

# A Formal Model of Object Mobility in Resource-Restricted Deployment Scenarios <sup>★</sup>

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

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

**Abstract.** Software today is often developed for deployment on different architectures, ranging from sequential machines via multicore and distributed architectures to the cloud. In order to apply formal methods, models of such systems must be able to capture different deployment scenarios. For this purpose, it is desirable to express aspects of low-level deployment at the abstraction level of the modeling language. This paper considers formal executable models of concurrent objects executing with user-defined cost models. Their execution is restricted by deployment components which reflect the execution capacity of groups of objects between observable points in time. We model strategies for object relocation between components. A running example demonstrates how activity on deployment components causes congestion and how object relocation can alleviate this congestion. We analyze the average behavior of models which vary in the execution capacity of deployment components and in object relocation strategies by means of Monte Carlo simulations.

## 1 Introduction

Software is increasingly often developed as a range of systems. Different versions of a software may provide different functionality and advanced features, depending on target users. In addition to such functional variability, software systems need to adapt to different *deployment scenarios*. For example, *operating systems* adapt to specific hardware and even to different numbers of available cores; *virtualized applications* are deployed on a varying number of (virtual) servers; and *services on the cloud* may need to adapt dynamically to the underlying cloud infrastructure. This kind of adaptability raises new challenges for the modeling and analysis of component-based applications [33]. To apply formal methods to such applications, it is interesting to lift aspects of low-level deployment concerns to the abstraction level of the modeling language. In this paper we propose abstract performance analysis for formal object-oriented models, in which objects may migrate between deployment components that are parametric in the amount of concurrent processing resources they provide to their objects.

The work presented in this paper is based on ABS [20], a modeling language for distributed concurrent objects which communicate by asynchronous

---

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

method calls. ABS is an executable language, but still allows abstractions (i.e., functions and abstract data types can be used to specify internal, sequential computations). ABS is a successor of Creol [21], simplifying that language by removing some features such as class inheritance and internal non-deterministic choice, but retaining a concurrent object model similar to Actors [1] and Erlang processes [5]: objects are inherently concurrent, with at most one process active per object. Concurrent objects and actors have attracted attention as an alternative to multi-thread concurrency in object-orientation (e.g., [9]), and been integrated with, e.g., Java [30, 32] and Scala [15]. ABS uses Creol’s *cooperative scheduling* of processes inside concurrent objects, which eliminates some common programming errors (specifically, race conditions are much harder to introduce inadvertently) and enables *compositional* verification of models [2, 12].

In order to capture deployment scenarios for ABS models, previous work by the authors proposes an extension of the ABS language with *deployment components* which are parametric in the amount of concurrent activity they allow within a time interval [23]. This allows us to analyze how the amount of concurrent execution resources allocated to a deployment component influences the performance of objects deployed on the component. For this purpose, we work with a notion of timed concurrent objects [8], extended to capture parametric concurrent activities between observable points in time. To validate and compare the concurrent behavior of models under restricted concurrency assumptions, the timed operational semantics of our ABS extension, defined in an SOS style [29], is expressed in rewriting logic [26], which enables the use of Maude [11] as a simulation and analysis tool for ABS models.

The contribution of this paper goes in three directions, compared to our previous work. First, we propose a formalization of *object mobility* in resource-restricted deployment scenarios. This allows models to capture dynamic object deployment, which was not expressible in our previous work. We show how object mobility naturally integrates in ABS in an elegant and simple way, and how it allows dynamic deployment scenarios such as load balancing strategies to be expressed and executed in parallel with the functional parts of the model. This technical contribution complements the work presented in [22], which formalizes load balancing by resource reallocation. Second, user-defined cost models for resource usage are introduced. Where our previous work used fixed cost models for processing capacity, user-defined cost models are given by functional expressions at the abstraction level of the modeling language and introduced in the models in the form of annotations, providing a separation of concerns between the functional aspect of the model and its resource consumption. Third, we extend our simulation tool to support Monte Carlo simulations; i.e., non-determinism in the semantics is resolved in the simulation tool by means of a sequence of pseudo-random numbers which is controlled by a seed when starting a simulation. In principle, this allows the possible execution paths of a model to be systematically inspected up to a given time, and allows us to analyze *average behavior* for models with user-defined cost. We demonstrate the use of Monte Carlo simulations to analyze the resource usage of distributed system models



`thumbnailImage` method of their `Session` instance, which has as an actual parameter the `size` of the image. After completing the service, the session object is returned to the agent’s pool. Our model defines a user-datatype `Pixels` as the unit to measure the `size` of an image.

Let the `thumbnailImage` method of a session have a certain computation cost and deadline calculated in terms of the parameter `size`; a service request is successful if it can be handled within the deadline. Let us assume that our service reduces the size of any given image to *150 pixels*, and that we are expecting to process an average of *2400 pixels* per time interval; then we calculated the cost as  $size/150$  and the deadline as  $size/2400$ . For simplicity, we abstract from the specific functionality of our service. The *cost annotation* in the while-loop expresses the granularity of resource consumption. In our model, the actual cost `cost` is decomposed into a number of cumulative steps. In contrast, an annotation with the full cost would express that the computation must happen within one time interval; e.g., `[Cost : cost] skip`.

