A Formal Model of User-Defined Resources in Resource-Restricted Deployment Scenarios [⋆]

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

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

Abstract. Software today is often developed for deployment on varying architectures. In order to model and analyze the consequences of such deployment choices at an early stage in software development, it seems desirable to capture aspects of low-level deployment concerns in high-level models. In this paper, we propose an integration of a generic cost model for resource consumption with deployment components in Timed ABS, an abstract behavioral specification language for executable object-oriented models. The actual cost model may be user-defined and specified by means of annotations in the executable Timed ABS model, and can be used to capture specific resource requirements such as processing capacity or memory usage. Architectural variations are specified by resource-restricted deployment scenarios with different capacities. For this purpose, the models have deployment components which are parametric in their assigned resources. The approach is demonstrated on an example of multimedia processing servers with a user-defined cost model for memory usage. We use our simulation tool to analyze deadline misses for given usage and deployment scenarios.

1 Introduction

Software systems often need to adapt to different deployment scenarios: *operating systems* adapt to different hardware, e.g., the number of processors; *virtualized applications* are deployed on varying (virtual) servers; and *services on the cloud* need to adapt dynamically to the underlying infrastructure. Such adaptability raises new challenges for the modeling and analysis of component-based systems.

In general, abstraction is a means to reduce complexity in a model [21]. In formal methods, the ability to execute abstract models was initially considered counter-productive because the models would become less abstract [13, 15], but recently abstract executable models have gained substantial attention and also been applied industrially in many different domains [30]. Specification languages range from design-oriented notations including UML structural diagrams, which are concerned with structural models of architectural deployment, to programming language close specification languages such as JML [6], which are best

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

suited to express functional properties. *Abstract executable modeling languages* are found in between these two abstraction levels, and appear as the appropriate abstraction level to express deployment decisions because they abstract from concrete data structures in terms of abstract data types, yet allow the faithful specification of a system's control flow and data manipulation. For the kind of properties we are considering in this paper, it is paramount that the models are indeed executable in order to have a reasonable relationship to the final code.

The abstract behavioral specification language ABS [7, 17] is such an executable modeling language with a formally defined semantics and a simulator built on the Maude platform [8]. ABS is an object-oriented language in which concurrent objects communicate by asynchronous method calls and in which different activities in an object are cooperatively scheduled. In recent work, we have extended ABS with time and with a notion of *deployment component* in order to abstractly capture resource restrictions related to deployment decisions at the modeling level. This allows us to observe, by means of simulations, the performance of a system model ranging over the amount of resources assigned to the deployment components, with respect to execution capacity [19, 20] and with respect to memory [2]. This way, the modeler gains insight into the resource requirements of a component, in order to provide a minimum response time for given client behaviors. In the approach of these papers, the resource consumption of the model was fixed by the ABS simulator (or derived by the COSTA tool [1]), so the only parameter which could be controlled by the modeler was the capacity of the deployment components.

In this paper, we take a more high-level approach to the modeling of resource usage and consider a generic cost model $\mathcal{M}$. We propose a way for the modeler to explicitly associate resource costs to different activities in a model via optional annotations in the language. This allows simulations of performance at an early stage in the system design, by further abstraction from the control flow and data structures of the system under development. Resource usage according to $\mathcal{M}$ may be specified abstractly (e.g., for whole methods), or more fine-grained and following the control flow in parts of the model which are of particular interest; it can also be refined along with the model. Resource annotations are specified by means of user-defined expressions; e.g., depending on the input parameters to a method in the model. Given a model with explicit resource annotations, we show how our simulation tool for Timed ABS may be used for abstract performance analysis of formal object-oriented models, to analyze the performance of the model depending on the resources assigned to the deployment components. The proposed approach and associated tool are illustrated on an example of a multimedia processing service with a cost model for memory allocation.

Paper overview. Section 2 introduces the Timed ABS modeling language, Section 3 presents the semantics of Timed ABS with user-defined resource annotations. Section 4 presents an application and simulation results of a photo and video processing service in Timed ABS. Section 5 discusses related work and Section 6 concludes the paper.

2 Models of Deployed Concurrent Objects in Timed ABS

Timed ABS is a timed abstract behavioral specification language for distributed concurrent objects. It is a timed extension of ABS [17], which builds on the Creol language [18] (used in our previous work [19, 20]) but provides a fully implemented functional sublanguage and support for software product line modeling [7]. Characteristic features of Timed ABS are that: (1) it allows abstracting from implementation details while remaining executable; i.e., a *functional sublanguage* over abstract data types is used to specify internal, sequential computations [17]; (2) it provides *flexible concurrency and synchronization mechanisms* by means of asynchronous method calls, release points in method definitions, and cooperative scheduling of method activations [9,18]; (3) it supports *user-provided deadlines* to method calls to express local QoS requirements [10]; and (4) it features *deployment components* with parametric resources to model deployment variability [2, 19, 20]. Compared to previous work on deployment components, we here extend the syntax and semantics of Timed ABS with *user-defined annotations* to express general cost-models for resource usage.

A Timed ABS *model* P, as shown in Figure 2, defines interfaces, classes, data types, and functions, and has a main block $\{\overline{T}\ \overline{x}; s\}$ to configure the initial state. Objects are dynamically created instances of classes; their declared fields are initialized to arbitrary type-correct values, but may be redefined in an optional method *init*. This paper assumes that models are well-typed, so method binding is guaranteed to succeed [17]. Intuitively, concurrent objects in Timed ABS have dedicated processors and live in a distributed environment with asynchronous and unordered communication. All communication is between named objects, typed by interfaces, by means of asynchronous method calls. (There is no remote field access.) Calls are asynchronous as the caller may decide at runtime when to synchronize with the reply from a call. Method calls may be seen as triggers of concurrent activity, spawning new activities (so-called *processes*) in the called object. Active behavior, triggered by an optional *run* method, is interleaved with passive behavior, triggered by method calls. Thus, an object has a set of processes to be executed, which stem from method activations. Among these, at most one process is *active* and the others are *suspended* in a process pool. Process scheduling is non-deterministic, but controlled by *processor release points* in a cooperative way. *Deployment components* restrict the natural concurrency of objects in Timed ABS by introducing resource-restricted execution contexts to capture different deployment scenarios. Every object in Timed ABS is associated with one deployment component.

