Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 266 (2010) 63–75
www.elsevier.com/locate/entcs

Resource Modeling for Timed Creol Models*

Rudolf Schlatte[a,b,1] Bernhard Aichernig[a,b,2]
Andreas Griesmayer[b,3] Marcel Kyas[c,4]

[a] *Institute for Software Technology, Graz University of Technology, Austria*
[b] *International Institute for Software Technology, United Nations University, Macao S.A.R., China*
[c] *Department of Computer Science, Freie Universität Berlin, Germany*

Abstract

This paper describes the semantics of a timed, resource-constrained extension of the Creol modeling language. Creol is an object-oriented modeling language with a design that is suited for modeling distributed systems. However, the computation model of Creol assumes infinite memory and infinite parallelism within an object. This paper describes a way to extend Creol with a notion of *resource constraints* and a way to quantitatively assess the effects of introducing resource constraints on a given model. We discuss possible semantics of message delivery under resource constraints, their implementation and their impact on the model. The method is illustrated with a case study modeling a biomedical sensor network.

Keywords: Modeling, Resource Constraints, Object Orientation, Embedded Systems, Creol

1 Introduction

Modeling is an important activity in the design phase of a software project. A formal model can be used to answer questions about a system's functionality, behavior and properties during the specification and implementation phase. By nature, a model focuses on specific aspects of the system-to-be; for reasons of simplicity and clarity, specific aspects of the eventual implementation are abstracted away in the model. Among the implementation details that are abstracted away are often processor, bandwidth and memory requirements of components of the system. However, these aspects are of high importance for example in embedded systems. As a consequence, having modeling languages that consider these aspects is desirable.

* This research was carried out as part of the EU FP6 project *Credo*: Modeling and analysis of evolutionary structures for distributed services (IST-33826).

[1] Email: rschlatte@iist.unu.edu
[2] Email: aichernig@ist.tugraz.at
[3] Email: agriesma@iist.unu.edu
[4] Email: marcel.kyas@fu-berlin.de

doi:10.1016/j.entcs.2010.08.049

This paper describes an enhancement of the modeling language Creol [7] for supporting the modeling of resource constraints, specifically restrictions on parallelism, call stack depth or memory consumption. Creol is an object-oriented modeling language with asynchronous communication primitives. A model in Creol consists of classes and objects; objects can have active and reactive behavior. Conceptually, each Creol object has its own processor and handles concurrency independently of other objects. Objects communicate solely via messages and control flow never leaves an object; instead, when a process issues a method call, the receiving object creates a new process that the calling process can synchronize with. These features of Creol make it very suitable for modeling systems of independent, cooperating agents, such as wireless sensor networks.

For modeling resource constraints, we assign each method a (possibly zero) amount of needed resources, and each class an amount of available resources. At runtime, method invocations take the needed amount of resources from the object's available resources. This abstract concept of resources can be used to restrict the amount of parallelism within an object (by giving each method a cost of 1 and the class a number of resources corresponding to the number of allowed concurrent threads), or to model a finite amount of memory or processing power to be claimed by running threads.

Various behaviors can be implemented when encountering lack of resources: delaying message delivery, blocking the sender or dropping the message. We give examples for these behaviors, show how to implement them in our rewrite rule-based system and discuss advantages and disadvantages as pertains to modeling.

The rest of this paper is structured as follows: Section 2 gives an overview of the features of the Creol language pertaining the modeling of distributed systems, Section 3 explains how the Creol semantics and interpreter were altered to allow modeling of resource constraints. Section 4 presents an extended example and some experiences gained from introducing resource constraints in a larger case study. Section 5 gives an overview of related work in this area, and Section 6 concludes the paper.

2 The Creol Language

Creol is an object-oriented modeling language for distributed and concurrent systems, with an operational semantics given in rewriting logic [11] that is executable on the Maude [2] rewriting engine. Creol is especially suited for modeling loosely-coupled, active and reactive communicating systems. This section gives an overview of the features of Creol that are important to understand the models presented in the paper; for a detailed description of Creol's features (data types, interfaces and co-interfaces, inheritance etc.) see for example [7].

A Creol model is composed of *classes* that implement *interfaces*; instance variables of classes are either a primitive type (numbers, string, Boolean), lists, sets, maps and tuples of types, or a reference to an interface [5]. Classes contain methods,

[5] Interfaces are elided from the code samples presented in this paper for reasons of brevity.

