Dynamic Resource Reallocation Between Deployment Components [⋆]

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

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

Abstract. Today's software systems are becoming increasingly configurable and designed for deployment on a plethora of architectures, ranging from sequential machines via multicore and distributed architectures to the cloud. Examples of such systems are found in, e.g., software product lines, service-oriented computing, information systems, embedded systems, operating systems, and telephony. To model and analyze systems without a fixed architecture, the models need to naturally capture and range over relevant deployment scenarios. For this purpose, it is interesting to lift aspects of low-level deployment concerns to the abstraction level of the modeling language. In this paper, the object-oriented modeling language Creol is extended with a notion of dynamic deployment components with parametric processing resources, such that processor resources may be explicitly reallocated. The approach is compositional in the sense that functional models and reallocation strategies are both expressed in Creol, and functional models can be run alone or in combination with different reallocation strategies. The formal semantics of deployment components is given in rewriting logic, extending the semantics of Creol, and executes on Maude, which allows simulations and test suites to be applied to models which vary in their available resources as well as in their resource reallocation strategies.

1 Introduction

Software systems today are increasingly being developed to be highly configurable, not only with respect to the functionality provided by a specific instance of the system but also with respect to the targeted deployment architecture. An example of a development method which attempts to systematize this variability, is software product line engineering [23]; in a product line, different software systems (or products) may be instantiated with different features and for different architectures. Deployment variability may be found in operating systems, which can be adapted to specific hardware and even to different numbers of available kernels; web shops, which are deployed on a varying number of servers and may even dynamically perform load balancing between these servers; and information systems within, e.g., healthcare or finance, which may run on a single computer,

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

in a distributed set-up, or even in the cloud. Software product lines raise new challenges for the performance analysis of component-based applications [27]. In this paper, we consider the performance analysis of object-oriented component or system models in deployment scenarios where the amount of processing resources available to a component may vary over time.

Our work is based on Creol [10, 17], a modeling language for concurrent objects communicating by asynchronous method calls. Creol has an operational semantics in rewriting logic [21] which is executable on Maude [9]. Concurrent objects resemble Actors [2] and Erlang [4] processes: Objects are inherently concurrent, conceptually each object has a dedicated processor, and there is at most one activity in an object at any time. This concurrency model is attracting attention as an alternative to multi-thread concurrency in object-orientation (e.g., [6]), and been integrated with, e.g., Java [26] and Scala [13]. Concurrent objects support compositional verification of concurrent software [3, 10], in contrast to multi-threading [1]. A distinguishing feature of Creol is its cooperative scheduling of method activations inside concurrent objects. Recently, Creol's notion of cooperative scheduling and asynchronous method calls has been integrated in Java by means of concurrent object groups [24].

This paper generalizes the idea of concurrent object groups to *dynamic deployment components* which are parametric in the amount of concurrent activity they allow within a time interval, and between which resources may be reallocated. Creol is extended with notions of timed execution and deployment components, which are integrated into Creol's operational semantics. This integration is non-trivial in that it must capture parametric concurrent activities within time intervals in terms of an interleaving concurrency semantics in order to execute the models on Maude. Deployment scenarios varying in the resources available to the deployment components, may be validated by means of test suites, executed on Maude. This allows the timed behavior of concurrent object models under restricted concurrency assumptions, as well as load balancing and process migration strategies between components, to be validated and compared.

Paper overview. Sect. 2 presents a timed version of Creol and Sect. 3 the dynamic deployment components. Sect. 4 illustrates this extension by the modeling and simulation of an example. Sect. 5 explains the operational semantics of the extended language and Sect. 6 discusses related work, and Sect. 7 concludes.

2 Concurrent Objects in Creol

Creol is an abstract behavioral modeling language for distributed active objects, based on asynchronous method calls and processor release points. In Creol, objects conceptually have dedicated processors and live in a distributed environment with asynchronous and unordered communication between objects. Communication is between named objects by means of asynchronous method calls; these may be seen as triggers of concurrent activity, resulting in new activities (so-called *processes*) in the called object. This section briefly introduces Creol (for further details see, e.g., [10,17]). Objects are dynamically created instances of

Syntactic categories.	*Definitions.*
C, I, m in Names	$IF ::= \textbf{interface}\ I\ \{\ [\overline{Sg}]\ \}$
g in Guard	$CL ::= \textbf{class}\ C\ [(\overline{I\ x})]\ [\textbf{implements}\ \overline{I}]\ \{\ [\overline{I\ x;}]\ \overline{M}\}$
s in Stmt	$Sg ::= I\ m\ ([\overline{I\ x}])$
x in Var	$M ::= Sg == [\overline{I\ x;}]\ \{\ s\ \}$
e in Expr	$g ::= b \mid x? \mid g \wedge g$
b in BoolExpr	$s ::= s; s \mid x := rhs \mid \textbf{release} \mid \textbf{await}\ g \mid \textbf{return}\ e$
	$\mid \textbf{if}\ b\ \textbf{then}\ \{\ s\ \}\ [\textbf{else}\ \{\ s\ \}] \mid \textbf{while}\ b\ \{\ s\ \} \mid \textbf{skip}$
	$e ::= x \mid b \mid \textbf{this} \mid \textbf{now} \mid \textbf{null}$
	$rhs ::= e \mid \textbf{new}\ C(\overline{e}) \mid [e]!m(\overline{e}) \mid [e.]m(\overline{e}) \mid x.\textbf{get}$

Fig. 1. The syntax of core Timed Creol. Terms such as $\overline{e}$ and $\overline{x}$ denote lists over the corresponding syntactic categories, square brackets [ ] denote optional elements. Expressions e and guards g are side-effect free; Boolean expressions b include comparison by means of equality, greater- and less-than operators. Expressions on other datatypes (strings, numbers) are written in the usual way and not contained in this figure.

classes, declared attributes are initialized to some arbitrary type-correct values. An optional *init* method may be used to redefine the attributes. Active behavior, triggered by an optional *run* method, is interleaved with passive behavior, triggered by method calls. Thus, an object has a set of processes to be executed, which stem from method activations. Among these, at most one process is *active* and the others are *suspended* on a process queue. Process scheduling is by default non-deterministic, but controlled by *processor release points* in a cooperative way. Creol is strongly typed: for well-typed programs, invoked methods are supported by the called object (when not *null*), such that formal and actual parameters match. This paper assumes that programs are well-typed.

Figure 1 gives the syntax for a core subset of Timed Creol (omitting, e.g., inheritance). A *program* consists of interface and class definitions and a `main` method to configure the initial state. IF defines an interface with name I and method signatures Sg. A class implements a list $\overline{I}$ of interfaces, specifying types for its instances. CL defines a class with name C, interfaces $\overline{I}$, class parameters and state variables x (of type I), and methods M. (The *attributes* of the class are both its parameters and state variables.) A method signature Sg declares the return type I of a method with name m and formal parameters $\overline{x}$ of types $\overline{I}$. M defines a method with signature Sg, a list of local variable declarations $\overline{x}$ of types $\overline{I}$, and a statement s. Statements may access class attributes, locally defined variables, and the method's formal parameters.

