Innovations in Systems and Software Engineering manuscript No.
DOI: 10.1007/s11334-012-012-0184-5

User-defined Schedulers for Real-Time Concurrent Objects

Joakim Bjørk · Frank S. de Boer · Einar Broch Johnsen · Rudolf Schlatte · S. Lizeth Tapia Tarifa

Received: 30 April 2011 / Accepted: 2 May 2012

Abstract Scheduling concerns the allocation of processors to processes, and is traditionally associated with low-level tasks in operating systems and embedded devices. However, modern software applications with soft real-time requirements need to control application-level performance. High-level scheduling control at the application level may complement general purpose OS level scheduling to fine-tune performance of a specific application, by allowing the application to adapt to changes in client traffic on the one hand and to low-level scheduling on the other hand. This paper presents an approach to express and analyze application specific scheduling decisions during the software design stage. For this purpose, we integrate support for application-level scheduling control in a high-level object-oriented modeling language, Real-Time ABS, in which executable specifications of method calls are given deadlines and real-time computational constraints. In Real-Time ABS, flexible application-specific schedulers may be specified by the user, i.e., developer, at the abstraction level of the high-level modeling language itself and associated with concurrent objects at creation time. Tool support for Real-Time ABS is based on an abstract interpreter which supports simulations and measurements of systems at the design stage.

Keywords Scheduling policies · real-time · modeling languages · object orientation · operational semantics

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

Joakim Bjørk
University of Oslo, Norway
E-mail: joakimbj@ifi.uio.no

Frank S. de Boer
CWI, Amsterdam, The Netherlands
E-mail: frb@cwi.nl

Einar Broch Johnsen
University of Oslo, Norway
E-mail: einarj@ifi.uio.no

Rudolf Schlatte
University of Oslo, Norway
E-mail: rudi@ifi.uio.no

S. Lizeth Tapia Tarifa
University of Oslo, Norway
E-mail: sltarifa@ifi.uio.no

1 Introduction

The scheduling problem concerns the allocation of available processors to unfinished processes. This is a non-trivial problem as processes in general are dynamically created, have possibly conflicting needs, and their execution time depends on the size of their input. Operating systems usually use heuristics to guess the optimal ordering of their processes, called scheduling policies. Despite many years of research on optimizing scheduling policies at the level of operating systems, scheduling is largely beyond the control of most existing high-level modeling and programming languages, whose purpose is to relieve the software developer from implementation and deployment details by means of suitable abstractions. However, in general-purpose programming, software engineers increasingly need to express and control not only functional correctness but also quality of service in their designs. On the other hand, operating systems by their very nature do not consider the specific requirements of different applications and this can greatly affect application-level performance. For optimal use of both hardware and software resources, we therefore cannot avoid leveraging scheduling and performance related issues from the underlying operat-

ing system to the software engineering level, which necessitates timed semantics for general-purpose modeling and programming languages [27]. Today, domain-specific languages for cyber-physical systems with soft (or firm) real-time requirements start to provide cooperative, non-preemptive scheduling with deadlines at the application level [38]. Furthermore, emerging support for reflection in middleware allows applications to dynamically control the creation of virtual processors and to define user-level schedulers [8]. Flexible application-specific schedulers are needed in new application domains with soft real-time requirements, such as multimedia streaming. To fully exploit increasing platform virtualization, a major challenge in software engineering is to find a balance between these two conflicting requirements of abstraction and deployment control.

This paper presents Real-Time ABS, a high-level abstract behavioral specification language for modeling distributed systems, which supports the integration of realizable abstractions of requirements related to deployment, e.g., scheduling and deadlines. We achieve this by modeling the components of the distributed system as loosely interacting concurrent objects. Concurrent objects are similar to the Actor model [1] and Erlang [6] processes in that they represent a *unit of distribution*; i.e., their interaction does not transfer control. Each concurrent object has a queue of method activations (stemming from method calls), among which at most one may be executing at any time. This model of computation is currently attracting interest due to its potential for distribution on multicore platforms, for example in terms of Scala actors [19], concurrent object groups in Java [36], Kelim lightweight threads [39], and concurrent Creol objects in Java [30].

An important aspect of the concurrent object model in Creol [25] and ABS [24] is that the scheduling of method activations from an object's queue is *cooperative* but *non-deterministic*. The cooperative scheduling is non-preemptive, but a method activation may yield control at explicitly declared points in its execution, leaving the object idle. When the object is idle, scheduling is non-deterministic in the sense that any enabled method activation may resume its execution. In Real-Time ABS, we extend this computation model with real-time and with the ability to refine the non-deterministic per-object scheduler by allowing each concurrent object to embody its own scheduling policy, tailored towards attaining its quality of service. These high-level scheduling policies provide run-time adaptability, which is beyond the state-of-the-art fixed-priority scheduling in operating systems. Thus, we both leverage and generalize scheduling issues from the operating system to the application level. Our approach is based on a two-level scheduling scheme (e.g., [12, 35, 40]). At the top level, objects on virtual servers decide on the scheduling of requests by encapsulating an application-level scheduling policy. However, these virtual servers may be realized in terms of one or several virtual or physical machines. The bottom level scheduling, which decides on the arbitration between the virtual servers, is not adressed in this paper.

By expressing per object scheduling policies at the software *modeling* level, quality-of-service and deployment requirements may be analyzed and resolved at design time. For example in a real-time setting, we must guarantee a maximum on average response-time (end-to-end deadlines) or a minimum on the level of system throughput. The main technical contributions of this paper are

- a real-time object-oriented modeling language associating schedulers with concurrent objects and deadlines with method calls, and its formal semantics;
- user-defined schedulers expressed at the abstraction level of the modeling language; and
- tool support based on an abstract interpreter.

The formal semantics of Real-Time ABS rigorously defines the real-time behavior of a system of concurrent objects. However, the expressivity of the Real-Time ABS modeling language means that models in Real-Time ABS are out of scope for automata-based model-checking techniques for full state-space exploration (such as, e.g., Uppaal [26]). Based on the formal semantics, we have therefore implemented a prototype abstract interpreter for Real-Time ABS in Maude [14], an executable rewriting logic framework which also supports real-time rewriting logic [31, 32]. Technically, user-defined schedulers are achieved by allowing a degree of reflection in the language, such that the runtime representation of the process queue of objects is lifted into an expression inside the modeling language itself. The abstract interpreter is not bound to any particular hardware or deployment platform; instead, its architecture is directly based on the concepts of Real-Time ABS. The abstract interpreter can be used for both systematic simulation and measurement of Real-Time ABS models. Systematic simulations allow us to simulate different runs for our executable models, even before creating a detailed implementation of the software. As for measurements, profiling techniques for Real-Time ABS can be used to, e.g., find bottlenecks, data and communication intensity, maximum queue sizes, race conditions, etc.

Related Work. Real-Time ABS is a real-time extension of ABS [24], which simplifies Creol [9, 25] by, e.g., ignoring class inheritance. This journal paper integrates

and extends two previous papers on timed concurrent objects by the authors [7, 10], and additionally introduces the concept of user-defined per object schedulers. A discrete time Creol interpreter was proposed in [7], in which the passage of time is indirectly observable by comparing observations of a global clock. In contrast, duration statements and deadlines for method calls were introduced and formalized with a real-time semantics in [10], without a language interpreter. In this paper, we follow the latter approach at the level of the surface syntax, additionally introduce duration guards which complement duration statements, and develop an abstract interpreter for the resulting language.

An encoding of real-time concurrent objects into timed automata was proposed in [10]. This encoding follows the approach of task automata [18], which allows schedulability analysis in Uppaal [26] by means of the Times Tool [3]. Complementing this work, we develop an abstract interpreter based on the formal semantics of real-time concurrent objects as proposed in this paper, by integrating the real-time model of Real-Time Maude [31, 32] with the Maude interpreter for ABS developed in the EU FP7 project HATS: Highly Adaptable and Trustworthy Software using Formal Models. Real-Time ABS provides a much more expressible language than that of, for example, the Times Tool. This additional expressive power is needed for the high-level description of complex data-intensive systems of real-time concurrent objects enhanced with user-defined schedulers, which is proposed in this paper.