The functional level of Timed ABS defines user-defined parametric datatypes and functions, as shown in Figure 1. The ground types T consist of built-in types B (such as Bool, Int, and $\mathsf{Resource}$), 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 [26]). In *datatype declarations* Dd, a datatype D has a set of constructors $Cons$, which have 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. *Expressions* e include Boolean

Basic Syntactic Categories.	
B	Built-in type names (e.g., String)
D	Datatype name
I	Interface name
N	Type variable

Definitions.

$$\begin{array}{rcl}
T & ::= & B \mid I \mid D \mid D\langle\overline{T}\rangle \\
A & ::= & N \mid T \mid D\langle\overline{A}\rangle \\
\mathit{Dd} & ::= & \textbf{data}\ D[\langle\overline{A}\rangle][=\overline{\mathit{Cons}}]; \\
\mathit{Cons} & ::= & \mathit{Co}[(\overline{A})] \\
F & ::= & \textbf{def}\ A\ \mathit{fn}[\langle\overline{A}\rangle](\overline{A\ x}) = e; \\
e & ::= & b \mid x \mid v \mid \mathit{Co}[(\overline{e})] \mid \mathit{fn}(\overline{e}) \mid \textbf{case}\ e\ \{\overline{br}\} \\
 & \mid & \textbf{this} \mid \textbf{thiscomp} \mid \textbf{deadline} \mid \textbf{now} \\
v & ::= & \mathit{Co}[(\overline{v})] \mid \textbf{null} \mid \ldots \\
\mathit{br} & ::= & p \Rightarrow e; \\
p & ::= & \_ \mid x \mid v \mid \mathit{Co}[(\overline{p})]
\end{array}$$

Fig. 1. Syntax for the functional level of Timed ABS. Terms like $\overline{e}$ and $\overline{x}$ denote (possibly empty) lists over the corresponding syntactic categories, square brackets [] denote optional elements.

expressions b, variables x, values v, the self-identifier **this**, constructor expressions $Co(\overline{e})$, function calls $fn(\overline{e})$, and case expressions **case** e $\{\overline{br}\}$. In Timed ABS, the expression **deadline** refers to the *remaining permitted execution time* of the current method activation, which is initially given by a deadline annotation at the method call site or by default. We assume that message transmission is instantaneous, so the deadline expresses the time until a reply is received; i.e., it corresponds to an *end-to-end* deadline. The expression **now** returns the current time (explained below) and **thiscomp** the deployment component to which the current object is associated. *Values* v are constructors applied to values $Co(\overline{v})$, values of the built-in types (e.g., integer and string), or **null**. *Case expressions* have a list of branches br of the form $p \Rightarrow e$, where p is a pattern. Branches are evaluated in the listed order. Patterns include wild cards _, variables x, values v, and constructor patterns $Co(\overline{p})$.

The concurrent object level of Timed ABS is given in Figure 2. 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 $\overline{x}$ of types $\overline{T}$, and methods $\overline{M}$. (The *fields* of the class are the union of class parameters and state variables.) A 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 variables $\overline{x}$ of types $\overline{T}$, and a statement s. Statements may access fields of the current class, locally defined variables, and the method's formal parameters.

Right hand side expressions rhs include object and component creation, method calls $e!m(\overline{e})$ where e evaluates to the callee and $\overline{e}$ to the actual parameters to the call, future dereferencing e.**get**, and (pure) expressions e. Method calls and future dereferencing are explained in detail below. An object may be created in the current deployment component, written **new** $C(\overline{e})$, or in a named component e, written **new** $C(\overline{e})$ **in** e. Deployment components are created by

$$
\begin{array}{ll}
\textit{Syntactic Categories.} & \textit{Definitions.} \\
C : \textsf{Class name} & P ::= \overline{Dd}\ \overline{F}\ \overline{IF}\ \overline{CL}\ \{\overline{T\ x};\ s\} \\
m : \textsf{Method name} & IF ::= \mathbf{interface}\ I\,\{\,\overline{Sg}\,\} \\
 & CL ::= [\,\llbracket an \rrbracket\,]\ \mathbf{class}\, C\,[(\overline{T\ x})]\,[\mathbf{implements}\,\overline{I}]\,\{\,\overline{T\ x};\ \overline{M}\} \\
 & Sg ::= T\ m\ ([\overline{T\ x}]) \\
 & M ::= [\,\llbracket an \rrbracket\,]\ Sg\ \{\overline{T\ x};\ \ s\,\} \\
 & an ::= \texttt{Deadline:}\ e \mid \texttt{Cost:}\ e \mid \texttt{Free:}\ e \\
 & g ::= b \mid x? \mid \mathbf{duration}(e) \mid g \wedge g \\
 & s ::= s; s \mid [\,\llbracket an \rrbracket\,]\ s \mid \mathbf{skip} \mid \mathbf{if}\ b\ \{\,s\,\}\,[\mathbf{else}\,\{\,s\,\}] \mid \mathbf{while}\ b\,\{\,s\,\} \\
 & \qquad \mid\ x = rhs \mid \mathbf{suspend} \mid \mathbf{await}\ g \mid \mathbf{duration}(e) \mid \mathbf{return}\ e \\
 & rhs ::= e \mid \mathbf{new}\ C\,(\overline{e})\ \ [\mathbf{in}\ e] \mid \mathbf{component}\,(e) \mid e.\mathbf{get} \mid e!m(\overline{e})
\end{array}
$$

Fig. 2. Syntax for the concurrent object level of Timed ABS.

component(e), where e evaluates to a number reflecting the amount of resources assigned to the component (i.e., the resource capacity of the component).

Statements are standard for sequential composition $s_1; s_2$, and **skip**, **if**, **while**, and **return** constructs. The statement **suspend** unconditionally releases the processor, suspending the active process. In **await** g, the guard g controls processor release and consists of Boolean conditions b, durations **duration**(e), and return tests $x?$ (see below). Just like functional expressions e, guards g are side-effect free. If the guard is not fulfilled, the processor is released and the process *suspended*. When the execution thread is idle, any process may be selected from the pool of suspended processes if it is ready to execute.

Communication in Timed ABS is based on asynchronous method calls, denoted by assignments $x = e!m(\overline{e})$ to future variables x. (Local calls are written **this**$!m(\overline{e})$.) After making an asynchronous call $x = e!m(\overline{e})$, the caller may proceed with its execution without blocking on the method reply. Here e evaluates to an object reference, and $\overline{e}$ are expressions providing actual parameter values for the method invocation. A future variable refers to a return value which has yet to be computed. There are two operations on future variables, which control synchronization in Timed ABS. First, the return value can be retrieved by the expression x.**get**, which blocks all execution in the object until the return value is available. Second, the guard **await** $x?$ suspends the active process unless the call associated with x has finished, allowing other processes in the object to execute in the meantime. Standard usages of asynchronous method calls include the statement sequence $x = e!m(\overline{e});\ x' = x$.**get** which encodes a *blocking call* (abbreviated $x' = o.m(\overline{e})$ and often referred to as a synchronous call, where no activity can take place in the calling object for the duration of the call), and the sequence $x = e!m(\overline{e})$; **await** $x?$; $x' = x$.**get** which encodes a non-blocking, *preemptible call*, abbreviated **await** $x' = e.m(\overline{e})$. If the return value of a call is of no interest, the call expression $e!m(\overline{e})$ may be used as a statement. In Timed ABS, it is the decision of the caller whether to call a method synchronously or asynchronously, and when to synchronize on the return value of a call.