In the `Agent` class, the attribute `sessions` stores a set of `Session` objects. (ABS has a datatype for sets, with operations `emptySet` to check for the empty set, denoted `EmptySet`, `select` to select an element of a non-empty set, and the usual `remove` and `Insert` operators). When a client requests a `Session`, the `Agent` takes a session from the available sessions if possible, otherwise it creates a new session. The method `free` inserts the session in the available sessions of the `Agent`, and is called by the session itself upon completion of a *thumbnail* service request. This model captures the architecture and control flow of a service oriented application, while abstracting from many functionality and implementation details (such as thread pools, data models, sessions spanning multiple requests, etc.) which can be added to the model if needed.

The main block of the model specifies the initial state for model execution as a *deployment scenario* in which an `Agent` object is deployed on a deployment component `server` (of the predefined type `DC`), which will also contain the `Session` objects. The parameter to the server specifies its execution capacity in terms of abstract *concurrent resources*, which reflect the amount of potential abstract execution cycles available to the objects deployed on the server between observable points in time. The agent creates concurrently executing `Session` objects on the same server as needed. It is easy to see that heavy client traffic may lead to congestion on the server, which may in turn cause a lot of unsuccessful requests to the service.

### 3 Models of Deployed Concurrent Objects in ABS

ABS is an abstract behavioral specification language for distributed concurrent objects [20]. Concurrent objects are, like Actors [1] and Erlang [5] processes, dynamically created and inherently concurrent. ABS is an object-oriented language, so objects are dynamically created instances of classes, with attributes initialized to default type-correct values. An optional *init* method may be used to redefine attributes. Objects are typed by interface and communicate by asyn-

chronous method calls, spawning concurrent activities in the called object. Active behavior, specified by an optional *run* method, is interleaved with passive behavior, triggered by such asynchronous 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*. The others are *suspended* on a queue. Process scheduling is by default non-deterministic, but controlled by *processor release points* in a cooperative way. ABS is strongly typed: for well-typed programs, invoked methods are supported by the called object (when not *null*), and formal and actual parameters match. We assume that programs are well-typed.

*Deployment components* were proposed in [23] to restrict the inherent concurrency of objects in ABS by mapping the logical concurrency to a model of physical computing resources. Deployment components abstract from the number and speed of the available physical processors by a notion of concurrent processing resource, reflecting the processing capacity of a component. Concurrent processing resources can be consumed in parallel or in sequential order, which reflects the number of processors and their speeds relative to the intervals between observable points in time. A simple time model suffices to define the points in time when the executing system is observable. How an object consumes resources depends on a *cost model*, which reflects the processing costs of different activities in the objects. In [23], we worked with a simplistic cost model which assigned a fixed cost to **skip** and to statements with write-access to memory. In [22], we introduced reflection into this component model, such that an object could inspect the *load* of its deployment component, and reallocate resources between deployment components. However, objects were statically deployed on a deployment component when they were created, and the same simplistic cost model was used.

In ABS, objects are deployed on deployment components with given amounts of resources. Objects deployed on a component may consume resources within a time interval until the component runs out of resources or the objects are otherwise blocked. This way, the logical concurrency model of a group of concurrent objects is controlled by their associated deployment component. A deployment component is parametric in the computational resources it offers to a group of dynamically created objects, which makes it easy to configure deployment scenarios varying in their concurrent resources.

In this paper, we generalize our previous approach by allowing a user-defined cost model in which the processing costs of a statement are given in terms of a cost expression  $e$  which depends on the current state of the object and the local variables of the active process. The expression is introduced into the ABS syntax as an optional annotation  $[\text{Cost} : e]s$  to statements  $s$ ; thus, we obtain a separation of concerns between the cost and functional behavior of models. Statements without annotations are given a default cost and models without annotations are valid models in the resource-restricted extension to ABS. Furthermore, the statement **goto**( $e$ ) is introduced to the language. This statement expresses object mobility such that an object may relocate to a target deployment component  $e$ . This way, deployment scenarios may be modeled in which

<i>Syntactic categories.</i>	<i>Definitions.</i>
$C, I, m$ in Names	$IF ::= \mathbf{interface} I \{ [Sg] \}$
$g$ in Guard	$CL ::= \mathbf{class} C [(I \bar{x})] [\mathbf{implements} \bar{I}] \{ [\bar{I} \bar{x};] \bar{M} \}$
$s$ in Stmt	$Sg ::= I m ([\bar{I} \bar{x}])$
$x$ in Var	$M ::= Sg \{ [\bar{I} \bar{x};] s \}$
$e$ in Expr	$g ::= b \mid x? \mid g \wedge g$
$b$ in BoolExpr	$s ::= s; s \mid [Cost : e] s \mid \mathbf{skip} \mid x = rhs$
$r$ in Resource	$\mid \mathbf{suspend} \mid \mathbf{await} g \mid \mathbf{while} b \{ s \} \mid \mathbf{goto}(e)$
	$\mid \mathbf{if} b \mathbf{then} \{ s \} [\mathbf{else} \{ s \}] \mid \mathbf{return} e$
	$e ::= x \mid b \mid r \mid \mathbf{this} \mid \mathbf{thiscomp} \mid \mathbf{now} \mid \mathbf{total}$
	$\mid \mathbf{load}(e) \mid \mathbf{random}(e)$
	$rhs ::= e \mid cm \mid \mathbf{new} C(\bar{e}) [\mathbf{in} e] \mid \mathbf{component} (e)$
	$cm ::= [e].m(\bar{e}) \mid [e].m(\bar{e}) \mid x.get$

**Fig. 2.** ABS syntax. Terms such as  $\bar{e}$  and  $\bar{x}$  denote lists over the corresponding syntactic categories, square brackets  $[]$  denote optional elements.

objects dynamically change deployment components. For readability, we present the syntax of the full language with the proposed extensions below.

