Lightweight Time Modeling in Timed Creol*

Joakim Bjørk Einar Broch Johnsen Olaf Owe Rudolf Schlatte
Department of Informatics, University of Oslo, Norway
{joakimbj,einarj,olaf,rudi}@ifi.uio.no

Creol is an object-oriented modeling language in which inherently concurrent objects exchange asynchronous method calls. The operational semantics of Creol is written in an actor-based style, formulated in rewriting logic. The operational semantics yields a language interpreter in the Maude system, which can be used to analyze models. Recently, Creol has been applied to the modeling of systems with radio communication, such as sensor systems. With radio communication, messages expire and, if sent simultaneously, they may collide in the air. In order to capture these and other properties of distributed systems, we extended Creol's operational semantics with a notion of time. We exploit the framework of a language interpreter to use a lightweight notion of time, in contrast to that needed for a general purpose specification language. This paper presents a timed extension of Creol, including the semantics and the implementation strategy, and discusses its properties using an extended example. The approach can be generalized to other concurrent object or actor-based systems.

1 Introduction

Actor-based systems consist of autonomous "actors" with explicit identity, which execute local tasks and asynchronously exchange messages [1, 2]. Actor-based systems are attractive for the modeling of distributed computing systems due to their separation of concerns between local computation on the one hand and communication and synchronization on the other hand, which fits with a natural way of understanding distributed systems. Creol is an object-oriented modeling language in which inherently concurrent objects exchange asynchronous method calls [12]. The semantics of Creol is defined in rewriting logic [17]; in fact, we have used Maude [8] as an underlying simulation platform for Creol models and extended the interpreter for visualizing the runtime state as well as for dynamic symbolic execution [11]. A novel feature of Creol is that method activations may explicitly suspend execution while waiting for some condition; e.g., the answer to a method call. This results in a very intuitive model of a computation unit which combines active and reactive behavior. In the semantics of Creol, asynchronous method calls are encoded using asynchronous message passing. Ignoring object-oriented features such as the late binding of method calls, a concurrent object in Creol may be understood as an actor in which tasks are executed using cooperative scheduling. Due to its close relationship with actor systems, Creol is a concurrent imperative language which actually supports compositional reasoning [7], in contrast to object-oriented languages like, e.g., Java.

In many cases, time influences the desired (or actual) behavior of systems. As an interesting example, wireless sensor systems behave according to timing properties. The radio unit of a sensor typically has different modes; the radio may be receiving, sending, or dormant. Unless the radio is in receiving mode, a message sent to the sensor will be lost. If two concurrently

*This research was done in the context of the EU project FP7-231620 *HATS*: Highly Adaptable and Trustworthy Software using Formal Methods (http://www.hats-project.eu).

P. C. Ölveczky (Ed.): First International Workshop on Rewriting Techniques for Real-Time Systems (RTRTS'10)
EPTCS 36, 2010, pp. 67–81, doi:10.4204/EPTCS.36.4

running sensors send messages at the same time, the messages may interfere with each other and their content is lost. These properties of wireless sensor networks introduce many interesting challenges for formal methods. A sensor radio with explicit timing parameter settings has been modeled and analyzed in Uppaal [21]. A plethora of new network protocols have been proposed for wireless sensor networks, due to their ad-hoc and self-organizing nature. Ölveczky et al. have shown how such protocols may be modeled and analyzed [14, 20], based on Real-Time Maude [18,19].

For *timed distributed systems*, time is either modeled by a global clock (or equivalently, local clocks which evolve with the same rate), or by local clocks. For simplicity we use a so-called *fictitious clock model* [4] based on a global clock, which allows us to ignore clock synchronization between objects. When modeling timed systems, one may consider different time domains from a partial ordering of events, via a discrete time domain to the detailed continuous time. Increasing the level of granularity of the time domain leads to an increased complexity of the models. For our purpose of grouping simultaneous (communication) events with an interleaving semantics, it suffices to consider a discrete time domain. The values need not correspond to values of real clocks, and are therefore usually chosen to be natural numbers that count steps. Effects such as radio broadcast are confined to a particular instance of time, and broadcast messages disappear as soon as time advances.

In this paper we present a timed version of Creol based on a fictitious clock model. The model is without local clocks. At the language level no additional syntax is needed apart from read-only access to the global clock, using the variable *now*. The major argument for keeping the time model lightweight is to keep the state space as small as possible for model checking purposes. As a case study we present a model of a wireless sensor network together with execution results. We will compare our time model for Creol to other formalisms such as Real-Time Maude.

The paper is structured as follows: Section 2 presents the modeling language Creol. In Section 3 Creol is extended with time. Both syntax and semantics are presented. An example is shown in Section 4. Section 5 gives a comparison of timed Creol to other timed models. Section 6 concludes the paper.

