Validating Timed Models of Deployment Components with Parametric Concurrency *

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. Many software systems today are designed without assuming a fixed underlying architecture, and may be adapted for sequential, multicore, or distributed deployment. Examples of such systems are found in, e.g., software product lines, service-oriented computing, information systems, embedded systems, operating systems, and telephony. Models of such systems need to capture and range over relevant deployment scenarios, so it is interesting to lift aspects of low-level deployment concerns to the abstraction level of the modeling language. This paper proposes an abstract model of deployment components for concurrent objects, extending the Creol modeling language. The deployment components are parametric in the amount of concurrency they provide; i.e., they vary in processing resources. We give a formal semantics of deployment components and characterize equivalence between deployment components which differ in concurrent resources in terms of test suites. Our semantics is executable on Maude, which allows simulations and test suites to be applied to a deployment component with different concurrent resources.

1 Introduction

Software systems today are increasingly developed to be highly configurable. A development method which attempts to systematize this variability is software product line engineering [25]; in a product line, different software systems (or products) may be instantiated with different features. To illustrate this approach to software development, consider software for cell phones. Products for different cell phones and service subscriptions are produced by selecting among features such as call forwarding, answering machine, text messaging, etc. In addition to this software variability, products often need to be adapted to different hardware or deployment scenarios. Examples of such variability are found in operating systems, which can be adapted to specific hardware and even to the different numbers of available cores; 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, in a distributed set-up, or even on the cloud. Software product lines raise new challenges for the performance analysis of component-based

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

applications [29]. In this paper, we apply performance analysis to formal models of object-oriented components or systems in deployment scenarios which vary in the amount of concurrent resources they can provide to the given component.

This work is based on Creol [11,19], a modeling language for distributed concurrent objects which communicate by asynchronous method calls and futures. Creol's operational semantics is given in rewriting logic [21] and is executable on Maude [10]. Concurrent objects are reminiscent of Actors [1] and Erlang [4]: 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 has attracted attention as an alternative to multi-thread concurrency in object-orientation (e.g., [7]), and been integrated with, e.g., Java [28] and Scala [14]. Concurrent objects support compositional verification of concurrent software [2, 11], in contrast to multi-threading. A particular 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 [26].

This paper generalizes the idea of concurrent object groups to *deployment components* which are parametric in the amount of concurrent activity they allow within a time interval. 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 semantics in order to execute the models in Maude. We characterize the equivalence of different deployment scenarios, varying in the concurrency resources of the deployment components, in terms of test suites of timed observable behavior and use Maude to run tests for our models. This allows the timed behavior of concurrent object models under restricted concurrency assumptions to be validated and compared.

Paper Overview. Sect. 2 presents a timed version of Creol, and Sect. 3 the deployment components with parametric concurrency. Sect. 4 illustrates the language by an example. Sect. 5 explains the operational semantics of Creol extended with time and deployment components. Sect. 6 presents testing and simulation results in the context of the example, Sect. 7 discusses related work, and Sect. 8 concludes the paper.

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 (processes) in the called object. This section briefly introduces Creol (for further details see, e.g., [11, 19]). Objects are dynamically created instances of classes, their declared attributes are initialized to some arbitrary type-correct values. An

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

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.

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. The scheduling of processes 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 declared fields.) 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 and 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 $s_1; s_2$, **skip**, **if**, **while**, and **return** e 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 expressions b over attributes and return tests $x?$ (see below). If g evaluates to false, the current process is *suspended* and the active process becomes idle. In this case, any enabled process may be chosen from the pool of suspended processes. 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. Explicit signaling is redundant.

Expressions rhs include declared variables x, Boolean expressions b, and object creation **new** $C(\overline{e})$. The reserved read-only variable **this** refers to the

object identifier and **now** 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 here, and explained when used in the examples.)

Communication in Creol is based on asynchronous method calls, denoted $e!m(\overline{e})$, and future variables. (Local calls are written $!m(\overline{e})$.) After making an asynchronous call $x := e!m(\overline{e})$, the caller may proceed with its execution without blocking on the method reply. Here x is a future variable, and e and $\overline{e}$ are expressions. A future variable refers to a return value which may still need to be computed. There are two operations on future variables, which control synchronization in Creol. First, the guard **await** x? suspends the active process unless 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. 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. In this paper we work with an extended version of the Creol language which includes an implicit time model [6], comparable to a system clock which updates every n milliseconds (representing a time interval). In this extension 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 suitable for guards in **await** statements. From an object's local perspective, the passage of time is indirectly observable via **await** statements in this model of timed behavior, and time is advanced when no other activity may occur. In this paper this model of time is used to handle the amount of concurrent activity allowed within a time interval in order to model resource constrains for different deployment scenarios.