Statements. Assignment $x := rhs$, sequential composition $s_1; s_2$, and **if**, **skip**, **while**, and **return** constructs are standard. The statement **release** unconditionally releases the processor by suspending the active process. In contrast, the guard g controls processor release in the statement **await** g, and consists of Boolean conditions b and return tests $x?$ (see below). If g evaluates to false, the current process is *suspended* and the execution thread becomes idle. When the execution thread is idle, any enabled process from the pool of suspended processes may be scheduled. Explicit signaling is therefore redundant.

Expressions rhs include declared variables x, object identifiers o, Boolean expressions b, and object creation **new** $C(\overline{e})$ and **null**. The specially reserved read-only variable **this** refers to the identifier of the object and **now** refers to the current clock value (explained below). Note that pure expressions are denoted by e and that remote access to attributes is not allowed. (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 in the core syntax, and explained when used in the examples.)

Communication in Creol is based on asynchronous method calls, denoted $o!m(\overline{e})$, and future variables. (Local calls are written $!m(\overline{e})$.) After making an asynchronous call $x := o!m(\overline{e})$, the caller may proceed with its execution without blocking on the call. Here x is a future variable, o is an object expression, and $\overline{e}$ are (data value or object) expressions. A future variable x refers to a return value which has yet to be computed. There are two operations on future variables, controlling external synchronization in Creol. 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 be scheduled). Second, the return value is retrieved by the expression x.**get**, which blocks all execution in the object until the return value is available. The statement sequence $x := o!m(\overline{e})$; $v := x$.**get** encodes a *blocking call*, abbreviated $v := o.m(\overline{e})$ (often referred to as a synchronous call), whereas the statement sequence $x := o!m(\overline{e})$; **await** x?; $v := x$.**get** encodes a non-blocking, *preemptable call.*

Time. We consider a discrete time model, comparable to a system clock which updates every n milliseconds. With this granularity of time, an object which executes a statement may, but need not observe that time has advanced. The expression **now** returns the present time, i.e., the global clock's value in the current state. Time values are totally ordered by the less-than operator; comparing two time values result in a Boolean value which may be used as a guard in **await** statements. From an object's local perspective the passage of time is indirectly observable; time can advance by either evaluating statements, blocking, or simply awaiting the passage of time. This model of time combined with Creol's blocking and non-blocking synchronization semantics, is powerful enough to express both process- and object-wide *progress* statements.

3 Dynamic Deployment Components

Creol's object model is inherently concurrent, which means that for the actual deployment of a program it is necessary to map the logical concurrency of the model to physical computing resources. For this purpose, we introduce a notion of *deployment component* into the modeling language, which abstracts from the number and speed of the physical processors available to the component by a notion of *concurrent resource.* The granularity of the global time model defines the points in time when the executing system is observable. Concurrent resources may be consumed in parallel or in sequential order, which reflects the number of processors and their speeds relative to the granularity of the time intervals

$$
\begin{aligned}
s &::= \ldots \mid \mathtt{\textbf{transfer}}(e, e)\\
e &::= \ldots \mid \mathtt{\textbf{mycomp}} \mid \mathtt{\textbf{available}} \mid \mathtt{\textbf{load}}(e)\\
rhs &::= \ldots \mid \mathtt{\textbf{component}}(r) \mid \mathtt{\textbf{new}}\ C\,(\overline{e})\ \mathtt{\textbf{in}}\ e
\end{aligned}
$$

Fig. 2. Extension of the syntax for deployment components (r in Resources) in Fig. 1.

of the model. Thus, the logical concurrency model of the 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 allows easy configuration of concurrent resources.

The execution inside a deployment component can be understood as follows. Let n be a natural number. Resources are modeled by a data type `Resource` which extends the natural numbers with an "unlimited resource" ω, such that resource consumption is captured by subtraction, where $\omega - n = \omega$. Within a time interval, a deployment component with r concurrent resources may execute up to n execution steps in parallel, where $n \leq r$. Consider a deployment component D instantiated with r resources and let G be the set of concurrent objects which currently reside in the deployment component. Let $A \subseteq G$ be a subset of the concurrent objects on the component, such that objects in A are able to perform an execution step in their current state. Provided $|A| \leq r$, every object in A may consume a resource, leaving $r' = r - |A|$ resources available on the component. If there are remaining resources (i.e., $r' > 0$) , another set of execution steps is performed if possible within the same time interval by repeating this procedure.

In the modeling language, an object exists in the context of a deployment component with a given amount of resources, and may have variables x of type `Component` which refer to deployment components. A new deployment component is created by the statement x`:=`**`component`**`(`r`)`, which allocates a given quantity of concurrent resources r to the component x (capturing the actual processing capacity of x) by correspondingly reducing the resources of the current deployment component. The set of concurrent objects residing on the components, representing the logically concurrent activities, may grow dynamically. When objects are created, they must reside inside a deployment component. The syntax for object creation is extended with an optional clause to specify the targeted deployment component in the expression **`new`** C`(`$\overline{e}$`)` **`in`** x**`.`** This expresses that the new C object will reside in the component x. Objects generated by a parent object residing in a component x will also reside in x unless otherwise specified by an **`in`** clause. Thus the behavior of a Creol model which does not statically declare additional deployment components can be captured by a root deployment component with ω resources.

In the context of a given deployment component dc, the expression **`mycomp`** returns dc, **`available`** returns the number of resources currently allocated to dc, and **`load`**(e) returns the average number of used resources in dc during the last e time intervals. The statement **`transfer`**(x, r) reallocates r resources from dc to a component x. The language extension is summarized in Fig. 2.

```
interface TelephoneService { Void call(Int duration); }

interface SMSService { Void sendSMS(); }

class TelephoneService implements TelephoneService {
    Void call(Int duration) {
        Time t; t := now;
        await now >= t + duration;
    }
}
class SMSService implements SMSService {
    Void sendSMS() { skip; }
}
```

Fig. 3. Creol interfaces and classes for the telephony and SMS services.

4 Example: Phone Services During New Year's Eve

At midnight on new year's eve the behavior of cellphone users briefly changes from normal usage (i.e., a fairly low number of calls and messages) to sending large numbers of SMS messages. We use this phenomenon to motivate and illustrate resource reallocation by means of two cooperating deployment components. The model consists of two services, `TelephoneService` and `SMSService`, and a number of handset clients interacting with either the telephony or messaging service. The interfaces and implementations of the two services are given in Fig. 3. The method `call` will be called synchronously; as a parameter the client provides a duration for the call. The method `sendSMS` will be called asynchronously. Note that this model abstracts from many details (e.g. data model, bandwidth, server internals), which can be added as needed. The model of the handset clients interoperating with the services is given in Fig. 4. Client behavior is regulated by a parameter `cycle`, which determines the frequency of phone calls and messages of the handset. Between time $t = 50$ and 70, `Handset` objects change behavior and send SMS messages in a rapid pace.