Time. In Timed ABS, we work with a discrete time model. The local passage of time is *explicitly expressed* using **duration** statements and **duration** guards. The statement **duration**(e) expresses the passage of e time units, and blocks the whole object. Similarly, the statement **await duration**(e) suspends the current process for e time units. Note that time can also pass during synchronization with a method invocation; this can block one process (via **await** x?) or the whole object (via $x' = x$.**get**). All other statements (standard assignments, **skip** statements, etc.) do not cause time to pass.

Deployment Components can be understood as resource-restricted execution contexts which allow us to specify and compare different execution environments for Timed ABS models. Deployment components are parametric in the amount of resources they make available to their objects, which makes it easy to compare the behavior of a model under different resource assumptions. Deployment components were originally introduced for processing resources in [20]. In contrast, resources in this paper are general, and it is up to the modeler to define a cost model which fits with the specific targeted resource.

Deployment components are integrated in Timed ABS as follows. Resources are understood as a quantitative measure of cost and modeled by the built-in data type Resource which extends the integers with an "unlimited resource" ω. We define addition and subtraction for resources as for integers and by $\omega+n = \omega$ and $\omega - n = \omega$ (for integers n). In Timed ABS, deployment components are of a type Component and can be dynamically created by the statement `x=`**`component`**`(`e`)`, which assigns a new deployment component with a given quantity e of resources to `x`. All objects in a Timed ABS model belong to some deployment component. The number of objects residing on a component may grow dynamically through object creation. The ABS syntax for object creation is therefore extended with an optional clause to specify the targeted deployment component; in the expression **new** `C(`e`)` **in** `x`, the new `C` object will reside in the component `x`. Objects generated without an **in**-clause reside in the same component as their parent object. The behavior of a Timed ABS model which does not statically declare deployment components is captured by a root deployment component with ω resources.

The execution inside a deployment component is restricted by the number of available resources in the component; thus the execution in an object may need to wait for resources to become available. In general, the usage of resources for objects in a deployment component depends on a specific cost model $\mathcal{M}$, which expresses how resources are used during execution. In Timed ABS, cost models are expressed by the user by associating resource usage with the execution of different statements in the model. During the execution of a statement with cost n, the associated component consumes n resources. In a deployment component with ω resources, a transition can always be executed immediately. In a deployment component with less than n assigned resources, the object permanently blocks. Otherwise, the object needs to wait until sufficient resources become available. When time advances, resources which are no longer in use by an object may be returned to its deployment component, as determined by the cost model $\mathcal{M}$. For example, for processor resources, all resources may be returned to the deployment component when time advances. In contrast, for memory resources, the time advance corresponds to deallocating memory cells (e.g., by running a garbage collector), and will return the used stack memory and possibly some portion of the heap. (Remark that the memory deallocated after a transition may be larger than the memory allocated before the transition, which corresponds to transitions which reduce the size of the heap.)

Technically, for a cost model $\mathcal{M}$ and a transition from state t to t', the resources to be consumed by the transition are given by a function $cost_{\mathcal{M}}(t, t')$ and the resources to be returned by time advance are given by a function $free_{\mathcal{M}}(t, t')$. The time advance will return to the deployment component the accumulated number of returned resources, as determined by the sum of $free_{\mathcal{M}}(t, t')$ for all transitions in that component since the last time advance. For example, for processor resources, all resources are available in each time step so $cost_{\mathcal{M}}(t, t') = free_{\mathcal{M}}(t, t')$. For memory resources, $free_{\mathcal{M}}(t, t') = cost_{\mathcal{M}}(t, t') + size(t') - size(t)$ if we denote by $size(t)$ the size of the heap for state t. The cost model $\mathcal{M}$ is determined by the exact definitions of $cost_{\mathcal{M}}(t, t')$ and $free_{\mathcal{M}}(t, t')$ for the statements in the language. In order to give the modeler explicit control over the use of resources, the statements of Timed ABS have zero cost by default, which can be overridden by annotations defining the cost model.

Annotations. We extend the syntax of Timed ABS with the following resource and deadline annotations: `Cost:` e, `Free:` e, and `Deadline:` e. Annotations *an* are optional and may be associated with class and method declarations, as well as with statements. *Deadline annotations* interact with time to reflect soft end-to-end deadlines; i.e., deadlines may be violated in the model. The annotation `Deadline:` e may only be associated with method calls and specifies the relative time before which a method activation should complete its execution. Method calls without annotations get the infinite deadline by default. In the sequel, we assume a model without misplaced annotations (no classes with deadlines etc).

Resource annotations interact with time and deployment components to express the resource management of a given cost model $\mathcal{M}$. By *default*, execution in an object does not affect the resources of the associated deployment component. Annotations `Cost:` e and `Free:` e will *override* this default. For example, a statement `[Cost:` e`] s` (where e evaluates to an integer r) can only be executed if r resources can be removed from the associated deployment component. Similarly, `[Free:` e`] s` (where e evaluates to r) can be executed immediately and expresses that the execution of `s` returns r resources to the deployment component; these resources become available at the next time advance. For class declarations the annotations `Cost:` e and `Free:` e may be used to specify the resource consumption associated with the creation of an object of that class and the resources to be freed after the object creation (on the deployment component associated with the new object). For method declarations, the annotation `Cost:` e may be used to override a default resource consumption reflecting the method activation, and `Free:` e may be used to override the default amount of resources to be freed after method activation. The detailed semantics is given in Fig. 6.

$$\begin{array}{rcl@{\qquad}rcl}
cn & ::= & \epsilon \mid obj \mid comp \mid msg \mid fut \mid cn\ \ cn & s & ::= & \mathbf{timer}(v) \mid \ldots \\
obj & ::= & o(\sigma, pr, q) & v & ::= & o \mid f \mid dc \mid \ldots \\
comp & ::= & dc(av, fr, tot) & pr & ::= & \{\sigma|s\} \mid idle \\
msg & ::= & m(o, \overline{v}, f, d) & \sigma & ::= & x \mapsto v \mid \sigma \circ \sigma \\
fut & ::= & f \mid f(v) & q & ::= & \epsilon \mid pr \mid q \circ q \\
tcn & ::= & \{cn\ \ cl(t)\} & & &
\end{array}$$

Fig. 3. Runtime syntax; here, o and f are object and future identifiers, d and c are the deadline and cost annotations.

3 Semantics

This section presents the operational semantics of Timed ABS as a transition system in an SOS style [27]. 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 [22]). A run is a possibly nonterminating sequence of rule applications.