3 Deployment Components with Parametric Concurrency

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 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 modelled by a data type `Resource` which extends the natural numbers with an "unlimited resource" ω. Resource consumption is captured by subtraction, where $\omega - n = \omega$. Within a time interval, a deployment component with r concurrent resources is able to 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 ($r' > 0$), another cycle of execution steps may be performed for r' within the time interval by repeating this procedure.

In the modeling language, a deployment component D is declared by associating a name to a given quantity of concurrent resources r, capturing the actual processing capacity of D. For simplicity in this paper, a deployment component is a static entity, in contrast to class declarations which act as templates for the dynamic generation of objects. A component is introduced by the syntax **`component`** $D(r)$, where D is the name of the component and r, of sort `Resource`, represents the concurrent resources of the component. The set of concurrent objects residing on the components, representing the logically concurrent activities, may grow dynamically. Thus, 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: **`new`** $C(\overline{e})$ **`in`** D. This expresses that the C object will reside in component D. Objects generated by a parent object residing in a component D will also reside in D 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 main deployment component with ω resources.

4 Example: A Distributed Shopping Service

We consider a simple model of a web shop (see Fig. 2). Clients connect to the shop by calling the `getSession` method of an `Agent` object. An `Agent` hands out `Session` objects from a dynamically growing pool. Clients call the `order` method of their `Session` instance, which in turn calls the `makeOrder` method of a `Database` object that is shared across all sessions. After completing the order, the session object is added to the agent's pool again. This scenario models the architecture and control flow of a database-backed website, while abstracting from many details (load-balancing thread pools, data model, sessions spanning multiple requests, etc.), which can be added to the model should the need arise.

In the implementation of the `Database` class, an order takes a minimum amount of time, and should be completed within a maximum amount of time.

```
1  interface Agent { Session getSession(); Void free(Session session);}
2  interface Session { Bool order(); }
3  interface Database { Bool makeOrder(); }
4
5  class Database(Nat min, Nat max) implements Database {
6      Bool makeOrder () {
7          Time t:=now;
8          await now >= t + min;
9          return now <= t + max; }
10 }
11 class Agent(Database db, Set[Session] available) implements Agent{
12     Session getSession() {
13         if isempty(available) {
14             return new Session(this, db); }
15        else { session:=choose(available);
16                available:=remove(session,available);return session;}}
17     Void free(Session session){available:=add(available,session);}
18 }
19 class Session(Agent agent, Database db) implements Session {
20     Bool order() {return db.makeOrder(); agent.free(this); }
21 }
```

Fig. 2. A web shop model in Creol.

The timing behavior of the database is configurable via the class parameters `min` and `max`. Line 8 implements the delay while processing the order, Line 9 calculates and returns the success status of the order (i.e., whether a timeout occurred). Note that a component with unlimited resources, will complete all orders in the minimum amount of time, just as expected. In the `Agent` class, the attribute `available` stores a set of `Session` objects. (Creol has a datatype for sets, with operations `isempty` to check for the empty set, denoted `{}`, `choose` to select an element of a non-empty set, and `remove` and `add` to remove or add an element to a set.) When a customer requests a `Session`, the `Agent` takes a session from the available sessions if possible (Line 15), otherwise it creates a new session (Line 14). The method `free` inserts a session in the `available` sessions of the `Agent`, and is called by the session itself upon completion of an order. Section 6 will show how to run this example on a deployment component.

5 Operational Semantics

The operational semantics of timed Creol with deployment components is presented as a transition system in an SOS style [24]. Transition rules apply to subsets of configurations (the standard context rules are not listed). For simplicity we assume that configurations can be reordered to match the left hand side of the rules (i.e., matching is modulo associativity and commutativity as in rewriting logic [21]). A run of the system is a possibly nonterminating sequence of rule applications. When auxiliary functions are used in the semantics, these are evaluated in between the application of transition rules in a run.