Scheduling has been studied extensively in many different research communities, including real-time [13, 16], parallel algorithms [5, 23], operating systems [22], measurement and modeling [20], and job scheduling [15]. These communities have mostly worked independently with somewhat different concerns, and with little cooperation. Traditional approaches to schedulability analysis, especially in operations research, can handle only a restricted range of events, e.g., periodic ones that are generated with fixed inter-arrival times. In this paper we integrate application-level scheduling of non-uniform dynamically created processes with an executable high-level real-time modeling language.

The state of the art in parallel and distributed programming is the multi-threading paradigm, as in Java and the pthread library (usually in C++). Despite its popularity, multi-threading has some generally recognized drawbacks; for example, shared memory access between threads makes it not suitable for programming distributed applications. In addition, as the thread of execution travels across object boundaries (through synchronous method calls), there is no natural borderline to decompose a program for distributed deployment. New programming languages, e.g., Scala and Erlang (as described in [19] and [28], respectively), employ new paradigms that are fundamentally different from and alleviate some of the shortcomings of mainstream languages like Java, but still provide very limited control to influence the underlying scheduling policy. Some amount of controllability is achieved by Real-Time Specification for Java (RTSJ) [17] which allows assigning priorities to Java threads which can be used by a Fixed Priority Scheduler in the JVM. Schoeberl shows how a system-wide fixed priority scheduler can be defined inside Java itself [37]. Erlang has an efficient scheduler which improves the execution performance, but lacks proper language support to influence the scheduling policy. We are not aware of languages which support per-object user-defined schedulers at the abstraction level of the high-level programming or modeling language, as proposed in Real-Time ABS.

2 Real-Time ABS

Real-Time ABS is a high-level modeling language for real-time concurrent objects with user-defined schedulers. This section presents the integration of real-time and concurrent object-oriented modeling in Real-Time ABS. Section 3 presents the integration of local schedulers into the language.

Real-Time ABS consists of a functional level and a concurrent object level. A Real-Time ABS *model* defines interfaces, classes, datatypes, and functions, and a *main block* to configure the initial state. Objects are dynamically created instances of classes; their attributes are initialized to type-correct default values (e.g., `null` for object references), but may be redefined in an optional method *init*. Datatypes and functions are specified at the functional level, this allows a model to abstract from concrete imperative data structures inside the concurrent objects.

The concurrent object level focuses on the concurrency, communication, and synchronization aspects of a model, given in terms of interacting, concurrent objects. In Real-Time ABS, every concurrent object has its own queue of processes corresponding to the method activations. All objects are executing concurrently and each object executes one process at a time; in the sequel we refer to this process as the active process of the object.

2.1 The Functional Level of Real-Time ABS

The functional level of Real-Time ABS is used to model data manipulation in an intuitive way, without com-

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

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

mitting to specific low-level imperative data structures at an early stage in the software design. The functional level defines user-defined parametric datatypes and functions, as shown in Fig. 1. 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 [33]). 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 expressions b, variables x, values v, the self-identifier **this**, constructor expressions $Co(\overline{e})$, function expressions $fn(\overline{e})$, and case expressions **case** e $\{\overline{br}\}$. The expression **destiny** refers to the concurrent object level and denotes the future in which the return from the current method activation shall be stored [9]. The expression **deadline** refers to the timed aspect of a model and denotes the relative deadline of the current method activation. *Values* v are constructors applied to values $Co(\overline{v})$ or **null** (omitted from Fig. 1 are values of basic types such as String or the rational numbers Rat). *Case expressions* have a list of branches $p \Rightarrow e$, where p is a pattern. Branches are evaluated in the listed order. Patterns include wild cards _, variables x, values v, and constructor patterns $Co(\overline{p})$. For simplicity, Real-Time ABS does not currently support operator overloading.

Example 1 (Dense Time in Real-Time ABS) We consider a dense time model represented by a type Duration that ranges over non-negative rational numbers and is used for time intervals, e.g. deadlines and computation costs. To express infinite (or unbounded) durations, there is a term `InfDuration` such that for all other durations d_1, d_2, $d_1 + d_2$ is smaller than `InfDuration`.

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

Predicates are defined in the obvious way, shown here are a unary predicate `isInfinite` and the binary less-than-or-equal relation `lte` on Durations.

```
def Bool isInfinite(Duration d) =
  case d { InfDuration => True; _ => False;};

def Bool lte(Duration d1, Duration d2) =
  case d1 { InfDuration => d2 == InfDuration;
    Duration(v1) => case d2 {
      InfDuration => True;
      Duration(v2) => v1 ≤ v2;};};
```