The evaluation rules for the functional level of Timed ABS are standard and have been omitted for brevity.

3.1 Runtime Configurations

The runtime syntax is given in Figure 3. We add identifiers for objects, components, and futures to the values v and an auxiliary statement $\mathbf{timer}(v)$ to the statements s. *Configurations* cn are sets of objects, components, messages, and futures. A *timed configuration* tcn adds a global clock $cl(t)$ to a configuration (where t is a time value). Timed configurations live inside curly brackets; thus, in $\{cn\ cl(t)\}$, cn captures the *entire* system configuration at time t. The associative and commutative union operator on (timed) configurations is denoted by whitespace and the empty configuration by ε. An *object* obj is a term $o(\sigma, pr, q)$ where o is the object's identifier, σ a substitution representing the object's fields, pr a process, and q a *pool of processes*. A *substitution* σ is a mapping from variables x to values v. Let $\sigma[x \mapsto v]$ denote the substitution such that $\sigma[x \mapsto v](y) = \sigma(y)$ if $x \neq y$ and $\sigma[x \mapsto v](x) = v$. For substitutions and process pools, concatenation is denoted by $\sigma_1 \circ \sigma_2$ and $q_1 \circ q_2$, respectively. A *process* pr is a structure $\{\sigma|s\}$, where s is a list of statements and σ a substitution representing the bindings of local variables and of two local system variables *destiny* (storing the future for the process's return value) and *deadline*, assuming no name conflicts. A process with an empty statement list is denoted by *idle*.

A *component* $comp$ is a structure $dc(av, fr, tot)$ with integers dc, av, and tot such that dc is the component's identifier, av the unallocated resources, fr the deallocated resources, and tot the resources initially assigned to the component. An *invocation message* msg is a structure $m(o, \overline{v}, f, d)$, where 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 the provided deadline of the call. A *future* fut

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 for object layout and method definitions.

3.2 A Transition System for Timed Configurations

General Expressions. Expressions are evaluated in the context of the fields of an object, the local variables of a process, and the current time. For simplicity, we let **this** and **thiscomp** be appropriately bound fields, and let *destiny* and *deadline* be appropriately bound local variables, assuming no name conflicts. We denote by $[\![e]\!]^t_\sigma$ the result of evaluating e in the context $\sigma[now \mapsto t]$.

Evaluating Guards. A guard g is evaluated to determine whether a given process starting with **await** g is ready to be scheduled. Given a substitution σ, a time t, and a configuration cn, we denote by $[\![g]\!]^{t,cn}_\sigma$ an evaluation function which reduces guards g to Boolean values, defined as follows: $[\![b]\!]^{t,cn}_\sigma = [\![b]\!]^t_\sigma$, $[\![g_1 \wedge g_2]\!]^{t,cn}_\sigma = [\![g_1]\!]^{t,cn}_\sigma \wedge [\![g_2]\!]^{t,cn}_\sigma$, $[\![\mathtt{duration}(e)]\!]^{t,cn}_\sigma = \text{true if } [\![e]\!]^t_\sigma \leq 0$, $[\![x?]\!]^{t,cn}_\sigma = \text{true if } [\![x]\!]^t_\sigma = f$ and $f(v) \in cn$ for some value v, otherwise $[\![x?]\!]^{t,cn}_\sigma = \text{false}$. (For details on the reduction of functional expressions and guards, see [17].)

Transition rules transform (timed) configurations into new (timed) configurations, and are given in Figures 4 and 6. Note that the clock value t is passed through all transition rules in which general expressions or guards are evaluated, since their evaluation may depend on the current time. We denote by a the substitution which represents the fields of an object and by l the substitution which represents the local variable bindings of a process. In the semantics, different assignment rules are defined for side effect free expressions (Assign1 and Assign2), object and component creation (New-Object1, New-Object2, and New-Component), method calls (Async-call), and future dereferencing (Read-Fut). Rule Skip consumes a **skip** in the active process $\{l|\mathbf{skip}; s\}$, which becomes $\{l|s\}$. 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 variables l or in the fields a, respectively. Rules Cond1 and Cond2 cover the two cases of conditional statements in the same way. (We omit the rule for **while**, which unfolds into the conditional.)

Scheduling. Rule Await1 consumes the **await** g statement at the head of the active process if g evaluates to true in the current object state a, time t, and configuration cn, rule Await2 adds a suspend statement to the head of the active process if the guard evaluates to false. The evaluation of guard expressions with future variables motivates the use of outermost brackets in this rule, which technically ensures that cn is the rest of the global system configuration. The rule Schedule applies when the active process is *idle*, if there is some process pr which is *ready* in the current object state a, time t, and configuration cn. If pr is a process, a a substitution, t a time value, and cn a configuration, we let $ready(pr, a, cn, t)$ denote a predicate which checks if pr is ready to execute (in the sense that the processes will not directly suspend or block the object's processor [18]). The outcome of the rule application is that pr becomes the active process and is removed from the process queue q. (Note that two operations

$$\begin{array}{c}\text{(Skip)}\\ o(a,\{l|\mathtt{\textbf{skip}};s\},q)\\ \to o(a,\{l|s\},q)\end{array}\qquad \overset{\text{(Assign1)}}{\dfrac{v=[\![e]\!]^{t}_{a\circ l}\quad x\in dom(l)}{\begin{array}{c}o(a,\{l|x=e;s\},q)\ cl(t)\to\\ o(a,\{l[x\mapsto v]|s\},q)\ cl(t)\end{array}}}\qquad \overset{\text{(Assign2)}}{\dfrac{v=[\![e]\!]^{t}_{a\circ l}\quad x\in dom(a)}{\begin{array}{c}o(a,\{l|x=e;s\},q)\ cl(t)\to\\ o(a[x\mapsto v],\{l|s\},q)\ cl(t)\end{array}}}$$

$$\overset{\text{(Cond1)}}{\dfrac{[\![e]\!]^{t}_{a\circ l}}{\begin{array}{c}o(a,\{l|\mathtt{\textbf{if}}\ e\ \{s_1\}\ \mathtt{\textbf{else}}\ \{s_2\};s\},q)\ cl(t)\\ \to o(a,\{l|s_1;s\},q)\ cl(t)\end{array}}}\qquad \overset{\text{(Cond2)}}{\dfrac{\neg[\![e]\!]^{t}_{a\circ l}}{\begin{array}{c}o(a,\{l|\mathtt{\textbf{if}}\ e\ \{s_1\}\ \mathtt{\textbf{else}}\ \{s_2\};s\},q)\ cl(t)\\ \to o(a,\{l|s_2;s\},q)\ cl(t)\end{array}}}$$