Timed runtime timed configurations *tcn*, given in Fig. 3, include one global clock and an untimed configuration *cn*; i.e., a set which consists of deployment

$$
\begin{array}{rcl@{\qquad}rcl}
cn & ::= & \epsilon \mid comp \mid object \mid msg & tcn & ::= & clock\ cn \\
 & & \mid future \mid cn\ cn & comp & ::= & dc(n, r, u) \\
object & ::= & ob(o, C, a, p, q) & msg & ::= & invoc(o, f, m, \overline{v}) \\
q & ::= & \emptyset \mid p \mid q \backslash p \mid enqueue(p, q) & future & ::= & fut(f, v) \\
p & ::= & \{l|s\} \mid select(q, a, cn, t) \mid \text{idle} & clock & ::= & cl(t) \\
v & ::= & o \mid f \mid \text{null} \mid b \mid t & & &
\end{array}
$$

Fig. 3. The syntax for timed runtime configurations.

components, objects, invocation messages, and futures. The associative and commutative union operator on configurations is denoted by whitespace and the empty configuration by ε. These configurations live inside curly brackets; in the term $\{tcn\}$, tcn captures the entire configuration. The global *clock* is a term $cl(t)$ where t is the current time. A *deployment component* is a term $dc(n, r, u)$ where n is the identifier of the component, r the (non-negative) number of available computing resources, and u the maximum number of resources which can be consumed before the clock advances. An *object* is a term $ob(o, C, a, p, q)$ where o is the object's identifier and C its class, a is an attribute mapping representing the object's fields, p is an *active process*, and q is a *queue of suspended processes*. In the fields a of an object o, the reserved field 'mycomp' is bound to the deployment component associated with o. A process p consists of a mapping l of local variable bindings and a list s of statements, and will be written as $\{l|s\}$ when convenient. An *invocation message* is a term $invoc(o, f, m, \overline{v})$ where o is the callee, f the future to which the call's result shall be returned, m the method name, and $\overline{v}$ lists the call's actual parameter values. A *future* is a term $fut(f, v)$ with identifier f and reply value v (which is $\perp$ when the future's reply value has not been received). Values are object and future identifiers, Boolean expressions, clock values, and null (as well as expressions in the functional language).

Evaluating Expressions. Given a substitution σ, a time t, and a configuration cn, denote by $[\![e]\!]^{cn}_{\sigma,t}$ a confluent and terminating reduction system which reduces expressions e to data values. Let $[\![\mathbf{now}]\!]^{cn}_{\sigma,t} = t$. Let $[\![x?]\!]^{cn}_{\sigma,t} = \text{true}$ if $[\![x]\!]^{cn}_{\sigma,t} = f$ and $fut(f, v) \in cn$ for some value $v \neq \perp$, otherwise $[\![x?]\!]^{cn}_{\sigma,t} = \text{false}$. The remaining cases are fairly straightforward, looking up values for declared variables in σ. The reduction of an expression always happens in the context of a given process, object state, and configuration. Thus, $\sigma = a \circ l$ (the composition of the fields a and the local variable bindings l), t is the current global time, and cn the current untimed configuration of the system (ignoring the object itself).

The Rules. The rewrite rules of the operational semantics transform state configurations into new configurations, and are given in Fig. 4. If a and l are mappings, denote by $\text{dom}(a)$ the domain of a; by $a(x)$ the value bound to x in a (assuming that $x \in \text{dom}(a)$); by $a[x \mapsto v]$ the extension of a in which x is bound to v (and $a[x \mapsto v](x') = a(x')$ if $x \neq x'$); and by $a \circ l$ the composed mapping in which $a \circ l(x) = l(x)$ if $x \in \text{dom}(l)$ (and $a \circ l(x) = a(x)$ if $x \notin \text{dom}(l)$). For simplicity, classes are not represented explicitly in the semantics, but may be seen as static

(SKIP)
$$\frac{a(\text{mycomp}) = n \quad r > 0}{ob(o, C, a, \{l|\text{skip}; s\}, q)\ dc(n, r, u) \\ \to ob(o, C, a, \{l|s\}, q)\ dc(n, r-1, u)}$$

(ASSIGN1)
$$\frac{x \in \text{dom}(l) \quad v = [\![e]\!]^{\varepsilon}_{(a \circ l),t} \quad a(\text{mycomp}) = n \quad r > 0}{ob(o, C, a, \{l|x := e; s\}, q)\ dc(n, r, u)\ cl(t) \\ \to ob(o, C, a, \{l[x \mapsto v]|s\}, q)\ dc(n, r-1, u)\ cl(t)}$$