Two Duration values can be added together. Since there is no operator overloading in ABS, we define the 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);};};
```

The operators ≤ and + are used for comparison and addition in the underlying datatype of rational numbers.

2.2 The Concurrent Object Level of Real-Time ABS

The concurrent object level of Real-Time ABS is used to specify concurrency, communication, and synchronization in the system design. The syntax of the concurrent object level 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 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 of the current class, locally defined variables, and the method's formal parameters. A program's main block is a method body $\{\overline{T}\ \overline{x}; s\}$. There are no type variables at the concurrent object level of Real-Time ABS.

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

Fig. 2 Syntax for the concurrent object level of Real-Time ABS.

Annotations in Real-Time ABS are used to extend a model with information to control the quantitative properties of the model, while maintaining a separation of concerns with the qualitative behavior of the model. Annotations a are optional and may be associated with class and method declarations, new objects, and method calls as follows. For *class declarations* and *object creation*, the annotation `Scheduler`: e may be used to override a default scheduling policy with a user-defined policy e. User-defined scheduling policies are explained in Section 3.

For *method declarations*, the annotation `Cost`: d may be used to override a default cost estimate for the method activation. Cost estimates are functions which depend on the arguments to method calls, so d is an expression over the formal parameters $\overline{x}$ of the method and will be evaluated for the actual parameter values to the call. For the purposes of this paper the cost estimate is assumed to be of type Duration, but one could also consider, e.g., memory or other resources. (Observe that cost expressions are estimations (i.e., they are specified as part of the design and not measured at runtime). We do not consider how to find good cost estimations in this paper. However, a companion tool COSTABS may in many cases automatically assist in inferring cost annotations for methods in Real-Time ABS models [2].) The cost provides a heuristics which may be used for the scheduling of method activations. Method declarations without cost annotations get a default of no cost.

For *method calls*, we consider two annotations. The annotation `Deadline`: d is of type Duration and provides a deadline for the method return. The deadline specifies the relative time before which the corresponding method activation should finish execution. The annotation `Critical`: b specifies the perceived level of criticality of the method activation: `True` indicates that the method activation is *hard* and should be given priority [11]. In contrast, missed deadlines for *soft* method activations are logged but do not influence scheduling. This way, the caller may decide on the deadline and criticality for the call. Calls without annotations get default values: infinite deadline and soft criticality.

Right hand side expressions rhs include object creation **new** $C(\overline{e})$, method calls $[o]!m(\overline{e})$ and $[o.]m(\overline{e})$, future dereferencing x.**get** and pure expressions e. Method calls and future dereferencing are explained below.

Statements are standard for sequential composition $s_1; s_2$, and **skip**, **if**, **while**, and **return** constructs. Assignments may be given optional annotations as explained above. The statement **suspend** unconditionally suspends the active process of an object by adding it to its queue from which subsequently an enabled process is taken for execution. In **await** g, the guard g controls suspension of the active process and consists of Boolean conditions b and return tests $x?$ (see below). Just like pure expressions e, guards g are side-effect free. If g evaluates to false, the executing process is suspended, i.e., added to the object's queue, and another process is taken from the queue for execution. By means of a user-defined scheduling policy enabled processes can be selected for execution from the object's queue.

Communication in Real-Time ABS is based on asynchronous method calls, denoted by assignments $f = o!m(\overline{e})$ to future variables f. (Local calls are written **this**$!m(\overline{e})$.) After making an asynchronous method call $x := o!m(\overline{e})$, the caller may proceed with its execution without blocking on the method reply. Here x is a future variable, o is an object expression, and $\overline{e}$ are (data value or object) expressions providing actual parameter values for the method invocation. The future variable x refers to a return value which has yet to be computed. There are two operations on future variables, which control synchronization in Real-Time ABS. First, the guard **await** $x?$ suspends the active process unless a return to the call associated with x has arrived, allowing other processes in the object to execute. Second, the return value is retrieved by the expression x.**get**, which blocks all execution in the object until the return value is available. In Real-Time 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. Standard usages of asynchronous method calls include the state-

ment sequence $x := o!m(\overline{e})$; $v := x.\mathbf{get}$ which encodes a *blocking call*, abbreviated $v := o.m(\overline{e})$ (often referred to as a synchronous call), and the statement sequence $x := o!m(\overline{e})$; **await** $x?$; $v := x.\mathbf{get}$ which encodes a non-blocking, *preemptible call*, abbreviated **await** $v := o.m(\overline{e})$. As usual, if the return value of a call is of no interest, the call may occur directly as a statement $o!m(\overline{e})$.

Time. In Real-Time ABS, the local passage of time is *explicitly expressed* using **duration** statements and **duration** guards. The statement **duration**(b, w) expresses the passage of time, given in terms of an interval between the best case b and the worst case w duration (assuming $b \leq w$), and blocks the whole object. The guard **duration**(b, w), when used inside an **await** statement expresses the suspension time of the process in terms of a similar interval, and lets other processes of the object run in the meantime. Note that time can also pass during synchronization with a method invocation; this can block one process (via **await** $f?$) or the whole object (via $x = f.\mathbf{get}$). All other statements (normal assignments, **skip** statements, etc.) do not cause time to pass.

Inside a method body, the read-only local variable **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. If declared, an integer variable `value` may be assigned a value to express the relative importance of the current method activation with respect to other method activations executing in the object [11]. (By default, the method activation is unimportant and has value zero.)

3 Scheduling Strategies in Real-Time ABS

Scheduling refers to the way processes are assigned to run on the available processors of a runtime system. We here present the general notions of process scheduling, following Buttazzo [11], and relate these to the context of concurrent objects in Real-Time ABS. A processor is assigned to various concurrent processes according to a predefined criterium which is called a *scheduling policy*. The realization of a scheduling policy as an algorithm which, at any time, determines the order in which processes are executed is called a *scheduling algorithm* or *scheduler*. The problem in our context is to define a scheduling algorithm to select the next process among an object's method activations when the object is idle. For this purpose, we shall represent (runtime) *processes* as a datatype in Real-Time ABS in such a way that we can express *scheduling algorithms* as functions inside the language.

Processes. Real-time systems are characterized by computational activities with timing constraints that must be met in order to achieve the desired behavior. A typical timing constraint on a process is the *deadline*, which represents the time before which the process should complete its execution. Depending on the consequence of a missed deadline, a process is usually called *hard* if a completion after its deadline can cause catastrophic consequences on the system and *soft* if missing its deadline decreases the performance of the system but does not jeopardize the system's correct behavior. In general, real-time processes are characterized by a number of well-known parameters. In the context of Real-Time ABS models, these parameters are either part of the *specification* supplied by the modeler, or else *measured* at runtime:

- *Arrival Time* r (measured at runtime): the time when a process becomes ready for execution.
- *Computation Time* c (specified together with the method definition): the time needed by the object to execute the process without interruption,
- *Relative Deadline* d (specified at method call site): the time before which a process should be completed to avoid damage or performance degradation. It is specified with respect to the arrival time.
- *Start Time* s (measured at runtime): the time at which a process starts its execution.
- *Finish Time* f (measured at runtime): the time at which a process finishes its execution.
- *Criticality crit* (specified at method call site): a parameter related to the consequence of missing a deadline (hard or soft).
- *Value* v (specified / set during process execution): a value representing the relative importance of a process with respect to other processes.

The following further parameters can be calculated from the parameters above:

- *Absolute Deadline* (D): similar to the relative deadline, but with respect to time *zero* (i.e., $D = r + d$).
- *Response Time* (R): the difference between the finish time and the arrival time (i.e., $R = f - r$).
- *Lateness* (L): the possible delay of the process completion with respect to the absolute deadline (i.e., $L = f - D$). Note that if the completion happens before the deadline then L is negative.
- *Tardiness* (E) or exceeding time: the time a process stays active after its deadline (i.e., $E = max(0, L)$).

– *Laxity* (X): the maximum time a process can delay its activation to complete within its deadline (i.e., $X = r - c$).

These parameters can be used to define different scheduling policies, scheduling algorithms, as well as for performance evaluation; e.g., the *lateness* can be used to observe the optimality of a given scheduling policy or algorithm.

Processes in Real-Time ABS. Based on the above parameters, we define datatypes for *time* `Time`, *process identifiers* `Pid`, and *processes* `Process` as follows:

```
data Time = Time(Rat);
data Pid ;
data Process =
 Proc(Pid pid, String  m, Time r, Duration c,
 Duration d, Time s, Time f, Bool crit, Int v);
```

Here, `m` represents the method name associated with the process, `r` represents the arrival time recorded when the method is bound, `c` represents the computation time or cost (the cost heuristics is an optional annotation to method declarations), `d` the relative deadline (an optional annotation of the method call), `s` the starting time, `f` the finishing time, `crit` the criticality (an optional annotation of the method call) and `v` the value representing the relative importance of the process (this is an assignable local variable in the process). `pid` is a token identifying the underlying process and does not have a syntax defined at the language level; this ensures that user-defined scheduling algorithms can under no circumstance choose a non-existing process to be run. For two `Time` values t_1 and t_2, denote by $t_1 \leq t_2$ their comparison defined by comparing their rational arguments, and by $t_1 - t_2$ their subtraction which is of type Duration. For easier readability, we also define observer functions for the process parameters, e.g. for the process name:

```
def String name(Process p) =
    case p {Proc(_,m,_,_,_,_,_,_,_) => m;};
```

The observer functions for process id, arrival time, cost, deadline, value, etc. are defined in the same way and are used in defining the scheduling policies. The lifting of runtime method activations into the datatype `Process` is explained in Section 4.1.

3.1 General Scheduling Policies

During the modeling of software systems, it is important to consider which scheduling policy will provide the best application-level performance for the corresponding system. There is no universal "best" scheduling policy, and it is possible to both define new scheduling policies and to combine different scheduling policies. In Real-Time ABS, a scheduler algorithm is a function which, given a non-empty list of processes, returns the next process to be executed. Some examples of schedulers in Real-Time ABS are given below.

Example 2 The *default* scheduler, which selects the first process in the list for execution, is defined as follows:

```
def Process default (List<Process> l) = head(l);
```

Example 3 Many well-known scheduling policies fit into a general pattern: they define a total ordering of processes depending on some ordering predicate. Such scheduling policies can be realized in Real-Time ABS using a pattern consisting of a function `scheduler` which takes a non-empty list of processes and returns the best process according to a comparison function *comp* implementing the ordering predicate. A helper function `schedulerH` compares the best process found so far to the remaining list of processes.

