tools for cost analysis and symbolic execution only consider sequential and untimed programs. In addition, programs must be fully de-

veloped before automated cost analysis can be applied. COSTABS [20] is a cost analysis tool for ABS which supports *concurrent* object-oriented programs, based on a novel notion of cost center. Our approach, in which the modeler specifies resource consumption in terms of cost annotations, could be supported by COSTABS to automatically derive cost annotations for the parts of a model that are fully implemented (see Example 7). In collaboration with Albert *et al.*, this approach has been applied for memory analysis of ABS models [25]. However, the generalization of that work for processing resources as well as for general, user-defined cost models, and its integration into the software development process currently remains future work. The separation of concerns between the resource capacity of the deployment layer and the resource consumption of the imperative layer may allow cost analysis and symbolic execution of concurrent timed programs. Extending our tool with symbolic execution allow the approximation of best- and worst-case response times for different deployment scenarios, depending on the available resources and the user load.

The work presented in this paper is based on a *maximal progress* semantics. A case study of the Fredhopper Replication Server [43], where costs were obtained by averaging observations from a real system, suggests that this gives fairly realistic results. Our framework has been extended to Monte Carlo simulations by adding a seed to the simulation tool [16]. An interesting extension of our work is to support statistical model checking [62, 63], for example by combining PVeStA [64] with our simulation tool in Maude. However, a stochastic model requires that meaningful probabilities are assigned to the different transitions of the language interpreter. One approach could be to assign probabilistic information to each deployment component, refining the notion of maximal progress. In addition, it is interesting to use stochastic modeling to specify end-user scenarios for our models.

9. Conclusion

This paper presents a simple and flexible approach to integrating deployment architectures and resource consumption into executable object-oriented models. The approach is based on a separation of concerns between the resource cost of performing computations and the resource capacity of the deployment architecture. The paper considers resources which abstractly reflect execution: each deployment component has a resource capacity per time interval and each computation step has a cost, specified by a user-defined cost expression or by a default. This separation of concerns between cost and capacity allows the performance of a model to be easily compared for

a range of deployment choices. By comparing deployment scenarios, many interesting questions concerning performance can be addressed already at an early phase of the software design.

The integration of deployment architectures into software models further allows application-level resource management policies to become an integral part of the software design. For deployment scenarios reflecting fixed architectures, it is natural to define the architecture as part of a model's main block. For deployment scenarios reflecting dynamic architectures, new deployment components may be dynamically created to model, e.g., virtualized machines initialized through a middleware layer or on the cloud. This paper explores two complementary approaches to load balancing between existing deployment components as part of the application-level resource management, both based on allowing objects to inspect the load of different parts of the deployment architecture. First, concurrent object groups may move between deployment components and, second, resources may be reallocated between deployment components.

Technically, the paper presents an extension of Real-Time ABS with a deployment layer, including linguistic primitives to express dynamic deployment architectures and resource management at the abstraction level of the modeling language as well as optional annotations with user-defined cost expressions to capture resource consumption. These primitives have been fully integrated with Real-Time ABS, which combines real-time and object-oriented models. The paper presents a complete formal semantics for the extended language and a number of examples to illustrate its usage. The presented semantics has been used to extend the ABS tool suite, which has been applied to obtain simulation results concerning performance for the presented examples.

Whereas most work on performance either specify timing or cost as part of the model (assuming a fixed deployment architecture) or measure the behavior of the compiled code deployed on an actual deployment architecture, the approach presented in this paper addresses a need in formal methods to capture models which vary over the underlying deployment architectures, for example to model deployment variability in software product lines and resource management of virtualized resource management for the cloud.

References

[1] K. Pohl, G. Böckle, F. Van Der Linden, Software Product Line Engineering: Foundations, Principles, and Techniques, Springer, 2005.

[2] S. M. Yacoub, Performance analysis of component-based applications, in: G. J. Chastek (Ed.), Proc. Second International Conference on Software Product Lines (SPLC'02), Vol. 2379 of Lecture Notes in Computer Science, Springer, 2002, pp. 299–315.

[3] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen, ABS: A core language for abstract behavioral specification, in: B. Aichernig, F. S. de Boer, M. M. Bonsangue (Eds.), Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010), Vol. 6957 of Lecture Notes in Computer Science, Springer, 2011, pp. 142–164.

[4] D. Caromel, L. Henrio, A Theory of Distributed Object, Springer, 2005.

[5] E. B. Johnsen, O. Owe, An asynchronous communication model for distributed concurrent objects, Software and Systems Modeling 6 (1) (2007) 35–58.