2 A Short Introduction to Creol

We first introduce the features of the object-oriented modeling language Creol which are necessary to understand the approach presented in this paper. A more detailed introduction to Creol can be found in, e.g., [7,12].

Creol features imperative programming constructs for distributed active objects, based on asynchronous method calls and processor release points. Asynchronous method calls may be seen as triggers of concurrent activity, resulting in new activities (processes) within the called object. Objects are dynamically created instances of classes, an *init* method is used to initialize the object's fields at creation time. Active objects encapsulate a current activity (a thread) and an internal activity pool. Active behavior, triggered by a *run* method, is interleaved with passive behavior (triggered by method calls) by means of processor release points. At each point in time at most one thread is active in each object. The scheduling of threads is by default non-deterministic, but more refined scheduling strategies may be given, as in [6]. The modeling language includes a functional expression language for values of basic data types, which will not be explained in detail. Objects are uniquely identified; communication takes place between

Syntactic categories.	*Definitions.*		
C, I, m in Names	IF	$::=$	$\mathbf{interface}\ I\ \mathbf{begin}\ [\mathbf{with}\ I\ \{Sg\}]\ \mathbf{end}$
t in Tag	CL	$::=$	$\mathbf{class}\, C\, [(\{x : I\})]$ $[\mathbf{implements}\, \{I\}]\, \mathbf{begin}\ \{\mathbf{var}\, x : I[:= e]\}\, \{[\mathbf{with}\, I]\, \{M\}\}\, \mathbf{end}$
g in Guard	M	$::=$	$Sg == [\mathbf{var}\ \{\{x\} : I[:= e]\};]\ s$
s in Stmt	Sg	$::=$	$\mathbf{op}\ m\ ([\mathbf{in}\ \{x : I\}][\mathbf{out}\ \{x : I\}])$
x in Var	g	$::=$	$b \mid t? \mid g \wedge g \mid g \vee g$
e in Expr	s	$::=$	$s; s \mid s[]s \mid x := e \mid x := \mathbf{new}\ C\ [(\{e\})]$
o in ObjExpr		$\mid$	$\mathbf{if}\ b\ \mathbf{then}\ s\ [\mathbf{else}\ s]\ \mathbf{end} \mid \mathbf{while}\ b\ \mathbf{do}\ s\ \mathbf{end} \mid \mathbf{await}\ g$
b in BoolExpr		$\mid$	$[t]![o.]m(\{e\}) \mid t?[(\{x\})] \mid [\mathbf{await}][o.]m(\{e\}; \{x\})$

Figure 1: A simplified language syntax. Terms such as $\{e\}$ and $\{x\}$ denote lists over the corresponding syntactic categories and square brackets denote optional elements.

named objects, and object references may be exchanged between objects. Object variables are typed by interfaces. The language 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 also includes call-backs from objects to their environment via the special *caller* variable.

Basic statements. Figure 1 displays a simplified formal syntax of Creol programs. Inheritance in both interfaces and classes is omitted for brevity. A program consists of interface and class definitions. Classes CL contain definitions of attributes x (with initial values) and methods M. A method contains a list of local variable declarations, and a statement s, which may access class attributes, locally defined variables, and the formal parameters of the method (given after the keywords **in** and **out**). An interface definition *IF* contains method signatures *Sg* associated with *co-interfaces* I given by a **with** clause. The co-interface I specifies the type of a client of *IF* and enables to express requirements on call-backs. Finally, a class implements a list of interfaces, thereby specifying the type of its instances. In order to allow call-backs, a method may use the implicit *caller* parameter typed by the co-interface of the method. Class parameters, input parameters, and the self-reference *this*, are read-only. Remote access to attributes is not allowed, method interaction is the only means of communication between objects. Assignment, **if**-, and **while**-constructs are standard. The box `[]` is the non-deterministic choice operator. Creol also includes standard string, numeric, and Boolean datatypes, as well as lists, sets, maps and tuples, and their standard operators.

The guard g controls processor release in the statement **await** g, and consists of Boolean conditions that contain return tests (see below). If g evaluates to false, the current activity is *suspended* and the execution thread becomes idle. When the execution thread is idle, any enabled activity may be chosen from the pool of suspended activities. Explicit signaling is therefore redundant. The *run* method of an object is called after initialization, and initiates active behavior. Release points in the run method allow activities in the activity pool to be scheduled.