Simulating this model in a scenario with ω resources leads to a *purely behavioral model*, in which each object acts according to its specification (as in normal Creol). Placing the SMS service in an environment with restricted resources will lead to observable overload during the midnight window, given sufficient clients to consume all its resources. (Recall that the load history of a deployment component over time can be extracted from a simulation run via its `Load` attribute.)

The proposed resource-related language constructs (i.e., **`available`**, **`load`**, and **`transfer`**) allow different load balancing schemes to be expressed. In Fig. 5 the `main` method defines an example scenario where each service runs in its own deployment component, created with 50 resources, and three client objects run in the unrestricted root component. Dynamic load balancing is captured by the `Balancer` class, an instance of which runs in parallel with the service in each component. This class implements a simple balancing strategy, transferring resources to its partner deployment component when receiving a `request` message (Line 12), and monitoring its own load and requesting assistance when

```
class Handset (Int cycle, TelephoneService ts, SMSService  smss) {
    Time created := now;
    Bool call := false;

    Void normalBehavior() {
        Time t := now;
        if (now > created + 50 ∧ now < created + 70) {
            !midnightWindow();
        } else {
            if (call) ts.call(1;) else smss!sendSMS()
            call := ¬call;
            await now >= t + cycle;
            !normalBehavior(); } }

    Void midnightWindow() {
        Time t := now;
        Int i := 0;
        if (now > created + 70) {
            !normalBehavior();
        } else {
            while (i < 10) { smss!sendSMS(); i := i+1; }
            await now > t;
            !midnightWindow(); } }

    op run() { !normalBehavior(); }
}
```

Fig. 4. The `Handset` class, implementing "Happy New Year" behavior. Before and after midnight, users alternate between short calls and sending single messages. During the midnight window ($50 \leq t \leq 70$), ten SMS per interval are sent.

needed (Line 9). Different, more involved or hierarchical, schemes for distributing resources among deployment components can be implemented similarly.

Figure 6 presents simulation results for this example scenario and for a scenario without load balancing, which shows that the available resources are sufficient for normal client behavior. In the load balancing scenario, the SMS service is overloaded between $t = 50$ and 53, at which time enough resources have been transfered from the telephony service to process the increased workload. After the load peak, the telephony service operates at capacity for one interval before receiving resources back from the SMS service. In the scenario without load balancing, the SMS service is overloaded during the whole load peak and another 12 time intervals while catching up with the backlog of delayed messages. Note that the functional part of the model was not changed between the two scenarios, and that more elaborate load balancing strategies can be added in similar ways.

5 Operational Semantics

The semantics of Creol is defined in rewriting logic (RL) [21], and Creol models can be analyzed using the rewrite tool Maude [9]. In a rewrite theory (Σ, E, L, R), the signature Σ defines the ground terms, E defines equations between terms, L is a set of labels, and R a set of labeled rewrite rules. Rewrite rules ap-

```
1  interface Balancer { Void setPartner(Balancer p);
   Void request(Component comp); }
2
3  class Balancer {
4      Balancer partner := null;
5      Void run () {
6          Time t := now;
7          await now > t;
8          if (partner≠null ∧ available<load(1)*0.9) {
9              partner.request(mycomp);}
10         !run(); }
11     Void request(Component comp) {
12         if (load(1) < available−10) {transfer(comp,available/2);} }
13     Void setPartner(Balancer p) { partner := p; }
14 }
15
16 Void main() {
17     Component smscomp := component(50);
18     Component telcomp := component(50);
19     SMSService sms := new SMSService() in smscomp;
20     TelephoneService tel := new TelephoneService() in telcomp;
21     Balancer smsb := new Balancer in smscomp;
22     Balancer telb := new Balancer in telcomp;
23     smsb.setPartner(telb);    telb.setPartner(smsb);
24     Client c := new Handset(1,tel,sms); c := new Handset(1,tel,sms);
25     c := new Handset(1,tel,sms); c := new Handset(1,tel,sms);
26 }
```

Fig. 5. A resource reallocation strategy and deployment configuration. Lines 21-23 initiate resource balancing; without these lines, the model runs with no functional changes but it has a different timing behavior due to overload in the SMS deployment component. Since `Handset` objects are active, references to them are not needed.

ply to terms of given sorts, specified in (membership) equational logic (Σ, E). When modeling computational systems, different system components are typically modeled by terms of suitable sorts and the global state configuration is a set of these terms. RL extends algebraic specification techniques with transition rules which capture the dynamic behavior of a system. A *conditional rewrite rule* **`crl`** `[`*name*`]`: $t \longrightarrow t'$ **`if`** *cond* transforms an instance of the pattern t to evolve into the corresponding instance of the pattern t', where the condition *cond* is a conjunction of rewrites and equations that must hold for the main rule to apply (the *name* identifies the rule). When auxiliary functions are needed, these can be defined in equational logic, and thus evaluated in between the state transitions [21]. In a *conditional equation* **`ceq`** $t \ = t'$ **`if`** *cond* the condition must similarly hold for the equation to apply. Unconditional rewrite rules and equations are denoted by the keywords **`rl`** and **`eq`**, respectively. Given an initial configuration, Maude supports simulation and breadth-first search through reachable states and model checking of finite reachable states for desired properties. In this paper, Maude is used as an interpreter for Creol's operational semantics to simulate and test Creol models.

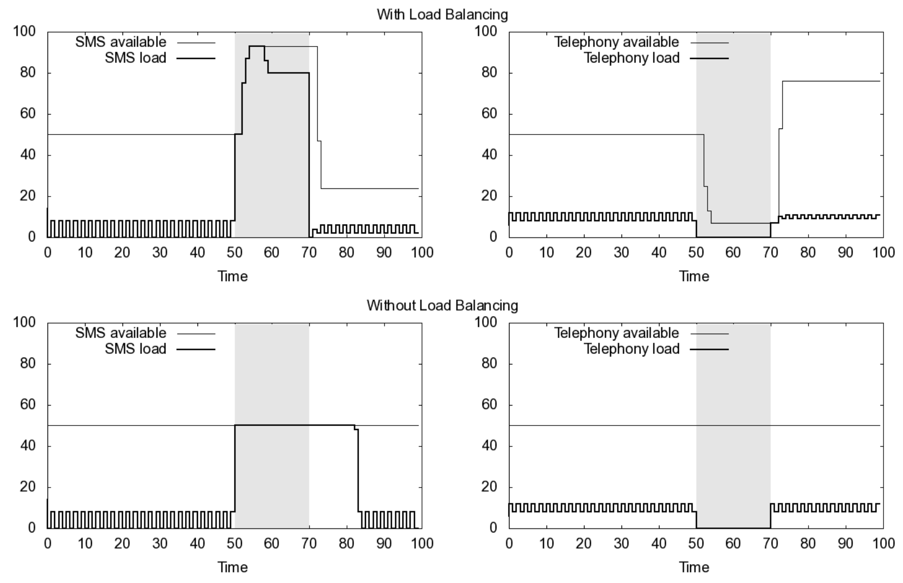

Fig. 6. Simulation of New Year's Eve behavior (SMS load spike between t=50 and t=70), with (top) and without resource balancing (bottom). The strategy of Fig. 5 distributes resources as needed between SMS component and telephony component.