[6] F. S. de Boer, D. Clarke, E. B. Johnsen, A complete guide to the future, in: R. de Nicola (Ed.), Proc. 16th European Symposium on Programming (ESOP'07), Vol. 4421 of Lecture Notes in Computer Science, Springer, 2007, pp. 316–330.

[7] G. A. Agha, ACTORS: A Model of Concurrent Computations in Distributed Systems, The MIT Press, Cambridge, Mass., 1986.

[8] P. Haller, M. Odersky, Scala actors: Unifying thread-based and event-based programming, Theoretical Computer Science 410 (2–3) (2009) 202–220.

[9] J. Armstrong, Programming Erlang: Software for a Concurrent World, Pragmatic Bookshelf, 2007.

[10] G. D. Plotkin, A structural approach to operational semantics, Journal of Logic and Algebraic Programming 60-61 (2004) 17–139.

[11] D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, P. Y. H. Wong, Modeling spatial and temporal variability with the HATS abstract behavioral modeling language, in: M. Bernardo, V. Issarny (Eds.), Proc. 11th Intl. School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM 2011), Vol. 6659 of Lecture Notes in Computer Science, Springer, 2011, pp. 417–457.

[12] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. L. Talcott (Eds.), All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, Vol. 4350 of Lecture Notes in Computer Science, Springer, 2007.

[13] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theoretical Computer Science 96 (1992) 73–155.

[14] E. B. Johnsen, O. Owe, R. Schlatte, S. L. Tapia Tarifa, Validating timed models of deployment components with parametric concurrency, in: B. Beckert, C. Marché (Eds.), Proc. International Conference on Formal Verification of Object-Oriented Software (FoVeOOS'10), Vol. 6528 of Lecture Notes in Computer Science, Springer, 2011, pp. 46–60.

[15] E. B. Johnsen, O. Owe, R. Schlatte, S. L. Tapia Tarifa, Dynamic resource reallocation between deployment components, in: J. S. Dong, H. Zhu (Eds.), Proc. International Conference on Formal Engineering Methods (ICFEM'10), Vol. 6447 of Lecture Notes in Computer Science, Springer, 2010, pp. 646–661.

[16] E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, A formal model of object mobility in resource-restricted deployment scenarios, in: F. Arbab, P. Ölveczky (Eds.), Proc. 8th International Symposium on Formal Aspects of Component Software (FACS 2011), Vol. 7253 of Lecture Notes in Computer Science, Springer, 2012, pp. 185–202.

[17] J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, User-defined schedulers for real-time concurrent objects, Innovations in Systems and Software Engineering 9 (1) (2013) 29–43.
URL http://dx.doi.org/10.1007/s11334-012-0184-5

[18] B. C. Pierce, Types and Programming Languages, The MIT Press, 2002.

[19] K. G. Larsen, P. Pettersson, W. Yi, UPPAAL in a nutshell, International Journal on Software Tools for Technology Transfer 1 (1–2) (1997) 134–152.

[20] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, COSTABS: a cost and termination analyzer for ABS, in: O. Kiselyov, S. Thompson (Eds.), Proc. Workshop on Partial Evaluation and Program Manipulation (PEPM'12), ACM, 2012, pp. 151–154.

[21] P. C. Ölveczky, J. Meseguer, Semantics and pragmatics of Real-Time Maude, Higher-Order and Symbolic Computation 20 (1–2) (2007) 161–196.

[22] J. Schäfer, A. Poetzsch-Heffter, JCoBox: Generalizing active objects to concurrent components, in: European Conference on Object-Oriented Programming (ECOOP 2010), Vol. 6183 of Lecture Notes in Computer Science, Springer, 2010, pp. 275–299.

[23] A. Welc, S. Jagannathan, A. Hosking, Safe futures for Java, in: Proc. Object oriented programming, systems, languages, and applications (OOPSLA'05), ACM Press, New York, NY, USA, 2005, pp. 439–453.

[24] W. Ahrendt, M. Dylla, A system for compositional verification of asynchronous objects, Science of Computer Programming 77 (12) (2012) 1289–1309.

[25] E. Albert, S. Genaim, M. Gómez-Zamalloa, E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, Simulating concurrent behaviors with worst-case cost bounds, in: M. Butler, W. Schulte (Eds.), FM 2011, Vol. 6664 of Lecture Notes in Computer Science, Springer, 2011, pp. 353–368.

[26] R. Schlatte, E. B. Johnsen, F. Kazemeyni, S. L. Tapia Tarifa, Models of rate restricted communication for concurrent objects, Electronic Notes in Theoretical Computer Science 274 (2011) 67–81.