Figure 2 gives the syntax of timed ABS with deployment components. A *program* consists of interface and class definitions and a *main block* to configure the initial state.  $IF$  defines an interface with name  $I$  and method signatures  $Sg$ . A class implements a set  $\bar{I}$  of interfaces, which specify types for its instances.  $CL$  defines a class with name  $C$ , interfaces  $\bar{I}$ , class parameters and state variables  $x$  (of type  $I$ ), and methods  $M$ . (The *attributes* of the class are both its parameters and declared fields.) A method signature  $Sg$  declares the return type  $I$  of a method with name  $m$  and formal parameters  $\bar{x}$  of types  $\bar{I}$ .  $M$  defines a method with signature  $Sg$ , a list of local variable declarations  $\bar{x}$  of types  $\bar{I}$ , and a statement  $s$ .

*Statements.* Assignment  $x = rhs$ , sequential composition  $s_1; s_2$ , **skip**, **if**, **while**, and **return**  $e$  are standard. The statement **goto**( $e$ ) moves the object to deployment component  $e$ . The statement **suspend** unconditionally releases the processor by suspending the active process. The guard  $g$  controls processor release in statements **await**  $g$ , and consists of Boolean expressions  $b$  over attributes and return tests  $x?$  (see below). If  $g$  evaluates to false, the current process is *suspended*. In this case, any enabled process from the pool of suspended processes may be activated. The scheduling of processes is *cooperative* in the sense that processes explicitly yield control and execution in one process may enable the further execution in another. The annotated statement  $[Cost : e] s$  expresses that the cost of executing  $s$  will be  $e$  resources, where  $e$  is evaluated in the current state of the object.

*Expressions*  $rhs$  include pure expressions  $e$ , communications  $cm$ , and the creation of deployment components and objects. The expression **component** ( $e$ ) creates a component with  $e$  concurrent resources. Resources are modeled by a type `Resource` which extends the natural numbers with an “unlimited resource”  $\omega$ . The set of concurrent objects deployed on a component, representing the

$$\begin{aligned}
cn &::= \epsilon \mid obj \mid msg \mid fut \mid cn \ cn \\
obj &::= o(\sigma, p, q) \\
p &::= \{\sigma \mid s\} \mid idle
\end{aligned}$$

**Fig. 3.** The syntax for timed runtime configurations.

logically concurrent activities, may grow dynamically. Object creation **new**  $C(\bar{e})$  has an optional clause **in**  $e$  to specify the targeted deployment component: here the  $C$  object is to be deployed on component  $e$ . (If the target component is omitted, the new object will be deployed on the same component as its parent. The behavior of ABS models without deployment restrictions on their functional behavior is captured by a main deployment component with  $\omega$  resources.)

*Pure expressions*  $e$  are variables  $x$ , Boolean expressions  $b$ , resources  $r$ , **this** (the object’s identifier) and **thiscomp** (the object’s current deployment component), and **now**, which returns the current time. Timed ABS uses an implicit time model [8], comparable to a system clock which updates every  $n$  milliseconds (representing a time interval). Time values are totally ordered by the less-than operator; comparing two time values results in a Boolean value suitable for guards in **await** statements. From an object’s local perspective, the passage of time is indirectly observable via **await** statements. Time advances when no other activity may occur. This model of time is used to handle the amount of concurrent activity allowed within a time interval in order to model resource constraints for different deployment scenarios. The total number of resources allocated to objects on the current deployment component are given by **total**, and the average load on the component for the last  $e$  time intervals by **load**( $e$ ). The expression **random**( $e$ ) returns some integer value between 0 and the value of  $e$ . (The full language includes a functional expression language with standard operators for data types such as strings, integers, lists, sets, maps, and tuples. These are omitted here, and explained when used in the examples.)

*Communications*  $cm$  are based on asynchronous method calls. After making an asynchronous call  $x = e!m(\bar{e})$ , the caller may proceed without waiting for the method reply. Here  $x$  is a *future variable*, which refers to a return value which may still need to be computed. Two operations on future variables control synchronization in ABS [20]. First, the guard **await**  $x?$  *suspends* the active process until a return to the call associated with  $x$  has arrived. This suspends execution of the process, but allows other processes to run. Second, the return value is retrieved by the expression  $x$ .**get**, which *blocks* all execution in the object until the return value is available. Two commonly used communication patterns are now explained; the statement sequence  $x = e!m(\bar{e}); y = x$ .**get** encodes a *blocking call*, conveniently abbreviated  $y = e.m(\bar{e})$  (often referred to as a synchronous call), whereas the statement sequence  $x = e!m(\bar{e});$  **await**  $x?$ ;  $y = x$ .**get** encodes a non-blocking, *preemptable call*.