Communication. After making an asynchronous method call $t!o.m(\{e\})$, the caller may proceed with its execution without blocking on the method reply. Here o is an object expression and $\{e\}$ are (data value or object) expressions. The tag t will be assigned a unique value that identifies the call, which may later be used to refer to that call in two different ways: First, the guard **await** $t?$ suspends the active activity unless a return to the call associated with t has arrived. Second, the return values are retrieved by the *reply statement* $t?(x)$, which is blocking the object until the return values are present. Local calls are written $t!m(\{e\})$. If no return values are desired by the caller, the tag may be omitted; e.g., $!o.m(\{e\})$. The sequence $t!o.m(\{e\})$; $t?(x)$ encodes a *blocking call*, abbreviated $o.m(\{e\};x)$ (often referred to as a synchronous call), whereas the call sequence $t!o.m(\{e\})$; **await** $t?$; $t?(x)$ encodes a non-blocking, *preemptable call*, abbreviated **await** $o.m(\{e\};x)$.

3 A Time Model for Creol

In order to reason about time in a Creol model, the concept of time has to be introduced to the language. Our timed Creol interpreter adds a datatype `Time` and its accompanying operations to the language.

A value of type `Time` can be obtained by evaluating the expression **now**, which returns the current time, i.e., the value of the global clock. Given a time value, other time values can be constructed by adding and subtracting duration values.

Time values form a total order, with the usual less-than operator semantics. Hence, two time values can be compared with each other, resulting in a Boolean value suitable for guards in **await** statements. While all other time values are constant, the result of comparing the expression **now** with another time value will change with the passage of time. As an example, given a tag `l`, the following Creol fragment

```
var t :Time :=now;
await l? ∧now <t +10 ; SL
[]
await now >t +10 ; EL
```

models both the normal (`SL`) and the timeout (`EL`) behavior of the synchronization with the method invocation associated with `l`. The box `[]` is the non-deterministic choice operator, which chooses one of its two statements if both are enabled, and blocks until at least one statement is enabled. (In the example above, the choice operator does not add non-deterministic behavior to the model, since the two guards are mutually exclusive.)

In this model of timed behavior, the passage of time needs never be made explicit in the model, as with e.g. a `tick` statement. Instead, passage of time is observed within **await** statements, and time is advanced when no other activity may occur. Note, though, that the semantics of this model of time, combined with Creol's blocking and non-blocking synchronization semantics, are powerful enough to express both activity- and object-wide `tick` statements: given the following method definition:

```
op tick(in duration: Int) == var t :Time :=now ; await now ≥t +duration,
```

The following two lines of timed Creol express activity- and object-wide `tick`, respectively, by using non-blocking and blocking synchronization on the tag `l`:

```
l!tick(1); await l?
l!tick(1); l?
```

The remainder of this section describes the implementation of this Creol time model as an operational semantics expressed in Maude.

3.1 Operational semantics

We introduce a clock that holds the current global time value. This value is accessible to Creol models via the `now` expression. The global clock is a structure containing an identifier and two natural numbers:

```
⟨O : Clock | time: T, limit: B ⟩
```

The first number, `T`, is the current time. `B` provides an upper time bound for model execution. `O` is a unique identifier for this clock object. The identifier is superfluous as models should only contain one clock object, but is included to allow the clock to follow standard object notation in Maude.

Each Creol object is represented as a Maude structure containing instance variable bindings, the currently running process, the process queue, and some housekeeping values. Maude equations and rewrite rules operate on these structures and implement the operational semantics of Creol.

As an example, here is an outline of the rule in the untimed interpreter that implements the caller side of asynchronous method calls (simplified for demonstration purposes – we leave out some details, such as generating the unique identifier `N` for the fresh future structure):

```
rl
 ⟨O : C | Att: A, Pr: { L | F :=call(O1, M, PL) ; SL }, PrQ: W ⟩
⟶
 ⟨O : C | Att: A, Pr: { L :: F ↦N | SL }, PrQ: W ⟩
 ⟨N : Future | Completed: false, Ref: 1, Value: emp ⟩
 invoc (O1, M, eval(PL, A :: L), O, N)
[label async-call] .
```

The object `O` of class `C` (with attributes `A`) has an active process with a state `L` and a first statement `call` to the method `M` of object `O1`. The result of that statement is to be stored in the local variable `F`. The rule `async-call` removes the call statement from the active process and updates the binding of the tag `F` with a reference to a new future `N` that holds the status of the method invocation (completed or not) and its return value. (A separate structure is necessary since Creol's Futures are first-class values that can be referenced from multiple objects.) Finally, the rule also generates an `invoc` structure for `O1` that will, in the rule `message-receive`, cause a process to be inserted into the callee's (`O1`'s) process queue `PrQ`.

```
rl
 ⟨O : C | Att: A, Pr: P, PrQ: W ⟩
 ⟨C : Class | Mtds: (MS, ⟨M : Method | Param: L, Code: SL ) ⟩
 invoc (O, M, DL, O1, N)
⟶
 ⟨O : C | Att: A, Pr: P, PrQ: (W, { L ↦ DL, caller ↦ O1, .result ↦ N | SL } ) ⟩
 ⟨C : Class | Mtds: (MS, ⟨M : Method | Param: L, Code: SL ) ⟩
[label message-receive] .
```

The `message-receive` rule, presented above, appends a new process into the receiving object's process queue. Method lookup is done by name in the class of the object (we elide handling of method inheritance for reasons of brevity); argument values `DL` are bound to the method's parameters `L`. Additionally, the `caller` attribute is set to a reference of the calling object `O1` and an internal field `.result` stores a reference to the future `N` that will store the result of the method invocation. Finally, the statement list of the process is initialized with the statement list `SL` of the method definition.