$$\overset{\text{(Await1)}}{\dfrac{[\![g]\!]^{t,cn}_{a\circ l}}{\begin{array}{c}\{o(a,\{l|\mathtt{\textbf{await}}\ g;s\},q)\ cl(t)\ cn\}\\ \to\{o(a,\{l|s\},q)\ cl(t)\ cn\}\end{array}}}\qquad \overset{\text{(Await2)}}{\dfrac{\neg[\![g]\!]^{t,cn}_{a\circ l}}{\begin{array}{c}\{o(a,\{l|\mathtt{\textbf{await}}\ g;s\},q)\ cl(t)\ cn\}\to\\ \{o(a,\{l|\mathtt{\textbf{suspend}};\mathtt{\textbf{await}}\ g;s\},q)\ cl(t)\ cn\}\end{array}}}$$

$$\overset{\text{(Schedule)}}{\dfrac{ready(pr,a,cn,t)\quad pr\in q}{\begin{array}{c}\{o(a,\text{idle},q)\ cl(t)\ cn\}\to\\ \{o(a,pr,(q\setminus pr))\ cl(t)\ cn\}\end{array}}}\qquad \begin{array}{c}\text{(Suspend)}\\ o(a,\{l|\mathtt{\textbf{suspend}};s\},q)\\ \to o(a,\text{idle},\{l|s\}\circ q)\end{array}$$

$$\overset{\text{(Duration)}}{\dfrac{v=[\![e]\!]^{t}_{a\circ l}}{\begin{array}{c}o(a,\{l|\mathtt{\textbf{duration}}(e);s\},q)\ cl(t)\\ \to o(a,\{l|\mathtt{\textbf{timer}}(v);s\},q)\ cl(t)\end{array}}}\qquad \overset{\text{(Timer)}}{\dfrac{v=0}{\begin{array}{c}o(a,\{l|\mathtt{\textbf{timer}}(v);s\},q)\\ \to o(a,\{l|s\},q)\end{array}}}\qquad \overset{\text{(Tick)}}{\dfrac{\text{canAdv}(cn,t)}{\begin{array}{c}\{cn\ cl(t)\}\\ \to\{\text{Adv}(cn)\ cl(t+1)\}\end{array}}}$$

$$\overset{\text{(Read-Fut)}}{\dfrac{f=[\![e]\!]^{t}_{a\circ l}}{\begin{array}{c}o(a,\{l|x=e.\mathtt{\textbf{get}};s\},q)\ f(v)\ cl(t)\\ \to o(a,\{l|x=v;s\},q)\ f(v)\ cl(t)\end{array}}}\qquad \overset{\text{(Return)}}{\dfrac{v=[\![e]\!]^{t}_{a\circ l}\quad f=l(\text{destiny})}{\begin{array}{c}o(a,\{l|\mathtt{\textbf{return}}(e);s\},q)\ f\ cl(t)\\ \to o(a,\{l|s\},q)\ f(v)\ cl(t)\end{array}}}$$

Fig. 4. Non-annotated transitions in the semantics of ABS.

manipulate a process pool q; $pr \circ q$ adds a process pr to q and $q \setminus pr$ removes pr from q.) Rule Suspend consumes the **suspend** primitive at the head of the active process $\{l|\textbf{suspend}; s\}$ and places the process $\{l|s\}$ in the process pool q, which leaves the active process *idle*.

Durations. In rule Duration, a statement **duration**(e) is reduced to the runtime statement **timer**(v), in which the expression e has been reduced to a value v. This statement blocks execution on the object until time value v has been decremented to 0 by means of the Tick rule (see below), after which rule Timer becomes applicable. Remark that time cannot advance beyond duration v before the statement has been executed.

Time advance. To simplify the logging of resource consumption, we consider a discrete time model in which time always advances by one unit. Rule Tick specifies how time advances in the system. In order to capture timed concurrent behavior with an interleaving semantics, we use a run-to-completion approach in which a transition must occur at the current time if possible. We follow the approach of Real-Time Maude [23, 24] and let time advance uniformly through the configuration *cn*. Auxiliary functions, defined in Fig. 5, specify the advancement of time and its effect. The predicate *canAdv* determines whether time may advance at the current state at the current time, reflecting the run-to-completion approach; intuitively, this means that given a configuration *cn* and a time t, $\mathit{canAdv}(\mathit{cn}, t)$ should only be true when no transition rule apart from Tick is applicable. (In the first line of the definition in Figure 5, we let cn' denote the base case: cn' is a configuration which does not contain objects or messages.) The function $\mathit{Adv}(\mathit{cn})$ specifies how the advancement of time affects different parts of the configuration *cn*. Both *canAdv* and *Adv* have the whole configuration as input but mainly consider objects and deployment components since these exhibit time-dependent behavior. The function *Adv* makes freed resources available in the deployment components (so they become available after the time advance, see below) and applies two auxiliary functions to each object; Adv_1 applies to the active process and Adv_2 applies to the process queue of the object. These functions decrement the *deadline* variable of the processes, reflecting that time has advanced. In addition, they call auxiliary functions Adv_3 and Adv_4 on the statement lists of the active and suspended processes, respectively. The functions decrement time values in the head of the statement list. The difference between the two functions lies in their treatment of the **`timer`** primitive: for an active process, the time value of the **`timer`** primitive is decremented, for a suspended process not. Finally, Adv_5 performs the similar time decrements on a guard expression (at the head of the statement list).

Annotations and Resources. Figure 6 presents the transition rules related to the creation of deployment components and resource annotations. We consider three kinds of annotations in this paper: *deadline*, *cost*, and *free*. The deadline annotation is associated with method calls, and is handled by the rule Async-Call (explained below). The activation of a method and the creation of a class may affect the resources at *both* the sender and receiver side. Receiver side annotations are associated with the declaration of methods and classes, while sender side annotations may be associated with any statement in the model.

Rule No-Annotations reduces a statement with an empty list of annotations to a statement without annotations. Rule Cost-Annotation controls resource usage, removing the specified amount e from the available resources of the deployment component in which the execution occurs (identified by the value of *thiscomp*). Observe that execution can only occur if there are enough resources available, given by the constraint $c \leq av$. Rule Free-Annotation similarly controls the freeing of resources, and adds the specified amount e to the resources which are available in the deployment component after the next time advance.