$$\begin{array}{c}
\text{(RESTRICTEDEXEC)} \\
\frac{\text{thiscomp}(o) = dc \quad \llbracket e \rrbracket_{\sigma o}^t = c \quad c \leq n}{o(\sigma, \{l|s\}, q) \text{ cl}(t) \text{ cn} \rightarrow o(\sigma', p', q') \text{ cl}(t) \text{ cn}'} \\
\frac{o(\sigma, \{l|[c\text{ost}:e]s\}, q) \text{ dc}(n, u, \bar{h}) \text{ cl}(t) \text{ cn}}{\rightarrow o(\sigma', p', q') \text{ dc}(n - c, u + c, \bar{h}) \text{ cl}(t) \text{ cn}'} \\
\\
\text{(RUNTOCOMPLETION)} \qquad \qquad \qquad \text{(RESET)} \\
\frac{cn \text{ cl}(t) \xrightarrow{!} cn' \text{ cl}(t) \quad cn' \xrightarrow{!}_{\tau} cn''}{\{cn \text{ cl}(t)\} \rightarrow_{\tau} \{cn'' \text{ cl}(t+1)\}} \qquad \frac{u > 0}{dc(n, u, \bar{h}) \rightarrow_{\tau} dc(n + u, 0, (\bar{h}; u))}
\end{array}$$

**Fig. 4.** A reduction semantics for timed resource-restricted execution.

### 3.1 Operational Semantics

The operational semantics of ABS is given as an SOS [29] style reduction system. We briefly outline the semantics here in order to explain the extension with user-defined cost annotations (the full details may be found in [20]). The runtime syntax is given in Fig. 3. A configuration  $cn$  consists of objects  $obj$ , messages  $msg$ , and futures  $fut$ . An object  $o(\sigma, p, q)$  has an identity  $o$ , a state  $\sigma$ , and active process  $p$ , and a queue of pending processes  $q$ . The active process consists of a list of statements  $s$  to be executed in the context of local variable bindings  $\sigma$ , unless the active process is *idle* (in which case a pending process from  $q$  is scheduled for execution). Messages represent method calls and futures represent method returns.

Given a reduction relation  $\rightarrow$ , a *run* is in general a possibly non-terminating sequence of terms  $t_0, t_1, \dots$  such that  $t_i \rightarrow t_{i+1}$ . Let  $t \xrightarrow{!} t'$  denote that  $t'$  is the final term of a terminating run from the initial term  $t$ ; i.e., there is no term  $t''$  such that  $t' \rightarrow t''$ . We shall denote by  $\rightarrow$  the reduction relation of ABS, which is defined inductively over the legal configurations  $cn$ . For an object  $o(\sigma, \{l|s\}, q)$ , there are in particular rules which reduce the head of the statement list  $s$ , defined by cases for the statements of ABS. In addition, there is a rule for binding a message  $msg$  to a method activation  $p$ , which is put into the object queue  $q$ , and for scheduling a suspended process from  $q$  when the active process is idle. (Observe that many processes may be schedulable at the same time, which leads to non-determinism in the semantics.) ABS objects are asynchronous in the sense that no reduction rules have two objects on the left hand side.

The runtime syntax of *timed* runtime configurations with deployment components is given by the following extension of the syntax of Figure 3:

$$tcn ::= \{ cl(t) \text{ cn} \} \qquad cn ::= dc(n, u, \bar{h}) \mid \dots$$

A *timed configuration* consists of a configuration  $cn$  and a clock  $cl(t)$  (where  $t$  is the current global time). Extended configurations  $cn$  may contain deployment components  $dc(n, u, \bar{h})$ , where  $dc$  is the identity of the component,  $n$  is the number of available processing resources,  $u$  the used resources, and  $\bar{h}$  the (possibly empty) sequence of resource usage over time. Observe that the standard ABS

reduction relation  $\rightarrow$  is not defined for active processes in which the head of the statement list is annotated. Figure 4 defines the extension to  $\rightarrow$  for such annotated statement lists, a reduction relation  $\rightarrow_\tau$  which expresses the effect of advancing time, and the timed resource-restricted reduction relation  $\rightarrow_r$ .

The rule `RESTRICTEDEXEC` extends the relation  $\rightarrow$  to capture the reduction of an object  $o$  in which the head of the statement list in the active process has an annotation of cost  $e$ . This can be done according to the standard rules for  $\rightarrow$  if the current deployment component of  $o$  has enough resources to do a reduction step. In this rule, we use `thiscomp(o)` to denote the current deployment component of  $o$ ,  $\llbracket e \rrbracket_\sigma^t$  to denote the evaluation of an expression  $e$  in the substitution  $\sigma$  at time  $t$ . Observe that the resources required to do the reduction are subtracted from the available resources of the deployment component and added to its used resources. Rule `RESET` expresses the effect of time advance on a deployment component; the available resources  $n$  are reset to amount of resources allocated to the component, and the history of resource consumption is extended with the the used resources  $u$  of the previous time interval.

The rule `RUNTOCOMPLETION` captures the timed resource-restricted reduction relation  $\rightarrow_r$  between timed configurations. Time advances from a timed configuration  $\{cn\ cl(t)\}$  by the reduction relation  $\rightarrow_r$  if  $cn$  can be reduced to a normal form  $cn'$  by application of  $\rightarrow$ , after which all deployment components in  $cn'$  have been reset by rule  $\rightarrow_r$ . A *run* of a timed, resource-restricted ABS model is a (possibly non-terminating) sequence of configurations  $tcn_0, tcn_1, tcn_2, \dots$  such that  $tcn_i \rightarrow_r tcn_{i+1}$ , which represent the configurations at the *observable points in time* during the execution of the timed resource-restricted ABS model. Observe that a non-terminating run by the ABS reduction relation  $\rightarrow$  corresponds to an infinitely fast execution in timed, resource-restricted ABS; there is no observable successor state.