[27] E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, A formal model of user-defined resources in resource-restricted deployment scenarios, in: B. Beckert, F. Damiani, D. Gurov (Eds.), Proc. International Conference on Formal Verification of Object-Oriented Software (FoVeOOS'11), Vol. 7421 of Lecture Notes in Computer Science, Springer, 2012, pp. 196–213.

[28] E. Albert, F. de Boer, R. Hähnle, E. B. Johnsen, C. Laneve, Engineering virtualized services, in: M. A. Babar, M. Dumas (Eds.), 2nd Nordic Symposium on Cloud Computing & Internet Technologies (NordiCloud'13), ACM, 2013, pp. 59–63.

[29] A. Igarashi, N. Kobayashi, Resource usage analysis, ACM Transactions on Programming Languages and Systems 27 (2) (2005) 264–313.

[30] M. Hennessy, A Distributed Pi-Calculus, Cambridge University Press, 2007.

[31] R. D. Nicola, G. L. Ferrari, U. Montanari, R. Pugliese, E. Tuosto, A process calculus for QoS-aware applications, in: J.-M. Jacquet, G. P. Picco (Eds.), Proc. 7th International Conference on Coordination Models and Languages (COORDINATION'05), Vol. 3454 of Lecture Notes in Computer Science, Springer, 2005, pp. 33–48.

[32] F. Barbanera, M. Bugliesi, M. Dezani-Ciancaglini, V. Sassone, Space-aware ambients and processes, Theoretical Computer Science 373 (1–2) (2007) 41–69.

[33] G. Lüttgen, W. Vogler, Bisimulation on speed: A unified approach, Theoretical Computer Science 360 (1–3) (2006) 209–227.

[34] R. Jain, The Art of Computer Systems Performance Analysis, John Wiley & Sons, Inc., 1991.

[35] X. Bai, M. Li, B. Chen, W.-T. Tsai, J. Gao, Cloud testing tools, in: J. Z. Gao, X. Lu, M. Younas, H. Zhu (Eds.), Proc. 6th Intl. Symposium on Service Oriented System Engineering (SOSE'11), IEEE, 2011, pp. 1–12.

[36] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, 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) (2011) 23–50.

[37] A. Nuñez, J. Vázquez-Poletti, A. Caminero, G. Castañé, J. Carretero, I. Llorente, iCanCloud: A flexible and scalable cloud infrastructure simulator, Journal of Grid Computing 10 (2012) 185–209.

[38] R. Buyya, M. Murshed, GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing, Concurrency and Computation: Practice and Experience 14 (2002) 1175–1220.

[39] R. N. Calheiros, M. A. Netto, C. A. D. Rose, R. Buyya, EMUSIM: an integrated emulation and simulation environment for modeling, evaluation, and validation of performance of cloud computing applications, Software: Practice and Experience 43 (5) (2013) 595–612.

[40] R. N. Calheiros, R. Buyya, C. A. F. De Rose, Building an automated and self-configurable emulation testbed for grid applications, Software: Practice and Experience 40 (5) (2010) 405–429.

[41] E. Deelman, G. Singh, M. Livny, G. B. Berriman, J. Good, The cost of doing science on the cloud: The Montage example, in: Proceedings of the Conference on High Performance Computing (SC'08), IEEE/ACM, 2008, pp. 1–12.

[42] E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, Modeling resource-aware virtualized applications for the cloud in Real-Time ABS, in: T. Aoki, K. Tagushi (Eds.), Proc. 14th International Conference on Formal Engineering Methods (ICFEM'12), Vol. 7635 of Lecture Notes in Computer Science, Springer, 2012, pp. 71–86.

[43] F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, P. Y. H. Wong, Formal modeling of resource management for cloud architectures: An industrial case study, in: F. D. Paoli, E. Pimentel, G. Zavattaro (Eds.), Proc. European Conference on Service-Oriented and Cloud Computing (ESOCC 2012), Vol. 7592 of Lecture Notes in Computer Science, Springer, 2012, pp. 91–106.

[44] E. Albert, F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, S. L. T. Tarifa, P. Y. H. Wong, Formal modeling and analysis of resource management for cloud architectures. an industrial case study using Real-Time ABS, Journal of Service-Oriented Computing and ApplicationsAvailable online: `http://dx.doi.org/10.1007/s11761-013-0148-0`. To appear.

[45] I. Epifani, C. Ghezzi, R. Mirandola, G. Tamburrelli, Model evolution by run-time parameter adaptation, in: Proc. 31st International Conference on Software Engineering (ICSE'09), IEEE, 2009, pp. 111–121.

[46] S. Balsamo, A. D. Marco, P. Inverardi, M. Simeoni, Model-based performance prediction in software development: A survey, IEEE Transactions on Software Engineering 30 (5) (2004) 295–310.