Auxiliary functions. Let $\mathit{class}(o)$ denote the class of an object with identifier o and let $\mathit{bind}(m, C, \overline{v}, f, d)$ return a process resulting from the activation of m in class C, with actual parameters $\overline{v}$, associated future f, and deadline d. If binding succeeds, the method's formal parameters are bound to $\overline{v}$, the reserved variables

$$
\begin{array}{ll}
\mathrm{canAdv}(cn', t) = \mathit{true} & \textit{cn' contains no objects or messages}\\
\mathrm{canAdv}(\mathit{msg}\ cn, t) = \mathit{false} & \textit{messages are instantaneous}\\
\mathrm{canAdv}(o(a, \{l|[\texttt{cost:}e, an]\ s\}, q)\ dc(av, \mathit{fr}, \mathit{tot})\ cn, t) & \textit{not enough resources}\\
\quad = \mathrm{canAdv}(o(a, \{l|[an]\ s\}, q)\ dc(av, \mathit{fr}, \mathit{tot})\ cn, t) & \textit{(other annotations ignored)}\\
\quad \wedge\, a(\mathrm{thiscomp}) = dc \wedge [\![e]\!]^t_{aol} = c \wedge c > av & \\
\mathrm{canAdv}(o(a, \{l|x = e.\mathbf{get}; s\}, q)\ f\ cn, t) & \textit{o is blocked and}\\
\quad = \mathrm{canAdv}(f\ cn, t) \wedge [\![e]\!]^t_{aol} = f & \textit{no value is available}\\
\mathrm{canAdv}(o(a, \{l|\mathbf{timer}(v); s\}, q)\ cn, t) & \textit{o must wait}\\
\quad = \mathrm{canAdv}(cn, t) \wedge v > 0 & \\
\mathrm{canAdv}(o(a, \mathrm{idle}, q)\ cn, t) & \textit{no ready processes}\\
\quad = \mathrm{canAdv}(cn, t) \wedge \neg\mathrm{ready}(pr, a, cn, t) \text{ for all } pr \in q & \\
\mathrm{canAdv}(\mathit{obj}\ cn, t) = \mathit{false} & \textit{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{Adv}(o(a, pr, q)\ cn) & = o(a, \mathrm{Adv}_1(pr), \mathrm{Adv}_2(q))\ \mathrm{Adv}(cn) & \\
\mathrm{Adv}(dc(av, \mathit{fr}, \mathit{tot})\ cn) & = dc(av + \mathit{fr}, 0, \mathit{tot})\ \mathrm{Adv}(cn) & \\
\mathrm{Adv}(cn) & = cn & \textit{otherwise}
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{Adv}_1(\mathrm{idle}) & = \mathrm{idle}\\
\mathrm{Adv}_1(\{l|[an]s\}) & = \{l[\mathrm{deadline} \mapsto l(\mathrm{deadline}) - 1]|\mathrm{Adv}_3(s)\}
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{Adv}_2(\emptyset) & = \emptyset\\
\mathrm{Adv}_2(\{l|[an]s\} \circ q) & = \{l[\mathrm{deadline} \mapsto l(\mathrm{deadline}) - 1]|[an]\mathrm{Adv}_4(s)\} \circ \mathrm{Adv}_2(q)
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{Adv}_3(s) & = \begin{cases} \mathbf{timer}(v - 1) & \text{if } s = \mathbf{timer}(v)\\ \mathbf{await}\ \mathrm{Adv}_5(g) & \text{if } s = \mathbf{await}\ g\\ \mathrm{Adv}_3(s_1) & \text{if } s = s_1; s_2\\ s & \text{otherwise} \end{cases}\\
\mathrm{Adv}_4(s) & = \begin{cases} \mathbf{await}\ \mathrm{Adv}_5(g) & \text{if } s = \mathbf{await}\ g\\ \mathrm{Adv}_4(s_1) & \text{if } s = s_1; s_2\\ s & \text{otherwise} \end{cases}\\
\mathrm{Adv}_5(g) & = \begin{cases} \mathrm{Adv}_5(g_1) \wedge \mathrm{Adv}_5(g_2) & \text{if } g = g_1 \wedge g_2\\ \mathbf{timer}(v - 1) & \text{if } g = \mathbf{timer}(v)\\ g & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 5. Functions controlling the advancement of time.

destiny and *deadline* are bound to f and d, respectively. Let $\mathit{annotations}(m, C)$ return the receiver side annotations associated with the definition of m in C. If m is *init*, the annotations of the class definition are returned. If there are no associated annotations, it returns ε. Let $\mathit{atts}(C, \overline{v}, o, dc)$ return the initial state of an instance of class C, in which the formal class parameters are bound to $\overline{v}$ and the reserved variables *this* and *thiscomp* are bound to o and dc. Let $\mathit{init}(C)$ return an activation of the *init* method of C, if defined, and call the *run* method, if defined. Otherwise it returns *idle*.

Method Calls. Rule Async-Call sends an invocation message of method m to $[\![e]\!]^t_{aol}$ with actual parameters $[\![\overline{e}]\!]^t_{aol}$, the unique identity f of a new future (since $\mathit{fresh}(f)$), and a *deadline* d obtained from the (possibly default) annotation. The identifier of the new future is placed as a marker in the configuration. In rule

$$\textsc{(No-Annotations)}\quad \frac{\begin{array}{c}\{o(a,\{l|s\},q)\ cn\}\\ \to\{o(a',pr',q')\ cn'\}\end{array}}{\begin{array}{c}\{o(a,\{l|[\varepsilon]\,s\},q)\ cn\}\\ \to\{o(a',pr',q')\ cn'\}\end{array}}$$

$$\textsc{(Cost-Annotation)}\quad \frac{\begin{array}{c}c\leq av\quad [\![e]\!]^t_{aol}=c\quad a(\text{thiscomp})=dc\\ \{o(a,\{l|[an]\,s\},q)\ dc(av-c,fr,tot)\ cl(t)\ cn\}\\ \to\{o(a',pr',q')\ dc(av',fr',tot)\ cl(t)\ cn'\}\end{array}}{\begin{array}{c}\{o(a,\{l|[\mathtt{cost}:e,an]\,s\},q)\ dc(av,fr,tot)\ cl(t)\ cn\}\\ \to\{o(a',pr',q')\ dc(av',fr',tot)\ cl(t)\ cn'\}\end{array}}$$

$$\textsc{(Free-Annotation)}\quad \frac{\begin{array}{cc}[\![e]\!]^t_{aol}=f & \{o(a,\{l|[an]\,s\},q)\ dc(av,fr,tot)\ cl(t)\ cn\}\\ a(\text{thiscomp})=dc & \to\{o(a',pr',q')\ dc(av',fr',tot)\ cl(t)\ cn'\}\end{array}}{\begin{array}{c}\{o(a,\{l|[\mathtt{free}:e,an]\,s\},q)\ dc(av,fr,tot)\ cl(t)\ cn\}\\ \to\{o(a',pr',q')\ dc(av',fr'+f,tot)\ cl(t)\ cn'\}\end{array}}$$

$$\textsc{(Activation)}\quad \frac{\begin{array}{c}\text{class}(o)=C\\ \text{annotations}(m,C)=an\\ \{l|s\}=\text{bind}(m,C,\overline{v},f,d)\end{array}}{\begin{array}{c}o(a,pr,q)\ m(o,\overline{v},f,d)\\ \to o(a,pr,\{l|[an]\,s\}\circ q)\end{array}}$$