The States. Following Maude conventions runtime objects are represented by terms $\langle o : C | \ldots,\ \texttt{Att}_i\texttt{:}\ x_i, \ldots\rangle$, where o is the identifier, C the class, and the object contains a set of attributes such that $\texttt{Att}_i$ is the name and x_i the current value of the i'th attribute. Variables are *slanted*, whereas constant parts of a term's syntax are in `typewriter` style. As before, $\overline{t}$ denotes a collection of terms t, either a list or a set depending on the context. Let `Emp` be the empty list and $\emptyset$ the empty set. In the rules below, all numbers are natural numbers (e.g., for time) except resources which are of sort `Resource`.

A state *configuration* is a set which consists of a global clock, deployment components, objects, classes, invocation messages, and futures. The associative and commutative union operator on configurations is denoted by whitespace and the empty configuration by `none`. The *entire* configuration lives inside curly brackets; thus, in the term $\{cn\}$ the variable cn captures the entire configuration. The global *clock* is a term $\langle t\texttt{: Clock} \mid \texttt{limit:}\ l\,\rangle$ where t is the current time and l the time limit considered in an execution. A *deployment component* is a term $\langle\, dc\texttt{: Comp} \mid \texttt{Free:}\ r\texttt{, Limit:}\ max\texttt{,}\ \ \texttt{Next:}\ next\texttt{,}\ \ \texttt{Load:}\ \overline{m}\,\rangle$ where dc is the identifier of the component, r the (non-negative) number of available computing resources, max the maximum number of resources which can be consumed before time advances, $next$ the maximum for the *next time interval*, and $\overline{m}$ the history of resource consumption over past time intervals.

An *object* is a term $\langle\, o\texttt{:}\ C \mid \texttt{Att:}\ \overline{a}\texttt{,}\ \ \texttt{Pr:}\ \{\overline{l} \mid \overline{s}\}\texttt{,}\ \ \texttt{PrQ:}\ \overline{w}\texttt{,}\ \ \texttt{Lcnt:}\ f\,\rangle$ where o is the identifier and C the object's class, its state is given by the attribute mapping

crl [*skip*] : $\langle o : C \mid \texttt{Pr}\colon \{\bar{l} \mid \texttt{skip;}\overline{s}\} \rangle\ \langle dc\colon \texttt{Comp} \mid \texttt{Free}\colon r \rangle$
$\longrightarrow \langle o : C \mid \texttt{Pr}\colon \{\bar{l} \mid \overline{s}\} \rangle\ \langle dc\colon \texttt{Comp} \mid \texttt{Free}\colon r-1 \rangle$ **if** $dc = \overline{a}[\texttt{mycomp}]$.

crl [*assign*] : $\langle o : C \mid \texttt{Att}\colon \overline{a},\ \texttt{Pr}\colon \{\bar{l} \mid x{:=}e\texttt{;}\overline{s}\} \rangle\ \langle t\colon \texttt{Clock} \mid \rangle\ \langle dc\colon \texttt{Comp} \mid \texttt{Free}\colon r \rangle$
$\longrightarrow$ **if** $x \in \texttt{dom}(\bar{l})$ **then** $\langle o : C \mid \texttt{Att}\colon \overline{a},\ \texttt{Pr}\colon \{\bar{l}[x \mapsto [\![e]\!]^{none}_{(\overline{a}\circ\bar{l}),t}] \mid \overline{s}\} \rangle$
else $\langle o : C \mid \texttt{Att}\colon \overline{a}[x \mapsto [\![e]\!]^{none}_{(\overline{a}\circ\bar{l}),t}],\ \texttt{Pr}\colon \{\bar{l} \mid \overline{s}\} \rangle$ **fi**
$\langle t\colon \texttt{Clock} \mid \rangle\ \langle dc\colon \texttt{Comp} \mid \texttt{Free}\colon r-1 \rangle$ **if** $dc = \overline{a}[\texttt{mycomp}]$.

crl [*return*] : $\langle o : C \mid \texttt{Att}\colon \overline{a},\ \texttt{Pr}\colon \{\bar{l} \mid \texttt{return}(e)\texttt{;}\overline{s}\} \rangle\ \langle t\colon \texttt{Clock} \mid \rangle$
$\langle n\colon \texttt{Fut} \mid \texttt{Done}\colon \texttt{false},\ \texttt{Value}\colon \bot \rangle\ \langle dc\colon \texttt{Comp} \mid \texttt{Free}\colon r \rangle$
$\longrightarrow \langle o : C \mid \texttt{Att}\colon \overline{a},\ \texttt{Pr}\colon \{\bar{l} \mid \overline{s}\} \rangle\ \langle n\colon \texttt{Fut} \mid \texttt{Done}\colon \texttt{true},\ \texttt{Value}\colon [\![e]\!]^{none}_{(\overline{a}\circ\bar{l}),t} \rangle$
$\langle t\colon \texttt{Clock} \mid \rangle\ \langle dc\colon \texttt{Comp} \mid \texttt{Free}\colon r-1 \rangle$ **if** $n = \bar{l}(\texttt{destiny}) \ \wedge\ dc = \overline{a}[\texttt{mycomp}]$.

rl [*release*] : $\langle o : C \mid \texttt{Pr}\colon \{\bar{l} \mid \texttt{release;}\overline{s}\},\ \texttt{PrQ}\colon \overline{w} \rangle$
$\longrightarrow \langle o : C \mid \texttt{Pr}\colon \texttt{idle},\ \texttt{PrQ}\colon \texttt{enqueue}(\{\bar{l} \mid \overline{s}\}, \overline{w}) \rangle$.

crl [*await1*] : $\{\langle o : C \mid \texttt{Att}\colon \overline{a},\ \texttt{Pr}\colon \{\bar{l} \mid \texttt{await}\ e\texttt{;}\overline{s}\} \rangle\ cn\ \langle t\colon \texttt{Clock} \mid \rangle\}$
$\longrightarrow \{\langle o : C \mid \texttt{Att}\colon \overline{a},\ \texttt{Pr}\colon \{\bar{l} \mid \overline{s}\} \rangle\ cn\ \langle t\colon \texttt{Clock} \mid \rangle\}$ **if** $[\![e]\!]^{cn}_{(\overline{a}\circ\bar{l}),t}$.