All rules for executing statements follow this general pattern. Specifically, note that these rules can execute mostly independently of the global clock value. The only exception is the Creol expression **now** which, given the presence of a clock ⟨O : Clock | time: T, limit: B ⟩, will evaluate to a term time(T). This in turn will influence Boolean guards involving the global time, hence the readiness of processes waiting on such a guard becoming true.

For instance, the timed version of the rule async-call looks as follows:

```
rl
 ⟨O' : Clock | time: T, limit: B ⟩
 ⟨O : C | Att: A, Pr: { L | F :=call(O1, M, PL) ; SL }, PrQ: W ⟩
⟶
 ⟨O' : Clock | time: T, limit: B ⟩
 ⟨O : C | Att: A, Pr: { L :: F ↦N | SL }, PrQ: W ⟩
 ⟨N : Future | Completed: false, Ref: 1, Value: emp ⟩
 invoc (O1, M, eval(PL, A :: L, T), O, N)
[label timed-async-call] .
```

The only difference to the untimed version is that the eval function gets an additional argument, namely the time value. Rule message-receive does not need to be changed for timed Creol at all, since no evaluation of expressions takes place in that rule.

Consequently, for the timed interpreter the global clock has to be added both to the left-hand and right-hand side of any rule that involves evaluating an expression. Here is the rule for evaluating an **await** statement in the timed interpreter:

```
crl
 ⟨O : C | Att: A, Pr: { L | await E ; SL }, PrQ: W ⟩
 ⟨O' : Clock | time: T, limit: B ⟩
 CN
⟶
 ⟨O : C | Att: A, Pr: { L | SL }, PrQ: W ⟩
 ⟨O' : Clock | time: T, limit: B ⟩
 CN
 if evalGuard(E, (A :: L), CN, T) asBool
[label await] .
```

If the condition E holds, as determined by the auxiliary equation evalGuard, then the **await** statement simply reduces to a skip. In contrast to the eval function above, evalGuard may need to consider futures in the environment, and gets the additional parameter CN. In contrast to the tick rule presented below, there is no necessity of referencing the whole system in CN here, since Maude will choose a subset as needed. Another rule in the interpreter, not presented here, has the task of suspending the active process if E evaluates to false.

Clock advancement. The global clock cannot advance freely. We specify run-to-completion semantics, requiring all objects to finish their actions as soon as possible. Hence, there are several restrictions to when the global clock may advance. Clock advancement is blocked if:

- An object has an active non-blocked process.
- Some process in the process queue of an object is enabled and ready to run.
- A message (method invocation) is "in flight" between objects.

We define a Maude operator *canAdvance* that traverses the configuration of objects and returns *false* if any of these conditions are true. A sketch of the its semantics, with some implementation details elided, looks as follows:

```
op canAdvance : Configuration Time→ Bool .

eq canAdvance (⟨ O : C | Att: A, Pr: P, PrQ: W ⟩ CN, T) =
 blocked(P, A, CN, T) and canAdvance(CN, T) .
```

```
eq canAdvance (⟨ O : C |Att: A, Pr: idle, PrQ: W ⟩ CN, T) =
 allBlocked(W, A, CN, T) and canAdvance(CN, T) .

eq canAdvance (⟨ M: Msg |⟩ CN, T) = false .

eq canAdvance (CN, T) = true [owise] .
```

The `blocked` operator reduces to *true* if the process `P` given as its first argument is not enabled. `allBlocked` returns *true* if all of the processes in the process queue `W` are blocked. Both these operators need the current configuration to determine the status of processes currently waiting on futures, which is why the configuration is an argument to `blocked` and `allBlocked`.

With the `canAdvance` equation, the tick rule of the global clock is straightforward. We increase the time by one time unit if the clock can advance, as long as the time limit is not exceeded. The definition of the tick rule is shown below (with `{ }` enclosing the whole configuration to produce a system, in the same manner as in Real-time Maude).

```
crl
  { CN ⟨O : Clock |time: T, limit: B ⟩ }
⟶
  { CN ⟨O : Clock |time: T +1, limit: B ⟩ }
if canAdvance (CN, T) and T < B
  [label tick] .
```

Here the importance of the `limit` attribute of the clock becomes clear: If all object activity has ceased, the clock is free to advance without any bound. The limit, which can be arbitrarily large, ensures that an unbounded Maude rewrite command terminates.