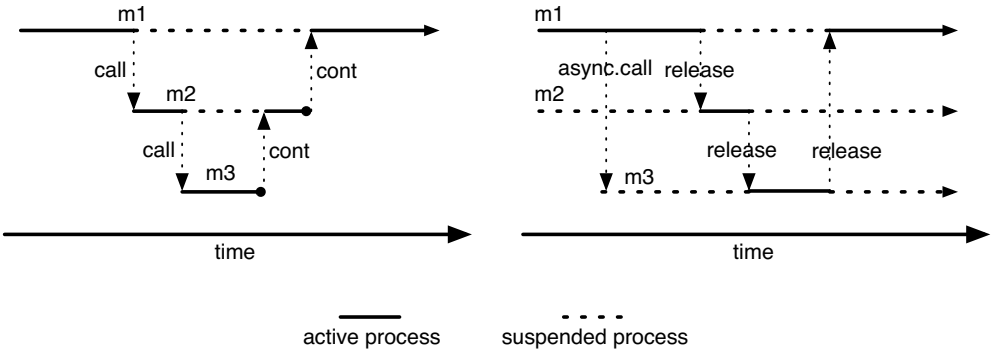

Fig. 1. Modeling function calls via Creol synchronous method calls (left) and parallelism via Creol asynchronous calls and release points (right).

methods can have multiple arguments and return values.

Creol contains the "standard" features of an object-oriented imperative programming language: implementation and interface inheritance; strings and various numeric primitive datatypes; lists, sets, maps and tuples; assignment, conditional and looping statements. All members of an object are private to the object and can only be accessed from another object via method calls.

The execution model of Creol has been derived from the *actor* model and uses *cooperative multi programming* for coordination. For method calls, the caller can choose whether to block and wait for a return value (synchronous call), to synchronize with the callee later (asynchronous call), or not to synchronize at all by ignoring the return value. In the second case a *future variable* [6,4] is used by the caller to poll the method invocation's termination and to obtain the return values, blocking if necessary. Additionally, the identity of the caller is available in most method bodies through the variable `caller`, which allows for call backs. Thus, Creol provides and allows to combine different styles of object interaction.

Each Creol object is executing its own thread of control, interleaving active and reactive behavior. All method calls (including self-calls) create a new *process* within the called object. At most one process is active for each object and has exclusive access to the objects attributes. Special statements allow to change between processes, which manifest the cooperative multiprogramming style. A process need not distinguish whether it was created from a synchronous or asynchronous call.

These features together allow Creol to model both concurrent and single-threaded control flows in a uniform way. Figure 1 illustrates the flow of control between 3 processes within one object in the synchronous and asynchronous case. Note that `m3` is called asynchronously in one case and synchronously in the other – a method has no way of determining whether it was called synchronously or asynchronously.

The syntax for method calls is as follows:

Synchronous call: `object.name(in-parameters;out-parameters)`

Asynchronous call: `future-variable!object.name(in-parameters)`

After a synchronous call returns, the output parameters contain the return value(s)

of the method. For an asynchronous call, the return value(s) are stored in the caller's future variable upon completion. If an asynchronous call does not return any value, there is an abbreviated call syntax that is for example used in line 14 of Figure 8.

Figure 1 also illustrates synchronization between processes within an object. Each Creol object conceptually contains its own processor and no data is shared between objects; hence, there are no inter-object process synchronization issues. Processes within an object use cooperative scheduling. The Creol statement `release` suspends the current process unconditionally, the statement `await` *`condition`* suspends the process until the condition evaluates to *true*. In Figure 1 (right side), the process `m1` dynamically creates a new process `m3`. Since it is an asynchronous call, `m3` does not begin running immediately; rather, `m1` continues after the call until it reaches a release point, at which point the object schedules another thread from its thread pool.

As a special case, a synchronous self-call unconditionally transfers control to the new process created by the self-call, and arranges for an unconditional transfer of control back to the calling process upon completion (Figure 1, left-hand side). Reading from a future variable *blocks* the whole object until the values arrive; hence, a synchronous method call in Creol can be seen as just an asynchronous call plus an immediate blocking read of the associated future variable. For synchronization without whole-object blocking, an `await` statement is used that *suspends* the process but allows other processes in the object to run.