```
def Process scheduler (List<Process> l) =
       schedulerH(head(l), tail(l));

def Process schedulerH(Process p1,
           List<Process> l1) =
 case l1 {
    Nil => p1;
    Cons(p2,l2) =>
        if (comp(p1, p2))
        then schedulerH(p1,l2)
        else schedulerH(p2,l2) };
```

In order to realize a specific scheduling policy, we follow the pattern above and replace *comp*`(p1,p2)` with a suitable comparison function for the desired policy (since Real-Time ABS does not support first-class function arguments, we open-code the concrete algorithm).

– *Earliest deadline first (edf)* is a dynamic scheduling policy that selects processes according to their absolute deadline. Processes with earlier deadlines will be executed with a higher priority.
  The scheduling function `edf(l)` is obtained by using the following comparison function:

  ```
  def Bool comp_edf(Process p1, Process p2) =
      lte(deadline(p1), deadline(p2));
  ```

– *First in, first out (fifo)* selects processes according to their arrival order. The scheduling function `fifo(l)` uses the following comparison function:

  ```
  def Bool comp_fifo(Process p1, Process p2) =
      arrival(p1) ≤ arrival(p2);
  ```

– *Fixed priority (fp)* selects the process with the greatest fixed assigned priority from the process queue. In our setting, it is natural to fix the priority of processes depending on the name of the activated methods. Let `fp(l)` be the scheduler defined as above, but using the following comparison function:

```
def Bool comp_fp(Process p1, Process p2) =
    weight(name(p1)) ≥ weight(name(p2));

def Int weight(String s) =
    case s  {"method1" => v1;
             "method2" => v2; ...};
```

– *Dynamic priority (dp)* is similar to *fp* but the priority is not fixed. Rather, the priority depends on the v (value) attribute of the processes, which can be modified by the process itself during execution. `dp(l)` is the scheduler defined as above using the following comparison function (where lower value gives higher priority):

```
def Bool comp_dp(Process p1, Process p2) =
    value(p1) ≤ value(p2);
```

– *Shortest job first (sjf)* selects the process with the least remaining computation time (cost) from the queue. Let `sjf(l)` be the scheduler defined as above, which uses the following comparison function:

```
def Bool comp_sjf(Process p1, Process p2) =
    lte(cost(p1), cost(p2));
```

Different scheduling policies may be combined. This may be illustrated by the following example.

Example 4 (Combined sjf and dp) We consider a combination of *sjf* and *dp*, where the process with the lowest cost is selected among the processes with the highest priority. In this case, we define a filter `highPri` and call `sjf` on the filtered list of processes:

```
def List<Process> highPri(List<Process> l1,
                          List<Process> l2) =
case l2 {
 Nil => l1;
 Cons(h,t) =>
   if (l1 == Nil)
   then highPri(Cons(h,Nil),t)
   else if (comp_dp (head(l1),h))
        then if (value(head(l1)) == value(h))
             then highPri(Cons(h,l1), t)
             else highPri(l1,t)
         else highPri(Cons(h,Nil),t) };

def Process sjfdp(List<Process> l) =
   sjf(highPri(Nil,l));
```

3.2 Conditional Scheduler

Here we show how to define a scheduler which changes the priority of a set of processes depending on the length of the process queue.

Example 5 We consider an object which has a set of processes to be scheduled by some scheduling algorithm `scheduler`. Some of these processes, e.g., a load balancer, need to have high priority when there is congestion inside the object. Let l be a non-empty process queue, n an integer representing the queue length limit (i.e., the process priority changes when the size of the queue l grows beyond n), and let *ccp* denote the set of method names for the processes with conditionally changing priority. Let `filter` be a function which filters processes with names contained in *ccp* from the process queue l.

```
def Process condScheduler(List<Process> l,
            List<String> ccp, Int n) =
   if (length(l) ≤ n || filter(ccp,l) = Nil)
   then scheduler(l)
   else scheduler(filter(ccp,l))
```

Note that the list *ccp* of method names as well as the the queue length limit n are highly application dependent. In fact, these parameters of the scheduling function can be state dependent in Real-Time ABS.

3.3 Scheduling Annotations in Real-Time ABS

In Real-Time ABS, there are three levels at which the scheduling policy of a concurrent object can be determined. There is a default system-level scheduling policy, as defined in Example 2. This default scheduling policy may be overridden for all instances of a class by providing a scheduling annotation for the class definition. This class-level scheduling policy may in turn be overridden for a specific instance by a scheduling annotation at the object creation statement. In this way, different instances of a class may have different scheduling policies, for example to improve the performance for different application specific user scenarios. In a scheduling annotation, the keyword **`queue`** is used to refer to the ABS datatype representation of the runtime process queue of the scheduled object. Additional formal parameters of a scheduling function are bound to object attributes. This means that the object state can influence the scheduling, Example 9 in Section 5 gives an example.

3.4 Monitors with Signal and Continue Discipline

A monitor [4, 21] is a high-level synchronization mechanism which protects shared state by only allowing access to the shared state through a given set of methods. The defining characteristic of a monitor is that its methods are executed with mutual exclusion; similar to processes in a Real-Time ABS object, at most one process may be executing at any time inside the monitor. Monitors use *condition variables* to delay processes until the monitors's state satisfies some Boolean condition. This condition variable is also used to awaken a delayed process when the condition becomes true. The

value of a *condition variable* is associated with a *fifo* queue of delayed processes. Processes which are blocked on *condition variables* get awakened by a signal call.

In Real-Time ABS, objects may be regarded as abstract monitors without the need for explicit signaling, which is guaranteed by the semantics and need not be the responsibility of the modeler. However, the order in which processes are activated in the process queue of a Real-Time ABS object does not guarantee a *fifo* ordering of the queue. When explicit signaling is desired, monitors with different signaling disciplines can be encoded. We now show how such an ordering may be explicitly enforced independent of the scheduling policy of the object (following [25]), and then how this synchronization code can be simplified by specifying a *fifo* scheduling policy for the object.

Example 6 Consider a class implementing a general monitor with the *signal and continue* discipline [4]; for simplicity the example is restricted to one condition variable. Without any assumption about the scheduling strategy for the process queue of the monitor, we need to introduce synchronization code to ensure that suspended processes are activated following a *fifo* ordering. This can be achieved using a triple $\langle s, d, q\rangle$ of natural numbers; where s represents available signals to the condition variable, d the number of the delayed process in the queue of the condition variable, and q the number of delayed processes that have been reactivated.

```
interface Monitor { Unit wait();
   Unit signal(); Unit signalAll();}

class MonitorImp() implements Monitor {
    Int s = 1; Int d = 0; Int q = 0;
    Unit wait() { Int myturn = d+1;
       d = d+1;  await (s>0 ∧ q+1==myturn);
       s=s-1; q=q+1;}
    Unit signal() { if (d>q) {s=s+1;} }
    Unit signalAll() {s=d-q;} }
```

Example 7 By ensuring a *fifo* scheduling policy for the monitor object, the synchronization code of Example 6 can be significantly simplified. Let l represent the length of the queue of waiting processes and s the number of available signals as before.