### 3.2 ABS Analysis Tool

The SOS semantics of timed, resource-restricted ABS has previously been translated to rewriting logic [26] and implemented in Maude [11], to provide an interpreter for ABS models for the fixed cost model of our previous work. The details of this rewriting logic semantics for ABS are reported in [22]. As a technical contribution of this paper, we have extended this interpreter to accommodate user-defined cost models, as defined above, and the `goto`-statement and `random`-expression proposed in this paper. Whereas the implementation of the `goto`-statement translates into an assignment of the `thiscomp` field of an object, the `random`-expression is implemented such that it depends on a sequence of pseudo-random numbers, controlled by a *seed* provided as an argument to the execution of the model. The sequence of pseudo-random numbers is also used to make *scheduling decisions* in the simulator of the ABS semantics; i.e., if an object has a list of  $n$  schedulable processes in its queue, the interpreter will select for execution the `random(n)`'th process from this list. This interpreter for the semantics of timed resource-restricted ABS is used as a basis for Monte Carlo simulations. Whereas our previous work on deployment components could only

```

...
interface Client { }

class AsyncClientImp (Int cycles, Int frequency, Pixels size, Agent a)
implements Client {
  Unit run() {
    Time t = now;
    Fut<Session> f = a!getsession(); await f?; Session s = f.get;
    s!thumbnailImage(size*(random(3)+1));
    cycles = cycles-1;
    if (cycles > 0) {
      Int jitter = 3-(random(4)+1);
      await duration(frequency+jitter, frequency+jitter);
      this!run(); }
    }
}
{// Main block
...
new AsyncClientImp(15,4,5000,a);
new AsyncClientImp(15,3,4000,a);
new AsyncClientImp(15,3,3000,a);
new AsyncClientImp(15,4,2000,a); }

```

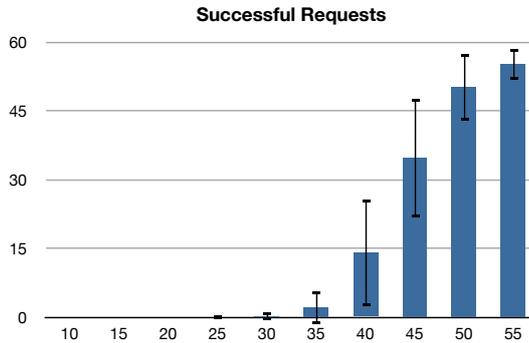
**Fig. 5.** A configurable asynchronous client which provides the workload scenario.

simulate one arbitrary run of the model, this extension allows us to simulate  $n$  runs with different sequences of pseudo-random numbers, which in principle allows us to exhaust the full state space of executions. The individual runs of the Monte Carlo simulations use the ABS interpreter, a query language allows us to extract information from the runtime states of these simulations, and to combine this information from the different runs of a deployment scenario. The use of this analysis tool is shown in the following sections.

## 4 Comparing Resource-Restricted Behaviors

In order to investigate the effects of specific deployment scenarios on the timing behavior of timed software models, we use the analysis tool to simulate and test ABS models. The *test purpose* for these scenarios using *Monte Carlo simulations* is to reach a conclusion on whether redeployment on a different configuration leads to an observable difference in timing behavior. We compare the behavior of ABS models with the same functional behavior and workload when the models are deployed on components with different amounts of resources.

We extend the example given in Section 2 with a workload scenario. Figure 5 shows the implementation of a *configurable asynchronous client*. The run method of an AsyncClientImp object has as actual parameters the *number of images* to process represented by the parameter `cycles`; the frequency of requests to the *thumbnail* service, represented by the parameter `frequency` and a *random jitter* value in the interval  $[-2, +2]$ ; and a *varying image size* given by the parameter `size` which varies the sizes of the images in the interval  $[size, 4 \cdot size]$ .



**Fig. 6.** Single-server simulation results: number of successful, i.e., non-timeout responses for 60 requests, with server capacity varying between 10 and 55. The numbers show mean and standard deviation of 100 runs for each server capacity.

Objects of the `AsyncClientImp` class are used to model the expected usage scenario and run with unlimited resources.

Figure 6 shows simulations results using four asynchronous clients running concurrently and making a total of 60 thumbnail request with a frequency ranging between  $[1,6]$  and image's size ranging over  $[2\ 000, 20\ 000]$  pixels. As we can see from Fig. 6, the server is basically unresponsive up to 35 resources, at 45 resources it can successfully handle approximately 50% of the requests, and above 50 resources it can successfully handle more than 75% of the requests.

## 5 Load Balancing Strategies

ABS models may be augmented with *load balancing strategies* with the aim of decreasing congestion and thus improving the overall quality of service compared to models with static deployment scenarios. Load balancing strategies may be expressed in ABS using the resource-related language constructs **total**, **load**, and **goto**.