2.1 Timed Creol

The base Creol language does not model time or progress, but recently Kyas and Johnsen designed a Creol extension for real-time constraints [8]. The extension is common and simple: the value of a global clock is accessible to all objects through the expression `now`, which behaves like a read-only global variable of type `Time`. Values of type `Time` can be stored in variables and compared with other `Time` values. There is no absolute notion of time; progress can be expressed by adding `Duration` values to observations to obtain other values of type `Time`. An advantage of this design is that specifications in timed Creol are shift invariant; i.e. properties involving time hold no matter at which point in (absolute) time the evaluation happens. Indeed, this time extension is inspired by the time model of the Ada programming language [13, Appendix D.8].

In contrast to the Ada programming language, Creol focuses on modeling and not on implementations. As such, a Creol model is a logical description and we ignore certain aspects like preemption due to interrupts in this paper. Interrupt handlers may be modeled by methods of singleton objects, which are invoked as a result of an interrupt signal. Thus, the method described in this paper allows to model the effect of interrupts without the need of taking the actual machine into account.

Expressing a time invariant in timed Creol looks as follows:

```
1  var t: Time := now;
2  SL
```

```
3  await now >= t + 10;
```

The `await` statement in line 3 guarantees that after evaluating the statement list *SL*, at least 10 units of time pass before the process can continue. (If the effects of *SL* should be *visible* only after 10 time units, then the `await` statement should be placed before *SL*.)

The semantics of Creol allow a process to be suspended indefinitely inside an `await` statement. To ensure forward progress of the system, timed Creol introduces the `posit` statement:

```
1  var t: Time := now;
2  SL
3  posit now <= t + 10;
```

Here, line 3 guarantees that evaluating *SL* takes *at most* 10 time units. A `posit` statement expresses a global property of the system and may result in a system that has no behavior at all; all `posit` statements are proof obligations on the statement level. For details and exact semantics of timed Creol, we once again refer to [8].

3 Implementing Resource Constraints

The execution semantics of Creol assume that an object can execute an arbitrary number of processes. Especially for small embedded systems, this assumption does not hold. It is therefore desirable to be able to model the operating constraints of real systems in Creol. This section presents the implementation approach that was taken to adapt the semantics and execution engine of Creol to deal with resource constraints.

Creol models can be animated on the Maude rewrite engine. In Maude, the execution state of the model is represented as a *state* containing terms representing Creol objects, classes and pending method invocations. The representation of an object `O` of class `C` is

```
< O : C | Att: AL, Pr: { BL | SL}, PrQ: PL >
```

`O`, `C`, `AL`, `BL`, `SL`, and `PL` are typed variables, where object `O` is an instance of class `C` with instance variables `AL` and an active process that consists of local variable bindings `BL` and a list of statements `SL`. The list of pending processes is represented by `PL`. Classes and method invocations have a similar Maude notation.

To implement resource-constrained objects, the notation for classes was updated to contain an attribute `RLimit`:

```
< C : Class | Inh: I, Param: P, Att: S, Mtds: M, RLimit: N >
```

The rest of the attributes are standard and are used for inheritance (`Inh`), constructor parameters (`Param`), Attributes (`Att`) and methods (`Mtds`). The new `RLimit` attribute tells how much memory / processing capacity objects of this class can supply to their processes.

Similarly, an additional method definition was introduced that specifies, in addition to the method name `M`, parameters `P`, local variables `A` and code `C`, how much

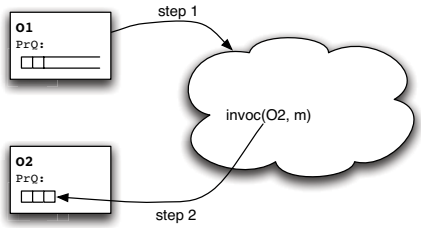

Fig. 2. Method calls in Creol. For restricted objects, the caller can be blocked (delay in step 1), the call can be delayed infinitely "in the cloud" (delay in step 2) or the call can be dropped (only step 1 happens).

resources a method needs when called:

```
< M : Method | Param: P, Att: A, Code: C, RNeed: N >
```