crl [*await2*] : $\{\langle o : C \mid \texttt{Att}\colon \overline{a},\ \texttt{Pr}\colon \{\bar{l} \mid \texttt{await}\ e\texttt{;}\overline{s}\} \rangle\ cn\ \langle t\colon \texttt{Clock} \mid \rangle\}$
$\longrightarrow \{\langle o : C \mid \texttt{Att}\colon \overline{a},\ \texttt{Pr}\colon \{\bar{l} \mid \texttt{release;await}\ e\texttt{;}\overline{s}\} \rangle\ cn\ \langle t\colon \texttt{Clock} \mid \rangle\}$ **if** $\neg[\![e]\!]^{cn}_{(\overline{a}\circ\bar{l}),t}$.

crl [*activate*] : $\{\langle o : C \mid \texttt{Att}\colon \overline{a},\ \texttt{Pr}\colon \texttt{idle},\ \texttt{PrQ}\colon \overline{w} \cup \{\{\bar{l} \mid \overline{s}\}\} \rangle\ cn\ \langle t\colon \texttt{Clock} \mid \rangle\}$
$\longrightarrow \{\langle o : C \mid \texttt{Att}\colon \overline{a},\ \texttt{Pr}\colon p,\ \texttt{PrQ}\colon \texttt{dequeue}(\overline{w}, p)\rangle\ cn\ \langle t\colon \texttt{Clock} \mid \rangle\}$
if $p = \texttt{select}(\overline{w}, \overline{a}, cn, t)$.

crl [*async-call*] : $\langle o : C \mid \texttt{Att}\colon \overline{a},\ \texttt{Pr}\colon \{\bar{l} \mid x{:=}e!m(\overline{e})\texttt{;}\overline{s}\},\ \texttt{Lcnt}\colon f \rangle\ \langle t\colon \texttt{Clock} \mid \rangle$
$\langle dc\colon \texttt{Comp} \mid \texttt{Free}\colon r \rangle$
$\longrightarrow \langle o : C \mid \texttt{Att}\colon \overline{a},\ \texttt{Pr}\colon \{\bar{l}[x \mapsto n] \mid \overline{s}\},\ \texttt{Lcnt}\colon \texttt{next}(f) \rangle\ \langle dc\colon \texttt{Comp} \mid \texttt{Free}\colon r-1 \rangle$
$\texttt{invoc}([\![e]\!]^{none}_{(\overline{a}\circ\bar{l}),t}, n, m, [\![\overline{e}]\!]^{none}_{(\overline{a}\circ\bar{l}),t})\ \langle n\colon \texttt{Fut} \mid \texttt{Done}\colon \texttt{false},\ \texttt{Value}\colon \bot \rangle\ \langle t\colon \texttt{Clock} \mid \rangle$
if $n = \texttt{label}(o, f) \wedge o \neq [\![e]\!]^{none}_{(\overline{a}\circ\bar{l}),t}\ \wedge\ dc = \overline{a}[\texttt{mycomp}]$.

rl [*bind-method*] : $\texttt{invoc}(o, n, m, \overline{d})\ \langle o : C \mid \texttt{PrQ}\colon \overline{w} \rangle$
$\langle C\colon \texttt{Class} \mid \texttt{Mtds}\colon (\overline{M} \cup \{\langle m\colon \texttt{Mtd} \mid \texttt{Prm}\colon \overline{x},\ \texttt{Att}\colon \bar{l},\ \texttt{Code}\colon \overline{s} \rangle\})\rangle$
$\longrightarrow \langle o : C \mid \texttt{PrQ}\colon \overline{w} \cup \{\{\bar{l}[\texttt{destiny} \mapsto n, \overline{x} \mapsto \overline{d}] \mid \overline{s}\}\} \rangle$
$\langle C\colon \texttt{Class} \mid \texttt{Mtds}\colon (\overline{M} \cup \{\langle m\colon \texttt{Mtd} \mid \texttt{Prm}\colon \overline{x},\ \texttt{Att}\colon \bar{l},\ \texttt{Code}\colon \overline{s} \rangle\})\rangle$.

crl [*receive-comp*] : $\langle o : C \mid \texttt{Att}\colon \overline{a},\ \texttt{Pr}\colon \{\bar{l} \mid x{:=}e\texttt{.get;}\overline{s}\} \rangle$
$\langle n\colon \texttt{Fut} \mid \texttt{Done}\colon \texttt{true},\ \texttt{Value}\colon d \rangle\ \langle dc\colon \texttt{Comp} \mid \texttt{Free}\colon r \rangle$
$\longrightarrow \langle o : C \mid \texttt{Att}\colon \overline{a},\ \texttt{Pr}\colon \{\bar{l} \mid x{:=}d\texttt{;}\ \overline{s}\} \rangle\ \langle n\colon \texttt{Fut} \mid \texttt{Done}\colon \texttt{true},\ \texttt{Value}\colon d \rangle$
$\langle dc\colon \texttt{Comp} \mid \texttt{Free}\colon r-1 \rangle$ **if** $n = [\![e]\!]^{none}_{(\overline{a}\circ\bar{l}),t} \wedge\ \texttt{dc} = \overline{a}[\texttt{mycomp}]$.

crl [*object-creation*] : $\langle o : C \mid \texttt{Att}\colon \overline{a},\ \texttt{Pr}\colon \{\bar{l} \mid x{:=}\texttt{new}\ B(\overline{e})\texttt{;}\overline{s}\} \rangle\ \langle t\colon \texttt{Clock} \mid \rangle$
$\langle dc\colon \texttt{Comp} \mid \texttt{Free}\colon r \rangle\ \langle B\colon \texttt{Class} \mid \texttt{Prm}\colon \overline{x},\ \texttt{Att}\colon \overline{a}_1,$
$\texttt{Mtds}\colon \overline{M} \cup \{\langle \texttt{init}\colon \texttt{Mtd} \mid \texttt{Prm}\colon \texttt{Emp},\ \texttt{Att}\colon \emptyset,\ \texttt{Code}\colon \overline{s}_1 \rangle\},\ \texttt{Ocnt}\colon g \rangle$
$\longrightarrow \langle o : C \mid \texttt{Att}\colon \overline{a},\ \texttt{Pr}\colon \{\bar{l} \mid x{:=}\texttt{newId}(B, g)\texttt{;}\overline{s}\} \rangle\ \langle B\colon \texttt{Class} \mid \texttt{Prm}\colon \overline{x},\ \texttt{Att}\colon \overline{a}_1,$
$\texttt{Mtds}\colon \overline{M} \cup \{\langle \texttt{init}\colon \texttt{Mtd} \mid \texttt{Prm}\colon \texttt{Emp},\ \texttt{Att}\colon \emptyset,\ \texttt{Code}\colon \overline{s}_1 \rangle\},\ \texttt{Ocnt}\colon \texttt{next}(g) \rangle$
$\langle \texttt{newId}(B, g) : B \mid \texttt{Att}\colon \overline{a}_1[\texttt{mycomp} \mapsto dc,\ \texttt{this} \mapsto \texttt{newId}(B, g), \overline{x} \mapsto [\![\overline{e}]\!]^{none}_{\overline{a}\circ\bar{l},t}],$
$\texttt{Pr}\colon \{\emptyset \mid \overline{s}_1\},\ \texttt{PrQ}\colon \emptyset,\ \texttt{Lcnt}\colon 0 \rangle\ \langle t\colon \texttt{Clock} \mid \rangle\ \langle dc\colon \texttt{Comp} \mid \texttt{Free}\colon r-1 \rangle$
if $dc = \overline{a}[\texttt{mycomp}]$.