```
[Scheduler: fifo(queue)]
class SimpleMonitorImp() implements Monitor {
    Int s = 1;  Int l = 0;
    Unit wait() { l = l+1;  await s>0;
       s=s-1; l=l-1;}
    Unit signal() { if (l>0) {s=s+1;}}
    Unit signalAll() {s = l;} }
```

This is an example of how a specific scheduling policy can influence object execution semantics.

$$
\begin{array}{rl@{\qquad}rl}
e & ::= \texttt{case2}\ v\ \{\overline{br}\} \mid \ldots & v & ::= o \mid f \mid \ldots \\
s & ::= \texttt{duration2}(d_1, d_2) \mid \ldots & & \\
cn & ::= \epsilon \mid obj \mid msg \mid fut \mid cn\ cn & tcn & ::= cn\ \mathrm{clock}(t) \\
fut & ::= f \mid \mathrm{fut}(f, v) & \sigma & ::= x \mapsto v \mid \sigma \circ \sigma \\
obj & ::= \mathrm{ob}(o, e, \sigma, pr, q) & pr & ::= \{\sigma | s\} \mid \mathrm{idle} \\
msg & ::= m(o, \overline{v}, f, d, c, t) & q & ::= \epsilon \mid pr \mid q \circ q
\end{array}
$$

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

4 Semantics

This section presents the operational semantics of Real-Time ABS as a transition system in an SOS style [34]. 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 [29]). A run is a possibly nonterminating sequence of rule applications. When auxiliary functions are used in the semantics, these are evaluated in between the application of transition rules in a run.

4.1 Runtime Configurations

The runtime syntax is given in Fig. 3. We extend expressions e with the runtime syntax **case2** v $\{\overline{br}\}$, statements with **duration2**(d, d), and values v with identifiers for objects and futures. For simplicity, we assume that all class and method declarations, as well as assignments for object creation and method calls are annotated as explained in Section 2 (i.e., the compiler inserts defaults and orders annotations where appropriate).

Configurations cn are sets of objects, invocation messages, and futures. A *timed configuration* tcn adds a global clock $clock(t)$ to a configuration (where t is a value of type Time). The global clock is used to record arrival and finishing times for processes. Timed configurations live inside curly brackets; thus, in $\{cn\}$, cn captures the *entire* runtime configuration of the system. 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 $ob(o, e, \sigma, pr, q)$ where o is the object's identifier, e is an expression of type Process representing a *scheduling policy*, σ a substitution representing the object's fields, pr is an (active) process, and q a *pool of processes*. A *substitution* σ is a mapping from variable names x to values v. For substitutions and process pools, concatenation is denoted by $\sigma_1 \circ \sigma_2$ and $q_1 \circ q_2$, respectively.

$$[\![b]\!]_\sigma = b$$
$$[\![x]\!]_\sigma = \sigma(x)$$
$$[\![v]\!]_\sigma = v$$
$$[\![Co(\overline{e})]\!]_\sigma = Co([\![\overline{e}]\!]_\sigma)$$
$$[\![\texttt{deadline}]\!]_\sigma = \sigma(\texttt{deadline})$$
$$[\![fn(\overline{e})]\!]_\sigma = \begin{cases} [\![e_{fn}]\!]_{\overline{x}\mapsto\overline{v}} & \text{if } \overline{e} = \overline{v} \\ [\![fn([\![\overline{e}]\!]_\sigma)]\!]_\sigma & \text{otherwise} \end{cases}$$

$$[\![\texttt{case } e\ \{\overline{br}\}]\!]_\sigma = [\![\texttt{case2 } [\![e]\!]_\sigma\ \{\overline{br}\}]\!]_\sigma$$
$$[\![\texttt{case2 } t\ \{p \Rightarrow e; \overline{br}\}]\!]_\sigma = \begin{cases} [\![e]\!]_{\sigma \circ match(p,t)} & \text{if } match(p,t) \neq \bot \\ [\![\texttt{case2 } t\ \{\overline{br}\}]\!]_\sigma & \text{otherwise} \end{cases}$$

Fig. 4 The evaluation of functional expressions.

In an *invocation message* $m(o, \overline{v}, f, d, c, t)$, 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, d and c are the provided deadline and cost of the call, and t is a time stamp recording the time of the call. A *future* is either an identifier f or a term $fut(f, v)$ 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.

Processes and Process Lifting. A *process* $\{\sigma|s\}$ consists of a substitution σ of local variable bindings and a list s of statements, or it is *idle*. By default, the local variables of a process include the variables `method` of type String, `arrival` of type Time, `cost` of type Duration, `deadline` of type Duration, `start` of type Time, `finish` of type Time, `critical` of type Bool, `value` of type Int, and `destiny` of type Name. Consequently, we can define a function *lift* which transforms the runtime representation of a process into the Real-Time ABS datatype of processes and a function *select* which returns the process corresponding to a given process identifier in a process queue, as follows:

$$\begin{aligned} lift(\{\sigma|s\}) = \texttt{Proc}(&\sigma(\texttt{destiny}), \sigma(\texttt{method}), \sigma(\texttt{arrival}), \\ &\sigma(\texttt{cost}), \sigma(\texttt{deadline}), \sigma(\texttt{start}), \sigma(\texttt{finish}), \\ &\sigma(\texttt{crit}), \sigma(\texttt{value})) \end{aligned}$$
$$select(pid, \varepsilon) = \text{idle}$$
$$select(pid, \{\sigma|s\} \circ q) = \begin{cases} \{\sigma|s\} & \text{if } \sigma(\texttt{destiny}) = pid \\ select(pid, q) & \text{otherwise} \end{cases}$$

The value of `destiny` is guaranteed to be unique, and is used to identify processes at the Real-Time ABS level.

4.2 A Reduction System for Expressions

The strict evaluation $[\![e]\!]_\sigma$ of functional expressions e, given in Fig. 4, is defined inductively over the data types of the functional language and is mostly standard, hence this subsection only contains brief remarks about some of the expressions. Let σ be a substitution which binds the name `deadline` to a duration value. For every (user-defined) function definition

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

the evaluation of a function call $[\![fn(\overline{e})]\!]_\sigma$ reduces to the evaluation of the corresponding expression $[\![e_{fn}]\!]_{\overline{x}\mapsto\overline{v}}$ 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 $p \Rightarrow e$ extends the current substitution σ with bindings that occurred during the pattern matching. Let the function $match(p, v)$ return a substitution σ such that $\sigma(p) = v$ (if there is no match, $match(p, v) = \bot$).

4.3 A Transition System for Timed Configurations

Evaluating Guards. Given a substitution σ and a configuration cn, we lift the evaluation function for functional expressions and denote by $[\![g]\!]^{cn}_\sigma$ a evaluation function which reduces guards g to data values (the state configuration is needed to evaluate future variables). Let $[\![g_1 \wedge g_2]\!]^{cn}_\sigma = [\![g_1]\!]^{cn}_\sigma \wedge [\![g_2]\!]^{cn}_\sigma$, $[\![\texttt{duration}(b, w)]\!]^{cn}_\sigma = [\![b]\!]_\sigma \leq 0$, $[\![x?]\!]^{cn}_\sigma = \text{true}$ if $[\![x]\!]_\sigma = f$ and $fut(f, v) \in cn$ for some value v (otherwise $f \in cn$ and we let $[\![x?]\!]^{cn}_\sigma = \text{false}$), and $[\![b]\!]^{cn}_\sigma = [\![b]\!]_\sigma$.

Auxiliary functions. If T is the return type of a method m in a class C, we let $bind(m, o, \overline{v}, f, d, b, t)$ return a process resulting from the activation of m in the class of o with actual parameters $\overline{v}$, callee o, associated future f, deadline d, and criticality b at time t. If binding succeeds, this process has a local variable `destiny` of type $\mathsf{fut}\langle T\rangle$ bound to f, the method's formal parameters are bound to $\overline{v}$, and the reserved variables `deadline` and `critical` are bound to d and b, respectively. Furthermore, `arrival` is bound to t and `cost` to $[\![e]\!]_{\overline{x}\mapsto\overline{v}}$ (or to the default 0 if no annotation is provided for the method). The function $atts(C, \overline{v}, o)$ returns the initial state of an instance of class C, in which the formal parameters are bound to $\overline{v}$ and the reserved variables `this` is bound to the object identity o. The function $init(C)$ returns an activation of the *init* method of C, if defined. Otherwise it returns the *idle* process. The predicate $fresh(n)$ asserts that a name n is globally unique (where n may be an identifier for an object or a future).

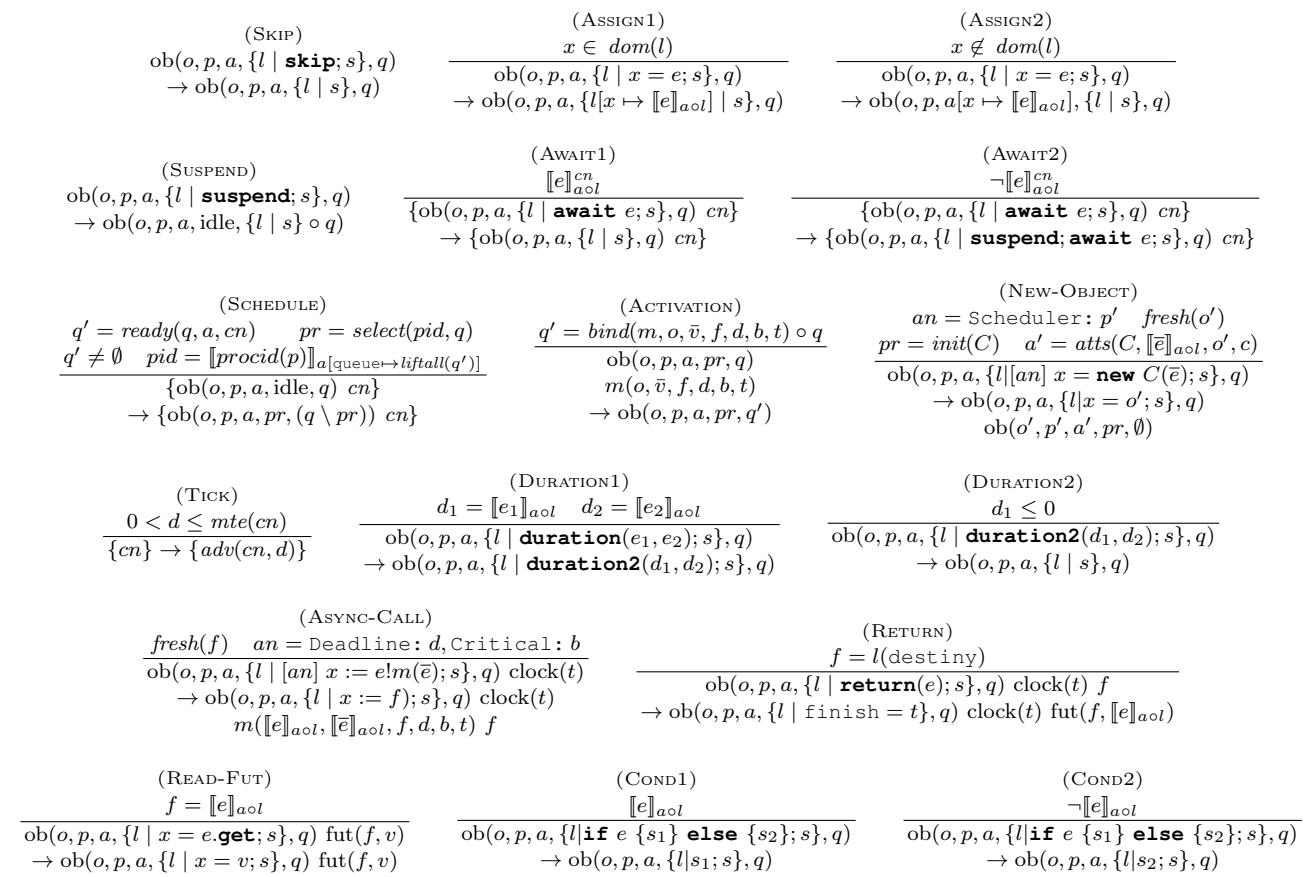

Fig. 5 The semantics of Real-Time ABS.

Transition rules transform state configurations into new configurations, and are given in Fig. 5. We denote by a the substitution which represents the attributes 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 creation (New-Object), method calls (Async-Call), and future dereferencing (Read-Fut). Rule Skip consumes a **skip** in the active process. Here and in the sequel, the variable s will match any (possibly empty) statement list. We denote by *idle* a process with an 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**-loops which unfolds into the conditional.)

Scheduling. Two operations manipulate a process pool q; $pr \circ q$ adds a process pr to q and $q \setminus pr$ removes pr from q. If q is a pool of processes, σ a substitution, t a time value, and cn a configuration, we denote by $ready(q, \sigma, cn)$ the subset of processes from q which are ready to execute (in the sense that the processes will not directly suspend or block the object [25]).

Scheduling is captured by the rule Schedule, which applies when the active process is *idle* and schedules a new process for execution if there are ready processes in the process pool q. We utilize a scheduling policy as explained in Section 3: in an object $ob(o, p, \sigma, idle, q)$, p is an expression representing the user-defined scheduling policy. This policy selects the process to be scheduled among the ready processes of the pool q.