(ASSIGN2)
$$\frac{x \in \text{dom}(a) \quad v = [\![e]\!]^{\varepsilon}_{(a \circ l),t} \quad a(\text{mycomp}) = n \quad r > 0}{ob(o, C, a, \{l|x := e; s\}, q)\ dc(n, r, u)\ cl(t) \\ \to ob(o, C, a[x \mapsto v], \{l|s\}, q)\ dc(n, r-1, u)\ cl(t)}$$

(COND1)
$$\frac{[\![e]\!]^{\varepsilon}_{(a \circ l),t}}{ob(o, C, a, \{l|\text{if } e \text{ then } s_1 \text{ else } s_2 \text{ fi}; s\}, q)\ cl(t) \\ \to ob(o, C, a, \{l|s_1; s\}, q)\ cl(t)}$$

(COND2)
$$\frac{\neg[\![e]\!]^{\varepsilon}_{(a \circ l),t}}{ob(o, C, a, \{l|\text{if } e \text{ then } s_1 \text{ else } s_2 \text{ fi}; s\}, q)\ cl(t) \\ \to ob(o, C, a, \{l|s_2; s\}, q)\ cl(t)}$$

(READ-FUT)
$$\frac{v \neq \bot \quad f = [\![e]\!]^{\varepsilon}_{(a \circ l),t} \quad a(\text{mycomp}) = n \quad r > 0}{ob(o, C, a, \{l|x := e.\text{get}; s\}, q)\ fut(f, v)\ dc(n, r, u)\ cl(t) \\ \to ob(o, C, a, \{l|x := v; s\}, q)\ fut(f, v)\ dc(n, r-1, u)\ cl(t)}$$

(BIND-MTD)
$$\frac{p' = \text{bind}(o, f, m, \overline{v}, C)}{ob(o, C, a, p, q)\ invoc(o, f, m, \overline{v}) \\ \to ob(o, C, a, p, \text{enqueue}(p', q))}$$

(ASYNC-CALL)
$$\frac{o' = [\![e]\!]^{\varepsilon}_{(a \circ l),t} \quad \overline{v} = [\![\overline{e}]\!]^{\varepsilon}_{(a \circ l),t} \quad \text{fresh}(f) \quad a(\text{mycomp}) = n \quad r > 0}{ob(o, C, a, \{l|x := e!m(\overline{e}); s\}, q)\ dc(n, r, u)\ cl(t) \\ \to ob(o, C, a, \{l|x := f; s\}, q)\ dc(n, r-1, u)\ cl(t)\ invoc(o', f, m, \overline{v})\ fut(f, \bot)}$$

(RETURN)
$$\frac{v = [\![e]\!]^{\varepsilon}_{(a \circ l),t} \quad r > 0 \quad a(\text{mycomp}) = n \quad l(\text{destiny}) = f}{ob(o, C, a, \{l|\text{return } e; s\}, q)\ cl(t)\ fut(f, \bot)\ dc(n, r, u) \\ \to ob(o, C, a, \{l|s\}, q)\ cl(t)\ fut(f, v)\ dc(n, r-1, u)}$$

(AWAIT1)
$$\frac{[\![g]\!]^{cn}_{(a \circ l),t}}{\{ob(o, C, a, \{l|\text{await } g; s\}, q)\ cl(t)\ cn\} \\ \to \{ob(o, C, a, \{l|s\}, q)\ cl(t)\ cn\}}$$

(AWAIT2)
$$\frac{\neg[\![g]\!]^{cn}_{(a \circ l),t}}{\{ob(o, C, a, \{l|\text{await } g; s\}, q)\ cl(t)\ cn\} \\ \to \{ob(o, C, a, \{l|\text{release}; \text{await } g; s\}, q)\ cl(t)\ cn\}}$$

(NEW-OBJECT)
$$\frac{\text{fresh}(o') \quad a(\text{mycomp}) = n \quad r > 0 \quad p = \text{init}(B) \quad a' = \text{atts}(B, \overline{v}, o', n)}{ob(o, C, a, \{l|x := \text{new } B(\overline{e}); s\}, q)\ cl(t)\ dc(n, r, u) \\ \to ob(o, C, a, \{l|x := o'; s\}, q)\ cl(t)\ ob(o', B, a', p, \emptyset)\ dc(n, r-1, u)}$$

(PROGRESS)
$$\frac{\text{canAdv}(cn, t)}{\{cn\ cl(t)\} \\ \to \{\text{Adv}(cn)\ cl(t+1)\}}$$

(RELEASE)
$$ob(o, C, a, \{l|\text{release}; s\}, q) \\ \to ob(o, C, a, \text{idle}, \\ \quad \text{enqueue}(\{l|s\}, q))$$