$$\textsc{(Async-Call)}\quad \frac{\begin{array}{c}\mathit{fresh}(f)\quad an=\mathtt{deadline}:e'\\ o'=[\![e]\!]^t_{aol}\quad \overline{v}=[\![\overline{e}]\!]^t_{aol}\quad d=[\![e']\!]^t_{aol}\end{array}}{\begin{array}{c}o(a,\{l|[an]\,x=e!m(\overline{e});s\},q)\ cl(t)\to\\ o(a,\{l|x=f);s\},q)\ cl(t)\ m(o',\overline{v},f,d)\ f\end{array}}$$

$$\textsc{(New-Component)}\quad \frac{\mathit{fresh}(dc)\quad v=[\![e]\!]^t_{aol}}{\begin{array}{c}o(a,\{l|x=\mathbf{component}(e);s\},q)\ cl(t)\\ \to o(a,\{l|x=dc;s\},q)\ dc(v,0,v)\ cl(t)\end{array}}$$

$$\textsc{(New-Object1)}\quad \frac{\begin{array}{c}\text{fresh}(o')\quad a'=\text{atts}(C,[\![\overline{e}]\!]^t_{aol},o',[\![e]\!]^t_{aol})\\ \{l'|s'\}=\text{init}(C)\quad \text{annotations}(\text{init},C)=an\end{array}}{\begin{array}{c}o(a,\{l|x=\mathbf{new}\ C(\overline{e})\ \mathbf{in}\ e;s\},q)\ cl(t)\to\\ o(a,\{l|x=o';s\},q)\ o'(p',a',\{l'|[an]\,s'\},\emptyset)\ cl(t)\end{array}}$$

$$\textsc{(New-Object2)}\quad \frac{v=a(\text{thiscomp})}{\begin{array}{c}o(a,\{l|x=\mathbf{new}\ C(\overline{e});s\},q)\to\\ o(a,\{l|x=\mathbf{new}\ C(\overline{e})\ \mathbf{in}\ v;s\},q)\end{array}}$$

Fig. 6. Annotated transitions in the semantics of ABS.

Activation the function $bind(m,C,\overline{v},f,d)$ binds a method call to object o in the class of o and $annotations(m,C)$ returns the annotations associated with the method declaration. This results in a new process $\{l|[an]s\}$ which is placed in the queue, where $l(\text{destiny})=f$, $l(\text{deadline})=d$, and where the formal parameters of m are bound to $\overline{v}$. Rule Return (in Figure 4) places the evaluated return expression v in the future f associated with the *destiny* variable of the process, expanding the future f to $f(v)$. Rule Read-Fut (in Figure 4) dereferences the future $f(v)$. Unless the future has a return value, this reduction is *blocked.*

Creation of Objects and Deployment Components. Rule New-Component creates a deployment component with a unique name dc and a specified number of resources e, which are initially available. Rule New-Object1 creates a new object with a unique name o' in a deployment component dc. The fields are given default values by $\text{atts}(C,\overline{v},o',dc)$, extended with the actual values $\overline{v}$ for the class parameters, o' for *this*, and dc for *thiscomp*. To instantiate the remaining fields, the process $\text{init}(C)$ is scheduled, with the appropriate class annotations. Rule New-Object2 reduces to an object creation without a deployment destination to an object creation in the current deployment component (i.e., **thiscomp**).

```
interface Server {
   Bool request(Int nFrames, Int bc, Int wc, Duration dl);
   Bool processframe(Int bc, Int wc);
}
class ServerImp implements Server {
   Bool request(Int nFrames, Int bc, Int wc, Duration dl) {
      List<Fut<Bool>> results = Nil;
      Bool result = True;
      while (nFrames > 0) {
        [Deadline: dl] Fut<Bool> r = this!processframe(bc, wc);
        results = Cons(r, results);
        nFrames = nFrames - 1;
        }
      while (~(results == Nil)){
        await (head(results))?;
        Bool fr = head(results).get; result = result && fr;
        results = tail(results);
      }
      return result;
   }
   Bool processframe(Int bc, Int wc) {
      [Cost: wc] await duration(bc) ;
      [Free: wc] return deadline() > 0;
   }
}

interface Client { }
class ClientImp (Server s, Int frequency, Int nJobs,
             Int nFrames, Int bc, Int wc, Duration dl)
implements Client {
   Unit run() {
     if (nJobs > 0) {
      Fut<Bool> res = s!request(nFrames, bc, wc, dl);
      await duration(frequency);
      nJobs = nJobs - 1;
      this!run(); await res?;
      }
   }
}
```

Fig. 7. Photo and video processing example.

4 Example

The semantics of Section 3 have been implemented in rewriting logic to define a language interpreter in the Maude tool [8] for Timed ABS with user-defined resources. This section briefly presents a working example to illustrate the usage of our approach, and simulation results obtained using the language interpreter.

Consider the model of a service for processing a job consisting of a number of work packets, e.g., batch-converting a number of photos or video frames (Figure 7). A client calls the `request` method of a `Server` object; the number of

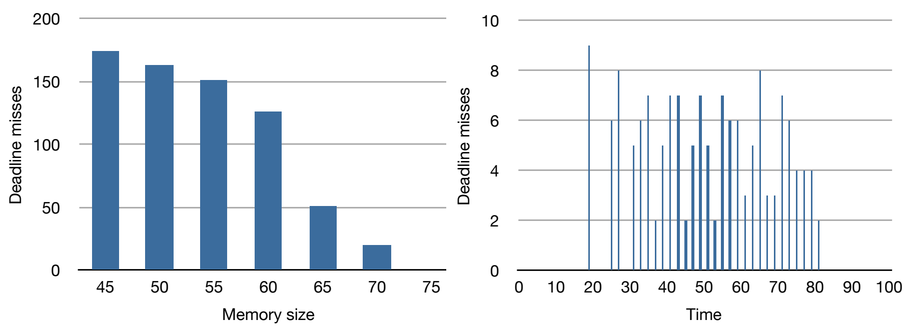

Fig. 8. Deadline misses as a function of memory size (left); deadline misses over time for memory size=55 (right)

frames, and a deadline for processing are given as parameters to the `request` method, as are (specifications of) the time and memory processing cost for each work unit. These last two parameters can be seen as abstractions of the size of the image. The class `ClientImp` can be used to simulate specific workloads, dispatching jobs of a given size with a certain frequency to the server.