4 Example: A Model of a Wireless Sensor Network

We used timed Creol to model the behavior of a wireless sensor network. A typical sensor network consists of a number of sensors, and a sink which collects data. The sensors record some sort of data and send it towards the sink via wireless links. Due to the limited power of the wireless signals, some sensors may not be directly connected to the sink; these sensors are dependent on other sensors to forward their messages. The routing of messages towards the sink may be done in many different ways to limit time consumption, energy consumption, number of messages sent etc. The example in this section uses a simple flooding algorithm: when a sensor senses data, it broadcasts it to all other nodes within range. When a sensor receives a message that it has not seen before, it is rebroadcast to all its neighbors. More involved routing algorithms exist where sensors selectively rebroadcast messages depending on whether they are on the path to the sink, but this simple algorithm suffices to show our results. A more complex routing protocol modeled in Creol is investigated in [16].

In our model, the nodes are not directly connected to each other. Instead, each node has a reference to a *Network* object which models the behavior of the transmission medium between nodes. This structure enables us to model collisions, message loss, selective retransmission, and the node topology (which nodes can be reached from each node) without local knowledge inside the node objects.

Figure 2 shows the interfaces of both the nodes and the network. The `broadcast` method of the network gets called by nodes when they wish to broadcast data; the `receive` method of a node is called by the network with data that is broadcast by another node.

The *run* method of a *Main* class (not given here) sets up the sensor network by creating all `Sensor` objects, one `Network` object, and establishing the topology. The sensors

```
interface Node
begin
  with Network
    // Receive data from network.  data format:  (id of originator, sequence no)
    op receive(in data: [Int,Int])
end

interface Network
begin
  with Any
    // Register 'node' as part of the network and as able to send to the
    // nodes in 'connections'
    op register(in node: Node, connections: List[Node])
  with Node
    op broadcast(in data: [Int,Int])
end
```

Figure 2: The interfaces of the sensor network

are created by statements such as `n1 :=new SensorNode(1, nw, 3);`. This statement results in the creation of a *SensorNode* object with *id* 1. It will belong to the network *nw* and do three sensings. Later the sensor will be registered in the network by a command like `nw.register(n1, [n2,n3];);` which states that the sensor *n1* should have a link to the nodes *n2* and *n3*. These links are not necessarily symmetrical; it is possible for a sensor to receive messages from another sensor but not be able to send to it in turn.

When an object of class *SensorNode*, shown in Figure 3, is generated, its `run` method executes (shown in line 29 ff.), choosing one of two behaviors. If the sensor has not yet done the number of sensings it is supposed to do, it may do a call to its own *sensing* method (line 25 ff.). The sensor data, which for simplicity is just a counter, is added to the `sendqueue` list together with the sensor's id. The `sendqueue` list contains all messages waiting to be sent by the sensor.

If there are elements in the `sendqueue` list, the sensor's `run` method may also call its own `sendOrForward` method (line 10 ff.). A call to this method will result in the broadcasting of the first message in the `sendqueue` list by a call to the `broadcast` method of the network object. If both sensing and sending is possible, then the node will perform a non-deterministic choice between the two actions. If none of these actions are possible, it will release the processor and wait.

When a sensor receives a message from another sensor, a call to the *receive* method is made by the network. If the sensor has not seen this message before, it is added to the `received` list, and additionally queued for re-sending.

Timed behavior was added to the node class by modifying the `sendOrForward` method. When executing that method, the current time is stored. We specify that sending a message takes one unit of time, so after broadcasting (and after synchronizing with the network to ensure broadcasting has finished), line 18 waits until that amount of time has passed before terminating the method. The `sending` field serves as a mutual exclusion flag (only one sending call can be active at any time); note how the cooperative scheduling of Creol makes this style of programming both safe and easy to understand. Also note that blocking the sending method in this way does not preclude the sensor from receiving messages, even while it is waiting for the network to accept its message.

The `SinkNode` class given in Figure 4 implements the same interface as the sensor nodes,

```
class SensorNode(id: Int, network: Network, noSensings: Int)
implements Node
begin
  var received: List[[Int,Int]] :=nil; // All previously received messages
  var sendqueue: List[[Int,Int]] :=nil; // Messages waiting to be sent
  var seqNo: Int :=0; // Running package seq.  no
  var sending: Bool :=false;

  // Forward (or send) a single message from the queue
  op sendOrForward ==
    var t: Time :=now;
    var l: Tag[ ];
    await sending = false;
    sending :=true;
    l!network.broadcast(head(sendqueue));
    sendqueue :=tail(sendqueue);
    await l?;
    await now >t;
    sending :=false

  op store(in data: [Int,Int]) ==
    sendqueue :=sendqueue ⊢data

  // Produce a sensing (of the environment) and store it locally
  op sense ==
    store((id,seqNo);;);
    seqNo :=seqNo +1

  op run ==
    await start = true;
    while true do
      await seqNo < noSensings; sense(;)
      []
      await #(sendqueue) >0; sendOrForward(;)
    end
  with Any
    op start == start :=true

  with Network
    op receive(in data: [Int,Int]) ==
      await start = true;
      if ¬(data in received) then // re-send if not seen before
        received :=received ⊢data;
        store(data;)
      end

end
```