(ACTIVATE)
$$\frac{p = \text{select}(q, a, cn, t)}{\{ob(o, C, a, \text{idle}, q)\ cl(t)\ cn\} \\ \to \{ob(o, C, a, p, q \backslash p)\ cl(t)\ cn\}}$$

Fig. 4. Semantics for timed Creol.

tables. Assume given functions $\text{bind}(o, f, m, \overline{v}, C)$ which returns a process resulting from the method activation of m in a class C with actual parameters $\overline{v}$, callee o and associated future f; $\text{init}(C)$ which returns a process initializing instances of class C; and $\text{atts}(C, \overline{v}, o, n)$ which returns the initial state of an instance of

class C with class parameters $\overline{v}$, identity o, and deployment component n. The predicate fresh(n) asserts that a name n is globally unique (where n may be an identifier for an object or a future). Let 'idle' denote any process $\{l|s\}$ where s is an empty statement list. Finally, we define different assignment rules for side effect free expressions (*assign1* and *assign2*), object creation (*new-object*), method calls (*async-call*), and future dereferencing (*read-fut*).

Rule *skip* consumes a **`skip`** in the active process and a resource in the object's deployment component. Here and in the sequel, the variable s will match any (possibly empty) statement list, the object's deployment component is given by a(mycomp), and $r > 0$ asserts that the deployment component has available resources. Rules *assign1* and *assign2* assign the value of expression e to a variable x in the local variables l or in the fields a, respectively, consuming a resource in the deployment component of the object. Rules *cond1* and *cond2* branch the execution depending on the value obtained by evaluating the expression e. (We omit the rule for while, which unfolds the while loop using an if-expression.)

Process Suspension and Activation. Three operations are used to manipulate a process queue q: enqueue(p, q) adds a process p to q, $q \setminus p$ removes the process p from q, and select(q, a, cn, t) selects a process from q (if q is empty, this is the idle process or no process is *ready* [19]). The actual definitions of these operations are left undefined; different definitions correspond to different scheduling policies for processes. Let $\emptyset$ denote the empty queue. Rule *release* suspends the active process to the process queue, leaving the active process idle. Rule *await1* consumes the await statement if the guard evaluates to true in the current state of the object, 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 [19].

Communication and Object Creation. Rule *async-call* sends an invocation message to o' with the unique identity f (by the condition fresh(f)) of a new future, the method name m, and actual parameters $\overline{v}$. Note that the return value of the new future f is undefined (i.e., $\bot$). This operation consumes a resource. Rule *bind-mtd* consumes an invocation method and places the process corresponding to the method activation in the process queue of the callee. Note that a reserved local variable 'destiny' is used to store the identity of the future associated with the call. Rule *return* places the return value into the call's associated future. This operation consumes a resource. Rule *read-fut* dereferences the future f in the case where $v \neq \bot$. This operation consumes a resource. Note that if this attribute is $\bot$ the reduction in this object is *blocked*. Finally, *new-object* creates a new object with a unique identifier o'. The object's fields are given default values by atts$(B, \overline{v}, o', n)$, extended with the actual values $\overline{v}$ for the class parameters, o' for this, and n for mycomp. In order to instantiate the remaining attributes, the process p is loaded (we assume that this process reduces to idle if init(B) is unspecified in the class definition, 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

$$
\begin{array}{lr}
\mathrm{canAdv}(cn', t) = \mathit{true} & \textit{cn' contains no objects or messages} \\
\mathrm{canAdv}(\mathit{msg}\ cn, t) = \mathit{false} & \textit{messages are instantaneous} \\
\mathrm{canAdv}(\mathit{ob}(o, C, a, p, q)\ \mathit{dc}(n, 0, u)\ cn, t) & \textit{no more resources} \\
\quad = \mathrm{canAdv}(\mathit{dc}(n, 0, u)\ cn, t) \wedge a(\mathrm{mycomp}) == n & \\
\mathrm{canAdv}(\mathit{ob}(o, C, a, \{l | x := f.\mathrm{get}; s\}, q)\ \mathit{fut}(f, \bot)\ cn, t) & \textit{o is blocked and} \\
\quad = \mathrm{canAdv}(\mathit{fut}(f, \bot)\ cn, t) & \textit{no value is available} \\
\mathrm{canAdv}(\mathit{ob}(o, C, a, \mathrm{idle}, q)\ cn, t) & \textit{no ready processes} \\
\quad = \mathrm{canAdv}(cn, t) \wedge \mathrm{select}(q, a, cn, t) == \mathrm{idle} & \\
\mathrm{canAdv}(\mathit{ob}(o, C, a, p, q)\ cn, t) = \mathit{false} & \textit{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{Adv}(\mathit{dc}(n, r, u)\ cn) & = \mathit{dc}(n, u, u)\ \mathrm{Adv}(cn) & \\
\mathrm{Adv}(cn) & = cn & \textit{otherwise}
\end{array}
$$