[47] A. Vulgarakis, C. C. Seceleanu, Embedded systems resources: Views on modeling and analysis, in: Proc. 32nd IEEE Intl. Computer Software and Applications Conference (COMPSAC'08), IEEE Computer Society, 2008, pp. 1321–1328.

[48] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, M. Stoelinga, Resource interfaces, in: R. Alur, I. Lee (Eds.), Proc. Third International Conference on Embedded Software (EMSOFT'03), Vol. 2855 of Lecture Notes in Computer Science, Springer, 2003, pp. 117–133.

[49] E. Fersman, P. Krcál, P. Pettersson, W. Yi, Task automata: Schedulability, decidability and undecidability, Information and Computation 205 (8) (2007) 1149–1172.

[50] M. M. Jaghoori, F. S. de Boer, T. Chothia, M. Sirjani, Schedulability of asynchronous real-time concurrent objects, Journal of Logic and Algebraic Programming 78 (5) (2009) 402–416.

[51] F. S. de Boer, M. M. Jaghoori, E. B. Johnsen, Dating concurrent objects: Real-time modeling and schedulability analysis, in: P. Gastin, F. Laroussinie (Eds.), Proc. 21st Intl. Conf. on Concurrency Theory (CONCUR), Vol. 6269 of Lecture Notes in Computer Science, Springer, 2010, pp. 1–18.

[52] H. Koziolek, Performance evaluation of component-based software systems: A survey, Performance Evaluation 67 (8) (2010) 634–658.

[53] C. U. Smith, L. G. Williams, Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software, Addison-Wesley, 2002.

[54] D. B. Petriu, C. M. Woodside, An intermediate metamodel with scenarios and resources for generating performance models from UML designs, Software and System Modeling 6 (2) (2007) 163–184.

[55] M. Verhoef, P. G. Larsen, J. Hooman, Modeling and validating distributed embedded real-time systems with VDM++, in: J. Misra, T. Nipkow, E. Sekerinski (Eds.), Proceedings of the 14th International Symposium on Formal Methods (FM'06), Vol. 4085 of Lecture Notes in Computer Science, Springer, 2006, pp. 147–162.

[56] R. Reussner, H. W. Schmidt, I. Poernomo, Reliability prediction for component-based software architectures, Journal of Systems and Software 66 (3) (2003) 241–252.

[57] S. Becker, H. Koziolek, R. Reussner, The Palladio component model for model-driven performance prediction, Journal of Systems and Software 82 (1) (2009) 3–22.

[58] L. Happe, B. Buhnova, R. Reussner, Stateful component-based performance models, Journal of Software and Systems Modeling. Available online: `http://dx.doi.org/10.1007/s10270-013-0336-6`. To appear.

[59] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Cost Analysis of Java Bytecode, in: 16th European Symposium on Programming, (ESOP'07), Vol. 4421 of Lecture Notes in Computer Science, Springer, 2007, pp. 157–172.

[60] S. Gulwani, K. K. Mehra, T. M. Chilimbi, SPEED: Precise and Efficient Static Estimation of Program Computational Complexity, in: Z. Shao, B. C. Pierce (Eds.), Proc. 36th Symp. on Principles of Programming Languages (POPL'09), ACM, 2009, pp. 127–139.

[61] B. Beckert, R. Hähnle, P. H. Schmitt (Eds.), Verification of Object-Oriented Software. The KeY Approach, Vol. 4334 of Lecture Notes in Artificial Intelligence, Springer, 2007.

[62] K. Sen, M. Viswanathan, G. Agha, On statistical model checking of stochastic systems, in: K. Etessami, S. K. Rajamani (Eds.), Proc. 17th International Conference on Computer Aided Verification (CAV'05), Vol. 3576 of Lecture Notes in Computer Science, Springer, 2005, pp. 266–280.

[63] A. David, K. G. Larsen, A. Legay, M. Mikucionis, D. B. Poulsen, J. van Vliet, Z. Wang, Statistical model checking for networks of priced timed automata, in: U. Fahrenberg, S. Tripakis (Eds.), Proc. 9th Intl. Conf. on Formal modeling and analysis of timed systems (FORMATS'11), Vol. 6919 of Lecture Notes in Computer Science, Springer, 2011, pp. 80–96.

[64] M. AlTurki, J. Meseguer, PVeStA: A parallel statistical model checking and quantitative analysis tool, in: A. Corradini, B. Klin, C. Cîrstea (Eds.), Proc. 4th International Conference on Algebra and Coalgebra in Computer Science (CALCO'11), Vol. 6859 of Lecture Notes in Computer Science, Springer, 2011, pp. 386–392.