Figure 3: The implementation of the sensor nodes

but has a different behavior. The major difference is that the sink has no *run* method, and hence no activity of its own. The `store` method of the sink counts the number of unique messages received, the `receive` method notes the time when the last message was received.

Figure 5 shows the implementation of the network. Its behavior is implemented by the `broadcast` method (line 15 ff.). We implemented different behavior for that method in the case of multiple senders in a time slot:

- No message collision (broadcast all messages)
- Collision sensing and re-send
- Message loss

```
class SinkNode(network: Network)
implements Node
begin
  var noStored: Int :=0;
  var received: List[[Int,Int]] :=nil;
  var lastReceived: Time;

  op init ==
    lastReceived :=now

  op store(in data: [Int,Int]) ==
    noStored :=noStored +1

  with Network
    op receive(in data: [Int,Int]) ==
      if (lastReceived <now) then lastReceived :=now end;
      if ¬(data in received) then // store if not seen before
        received :=received ⊢data;
        store(data;)
      end
end
```

Figure 4: The implementation of the sink node

Collision behavior	Topology	No. received by sink	Timestamp of last transmission
no interference	linear	12	14
	mixed	12	14
	star	12	2
resend	linear	12	60
	mixed	12	38
	star	12	12
drop	linear	0	–
	mixed	2	4
	star	2	2

Table 1: Timing results for different network behavior

The `broadcast` implementation of Figure 5 shows the second behavior (re-sending messages). Conceptually, re-sending means that the sensor broadcasts the message until it is successful (no collisions happened). This is implemented by the field `lastTransmission` in the network, which in conjunction with line 23 only lets one `broadcast` method call complete per time slot. (We make the optimistic assumption that the first broadcast actually succeeds – i.e. the sensors' antennas sense another transmission going on and the sensors abort their own attempt at sending.)

The model's behavior for message collision can be adapted easily. If line 23 is removed, multiple sensors can send at the same time. If line 23 is changed to

```
if lastTransmission = now then recs :=nil end;
```

all messages sent in a time slot after the first one are dropped.

Table 1 shows the effect of these different network behaviors on a network consisting of four sensor nodes sending three messages each. Three different topologies were simulated: the edge cases of each node having a direct connection to the sink in a star shaped network, and all nodes

```
class BroadcastNetwork()
 implements Network
begin
 var nodesConns: Map[Node, List[Node]] :=empty();
 var lastTransmission: Time;

 op init ==
  lastTransmission :=now

 with Any
  op register(in node: Node, connections: List[Node]) ==
   nodesConns :=insert(nodesConns, node, connections)

 with Node
  op broadcast(in data: [Int,Int]) ==
   var rec: Node;
   var recs: List[Node] :=nil;

   if caller in nodesConns then
    recs :=get(nodesConns, caller)
   end;

   await now >lastTransmission;
   lastTransmission :=now;

   while ¬isempty(recs) do
    rec :=head(recs);
    recs :=tail(recs);
    if rec ≠caller then
     !rec.receive(data)
    end
   end
end
```

Figure 5: The implementation of the network

being arranged in a linear fashion, and a more typical "mixed" network, with two nodes being able to reach the sink, the other nodes having to send through them (see Figure 6). For the mixed topology, only two messages manage to reach the sink at all in case of message loss upon collision; interference without message loss more than doubles the time until all messages reach the sink. The linear configuration exhibits the worst behavior, with potentially no message reaching the sink at all without re-sending messages upon collision.

5 Related Work

Actor systems have been used to model mobile ad-hoc networks, which are similar to our biomedical sensor networks, in [9], but this work does not include a notion of time in its model. Many

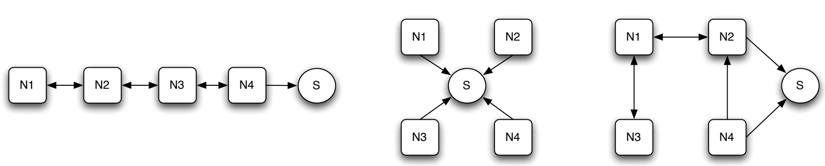

Figure 6: Network topologies: linear, star, mixed (from left to right)

approaches to timed modeling, for example the UPPAAL tool [5] use variants of timed automata [3]. We decided to take a different approach, namely to augment existing behavioral models with timing properties. Kyas and Johnsen [15] describe a timed extension to Creol that is similar to the one proposed in this paper. Their approach is a bit more expressive, since time constraints can be expressed as intervals in the model, compared to our approach, where only a minimum passed time can be declared and the semantics do not enforce progress. On the other hand, we give an operational semantics of timed Creol instead of a denotational one, along with an interpreter that implements "earliest possible completion" semantics for the language.