Fig. 5. Auxiliary functions controlling time advance. Here, *msg* denotes a message and cn' ranges over message- and object-free configurations.

object creation in a named deployment component differs from *new-object* only on this point, and is omitted from the presentation.

Advancing Time. We capture a *run-to-completion* semantics for concurrent execution within the resource bounds of deployment components: all objects must finish their actions as soon as possible if resources are available. In order to reflect timed concurrent execution with an interleaving semantics, time cannot advance freely. Time advance is regulated by a predicate `canAdv`, ranging over configurations and time (see Fig. 5), which can be explained as follows:

- For simplicity, we assume that invocation messages do not take time. Therefore, time may *not* advance when a message is on its way.
- Time may *not* advance if any deployment component has remaining resources and any of the component's objects o may perform an action. There are three cases:
  1. the active process in o is blocked on a value that 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.
- If all deployment components have run out of resources or none of their objects can proceed, then time can advance.

If there can be no activity in any object and no messages are in transit, then time may advance. (A timed model of communication may be obtained by introducing explicit delays in the model, associated with specific method calls, see Sect. 4.) Time advance is captured by the rewrite rule *progress* in Fig. 4, which updates the global clock. Once time has advanced, the deployment components get their resources refreshed for the next cycle of computation. This is done by an auxiliary function `Adv` defined in Fig. 5, which updates a configuration by resetting the free resources of each deployment component to the specified limit. Observe that for simplicity we here advance time with a single unit. It is of course straightforward to add an attribute *delta* which allows larger increments. However, this may lead to incompleteness for search in the timed models [22].

```
class SyncClient(Agent a,Nat c){
  Void run {
    Time t := now;
    Session s := a.getsession();
    Bool result := s.order();
    await now >= t + c; !run(); } }
```

```
class PeriodicClient(Agent a,Nat c){
  Void run {
    Time t := now;
    Session s := a.getsession();
    Fut(Bool) rc:= s!order();
    await now >= t + c;
    !run();
    await rc?; Bool r := rc.get; } }
```

```
component shop(10)
Void main() {
   Database db := new Database(5, 10) in shop;
   Agent a := new Agent(db, {}) in shop;
   PeriodicClient c := new PeriodicClient(a, 5); }
```

Fig. 6. Deployment environment and client models of the web shop example.

6 Simulating and Testing the Example

The operational semantics presented in Section 5 can be directly represented in rewriting logic [21], which allows models to be analyzed using the rewrite tool Maude [10]. 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 in order to simulate and test Creol models. The web shop example of Section 4 is now extended by specifying a deployment component and an environment in order to obtain testing and simulation results. Figure 6 shows how the web shop may be deployed: a *deployment component* `shop` is declared with 10 resources available for objects executing inside `shop`. The initial system state is given by the `main` method, which creates a single database, with 5 and 10 as its minimum and maximum time for orders, an `Agent` instance, and (in this example) one client outside of `shop`. The classes `SyncClient` and `PeriodicClient` model customers of the shop. `PeriodicClient` initiates a session and periodically calls `order` every `c` time intervals; `SyncClient` sends an `order` `c` time intervals after the last call returned.

Figure 7 displays the results of two sets of simulation runs over 100 clock cycles. For synchronous clients, 10 to 50 clients and 10 to 50 resources on the `shop` deployment unit were used. Starting with 20 clients, the number of requests goes up linearly with the number of resources, indicating that the system is running at full capacity. Moreover, the number of successful requests decreases somewhat with increasing clients since communication costs also increase. For the periodic case, the system gets overloaded much more quickly since clients will have several pending requests; hence, only 2 to 10 periodic clients were simulated. It can be seen that the system becomes completely unresponsive quickly when flooded with requests.