Fig. 7. A timed rewriting logic semantics for Creol. In the rewrite rules, the variable r ranges over non-zero natural numbers to ensure that resource values are non-negative. The rules for the **`if`** and **`while`** statements are standard and not shown in this figure.

$\overline{a}$ (i.e., a single *binding* a binds a value to a declared variable), a *process* $\{\overline{l}\ |\overline{s}\}$ consists of a mapping $\overline{l}$ of local variable bindings and a list $\overline{s}$ of statements. The set $\overline{w}$ of (suspended) processes represents the process queue and the attribute f is used to ensure that futures created by the object have unique identifiers (next(f) provides a new fresh value).

A *class* is a term $\langle C\texttt{: Class} \,|\, \texttt{Prm: }\overline{x}\texttt{, Att: }\overline{a}\texttt{, Mtds: }\overline{M}\texttt{, Ocnt: }g\,\rangle$ where C is the identifier, $\overline{x}$ the list of formal parameters, $\overline{a}$ maps declared attributes to default values, and $\overline{M}$ is the set of method definitions of the form $\langle m\texttt{: Mtd} \,|\, \texttt{Prm: }\overline{x}\texttt{, Att: }\overline{l}\texttt{, Code: }\overline{s}\,\rangle$. Here, m is the method name, $\overline{x}$ the formal parameter list, $\overline{l}$ the mapping of local variables to initial (default) values, and $\overline{s}$ a sequence of statements. The attribute g is used to create objects with unique identifiers.

An *invocation message* is a term $\texttt{invoc(}o\texttt{, }n\texttt{, }m\texttt{, }\overline{d}\texttt{)}$ where o is the callee, n the future to which the call's result is returned, m the method name, and $\overline{d}$ the call's actual parameter values. A *future* is a term $\langle n\texttt{: Fut} \,|\, \texttt{Done: }b\texttt{, Value: }d\,\rangle$ where n is the identifier, b a Boolean flag indicating whether the future's reply value has been received, and d the reply value.

Evaluating Expressions. Given a substitution σ, a time t and a configuration cn, we denote by $[\![e]\!]^{cn}_{\sigma,t}$ a confluent and terminating reduction system which reduces an expression e to a data value. Let $[\![\textbf{now}]\!]^{cn}_{\sigma,t} = t$, $[\![\textbf{mycomp}]\!]^{cn}_{\sigma,t} = \sigma[\textbf{mycomp}]$, the equations below define availability and resource load:

$$\begin{array}{llll}
\textbf{ceq} & [\![\texttt{\textbf{available}}]\!]^{cn\texttt{<}dc\ \texttt{:Comp|Limit: }r\texttt{>}}_{\sigma,t} & = r & \textbf{if}\ dc = \sigma\texttt{[}\textbf{mycomp}\texttt{]} \\
\textbf{ceq} & [\![\texttt{\textbf{load}}(n)]\!]^{cn\texttt{<}dc\ \texttt{:Comp|Load: }\overline{m}\texttt{>}}_{\sigma,t} & = avg(\overline{m}, n) & \textbf{if}\ dc = \sigma\texttt{[}\textbf{mycomp}\texttt{]}
\end{array}$$

$$\begin{array}{lll}
\textbf{eq} & avg(emp, n) & = 0 \\
\textbf{eq} & avg(\overline{m} \circ m, n) & = \textbf{if}\ \ n > 0\ \ \textbf{then}\ \ avg(\overline{m}, n-1) + m/min(n, length(\overline{m})) \\
 & & \qquad\qquad\quad \textbf{else}\ \ 0\ \ \textbf{fi}
\end{array}$$

where $avg(\overline{m}, n)$ calculates the average number of used resources during the last n time intervals (or the average of $\overline{m}$ if its length is shorter than n). Let $[\![\mathrm{x?}]\!]^{cn}_{\sigma,t} = \texttt{true}$ if $[\![\mathrm{x}]\!]^{cn}_{\sigma,t} = n$ and there is a future $\langle n\texttt{: Fut} \,|\, \texttt{Done: true, Value: }d\,\rangle$ in cn (for some value d), otherwise $[\![\mathrm{x?}]\!]^{cn}_{\sigma,t} = \texttt{false}$. The remaining cases of $[\![\mathrm{e}]\!]^{cn}_{\sigma,t}$ are fairly straightforward, looking up values for declared variables in σ. Expressions are always reduced inside an object in a given state configuration. Thus, $\sigma = \overline{a} \circ \overline{l}$, the composition of the object state $\overline{a}$ and the local variable bindings $\overline{l}$, the time t is the current global time, and the configuration cn is the current global configuration (ignoring the object itself). This ensures that **now**, **mycomp**, **available**, and **load(**n**)**, as well as reply guards and declared variables, are evaluated correctly in the state of the program.

Transitions. Rewrite rules transform state configurations into new configurations, and are given in Fig. 7. In the presentation of a rule, we follow the convention of Full Maude [9] and hide attributes in runtime objects unless they are needed for that specific rule. Rule *skip* consumes a **skip** in the active process and a resource in its deployment component. Rule *assign* evaluates an expression e and assigns the value to a variable x in the local state $\overline{l}$ or in the attributes $\overline{a}$, as appropriate, consuming a resource in its deployment component. (The rules for **if** and **while** statements are omitted from the presentation.)

Process suspension and activation. Three operations are used to manipulate the process queue $\overline{w}$: enqueue$(p, \overline{w})$ adds a process p to $\overline{w}$, select$(\overline{w}, \overline{a}, cn, t)$ selects a process from $\overline{w}$ (if $\overline{w}$ is empty or no process is *ready* [17], this is the idle process), and dequeue$(\overline{w}, p)$ removes the process p from $\overline{w}$. The actual definitions of enqueue and select are left undefined; different definitions correspond to different scheduling policies for processes and can be used to locally express, e.g., priority or fairness. Rule *release* suspends the active process to the process queue. We denote by `idle` the idle process. Rule *await1* consumes the await statement if the guard evaluates to true in the current state, rule *await2* adds a `release` statement in order to suspend the process if the guard evaluates to false. Rule *activate* selects a process from the process queue for execution if this process is *ready* to execute, i.e., if it would not directly be resuspended or block the processor [17].