In order to apply the scheduling policy p, which is defined for the datatype Process in Real-Time ABS, to the runtime representation q of the process pool, we lift the processes in q to values of type Process. Let the function *liftall* recursively transform a pool q of processes to a value of type List⟨Process⟩ by repeatedly applying *lift* to the processes in q. The process identifier of the scheduled process is used to *select* the runtime representation of this process from q.

Note that in order to evaluate guards on futures, the configuration cn is passed to the *ready* function. This explains the use of brackets in the rules, which ensures that cn is bound to the rest of the global system

configuration. The same approach is used to evaluate guards in the rules Await1 and Await2 below.

Rule Suspend 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 in order to suspend the process if the guard evaluates to false.

In rule Activation the function $\text{bind}(m, o, \overline{v}, f, d, c, b, t)$ binds a method call to object o in the class of o. This results in a new process $\{l|s\}$ which is placed in the queue, where $l(\texttt{destiny}) = f$, $l(\texttt{method}) = m$, $l(\texttt{arrival}) = t$, $l(\texttt{cost}) = c$, $l(\texttt{deadline}) = d$, $l(\texttt{start}) = 0$, $l(\texttt{finish}) = 0$, $l(\texttt{crit}) = b$, $l(\texttt{value}) = 0$, and where the formal parameters of m are bound to $\overline{v}$.

Durations. A statement **duration**(e_1, e_2) is reduced to the runtime statement **duration2**(d_1, d_2), in which the expressions e_1 and e_2 have been reduced to duration values. This statement blocks execution on the object until the best case execution time has passed; i.e., until at least the duration d_1 has passed. Remark that time cannot pass beyond duration d_2 before the statement has been executed (see below).

Method Calls. Rule Async-Call sends an invocation message to $[\![e]\!]_{a o l}$ with the unique identity f of a new future (since $\text{fresh}(f)$), the method name m, and parameter values $\overline{v}$. The identifier of the new future is placed in the configuration, and is bound to a return value in Return. The annotations are used to provide a deadline and a criticality which are passed to the callee with the invocation message. (The global clock provides a time stamp for the call.) Rule Return places the evaluated return expression in the future associated with the *destiny* variable of the process, and ends execution after recording the time of process completion in the `finish` variable. Rule Read-Fut dereferences the future $\text{fut}(f, v)$. Note that if the future lacks a return value, the reduction in this object is *blocked*.

Object creation. Rule New-Object creates a new object with a unique identifier o'. The object's fields are given default values by $atts(C, [\![\overline{e}]\!]_{a o l}, o', c)$, extended with the actual values $\overline{e}$ for the class parameters (evaluated in the context of the creating process) and o' for **this**. In order to instantiate the remaining attributes, the process *pr* is active (we assume that this process reduces to *idle* if $init(C)$ is unspecified in the class definition, and that it asynchronously calls `run` if the latter is specified). The object gets the scheduler in the annotation *an* (which is copied from the class or system default if a scheduler annotation is not provided).

Time advance. Rule Tick specifies how time can advance in the system. We adapt the approach of Real-Time Maude [31, 32] to Real-Time ABS and specify a

$$
\begin{array}{l}
mte(cn_1\ cn_2) = min(mte(cn_1), mte(cn_2)) \\
mte(\text{ob}(o,p,a,pr,q)) = \begin{cases} mte(pr) & \text{if } pr \neq \text{idle} \\ mte(q) & \text{if } pr = \text{idle} \end{cases} \\
mte(q_1, q_2) = min(mte(q_1), mte(q_2)) \\
mte(\{l|s\}) = \begin{cases} w & \text{if } s = \textbf{duration2}(b,w); s_2 \\ mte(g) & \text{if } s = \textbf{await}\ g; s_2 \\ 0 & \text{if } s \text{ is enabled} \\ \infty & \text{otherwise} \end{cases} \\
mte(g) = \begin{cases} max(mte(g_1), mte(g_2)) & \text{if } g = g_1 \wedge g_2 \\ w & \text{if } g = \textbf{duration}(b,w) \\ 0 & \text{if } g \text{ evaluates to true} \\ \infty & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{l}
adv(cn_1\ cn_2, d) = adv(c_1, d)\ adv(c_2, d) \\
adv(\text{ob}(o,p,a,pr,q), d) = \text{ob}(o,p,a,adv(pr,d),adv(q,d)) \\
adv((q_1,q_2), d) = adv(q_1,d), adv(q_2,d) \\
adv(\{l|s\}, d) = \{l[\texttt{deadline} \mapsto l(\texttt{deadline}) - d] | adv(s,d)\} \\
adv(s,d) = \begin{cases} \textbf{duration2}(b-d, w-d) & \text{if } s = \textbf{duration2}(b,w) \\ \textbf{await}\ adv(g,d) & \text{if } s = \textbf{await}\ g \\ adv(s_1,d) & \text{if } s = s_1; s_2 \\ s & \text{otherwise} \end{cases} \\
adv(g,d) = \begin{cases} adv(g_1,d) \wedge adv(g_2,d) & \text{if } g = g_1 \wedge g_2 \\ \textbf{duration}(b-d, w-d) & \text{if } g = \textbf{duration}(b,w) \\ g & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 6 Functions controlling the advancement of time. Trivial cases for terms *msg*, *fut* have been omitted.

global time which advances uniformly throughout the global configuration *cn*, combined with two auxiliary functions: $adv(cn, d)$ specifies how the advance of time with a duration d affects different parts of the configuration *cn*, and $mte(cn)$ defines the maximum amount that global time can advance. At any time, the system can advance by a duration $d \leq mte(cn)$. However, we are not interested in advancing time by a duration 0, which would leave the system in the same state.

The auxiliary functions *adv* and *mte* are defined in Fig. 6. Both have the whole configuration as input but consider mainly objects since these exhibit time-dependent behavior. The function *mte* calculates the maximum time increment such that no "interesting" occurrence (i.e., worst-case duration expires, duration guard passes) will be missed in any object. Observe that for statements which are not time-dependent, the maximum time elapse is 0 if the statement is enabled, since these statements are instantaneous, and infinite if not enabled, since time may pass when the object is blocked. Hence, *mte* returns the minimum time increment that lets an object become "unstuck", either by letting its active process continue or enabling one of its suspended processes. The function *adv* updates the active and suspended processes of all objects, decrementing all `deadline` values as well as the values in **duration** statements and **duration** guards at the head of the statement list in processes.

5 Case Studies and Simulation Results

A tool for Real-Time ABS is implemented, extending the existing ABS interpreter and tool chain. The parser, type-checker, and code generator have been implemented in Java using the JastAdd toolkit. Real-Time ABS models run on top of the Maude rewrite engine [14], with a model-independent interpreter almost directly implementing the semantics of Sec. 4. Code editing is supported in either the Emacs editor or the Eclipse integrated development environment, both of which support the ABS language after installing a plugin.

The examples below are based on the common scenario of a *server* receiving a series of *method calls* from an environment over time. These method calls result in method activations which have associated deadlines and costs. The examples further demonstrate the use of the Real-Time ABS tool.

Example 8 Consider the ABS model of a service, which is used by clients by calling the `request` method of a `Server` object (Fig. 7). For simplicity, we abstract from the specific functionality of our service (e.g., resizing photos or video of varying sizes) and let the `request` method of a server have a certain `duration` instead which is given as a parameter to the method. This duration reflects the cost of execution of the service in terms of its inputs best case `bc` and worst case `wc` execution time. A request to the server is successful (captured by the return value of the method) if it can be handled within the deadline which is given as an annotation at the call site in the `ClientImp` implementation.

The class `ClientImp` takes a number of jobs and dispatches them with a certain frequency to the server's `request` method with the given deadline. The class `ServerImp` contains a field `history` that is recording the scheduling sequence of the jobs and their lateness. The number of received and successful responses to `request` calls are recorded in the two variables `replies` and `successes` in the `ClientImp` class.

We simulate the model with a usage scenario in which objects for *photo clients* and *video clients* send a total of 70 jobs to the server. Approximately 70% of the jobs are cheap (i.e., processing photo) and 30% are expensive (i.e., processing video). In order to avoid infinite runs in the simulations, executions are set to stop at a given time limit. This allows us to observe the model's behavior up to a certain point in time. Figure 8 shows the number of failures to request calls (i.e., the missed deadlines) with respect to time and compares the performance of three different schedulers *fifo*, *edf*, and *sjf*, where *sjf* scheduler gives a better performance with respect to the other two schedulers.