Testing Timed Observable Behavior. In software testing, a formal model can be used both for test case generation and as a test oracle to judge test outcomes [17]. For example, test case generation from formal models of communication protocols can ensure that all possible sequences of interactions specified

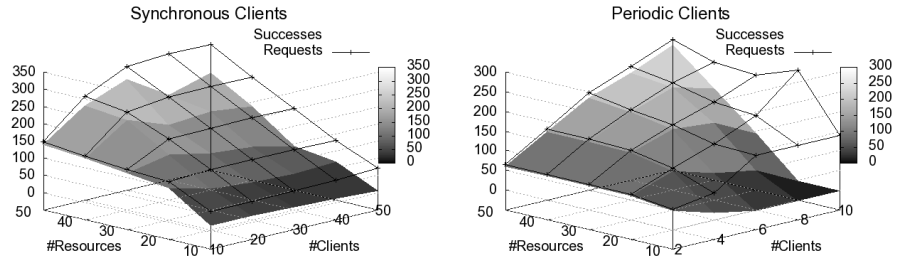

Fig. 7. Number of total and successful requests, depending on the number of clients and resources, for synchronous (left) and periodic (right) clients.

by the protocol are actually exercised while testing a real system. Using formal models for testing is most widely used in functionality testing (as opposed to e.g. load testing, where stability and timing performance of the system under test is evaluated), but the approaches from that area are applicable to formally specifying and testing timing behavior of software systems as well [16].

In this paper, we model and investigate the effects of specific deployment component configurations on the timing behavior of timed software models. The *test purpose* in this scenario is to reach a conclusion on whether redeployment on a different configuration leads to an observable difference in timing behavior. Both *model* and *system under test* are Creol models of the same system, but running under different deployment configurations. In our example, the client object(s) model the expected usage scenario; results about test success or failure are relative to the expected usage. As *conformance relation* we use trace equivalence. This simple relation is sufficient since model and system under test have the same internal structure, hence we do not need to test for input enabledness, invalid responses etc. In our case, traces are sequences of communication events, i.e. method invocations and responses annotated with the time of occurrence, which are recorded on both the model and the system under test and then compared after the fact (off-line testing).

Running the model with five `SyncClients` (see Fig. 6) but with unlimited resources in the component results in a trace $\langle 10, t\rangle$, $\langle 15, t\rangle$, $\langle 20, t\rangle$, ... (where each tuple contains $\langle$*response time, success*$\rangle$). Deploying with 50 resources results in the same trace, whereas running with 20 resources results in a trace $\langle 12, t\rangle$, $\langle 17, t\rangle$, $\langle 22, t\rangle$, ... If the model and system under test have identical untimed behavior, we conclude that a system without resource limits and a deployment component with 50 resources behave equivalently under the assumed workload, whereas deploying with 20 resources will lead to observably different behavior.

7 Related Work

The concurrency model provided by concurrent objects and Actor-based computation, in which software units with encapsulated processors communicate

asynchronously, is increasingly attracting attention due to its intuitive and compositional nature (e.g., [1, 2, 4, 7, 11, 14, 28]). A distinguishing feature of Creol is the cooperative scheduling between asynchronously called methods [19], which allows active and reactive behavior to be combined within objects as well as compositional verification of partial correctness properties [2, 11]. Creol's model of cooperative scheduling has recently been generalized to concurrent object groups in Java [26] by restricting to a single activity within the group. Our paper generalizes concurrent object groups to a resource-constrained deployment components, in which the allowed activity per time interval is parametric in concurrent resources, using a time version of Creol [6]. This allows us to abstractly model the effect of deploying concurrent object groups on deployment components with different amounts of processing capacity. A companion paper considers deployment components and resources as first-class citizens of the language [20], which allows load balancing strategies to be modeled in Creol.

Techniques and methodologies for predictions or analysis of non-functional properties are based on either *measurement* and *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 [12]. A survey of model-based performance analysis techniques is given in [5]. Formal systems using process algebra, Petri Nets, game theory, and timed automata (e.g., [8, 9, 13, 15]) have been applied in the embedded software domain, but also to the schedulability of processes in concurrent objects [18]. The latter 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 [23] informally define the Core Scenario Model (CSM) to solve questions that arise in performance model building. CSM has a notion of resource context, which reflects an operation's set of resources. CSM aims to bridge the gap between UML and techniques to generate performance models [5]. Closer to our work is M. Verhoef's extension of VDM++ for simulation of embedded real-time systems [27], in which architectures are explicitly modelled using CPUs and buses, and resources are bound to the CPUs. However, the underlying object models and operational semantics differ. VDM++ has multi-thread concurrency, preemptive scheduling, and a strict separation of synchronous method calls and asynchronous signals, in contrast to our work with concurrent objects, cooperative scheduling, and caller decided synchronization. In contrast to our fairly succinct semantics, the extension to VDM++ is embedded into VDM++ itself and defined in terms of 100 pages of VDM++ specifications [27].