We illustrate how ABS models may be augmented with load balancing strategies using the running example of the thumbnail image service, and compare the results of load balancing to the results for basic deployment scenarios presented in Section 4. In this section we model and explore two different load balancing strategies; (1) a *load-balancing agent* which moves sessions to a backup server when the load on the main server is above a given threshold, and (2) *self-monitoring sessions* which move themselves to the backup server if the processing of their requests takes more than a given time limit. Both of these dynamic deployment scenarios are analyzed using the same workload scenario as in Section 4. Other, more elaborate load-balancing strategies may be modeled in the same style.

```

interface Agent {Session getsession(); Unit free(Session session);}
interface Session {Bool thumbnailImage(Pixels size);
                  Unit moveto(DC server);}

class SessionImp(Agent agent) implements Session {
  Time start = now;
  Bool thumbnailImage(Pixels size){ ... } // As before
  Unit moveto(DC server){if (thiscomp != server){
    [Cost: 1] goto(server);[Cost: 1] skip;}}

class AgentImp(DC origserver, DC backupserver) implements Agent {
  Set<Session> sessions = EmptySet;
  Unit free(Session session){ ... // As before
    session!moveTo(origserver);}

  Session getsession() { Session session;
    if (emptySet(sessions)){session = new SessionImp(this);}
    else {session = select(sessions);
      sessions = remove(session,sessions);}
    if ((total - load(4)) < total/3){ // Move session to backup server
      session!moveto(backupserver);}
    return session;}}

```

**Fig. 7.** An agent which performs load-balancing for the thumbnail image service.

Figure 7 shows the ABS model of a load-balancing agent which moves sessions to a backup server when the load on the main server increases beyond  $2/3$  of the total resources allocated to the main server. This is a simple load balancing strategy which tries to minimize the amount of work done on the backup server, while maintaining an acceptable quality of service. The cost annotation of the **goto** statement expresses the cost of moving the object; i.e., the marshalling of the object. The cost annotation of the succeeding **skip** statement expresses the corresponding cost of demarshalling, which take place on server. For simplicity, we have here set both cost values to 1. Figure 8 shows an ABS model of self-monitoring session objects which move themselves to the backup server if the execution of the current request takes more than a given amount of time (the *limit*). Here, the active method *run* serves as a monitor. Once the session has moved, the monitor sets *timeToMove* to infinite time  $\infty$  to ensure that it will not be applied again to the same request. The next request resets *timeToMove* to the limit again, which reenables the monitor.

*Simulations of Load-Balancing Deployment Scenarios.* For the simulations of the running example augmented with load balancing strategies, we added a second deployment component to the initial configurations of Section 4, and let both deployment components have the same capacity. Figure 9 summarizes all three scenarios (single server, smart agent, and self-balancing sessions) with deployment component capacities ranging from 10 to 55. It can be seen that the load balancing strategies outperform the single server in all cases (as they should, since these scenarios have twice the total number of resources). The simulations

```

interface Agent { Session getSession(); Unit free(Session session);}
interface Session { Bool thumbnailImage(Pixel size); }

class SessionImp(Agent agent, Time limit, DC backupserver)
implements Session {
  Time start = now; Bool active = False;
  Time timeToMove = limit; DC origserver = thiscomp;

  Bool thumbnailImage(Pixels size) { // With monitor
    Int cost = size/150; Int deadline = size/2400;
    start = now; active = True; timeToMove= now+limit;
    while (cost > 0) {[Cost: 1] cost = cost - 1; suspend;}
    active = False; agent!free(this);
    Bool success = (now-start) <= deadline;
    if (thiscomp != origserver){[Cost: 1] goto(server);[Cost: 1] skip;}
    return (success);}

  Unit run() { // The monitor
    await (active ^ now > timeToMove);
    [Cost: 1] goto(backupserver); [Cost: 1] skip;
    timeToMove = ∞;
    this!run();}
  }

class AgentImp(Time limit, DC backupserver) implements Agent {
  // Same as in Fig. 1 but creating Session objects
  // with a parameter backupserver
}

```

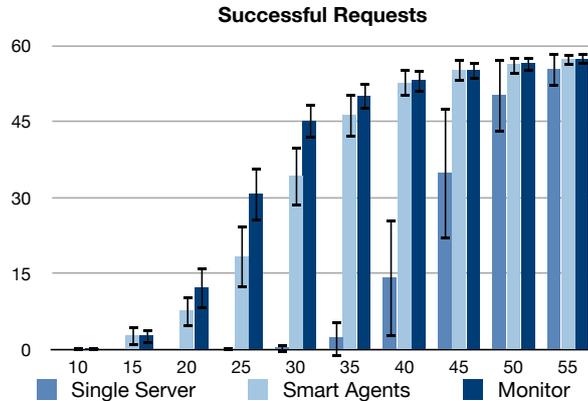
**Fig. 8.** Self-monitoring session objects for a thumbnail image service.

show that under constrained scenarios, the monitor strategy outperforms the smart agent for our example model and usage scenario.

Our simulation tool can also record behavior of models over time. Figure 10 shows the time progression of the average load (of 100 simulation runs) on the two deployment components under both balancing strategies, with main and backup deployment component both running with 35 resources. The simulation runs vary quite a bit, with standard deviation around 15 for all servers, except for the main server under the monitor strategy, which exhibits a standard deviation of approximately 10 throughout the run. It can also be seen that the server utilization under the monitor strategy is more stable, with the main server load around 30 (of 35) on average, and less work being transferred to the backup server. In summary, our simulation tools can be used both for quantitative insights into aggregated model behavior, and for understanding of timed behavior of models.