With $limit(O)$ the resource limit of an object O (as determined by its class), $P(O)$ the object's set of active processes, and $cost(P)$ the cost of a process or 0 if the process has no resource cost, the following invariant needs to hold for all constrained objects:

$$\sum_{p \in P(O)} cost(p) \leq limit(O)$$

The *dynamic semantics* of Creol is given as a set of Maude *rewrite rules* operating on parts of this state. When a rule of the form `<C1> => <C2>` is executed, the part of the state matching `<C1>` is replaced by `<C2>`. For example, the rewrite rule for the `skip` statement of Creol looks as follows:

```
  < O : C | Att: AL, Pr: { BL | skip ; SL}, PrQ: PL >
=>
  < O : C | Att: AL, Pr: { BL | SL}, PrQ: PL >
```

The left-hand side of this rule matches any object with an active process having `skip` as its next statement. Such an object is replaced with an object identical in every way except that the `skip` statement is removed and the remaining statement list `SL` left for execution. Rules for other statements follow the same pattern, but typically have more effect, such as rebinding variables, creating, destroying and scheduling processes or creating new objects.

3.1 Possible Semantics of Message Delivery

Delivering a message to an unconstrained object, or to an object that has enough free resources, always succeeds. A new process is created and will be scheduled by the object in due time. However, when the object cannot accept the message and create a process, various behaviors are possible:

(i) The message delivery can be *delayed* until the callee can accept it, without the caller being blocked.

(ii) The caller can be *blocked* until the callee can accept the message.

(iii) The message can be *dropped*; if the callee cannot accept it, the message is lost.

Figure 3 shows a simplified version of the rule for creating a new process in an instance of a restricted class. A new process is created only when adding it to the

```
  < C : Class | Inh: I, Param: AL, Att: S1, Mtds: MS, RLimit: N >
  < O : C | Att: S, Pr: P, PrQ: PL >
  invoc(O, m, param)
=>
  < C : Class | Inh: I, Param: AL, Att: S1, Mtds: MS, RLimit: N >
  < O : C | Att: S, Pr: P, PrQ: (PL, createProcess(m, param) >
if nResources(P, PL, m) < N and idleOrSelfcall(P)
```

Fig. 3. The (slightly simplified) conditional rewrite rule for creating a new process `m` in a constrained object `O`. A process is created if there are enough resources available and if the object can accept a method invocation (i.e., is idle or its current process issued the call).

```
  < O' : C' | Att: S', Pr: { BL | call(O, m, param) ; SL' }, PrQ: PL' >
  < C : Class | Inh: I, Param: AL, Att: S1, Mtds: MS, RLimit: N >
  < O : C | Att: S, Pr: P, PrQ: PL >
=>
  < O' : C' | Att: S', Pr: { BL | SL' }, PrQ: PL' >
  < C : Class | Inh: I, Param: AL, Att: S1, Mtds: MS, RLimit: N >
  < O : C | Att: S, Pr: P, PrQ: PL >
  invoc(O, m, param)
if nResources(P, PL, m) < N
```

Fig. 4. The (slightly simplified) conditional rewrite rule for invoking a method of object `O` from object `O1`. Evaluation of the rule is delayed until `O` is in a position to create a process `m`.

```
  < C : Class | Inh: I, Param: AL, Att: S1, Mtds: MS, RLimit: N >
  < O : C | Att: S, Pr: P, PrQ: PL >
  invoc(O, m, param)
=>
  < C : Class | Inh: I, Param: AL, Att: S1, Mtds: MS, RLimit: N >
  < O : C | Att: S, Pr: P, PrQ: PL >
[owise]
```

Fig. 5. The rule implementing message loss, working in concert with the process creation rule of Figure 3. The left-hand sides of both rules are identical, but this rule only applies if no other rule matches the left-hand side (via Maude's `[owise]` attribute).

object's process queue does not exceed the available resources. This rule implements delayed message delivery.

To implement a delay of the message sender, another rule has to be added that is shown in Figure 4. This rule blocks the sender until the receiver can accept the message.

To implement message loss in a Creol model, yet another rule has to be added; a simplified version is shown in Figure 5. This rule works in concert with the process creation rule of Figure 3; the `[owise]` Maude attribute guarantees that the invocation is only dropped if the process cannot be created.