Communication and object creation. Rule *async-call* sends an invocation message to a callee with the actual method parameters and the identity of a future in which to place the method's return value. The caller creates the future associated with the call, with a unique identity `label`(o, f) constructed from the caller's own identity o and the local attribute f. The future's `Done` attribute is initially `false` and the return value is undefined (i.e., $\perp$). This operation consumes a resource. Rule *bind-method* transforms a method invocation into a corresponding process, placed in the process queue of the callee. The reserved local variable `destiny` stores the identity of the call's future. Rule *return* puts the return value into the future associated with the call (the `destiny`-variable refers to the appropriate future) and sets the future's `done` attribute to true. This operation consumes a resource. Rule *receive-comp* dereferences the future variable n in the case where the future's `Done` attribute is `true`. Note that if this attribute is `false` the reduction in this object is *blocked*. This operation consumes a resource. Finally, *object-creation* creates a new object with a unique identifier `newId`(B, g) constructed from the class identifier B and the local attribute g. The object's state is generated from default values for state attributes, extended with the actual values for `this` and the class parameters. The `init` method is loaded (we assume that this method reduces to `skip` if unspecified and that it asynchronously calls `run` if the latter is specified). This operation consumes a resource. Note that the new object inherits the deployment component of its creator. The rule for object creation in a named deployment component differs from *object-creation* only on this point, and is not presented.

Advancing time. We define a *run-to-completion* semantics for execution with the resource bounds of deployment components: objects must execute when possible if resources are available. To capture timed concurrent execution with an interleaving semantics, time cannot advance freely but is restricted as follows:

– For simplicity, we assume that invocation messages do not take time. Therefore, time may *not* advance while a message is on its way.

$\mathbf{eq}\ \texttt{canAdv}(cn',t) = \texttt{true}\ .$ $\qquad$ *//cn′ contains no objects or messages*
$\mathbf{eq}\ \texttt{canAdv}(msg\ cn,t) = \texttt{false}\ .$ $\qquad$ *//messages are instantaneous*
$\mathbf{eq}\ \texttt{canAdv}(\langle o : C \mid \rangle\ \langle dc\colon \texttt{Comp} \mid \texttt{Free: } 0\ \rangle\ cn,t)$ $\qquad$ *//no more resources*
$\quad = \texttt{canAdv}(\langle dc\colon \texttt{Comp} \mid \texttt{Free: } 0\ \rangle\ cn,t)\ .$
$\mathbf{eq}\ \texttt{canAdv}(\langle o : C \mid \texttt{Pr: } \{\bar{l} \mid n.\texttt{get;}\overline{s})\}\rangle$ $\qquad$ *//o is blocked, value not available*
$\quad \langle n\colon \texttt{Fut} \mid \texttt{Done: false}\ \rangle\ cn,t) = \texttt{canAdv}(\langle n\colon \texttt{Fut} \mid \texttt{Done: false}\ \rangle\ cn,t)\ .$
$\mathbf{ceq}\ \texttt{canAdv}(\langle o : C \mid \texttt{Att: } \overline{a}, \texttt{Pr: idle}, \texttt{PrQ: } \overline{w}\ \rangle\ cn,t)$ $\qquad$ *//no ready processes*
$\quad = \texttt{canAdv}(cn,t)\ \mathbf{if}\ \texttt{select}(\overline{w},\overline{a},cn,t) = \texttt{idle}.$
$\mathbf{eq}\ \texttt{canAdv}(\langle o : C \mid\ \rangle\ cn,t) = \texttt{false}\ [\mathbf{owise}]\ .$

$\mathbf{eq}\ \texttt{Adv}(\langle dc\colon \texttt{Comp} \mid \texttt{Free: } r, \texttt{Limit: } max,\ \texttt{Next: } next,\ \texttt{Load: } \overline{m}\ \rangle\ cn) =$
$\langle dc\colon \texttt{Comp} \mid \texttt{Free: } next, \texttt{Limit: } next,\ \texttt{Next: } next,\ \texttt{Load: } \overline{m} \circ max - r\ \rangle\ \texttt{Adv}(cn)\ .$
$\mathbf{eq}\ \texttt{Adv}(cn) = cn\ [\mathbf{owise}]\ .$

$\mathbf{crl}\ [progress]\colon \{cn\ \langle t\colon \texttt{Clock} \mid \texttt{limit: } l\ \rangle\} \longrightarrow \{\texttt{Adv}(cn)\ \langle t+1\colon \texttt{Clock} \mid \texttt{limit: } l\ \rangle\}$
$\mathbf{if}\ \texttt{canAdv}(cn,t) \wedge t < l\ .$

$\mathbf{crl}\ [resource\text{-}transfer]\colon\ \langle o : C \mid \texttt{Att: } \overline{a},\ \texttt{Pr: } \{\bar{l} \mid \texttt{transfer}(e,e_1)\texttt{;}\overline{s}\}\ \rangle$
$\langle dc_1\colon \texttt{Comp} \mid \texttt{Next: } nl\ \rangle\ \langle dc_2\colon \texttt{Comp} \mid \texttt{Next: } nl_1\ \rangle\ cn\ \ \langle t\colon \texttt{Clock} \mid \rangle$
$\longrightarrow \langle o : C \mid \texttt{Att: } \overline{a},\ \texttt{Pr: } \{\bar{l} \mid \overline{s}\}\ \rangle\ cn\ \ \langle t\colon \texttt{Clock} \mid \rangle\ \langle dc_1\colon \texttt{Comp} \mid \texttt{Next: } nl - d\ \rangle$
$\langle dc_2\colon \texttt{Comp} \mid \texttt{Next: } nl_1 + d\ \rangle\ \mathbf{if}\ [\![e]\!]^{cn}_{\overline{a}\circ\bar{l},t} = dc_2 \wedge [\![e_1]\!]^{cn}_{\overline{a}\circ\bar{l},t} = d \wedge nl \geq d \wedge dc_1 = \overline{a}[\texttt{mycomp}]\ .$

Fig. 8. Advancing time and transferring resources. The variable *msg* denotes a message, r a non-zero natural number, and *cn'* a message- and object-free configuration.

– If a deployment component has run out of resources, none of its objects may proceed, and time can advance.
– If a deployment component has remaining resources and one of the component's objects o may execute, time may *not* advance. There are three cases:
  1. The active process in o is blocked, but the value has become available.
  2. The active process in o is idle, but a suspended process can be activated.
  3. The active process in o is not blocked.

A predicate canAdv, defined recursively over configurations (see Fig. 8), formalizes these restrictions on time advance in an interleaving semantics for timed concurrent execution. Time may not advance if some object can execute, expressed by the **owise** equation for canAdv. (The keyword **owise** in Maude expresses that an equation is chosen only when no other equation applies.) Finally time may advance if no object can execute and there are no messages, which is captured by the first equation for canAdv. Once time advances, the global clock is updated and the deployment components get their resources refreshed for the next time interval. This is done by an auxiliary function Adv defined in Fig. 8, which updates a configuration by resetting the free resources of each deployment component to the limit specified by *next* and extending the load history of the components. (Here, $\overline{m} \circ m$ appends m to the sequence $\overline{m}$.)

The advancement of time is captured by the rewrite rule *progress* in Fig. 8. Observe that for simplicity time advances with a single unit. It would be straightforward to allow larger increments. In order to ensure termination of model execution, a *limit* has been added to the global clock and we only consider execution sequences up to this limit in time.