5.1 Real-time Maude

Real-time Maude is an extension of Full Maude [8] introducing a concept of time to support real-time rewrite theories. It supports both discrete and dense time domains. New datatypes for time are introduced, and the user must specify what time model to use, or make a new one. In order to make time advance at the same pace throughout the system, an encapsulation of the configuration is required, and the sort `GlobalSystem` is introduced as follows:

```
op {_} : System → GlobalSystem .
```

Time advances by applying a *tick* rule. A typical *tick* rule would be something like:

```
crl [tick] : {SYSTEM} ⟶ {delta(SYSTEM, R)} in time R if R ≤mte(SYSTEM) .
```

The *delta* function propagates the time elapse throughout the configuration and the *mte* function calculates the maximum time elapse. Both these functions must be defined by the user. `R` is of type *Time* and is a random number not greater than the maximum time elapse of the system. If it is desired, the time may advance with a fixed number of time steps by replacing `R` with a number, and omitting the if part of the tick rule.

In contrast to Real-time Maude, our extension of Creol is targeting distributed concurrent objects with a simple notion of abstract discrete time. This gives a lightweight formalization of time which is strong enough to model time-outs as well as synchronized behavior suitable for wireless communication. In our setting the *tick* rule is simpler than in Real-time Maude in the sense that it can be formulated without use of auxiliary functions depending on user-defined equations. Thus the Creol programmer needs not understand the technicalities of the tick rule nor the time model. The notion of time is fully predefined, and the programmer's role is to use time-outs when modeling timed systems.

Our time model could in principal be implemented in Real-time Maude, but the *delta* and *mte* equations would be complicated due to the internal structure of Creol objects, and take into account time-outs, *await* statements and blocking calls appearing in the imperative code in the objects and also in the object's internal process queue.

Previous work on wireless sensor modeling in Real-Time Maude includes statistical model checking of wireless sensor network algorithms, and has demonstrated that this approach can be used to detect flaws in non-trivial wireless sensor network algorithms [14, 20]. The present work is not considering probabilistic modeling; however, in current work towards probabilistic Creol semantics we are trying to exploit this approach.

6 Conclusions

We have presented an extension of Creol with a lightweight model of time which supports the modeling of radio communication and time-dependent cooperative scheduling of tasks. Technically, this is done by allowing read-only access to a global clock, and thereby time related guards including lower and upper timer bounds. Currently, the time model does not support continuous time. Exploiting Creol's concept of asynchronous methods calls, time-outs may be used to model the passing of time while blocking the processor, as well as the passing of time while releasing the processor. This means that the passing of time can be tightly integrated into a Creol program. Due to the non-determinism in Creol, it is easy to capture distributed systems where the concurrent objects are progressing at independent speeds. This allows the modeling of "best-effort" systems where a time-out may be chosen some time after the given time limit.

The model presented here is a simplification of earlier timed versions of Creol [13], where both local and global time were modeled. Local time allowed a more direct implementation of objects progressing at independent speeds and of timing of communication events [13]. The present model is without local clocks and gives the programmer more freedom to model the progression of time. Another advantage is the ease with which timing information can be added to existing models. The model in Section 4 started out as an untimed model; augmenting it with timing constraints was a matter of adding less than 10 lines of code in total. We believe that this ease of expressing timing constraints is a valuable tool for the modeler.

Our semantic model of timed Creol is not suitable for modeling of systems where clock drift is relevant. It is suitable when one may assume perfect clock synchronization between nodes. Experiences with the more low-level semantics with local clocks given in [13] showed that this model was difficult to analyze with the Maude tools due to the large state space. The motivation behind the current approach was to achieve a model with a smaller state space, and still expressive enough to cover interesting Creol models. The investigation of the applicability of Maude analysis tools is still future work.

The simplicity of our model seems promising with respect to automated analysis including state space exploration and model checking tools. In fact, timed simulation of an untimed model will in many cases have less states than the corresponding untimed simulation, due to the constraints imposed by the global clock. A main motivation behind the design of Creol is simplicity of reasoning and simplicity of composition rules. In contrast to the standard multi-threaded concurrency model of object-oriented systems, Creol has a compositional proof system [7]. For partial correctness, the simplicity of the reasoning system for Creol can be preserved for the extension of Creol with time-outs. The time-outs can be handled in the reasoning system as a special kind of *await* statements, using the approach outlined in [10].