## 6 Related Work

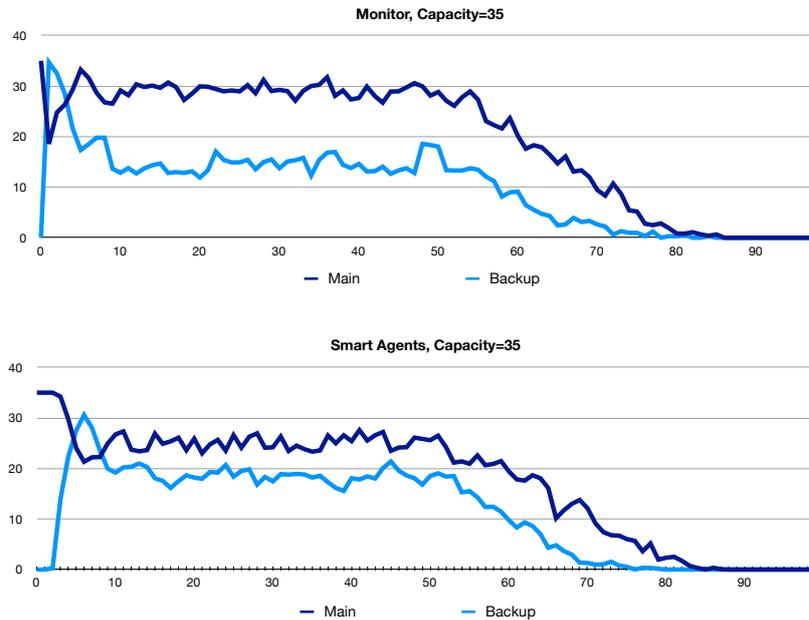
Asynchronously communicating software units, known from Actors and Erlang, are interesting due to their inherent compositionality. Concurrent objects with asynchronous method calls and futures combine asynchronous communication with object orientation [2, 9, 30, 32]. In these models, each software unit is also



**Fig. 9.** Simulation results of single server, smart agent and monitoring load balancing strategies: number of successful, i.e., non-timeout responses for 60 requests, with server capacity varying between 10 and 55. The numbers show mean and standard deviation of 100 runs for each server capacity for each scenario.

a unit of concurrency. There is a vast literature on formal models of mobility, based on, e.g., agents, ambient calculi, and process algebras, which is typically concerned with maintaining correct interactions between the moving entities with respect to, e.g., security, link failure, or location failure. For non-functional properties, access to shared resources have been studied through type and effect systems (e.g., [17, 18]), QoS-aware processes proposed for negotiating contracts [27], and space control achieved by typing for space-aware processes [7]. Closer to our work, timed synchronous CCS-style processes can be compared for speed using faster-than bisimulation [25], albeit without notions of mobility or location. We are not aware of other formal models connecting execution capacities to locations as in the deployment components studied in our paper.

This paper is part of ongoing work on resource-restricted execution contexts for concurrent objects [4, 22, 23]. Whereas [4] considers memory usage, deployment components with parametric concurrent resources were introduced in [23], extending work on a timed rewriting logic semantics for Creol [8]. A follow-up paper considers resources as first-class citizens of the language, formalizing the semantics of ABS with resource reallocation in rewriting logic [22]. In contrast, the present paper considers object mobility using a **goto** statement to allow an object to move to another deployment component, formalized in a more abstract SOS semantics. Relocation is possible due to the inherent compositionality of concurrent objects [12]: processes are encapsulated inside objects and the state of other objects can only be accessed through asynchronous method calls. This way the object is in control of its own location, which fits with the encapsulation of both state and control in the concurrent object model. Resource realloca-



**Fig. 10.** Main and backup server with 35 resources, using the monitoring (top) and smart agent (bottom) load-balancing strategies. Mean value of 100 runs plotted, standard deviation was around 15 resources (not plotted) throughout, with different runs exhibiting load spikes at different points in time.

tion and object mobility are in a sense complementary means to achieve load balancing: both have applications where they seem most natural.

Techniques and methodologies for predictions 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, e.g., JMeter or LoadRunner. Model-based approaches allow abstraction from specific system details, but depend on parameters provided by domain experts [13]. A survey of model-based performance analysis techniques is given in [6]. Formal approaches using Petri Nets, game theory, and timed automata (e.g., [10, 14, 24]) have been applied in the embedded software domain, but also to the schedulability of tasks in concurrent objects [19]. That work complements ours as it does not consider resource restrictions on the concurrency model, but associates deadlines with method calls.

Work on object-oriented models with resource constraints is more scarce. Based on a UML profile for schedulability, performance and time, the informally defined Core Scenario Model (CSM) [28] targets questions in performance model building. CSM has a notion of resource context, which reflects the set of resources used by an operation. CSM aims to bridge the gap between UML specifications and techniques to generate performance models [6]. UML models

with stochastic annotations for performance prediction have been proposed for components [16]. Closer to our work is a VDM++ extension to simulate embedded real-time systems [31], in which architectures are explicitly modeled using CPUs and buses, and resources statically bound to the CPUs. However, their work does not address relocation and load balancing strategies.

## 7 Discussion and Future Work

As software is increasingly developed to be deployed on a variety of architectures, it is important to be able to analyze the behavior of a model under different resource assumptions. ABS uses deployment components with parametric resources to express deployment scenarios for high-level executable models. This paper proposes a primitive for relocating concurrent objects between deployment components, expressed at the abstraction level of ABS, which integrates with the formal framework of deployment components in an elegant and simple way. Furthermore, we consider the problem of modeling systems with different load balancing strategies by allowing objects to move between deployment components, depending on the work load of their component. We demonstrate how a simple language extension is sufficient to naturally express dynamic object relocation strategies in this setting; our example shows how traffic on deployment components may cause congestion in the model, resulting in performance degradation for given deployment scenarios, and how load balancing strategies can be used to dynamically alleviate the congestion and thus to improve the overall performance of the model in a given deployment scenario. For the analysis of the deployment scenarios, we have extended ABS with a random expression and our simulation tool to do Monte Carlo simulations, which allows us to observe average behavior for the deployment scenarios.