All of the possible behaviors of message delivery are meaningful in some context. Dropping messages comes closest to the behavior of a system of loosely-coupled components, such as a network of wireless sensors or the datagram level in a TCP/IP network. On the other hand, this model behavior requires extensive changes of the Creol model, compared to an unconstrained model, that are not necessary for the other two possible behaviors. Specifically, every method call that expects a return value has to be implemented with a timeout and an error path:

```
1  var t: Time := now; var l: Label[Int]; var result: Int;
2  var success: Bool;
3
4  l!o.m();
5  await l?; l?(result); success := true
6  []
7  await now >= t + 10; success := false;
```

```
8  if (success)
9    ... // use 'result' here
10 else
11   ... // recover from timeout here
12 end
```

Delaying message delivery and suspending the sending process both model reliable message delivery. Delivery delay models a "smart" network, or an application-transparent buffer-and-resend layer of the sender. Blocking the sender models a tightly-coupled system, probably implemented on a single machine, where querying the receiver's state does not incur sending a message.

3.2 Modeling with Resource Constraints

Resource constraints, as described in this section, can be used to model and validate a variety of behavior:

Restricted parallelism : To model an object that has restricted parallelism, assign each method a cost of 1 and the class a limit corresponding to the maximum number of running processes.

Recursion depth : To validate that a model run does not exceed a certain depth of self-calls, assign all involved methods a cost of 1 and the class a limit corresponding to the maximum recursion depth. If the methods contain release points, outside calls (that can be used to model interrupts) also factor in the maximum depth.

Memory consumption : Assign the class the amount of memory that is available, and each method its memory cost. This assumes that an object models a physical processor, for example a sensor node.

4 Case Study

A wireless sensor network is a wireless network consisting of spatially distributed autonomous devices using sensors to cooperatively monitor physical, environmental or biomedical conditions, such as temperature, sound, vibration, pressure, motion, pollutants or biomedical signals at different locations. Sensor networks have been an active area of research for more than a decade, with applications in e.g. medicine, military, oil and gas, and smart buildings. A biomedical sensor network (BSN) consists of small, low-power and multi-functional sensor nodes that are equipped with biomedical sensors, a processing unit and a wireless communication device. Each sensor node has the abilities of sensing, computing and short-range wireless communication. Due to BSNs' reliability, self-organization, flexibility, and ease of deployment, the applications of BSNs in medical care are growing fast.

The case study presented in this section was developed as part of the Credo project [3]. It models a sensor network consisting of a set of *sensor nodes* and one *sink node*. Sensor nodes are actively monitoring their environment and sending out their measurements. In addition, sensor nodes have the task of routing messages from neighboring sensor nodes towards the sink node. The sink node, typically connected to back-end processing, receives data from all nodes but does not create

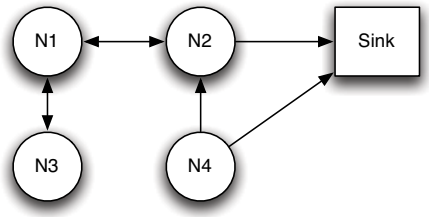

Fig. 6. A biomedical sensor network: 4 sensor nodes and one sink node. Note that the connections are not necessarily symmetric: Node 4 transmits with more power and can therefore reach Node 2, but Node 2 cannot reach Node 4.

```
1 class SensorNode(id: Int, network: Network)
2 begin
3   var received: List[[Int,Int]] := nil
4   var outgoing: List[[Int,Int]] := nil
5   var noSensings: Int := 3  // No. of sensings to do
6   var seqNo: Int := 0 // Running package seq. no
7
8   op transmit ==
9     !network.broadcast(head(outgoing));
10    outgoing := tail(outgoing)
11  op queue(in data: [Int,Int]) ==
12    outgoing := outgoing |- data
13  op sense ==
14    queue((id,seqNo);); // dummy value
15    seqNo := seqNo + 1
16  op run ==
17     while true do
18       await seqNo < noSensings; sense(;) // read sensor
19       [] // nondeterministic choice
20       await #(outgoing) > 0; transmit(;);
21       release
22     end
23  with Network // receive data from outside
24    op receive(in data: [Int,Int]) ==
25      if ~(data in received) then
26        queue(data;)
27        received := received |- data;
28      end
29 end
```

Fig. 7. Model of a sensor node. The `receive` method, called by the network, implements reactive behavior, the `run` method implements the node's active behavior.

any data itself.