```
interface Server {
 Bool request(String job, Rat bc, Rat wc); }

data Log = Log(String job, Time completiontime,
               Duration jobdeadline);

[Scheduler: sjf(queue)]
class ServerImp implements Server {
List<Log> history = Nil;
 [Cost: Duration(wc)]
 Bool request(String job, Rat bc, Rat wc) {
  duration(bc,wc);
  history = Cons(Log(job, now, deadline),history);
  return (durationValue(deadline) > 0); } }

interface Client { }

class ClientImp (String job, Int cycles,
 Int frequency, Duration bc, Duration wc,
 Duration limit, Server s) implements Client {
  Int replies = 0; Int successes = 0;
  Unit run() { await duration(frequency,frequency);
   [Deadline: limit]
   Fut<Bool> res = s!request(job,
           durationValue(bc),durationValue(wc));
   cycles = cycles - 1;
   if (cycles>0){ this!run(); }
   await res?; replies = replies + 1;
   Bool result = res.get;
   if (result){successes = successes+1;}
 } }
{ // Main block:
 Server s = new ServerImp();
 Client photo = new ClientImp("Photo",10,15,
     Duration(2),Duration(2),Duration(40),s);
 Client video = new ClientImp("Video",4,40,
     Duration(15),Duration(15),Duration(80),s);
 ...
} }
```

Fig. 7 A model of photo and video processing. The server class as shown uses *sjf* (shortest job first) scheduling.

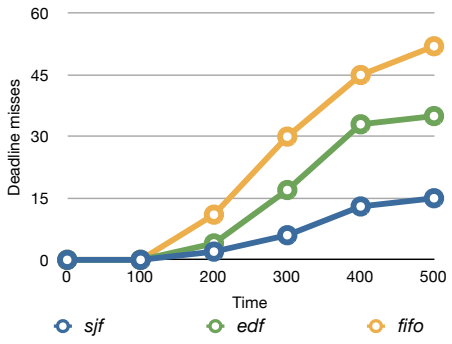

Fig. 8 Simulation results for Example 8: Comparison of the number of missed deadlines with respect to time for *fifo*, *edf*, and *sjf* schedulers.

Example 9 In the previous example, the *sjf* algorithm performed best overall, but this came at the cost of penalizing expensive jobs, who got a large portion of the total deadline misses. We now modify the model with an application-specific scheduler that can be pa-

```
// A scheduler which switches strategy
// based on the length of the queue
def Process lengthsensitive(Int limit,
                            List<Process> l) =
  if (length(l)<limit)
    then sjf(l)
    else fifo(l);