As a technical contribution of this paper, we have extended ABS with support for user-defined cost models in terms of annotations, which provide a much more flexible framework for expressing processing cost than in our previous work. For example, the cost of an assignment may depend on the cost of evaluating a function on the right hand side of the assignment, which again may depend on the size of the input to the function. While this is expressible by user annotations as proposed in this paper, it leaves significant responsibility with the modeler. In future work, we will consider how the modeler may be assisted in this task by means of tools. In particular, static analysis techniques may in many cases be applicable to approximate the actual cost of a statement in terms of worst-case upper bounds (e.g., following [3]). In a recent paper [4], we have shown how static analysis may be combined with simulation for the memory analysis of untimed ABS models. However, it remains to combine this approach with user-defined cost models and time, and to integrate the tools. In a more long term perspective, we are interested in how to combine different user-defined resources in the same model.

## References

1. G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems*. The MIT Press, Cambridge, Mass., 1986.
2. W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programming*, 2010. In press.
3. 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.
4. 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 M. Butler and W. Schulte, editors, *FM 2011*, volume 6664 of *Lecture Notes in Computer Science*, pages 353–368. Springer, June 2011.
5. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
6. 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.
7. F. Barbanera, M. Bugliesi, M. Dezani-Ciancaglini, and V. Sassone. Space-aware ambients and processes. *Theoretical Computer Science*, 373(1–2):41–69, 2007.
8. J. Bjørk, E. B. Johnsen, O. Owe, and R. Schlatte. Lightweight time modeling in Timed Creol. *Electronic Proceedings in Theoretical Computer Science*, 36:67–81, 2010. *Proceedings of 1st Intl. Workshop on Rewriting Techniques for Real-Time Systems (RTRTS 2010)*.
9. D. Caromel and L. Henrio. *A Theory of Distributed Object*. Springer, 2005.
10. A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Resource interfaces. In R. Alur and I. Lee, editors, *Proc. Third Intl. Conf. on Embedded Software (EMSOFT’03)*, volume 2855 of *Lecture Notes in Computer Science*, pages 117–133. Springer, 2003.
11. 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 *Lecture Notes in Computer Science*. Springer, 2007.
12. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Proc. 16th European Symposium on Programming (ESOP’07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, Mar. 2007.
13. I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by runtime parameter adaptation. In *Proc. 31st Intl. Conf. on Software Engineering (ICSE’09)*, pages 111–121. IEEE, 2009.
14. E. Fersman, P. Krcál, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.
15. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.
16. J. Happe, H. Koziol, and R. Reussner. Parametric performance contracts for software components with concurrent behaviour. In *Proc. Third Intl. Workshop on Formal Aspects of Component Software (FACS’06)*, volume 182 of *Electronic Notes in Theoretical Computer Science*, pages 91–106, 2007.
17. M. Hennessy. *A Distributed Pi-Calculus*. Cambridge University Press, 2007.
18. A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 27(2):264–313, 2005.

19. 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.
20. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th Intl. Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.
21. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
22. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Dynamic resource reallocation between deployment components. In J. S. Dong and H. Zhu, editors, *Proc. Intl. Conf. on Formal Engineering Methods (ICFEM’10)*, volume 6447 of *Lecture Notes in Computer Science*, pages 646–661. Springer, Nov. 2010.
23. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Validating timed models of deployment components with parametric concurrency. In B. Beckert and C. Marché, editors, *Proc. Intl. Conf. on Formal Verification of Object-Oriented Software (FoVeOOS’10)*, volume 6528 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2011.
24. M. Katelman, J. Meseguer, and J. C. Hou. Redesign of the LMST wireless sensor protocol through formal modeling and statistical model checking. In G. Barthe and F. S. de Boer, editors, *Proc. 10th Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS’08)*, volume 5051 of *Lecture Notes in Computer Science*, pages 150–169. Springer, 2008.
25. G. Lüttgen and W. Vogler. Bisimulation on speed: A unified approach. *Theoretical Computer Science*, 360(1–3):209–227, 2006.
26. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
27. R. D. Nicola, G. L. Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A process calculus for QoS-aware applications. In J.-M. Jacquet and G. P. Picco, editors, *Proc. 7th Intl. Conf. on Coordination Models and Languages (COORDINATION’05)*, volume 3454 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2005.
28. 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.
29. G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.
30. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *European Conf. on Object-Oriented Programming (ECOOP 2010)*, volume 6183 of *Lecture Notes in Computer Science*. Springer, June 2010.
31. M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Proceedings of the 14th Intl. Symposium on Formal Methods (FM’06)*, volume 4085 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2006.
32. A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. Object oriented programming, systems, languages, and applications (OOPSLA’05)*, pages 439–453, New York, NY, USA, 2005. ACM Press.
33. S. M. Yacoub. Performance analysis of component-based applications. In G. J. Chastek, editor, *Proc. Second Intl. Conf. on Software Product Lines (SPLC’02)*, volume 2379 of *Lecture Notes in Computer Science*, pages 299–315. Springer, 2002.