6 Related Work

Concurrent objects and Actors, in which software units with encapsulated processors communicate asynchronously, increasingly attract attention due to an intuitive and compositional concurrency model [2–4, 6, 10, 13, 26]. Creol proposes cooperative scheduling between asynchronously called methods [17], which allows active and reactive behavior to be combined within objects as well as compositional verification of partial correctness properties [3, 10]. This model of cooperative scheduling has recently been generalized to concurrent object groups in Java [24]. This paper further generalizes concurrent object groups to resource-constrained deployment components, where group activity per time interval is parametric in concurrent resources, using a time model which simplifies previous work [18]. The approach abstractly models the effect of deploying concurrent object groups on deployment components which vary in processing capacity.

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 intricacies, but depend on parameters provided by domain experts [11]. A survey of model-based performance analysis techniques is given in [5]. Formal approaches using process algebra, Petri Nets, game theory, and timed automata (e.g., [7, 8, 12, 15, 19, 20]) have been applied in the embedded software domain, but also to the schedulability of tasks in concurrent objects [16]. That work complements ours as it does not consider resource restrictions on the concurrency model, but associates deadlines with method calls.

Work on modeling object-oriented systems with resource constraints is more scarce. Using the UML SPT profile for schedulability, performance and time, Petriu and Woodside [22] informally define the Core Scenario Model (CSM) to solve questions that arise in performance model building. CSM has a notion of resource context, which reflects the set of resources used by an operation. CSM aims to bridge the gap between UML specifications and techniques to generate performance models [5]. UML models with stochastic annotations for performance prediction have been proposed for components [14]. Closer to our work is M. Verhoef's extension of VDM++ for simulation of embedded real-time systems [25], in which architectures are explicitly modeled using CPUs and buses, and resources are statically bound to the CPUs. Our work extends these ideas with dynamic load balancing strategies expressed in the modeling language and running in parallel with the behavioral parts of the model.

7 Conclusions and Future Work

We present a modeling framework which formalizes a high-level understanding of deployment concerns, reflecting the execution capabilities of underlying architectures. This framework is based on an abstract notion of execution resource, such that each component has an associated amount of available resources which can be used within a time interval. The framework is given as an extension of the object-oriented language Creol, allowing the dynamic creation of deployment components and the dynamic reallocation of resources, such that redistribution strategies can be expressed in terms of the load and the available resources of components. Resources and deployment components have been naturally integrated as first-class values at the abstraction level of the modeling language, including constructs to transfer resources, create deployment components and place new objects in given deployment components, as well as to check the current load of a component and its available resources. The extended language has been formalized by a timed operational semantics in rewriting logic. Rewriting logic semantics are directly executable in Maude, which allows the tool-supported simulation and analysis of models directly based on the operational semantics.

As shown by an example, the approach is compositional in the sense that the software controlling allocation and reallocation of resources can (but need not) be completely separated from the rest of the code. Classes express particular reallocation strategies, and one strategy object is created in each component that should be controlled by that strategy. It is easy to replace a strategy by another, to reuse strategies, and to apply different strategies to different components. This flexibility is valuable for software development with high needs for deployment configurability; for example in software product lines, variability in resources and reallocation strategies allow products to be deployed on different architectures while maintaining, e.g., response time requirements.

The proposed notions of resource and time stem from the need for abstract models which do not assume a fixed deployment scenario, yet support tool-based formal analysis and model exploration. However, our approach may be extended with more fine-grained notions of resources and resource consumption; e.g., using resource profiles for specific deployment scenarios. In future work, we plan to develop case studies using reallocation strategies based on gossiping, peer-to-peer, and hierarchical structures, as well as object migration. Furthermore, it is interesting to combine the simulation-based approach with concrete values in Maude with symbolic execution techniques for resource consumption and reallocation.

References

1. E. Ábrahám-Mumm, F. S. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept. In *Proc. FOSSACS'02*, LNCS 2303, pages 5–20. Springer, Apr. 2002.
2. G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems.* The MIT Press, Cambridge, Mass., 1986.
3. W. Ahrendt and M. Dylla. A verification system for distributed objects with asynchronous method calls. In K. Breitman and A. Cavalcanti, editors, *Proc. ICFEM'09*, LNCS 5885, pages 387–406. Springer, 2009.
4. J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.
5. S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. on Software Engineering*, 30(5):295–310, 2004.

6. D. Caromel and L. Henrio. *A Theory of Distributed Object.* Springer, 2005.
7. A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Resource interfaces. In *Proc. EMSOFT'03*, LNCS 2855, pages 117–133. Springer, 2003.
8. X. Chen, H. Hsieh, and F. Balarin. Verification approach of metropolis design framework for embedded systems. *Intl. J. Parallel Programming*, 34(1):3–27, 2006.
9. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.
10. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, LNCS 4421, pages 316–330. Springer, Mar. 2007.
11. I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by runtime parameter adaptation. In *Proc. ICSE'09*, pages 111–121. IEEE, 2009.
12. E. Fersman, P. Krcál, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Inf. and Comp.*, 205(8):1149–1172, 2007.
13. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.
14. J. Happe, H. Koziolek, and R. Reussner. Parametric performance contracts for software components with concurrent behaviour. In *Proc. 3rd Intl. Workshop on Formal Aspects of Component Software (FACS'06)*, ENTCS 182:91–106, 2007.
15. M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM Trans. on Prog. Languages and Systems*, 24(5):566–591, 2002.
16. M. M. Jaghoori, F. S. de Boer, T. Chothia, and M. Sirjani. Schedulability of asynchronous real-time concurrent objects. *Journal of Logic and Algebraic Programming*, 78(5):402–416, 2009.
17. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
18. E. B. Johnsen, O. Owe, J. Bjørk, and M. Kyas. An object-oriented component model for heterogeneous nets. In *Proc. Formal Methods for Components and Objects (FMCO 2007)*, LNCS 5382, pages 257–279. Springer, 2008.
19. M. Katelman, J. Meseguer, and J. C. Hou. Redesign of the lmst wireless sensor protocol through formal modeling and statistical model checking. In *Proc. FMOODS'08*, LNCS 5051, pages 150–169. Springer, 2008.
20. J.-P. Katoen, C. Baier, and D. Latella. Metric semantics for true concurrent real time. *Theoretical Computer Science*, 254(1–2):501–542, 2001.
21. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
22. 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.
23. K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques.* Springer, 2005.
24. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *Proc. ECOOP 2010*, LNCS 6183. Springer, June 2010.
25. M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In *Proc. Formal Methods (FM'06)*, LNCS 4085, pages 147–162. Springer, 2006.
26. A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. OOPSLA'05*, pages 439–453. ACM, 2005.
27. S. M. Yacoub. Performance analysis of component-based applications. In *Proc. Software Product Lines (SPLC'02)*, LNCS 2379, pages 299–315. Springer, 2002.