Connectivity is modeled by a *network object*. This object does not correspond to a physical artifact, but represents the topological arrangement of nodes and their connectivity and also models the behavior of broadcasting a message from a node to its neighboring nodes. Figure 6 shows an arrangement of sensor nodes and sink node.

In our model, sensor node objects have active behavior: after creation, they transmit a sequence of measurements and then switch to idle (reactive) behavior, only listening for and retransmitting messages.

Figure 7 shows the model of a sensor node. Its active behavior is implemented by the `run` method starting at line 16. A sensor node has two functions: read sensor values (method `sense`) and send them to neighboring nodes (method `transmit`), and receive and re-send values from other nodes in the network (methods `receive` and again `transmit`). The nondeterministic choice operator (`[]`) in line 19 chooses between reading a sensor value and transmitting a value that can either originate

```
1 class Network
2  pragma Max_resources(1)
3 begin
4  // All nodes in network that have registered and
5  // their connections.
6  var nodesConns: Map[Node, List[Node]] := empty()
7 [...]
8  // Broadcast a message from a node to its neighbors
9  with Node
10   op broadcast(in data: [Int,Int]) pragma Need_resources(1) ==
11    var receivers: List[Node] := get(nodesConns, caller)
12    while ~isempty(receivers) do
13     if head(receivers) /= caller then
14      !head(receivers).receive(data)
15     end
16     receivers := tail(receivers);
17    end
18 end
```

Fig. 8. Model of the network. (Code to initialize the connection map `nodesConns` elided.) Only one `broadcast` method can be called simultaneously because of the specified resource availability.

from the node itself or from the network. The method `receive` models the receiving part of the node's behavior and is called from the `Network` object.

Figure 8 shows the network model. This class does not model a physical object; instead it describes and implements the topology of co-operating `Node` objects; i.e., which other nodes will receive a message broadcast by some node. Line 6 shows the data structure containing the connection map, the method starting in line 10 implements the network's behavior. The pragma statements in lines 2 and 10 restricts objects of this class to have only one concurrent running `broadcast` method.

4.1 Results

The classes `SensorNode` (Figure 7) and `Network` (Figure 8), together with a class `SinkNode` (not shown) implement a simple flooding routing protocol. In the original case study, *network collisions* were not considered – an arbitrary number of nodes were allowed to broadcast data at the same time.

An obvious way to model the incremental-backoff strategy of resending packets on collision is to restrict the network to only one `broadcast` method at a time. This is very straightforward using the presented resource limit framework – simply assign a cost of 1 to the `broadcast` method and a single resource to the class, while delaying message delivery but allowing the sender to continue running (with these semantics, the timeout-and-resend behavior is implicit, allowing the original model's code to stay in place). With these constraints in place, the original model deadlocked.

The cause of the deadlock was identified in line 14. The original model had a synchronous call to the receiving node's `receive` method at that point. This serialized message delivery, forcing the receiving node to finish processing the message (including a recursive call to `Network.broadcast`) before the next node would even receive the message. Converting the synchronous call to an asynchronous call allowed the model to run to completion.

While the functional aspects of the model were correct (all messages arrived at

the sink node during a simulated run), constraining the `Network` class uncovered what is arguably a modeling error – since that class models the behavior of the "air space" between nodes, messages broadcast by one node will reach all of that node's neighbors at the same time. This is modeled via asynchronous calls. Constraining the network to its intended behavior helped uncover this modeling error.

5 Related Work

Modeling bounded computing resources is relevant because micro-controllers expose the programmer to a bounded call depth, either explicitly or because of memory constraints. For example, the PIC family of micro-controllers has an explicit maximal call depth between 2 and 31, depending on the model. Version 2 of TinyOS [9], an operating system for wireless sensor networks, contains `tos-ramsize`, a tool for *static stack depth analysis* calculating worst-case memory usage by summing stack usage at call points for the longest path through the call graph, and adding stack usage of all interrupt handlers. The theory behind this tool is presented in [12]. Being a simple tool, `tos-ramsize` does not handle recursion. McCartney and Sridhar [10] present `stack-estimator`, a similar tool for the TinyThread library. The value of our work is that resource constraints can be expressed already on the modeling level, and that the model can be validated by simulation. Also, we present a unified approach to modeling call stack depth and restricted parallelism in a model.