8 Conclusions and Future Work

This paper proposes a formal model of concurrent processing resources for the deployment of timed object-oriented components. We extend Creol with a notion

of deployment component which is parametric in its concurrent resources per time interval and formalize the operational semantics of object execution on deployment components. Based on this formalization, we use the rewriting logic tool Maude to validate resource requirements that are needed to maintain the timed behavior of concurrent objects deployed with restricted resources.

The proposed model of deployment components is simple and flexible. The time granularity is defined implicitly by the use of time outs, allowing several statements to be executed in one time interval. In contrast, the execution cost of basic statements is fixed (abstracting from the evaluation of expressions). With a single resource, at most one basic statement can be executed in a deployment component within a time interval. With multiple resources, all resources are used within the time interval if possible. This proposed resource model does not describe component scheduling policies, and abstracts from the cost of processor swapping and internal control flow. The model may be refined by associating deadlines to method calls and by defining explicit scheduling policies [18]; by computing worst-case costs for the evaluation of expressions [3] and internal control flow; by reconfiguration in terms of object mobility; as well as stronger analysis methods such as, e.g., static analysis and bisimulation techniques.

The abstract notion of resource proposed in this paper reflects computational limitations of concurrent or interleaved activity. Combined with a flexible time model, this resource model can express interesting non-functional system properties, as illustrated by the example. Whereas most work on performance analysis assumes a fixed underlying architecture, approaches such as the one presented in this paper address a need in formal methods, in order to capture models which vary over underlying architectures, for example in software product lines.

References

1. G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems.* The MIT Press, 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, G. Puebla, and D. Zanardini. COSTA: Design and implementation of a cost and termination analyzer for Java bytecode. In *Proc. FMCO'07*, *LNCS* 5382, pages 113–132. Springer, 2007.
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 Transactions on Software Engineering*, 30(5):295–310, 2004.
6. J. Bjørk, E. B. Johnsen, O. Owe, and R. Schlatte. Lightweight time modeling in Timed Creol. *Electronic Proc. in Theoretical Computer Science*, 36:67–81, 2010. *Proc. Intl. Workshop on Rewriting Techniques for Real-Time Systems (RTRTS'10).*
7. D. Caromel and L. Henrio. *A Theory of Distributed Object.* Springer, 2005.
8. A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Resource interfaces. In *Proc. EMSOFT*, *LNCS* 2855, pages 117–133. Springer, 2003.

9. X. Chen, H. Hsieh, and F. Balarin. Verification approach of Metropolis design framework for embedded systems. *Intl. J. of Parallel Prog.*, 34(1):3–27, 2006.
10. 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.
11. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Proc. ESOP'07*, *LNCS* 4421, pages 316–330. Springer, Mar. 2007.
12. I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *Proc. ICSE'09*, pages 111–121. IEEE, 2009.
13. E. Fersman, P. Krcál, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Inf. and Comp.*, 205(8):1149–1172, 2007.
14. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.
15. M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM TOPLAS* 24(5):566–591, 2002.
16. A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Testing real-time systems using UPPAAL. In *Formal Methods and Testing*, *LNCS* 4949, pages 77–117. Springer, 2008.
17. R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. A. Vilkomir, M. R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2), 2009.
18. 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.
19. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
20. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Dynamic resource reallocation between deployment components. In *Proc. ICFEM*, *LNCS* 6447, pages 646–661. Springer, Nov. 2010. To appear.
21. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
22. P. C. Ölveczky and J. Meseguer. Abstraction and completeness for Real-Time Maude. In *Proc. WRLA*, *ENTCS* 176: 5–27. Elsevier, 2007.
23. 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.
24. G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.
25. K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques.* Springer, 2005.
26. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *Proc. ECOOP*, *LNCS* 6183, pages 275–299. Springer, 2010.
27. M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In *Proc. of the 14th Intl. Symposium on Formal Methods (FM'06)*, *LNCS* 4085, pages 147–162. Springer, 2006.
28. A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. OOPSLA'05*, pages 439–453. ACM Press, 2005.
29. S. M. Yacoub. Performance analysis of component-based applications. In *Proc. SPLC*, *LNCS* 2379, pages 299–315. Springer, 2002.