Simulating Resource-Restricted Behaviors. A simple scenario illustrates simulation of resource restricted behavior in Timed ABS, with the following client:

```
new ClientImp (s, 1, 1, 3, 3, 3, Duration(8));
```

These values for the class parameters correspond to 1 job with 3 frames, best and worst case cost 3 for each frame, and a deadline at time 8. We couple this client with a server `s` running in a deployment component with 3 memory units results in one missed deadline. Since the available memory allows only one process to run at a time, the three frames are processed in sequence and the last one cannot meet the deadline. Increasing available memory to 6 allows the job to run within the specified deadline.

Let us now consider a more involved scenario, with two clients running a job with 10 frames every 6 time units, for a total of 10 jobs, which would be done in time $t = 60$ for an unconstrained server. Figure 8 shows the results of simulating this scenario, varying over the resources available on the servers. The number of missed deadlines varies with the amount of memory available for processing, up to 75 memory units which are enough to process the given workload scenario without deadline misses. The behavior of the server over time can also be visualized; Figure 8 also shows deadline misses over time on memory size 55 for the same scenario.

5 Related Work

The concurrency model provided by concurrent objects and Actor-based computation, in which software units with encapsulated processors communicate asynchronously, is increasingly getting attention due to its intuitive and compositional nature [3, 14, 29]. This compositionality allows concurrent objects to be naturally distributed on different locations, because only the local state of a concurrent object is needed to execute its methods. In previous work [2, 19, 20], the authors have introduced *deployment components* as a modeling concept which captures restricted resources shared between a group of concurrent objects, and shown how components with parametric resources may be used to capture a model's behavior for different assumptions about the available resources.

Techniques for prediction or analysis of non-functional properties are based on either *measurement* or *modeling*. Measurement-based approaches apply to existing implementations, using dedicated profiling or tracing tools like JMeter or LoadRunner. Model-based approaches allow abstraction from specific system intricacies, but depend on parameters provided by domain experts [11]. A survey of model-based performance analysis techniques is given in [4]. Formal systems using process algebra, Petri Nets, game theory, and timed automata have been used in the embedded software domain (e.g., [5,12]), but also to the schedulability of processes in concurrent objects [16]. The latter work complements ours as it does not consider restrictions on shared deployment resources, but associates deadlines with method calls with abstract duration statements.

Work on modeling object-oriented systems with resource constraints is more scarce. Using the UML SPT profile for schedulability, performance, and time, Petriu and Woodside [25] 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 [4]. Closer to our work is M. Verhoef's extension of VDM++ for embedded real-time systems [28], in which architectures are explicitly modeled using CPUs and buses. The approach uses fixed resources targeting the embedded domain, namely processor cycles bound to the CPUs, while we consider more general resources for arbitrary software.

6 Discussion and Future Work

We have described a generic way of adding resource constraints to a executable behavioral model. This expands on previous work, where the cost of execution was bound to a specific resource and directly fixed in the language semantics. In contrast, this paper generalized the approach by enabling the specification of resource costs as part of the software development process, supported by explicit user-defined cost statements expressed in terms of the local state and the input parameters to methods. This way, the cost of execution in the model can be adapted by the modeler to a specific cost scenario. Our extension to ABS allows us to abstractly model the effect of deploying concurrent objects on deployment components with different amounts of assigned resources at an early stage in the software development process, before modeling the detailed control flow of the targeted system.

This approach gives the modeler full freedom to provide a user-defined cost model inside ABS, albeit with the overhead of manual resource management. For specific resources, more specialized notations can make expressing resource constraints easier—for example, for CPU (processing) resources, cost and free are always the same amount, while for power consumption resources are never freed, except when modeling an operator recharging a battery. Alternatively, our approach can be supported by a cost analysis tool such as COSTA [1]). In fact, we have combined our approach with automatic resource analysis for a predefined cost model for memory [2]. However, the generalization of that work for general, user-defined resources and its integration into the software development process remains future work.

Another extension of this work is to strengthen the available analysis methods via a number of language extensions: User-defined schedulers for ABS, probabilistic scheduling and load balancing of resources and objects will provide more precise simulation results given knowledge of the deployment platform and show the range of possible behaviors with respect to system performance. Prototype implementations of these features exist, and our current work is dedicated to integrating them in the base language and tools.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning*, 46:161–203, 2011.
2. E. Albert, S. Genaim, M. Gómez-Zamalloa, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Simulating concurrent behaviors with worst-case cost bounds. In *FM 2011*, volume 6664 of *LNCS*, pages 353–368. Springer, June 2011.
3. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
4. S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.
5. A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Resource interfaces. In *Proc. EMSOFT'03*, volume 2855 of *LNCS*, pages 117–133. Springer, 2003.
6. P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Proc. FMCO'05*, volume 4111 of *LNCS*, pages 342–363. Springer, 2005.
7. D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Y. H. Wong. Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In *Proc. SFM 2011*, volume 6659 of *LNCS*, pages 417–457. Springer, 2011.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.
9. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Proc. ESOP'07*, volume 4421 of *LNCS*, pages 316–330. Springer, Mar. 2007.

10. F. S. de Boer, M. M. Jaghoori, and E. B. Johnsen. Dating concurrent objects: Real-time modeling and schedulability analysis. In *Proc. CONCUR'10*, volume 6269 of *LNCS*, pages 1–18. Springer, Sept. 2010.
11. I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *Proc. ICSE'09*, pages 111–121. IEEE, 2009.
12. E. Fersman, P. Krcál, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.
13. N. E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, pages 323–334, September 1992.
14. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.
15. I. Hayes and C. Jones. Specifications are not (Necessarily) Executable. *Software Engineering Journal*, pages 330–338, November 1989.
16. M. M. Jaghoori, F. S. de Boer, T. Chothia, and M. Sirjani. Schedulability of asynchronous real-time concurrent objects. *Journal of Logic and Algebraic Programming*, 78(5):402–416, 2009.
17. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proc. FMCO'10*, volume 6957 of *LNCS*, pages 142–164. Springer, 2011.
18. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
19. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Dynamic resource reallocation between deployment components. In *Proc. ICFEM'10*, volume 6447 of *LNCS*, pages 646–661. Springer, Nov. 2010.
20. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Validating timed models of deployment components with parametric concurrency. In *Proc. FoVeOOS'10*, volume 6528 of *LNCS*, pages 46–60. Springer, 2011.
21. J. Kramer. Is Abstraction the Key to Computing? *Communications of the ACM*, 50(4):37–42, 2007.
22. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
23. P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285(2):359–405, 2002.
24. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1–2):161–196, 2007.
25. D. B. Petriu and C. M. Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Software and System Modeling*, 6(2):163–184, 2007.
26. B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
27. G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.
28. M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In *Proc. FM'06*, volume 4085 of *LNCS*, pages 147–162. Springer, 2006.
29. A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. OOPSLA'05*, pages 439–453, New York, NY, USA, 2005. ACM Press.
30. J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):1–36, October 2009.