Foster et al. [5] make a strong case for checking a model under resource constraints via an example deadlocks in a proven-deadlock-free web service deployment. These deadlocks arose because of thread starvation – the proof of deadlock-freedom did not take the maximum number of threads of the underlying implementation into account. In their approach, the underlying BPEL (Business Process Execution Language) web service orchestration and the thread pool of the system that executes the service requests are modeled together as a labeled transition system. Model checking is then used to ascertain deadlock freedom under resource constraints. Another extensive work using automata to model resource consumption is Chakrabarti et al. [1], where interface automata are used to express the behavior and resource consumption of components, and a compositional game approach is used to calculate the behavior and resource consumption of a composition of components.

Our work deals with modeling systems on a lower level of abstraction than using automata models, using Creol [7], an imperative, object-oriented modeling language with asynchronous communication between objects. Similar work was done by Verhoef et al. [14], who use the timed variant of the modeling language VDM++ to model distributed embedded systems. They model processing time, schedulability and bandwidth resources by enriching timed VDM++ with a notion of CPUs, communication buses and asynchronous communication, and loosening the global time model of standard timed VDM++. Creol supports many of the changes necessary for modeling distributed systems in the core language already. Kyas and Johnsen [8] use Creol to model timing aspects of wireless sensors, but do not consider resource constraints of that platform.

6 Conclusion

This paper presented a flexible way of adding resource constraints to a behavioral model written in Creol. These constraints can be used to model restricted parallelism, recursion depth, memory usage or processing resources. Adding resource constraints to an existing Creol model requires only one annotation per class and one annotation per constrained method, except when modeling recovery from message loss, where the error-handling code has to be added. We believe that the approach is easily adaptable for VDM++ and similar modeling languages. The value of our approach lies in the ease in which it can be added to an existing, unconstrained model, and in the way different behaviors of message delivery can be explored using one same model. It should be noted that the results obtained by executing the model are sound but not necessarily complete – while the presence of deadlocks caused by resource constraints can be shown, their absence can only be proven by model-checking, which restricts the size of the model. Nevertheless, experience has shown that the approach can give valuable insight into the behavior of a system that is confronted with limited computing resources.

Acknowledgement

The BSN case study used in this paper was authored by Wolfgang Leister, Xuedong Liang and Bjarte M. Oestvold as part of the EU FP6 project *Credo*.

References

[1] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Resource interfaces. In R. Alur and I. Lee, editors, *Embedded Software*, volume 2855 of *LNCS*, pages 117–133. Springer-Verlag, 2003.

[2] 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.

[3] Credo project website. http://www.cwi.nl/projects/credo/. Last accessed May 25, 2009.

[4] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proceedings of the 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 316–330. Springer-Verlag, Mar. 2007.

[5] H. Foster, W. Emmerich, J. Kramer, J. Magee, D. S. Rosenblum, and S. Uchitel. Model checking service compositions under resource constraints. In I. Crnkovic and A. Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE)*, pages 225–234. ACM, 2007.

[6] R. H. Halstead. MULTILISP: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.

[7] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):39–58, Mar. 2007.

[8] M. Kyas and E. B. Johnsen. A real-time extension of creol for modelling biomedical sensors. In *FMCO 2009: Software Technologies Concertation on Formal Methods for Components and Objects, 20 – 24 October 2008, Sophia-Antipolis, France*. Springer-Verlag, 2009. To be published.

[9] P. Levis and D. Gay. *TinyOS Programming*. Cambridge University Press, 2009.

[10] W. P. McCartney and N. Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In A. T. Campbell, P. Bonnet, and J. S. Heidemann, editors, *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 167–180. ACM, 2006.

[11] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[12] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. *ACM Trans. Embedded Comput. Syst.*, 4(4):751–778, 2005.

[13] S. T. Taft, R. A. Duff, R. L. Brukardt, E. Ploedereder, and P. Leroy, editors. *Ada 2005 Reference Manual: Language and Standard Libraries, International Standard ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1*, volume 4348 of *LNCS*. Springer-Verlag, Berlin/Heidelberg, 2006.

[14] M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *LNCS*, pages 147–162. Springer-Verlag, 2006.