[Scheduler: lengthsensitive(limit,queue)]
class ServerImp (Int limit) implements Server {
  ... // (implementation unchanged)
}
```

Fig. 9 The model from Fig. 7, with an application-specific scheduler which adapts to server load.

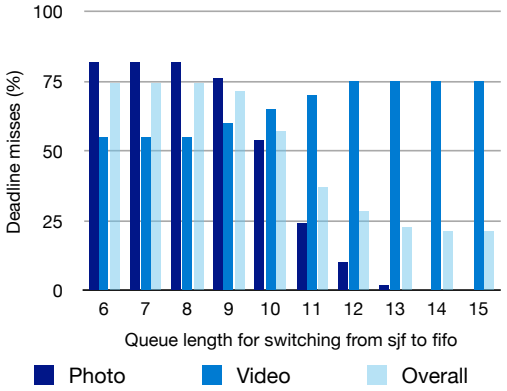

Fig. 10 Application-specific scheduling for the server example: a conditional scheduler switches between *fifo* and *sjf* behavior, depending on object state and server load.

rameterized to balance overall performance and fairness between large and small jobs.

Fig. 9 shows the modifications made to the model from Fig. 7. The new scheduler `lengthsensitive` switches between *sjf* and *fifo* behavior as the length of the process queue crosses a specified threshold. Note that the cutoff value `limit` is taken from the object state and is passed in as a second parameter to the scheduler. This shows how the state of an object can influence its scheduling decisions.

Switching between *fifo* and *sjf* represents a trade-off between favoring short and long jobs in our usage scenario. The new `limit` parameter to the class `ServerImp` lets the modeler influence the ratio of deadline misses, and hence QoS, for each job type. Fig. 10 presents simulation results for varying cutoff points. We record results and show deadline misses as percentage of overall submitted jobs for large and small jobs separately. It can be seen that for `limit` ≤ 6, the system performs identically to a system using *sjf* scheduling. For `limit` ≥ 15, the behavior is the same as *fifo*. The `limit` parameter can thus be used to influence overall quality of service, and to dynamically adjust an object's behavior for changing workloads. Note that we simulate with a constant value for `limit`, but it is straightforward to, e.g., model a monitor object running in parallel with the server that adjusts this parameter at runtime to adjust scheduling behavior based on performance measuring.

6 Conclusion

Whereas scheduling has traditionally been studied in the context of operating systems, modern software applications with soft real-time requirements need flexible application-specific schedulers to control application-level performance. This paper has presented Real-Time ABS, a real-time object-oriented modeling language in which user-defined schedulers may be associated with concurrent objects and deadlines with method calls. We have defined a formal semantics for Real-Time ABS and shown how user-defined schedulers may be expressed at the abstraction level of the modeling language and integrated in the formal semantics. A tool based on an abstract interpreter has been implemented for Real-Time ABS, which can be used for simulation and measurements for Real-Time ABS models. A series of examples demonstrate modeling with user-defined schedulers in Real-Time ABS and the use of this tool. Recent complementary work has shown how schedulability analysis for Real-Time ABS can be done by means of an encoding into timed automata [10] and how cost estimates for methods in Real-Time ABS can be inferred using the COSTABS tool [2]. The integration of this work in our tool is currently underway.

References

1. Agha, G.A.: ACTORS: A Model of Concurrent Computations in Distributed Systems. The MIT Press, Cambridge, Mass. (1986)
2. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Cost analysis of concurrent OO programs. In: Proc. 9th Asian Symposium on Programming Languages and Systems (APLAS 2011), *Lecture Notes in Computer Science*, vol. 7078, pp. 238–254. Springer (2011).
3. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: TIMES - a tool for modelling and implementation of embedded systems. In: Proc. 8th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002), *Lecture Notes in Computer Science*, vol. 2280, pp. 460–464. Springer (2002)
4. Andrews, G.R.: Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley (1999)
5. Angerer, C.M., Gross, T.R.: Static analysis of dynamic schedules and its application to optimization of parallel programs. In: Proc. 23rd Languages and Compilers

for Parallel Computing (LCPC 2010), *Lecture Notes in Computer Science*, vol. 6548, pp. 16–30. Springer (2010)
6. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)
7. Bjørk, J., Johnsen, E.B., Owe, O., Schlatte, R.: Lightweight time modeling in Timed Creol. *Proc. 1st Intl. Workshop on Rewriting Techniques for Real-Time Systems (RTRTS 2010).* Electronic Proceedings in Theoretical Computer Science **36**, 67–81 (2010).
8. Blair, G.S., Coulson, G., Robin, P., Papathomas, M.: An architecture for next generation middleware. In: Proc. IFIP Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), pp. 191–206. Springer (1998)
9. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: R. de Nicola (ed.) Proc. 16th European Symposium on Programming (ESOP'07), *Lecture Notes in Computer Science*, vol. 4421, pp. 316–330. Springer (2007)
10. de Boer, F.S., Jaghoori, M.M., Johnsen, E.B.: Dating concurrent objects: Real-time modeling and schedulability analysis. In: Proc. 21st Intl. Conf. on Concurrency Theory (CONCUR), *Lecture Notes in Computer Science*, vol. 6269, pp. 1–18. Springer (2010)
11. Buttazzo, G.: Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 2 edn. Kluwer Academic Publishers (2004)
12. Chakravarti, A.J., Baumgartner, G., Lauria, M.: Application-specific scheduling for the organic grid. In: Proc. 5th IEEE/ACM Intl. Workshop on Grid Computing (GRID'04), pp. 146–155. IEEE Press (2004)
13. Cheng, A.M.K.: Real-Time Systems: Scheduling, Analysis, and Verification. John Wiley & Sons, Inc. (2002)
14. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, *Lecture Notes in Computer Science*, vol. 4350. Springer (2007)
15. Coffman, E.G.: Computer and Job-Shop Scheduling Theory. John Wiley & Sons, Inc. (1976)
16. David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski:, A.: ECDAR: An environment for compositional design and analysis of real time systems. In: Proc. 8th Automated Technology for Verification and Analysis (ATVA 2010), *Lecture Notes in Computer Science*, vol. 6252, pp. 365–370. Springer (2010)
17. Dibble, P.C.: Real-Time Java Platform Programming, 2 edn. BookSurge Publishing (2008)
18. Fersman, E., Krcál, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Information and Computation **205**(8), 1149–1172 (2007)
19. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science **410**(2–3), 202–220 (2009)
20. Harchol-Balter, M., Schroeder, B., Bansal, N., Agrawal, M.: Size-based scheduling to improve web performance. ACM Transactions on Computer Systems **21**(2), 207–233 (2003)
21. Hoare, C.A.R.: Monitors: an operating system structuring concept. Commun. ACM **17**, 549–557 (1974)
22. Hsiung, P.A., Huang, C.H., Chen, Y.H.: Hardware task scheduling and placement in operating systems for dynamically reconfigurable soc. Journal of Embedded Computing **3**(1), 53–62 (2009)
23. Hsu, C.H., Chen, S.C.: A two-level scheduling strategy for optimising communications of data parallel programs in clusters. Intl. Journal of Ad Hoc and Ubiquitous Computing **6**(4), 263–269 (2010)
24. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Proc. 9th Intl. Symposium on Formal Methods for Components and Objects (FMCO 2010), *Lecture Notes in Computer Science*, vol. 6957, pp. 142–164. Springer (2011).
25. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling **6**(1), 35–58 (2007)
26. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer **1**(1–2), 134–152 (1997)
27. Lee, E.A.: Computing needs time. Communications of the ACM **52**(5), 70–79 (2009)
28. Logan, M., Merritt, E., Carlsson, R.: Erlang and OTP in Action. Manning Publications (2010)
29. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**, 73–155 (1992)
30. Nobakht, B., de Boer, F.S., Jaghoori, M.M., Schlatte, R.: Programming and deployment of active objects with application-level scheduling. In: Proc. Symposium on Applied Computing (SAC). ACM (2012).
31. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. Theoretical Computer Science **285**(2), 359–405 (2002)
32. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation **20**(1–2), 161–196 (2007)
33. Pierce, B.C.: Types and Programming Languages. The MIT Press (2002)
34. Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Programming **60-61**, 17–139 (2004)
35. Santoso, J., van Albada, G.D., Nazief, B.A.A., Sloot, P.M.A.: Simulating job scheduling for clusters of workstations. In: M. Bubak, H. Afsarmanesh, R. Williams, L.O. Hertzberger (eds.) 8th Intl. Conf. on High-Performance Computing and Networking (HPCN Europe 2000), *Lecture Notes in Computer Science*, vol. 1823, pp. 395–406. Springer (2000)
36. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: European Conf. on Object-Oriented Programming (ECOOP 2010), *Lecture Notes in Computer Science*, vol. 6183, pp. 275–299. Springer (2010)
37. Schoeberl, M.: Real-time scheduling on a Java processor. In: Proc. 10th Intl. Conf. on Real-Time and Embedded Computing Systems and Applications (RTCSA) (2004)
38. Sorensen, A., Gardner, H.: Programming with time: cyber-physical programming with Impromptu. In: W.R. Cook, S. Clarke, M.C. Rinard (eds.) Proc. Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA), pp. 822–834. ACM (2010)
39. Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for Java. In: J. Vitek (ed.) Proc. 22nd European Conf. on Object-Oriented Programming (ECOOP 2008), *Lecture Notes in Computer Science*, vol. 5142, pp. 104–128. Springer (2008)
40. Tchernykh, A., Ramírez-Alcaraz, J.M., Avetisyan, A., Kuzjurin, N., Grushin, D., Zhuk, S.: Two level job-scheduling strategies for a computational grid. In: R. Wyrzykowski, J. Dongarra, N. Meyer, J. Wasniewski (eds.) 6th Intl. Conf. on Parallel Processing and Applied Mathematics (PPAM), *Lecture Notes in Computer Science*, vol. 3911, pp. 774–781. Springer (2005)