The extension of Creol with time does not depend on very specific features of the language. Creol was chosen as a base language to allow exploitation of its executable Maude interpreter by extending it with time and experimenting with case studies. The presented lightweight model of time seems applicable to the wider setting of distributed concurrent object or actor-based systems where communication is by message passing.

Acknowledgments The original, untimed BSN sensor network model in Creol was created by Wolfgang Leister and Xuedong Liang. The authors wish to thank S. Lizeth Tapia Tarifa and the anonymous reviewers for valuable comments.

References

[1] Gul A. Agha (1986): *ACTORS: A Model of Concurrent Computations in Distributed Systems*. The MIT Press, Cambridge, Mass.

[2] Gul A. Agha, Ian A. Mason, Scott F. Smith & Carolyn L. Talcott (1997): *A foundation for actor computation*. *Journal of Functional Programming* 7(1), pp. 1–72.

[3] Rajeev Alur & David L. Dill (1994): *A Theory of Timed Automata*. *Theoretical Computer Science* 126(2), pp. 183–235.

[4] Rajeev Alur & Thomas A. Henzinger (1992): *Logics and Models of Real Time: A Survey*. In: Jaco W. de Bakker, Cornelis Huizing, Willem-Paul de Roever & Grzegorz Rozenberg, editors: *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, *Lecture Notes in Computer Science* 600, Springer, pp. 74–106.

[5] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson & Wang Yi (1995): *UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems*. In: Rajeev Alur, Thomas A. Henzinger & Eduardo D. Sontag, editors: *Hybrid Systems*, *Lecture Notes in Computer Science* 1066, Springer, pp. 232–243.

[6] Frank S. de Boer, Tom Chothia & Mohammad Mahdi Jaghoori (2010): *Modular Schedulability Analysis of Concurrent Objects in Creol*. In: Farhad Arbab & Marjan Sirjani, editors: *Third IPM International Conference on Fundamentals of Software Engineering (FSEN'09)*, *Lecture Notes in Computer Science* 5961, Springer, pp. 212–227.

[7] Frank S. de Boer, Dave Clarke & Einar Broch Johnsen (2007): *A Complete Guide to the Future*. In: Rocco de Nicola, editor: *Proc. 16th European Symposium on Programming (ESOP'07)*, *Lecture Notes in Computer Science* 4421, Springer, pp. 316–330.

[8] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer & José F. Quesada (2002): *Maude: Specification and Programming in Rewriting Logic*. *Theoretical Computer Science* 285, pp. 187–243.

[9] Jessie Dedecker & Werner Van Belle (2004): *Actors for Mobile Ad-Hoc Networks*. In: Laurence Tianruo Yang, Minyi Guo, Guang R. Gao & Niraj K. Jha, editors: *Embedded and Ubiquitous Computing, International Conference EUC 2004, Aizu-Wakamatsu City, Japan, August 25-27, 2004, Proceedings*, *Lecture Notes in Computer Science* 3207, Springer, pp. 482–494.

[10] Johan Dovland, Einar Broch Johnsen & Olaf Owe (2005): *Verification of Concurrent Objects with Asynchronous Method Calls*. In: *Proceedings of the IEEE International Conference on Software Science, Technology & Engineering(SwSTE'05)*, IEEE Computer Society Press, pp. 141–150.

[11] Andreas Griesmayer, Bernhard Aichernig, Einar Broch Johnsen & Rudolf Schlatte (2009): *Dynamic Symbolic Execution for Testing Distributed Objects*. In: Catherine Dubois, editor: *Proc. 3rd Intl. Conference on Tests and Proofs (TAP'09)*, *Lecture Notes in Computer Science* 5668, Springer-Verlag, pp. 105–120.

[12] Einar Broch Johnsen & Olaf Owe (2007): *An Asynchronous Communication Model for Distributed Concurrent Objects*. *Software and Systems Modeling* 6(1), pp. 35–58.

[13] Einar Broch Johnsen, Olaf Owe, Joakim Bjørk & Marcel Kyas (2008): *An Object-Oriented Component Model for Heterogeneous Nets*. In: Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf & Willem-Paul de Roever, editors: *Proc. 6th International Symposium on Formal Methods for Components and Objects (FMCO 2007)*, *Lecture Notes in Computer Science* 5382, Springer-Verlag, pp. 257–279.

[14] Michael Katelman, José Meseguer & Jennifer C. Hou (2008): *Redesign of the LMST Wireless Sensor Protocol through Formal Modeling and Statistical Model Checking*. In: Gilles Barthe & Frank S. de Boer, editors: *Proc. 10th Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'08)*, *LNCS* 5051, Springer, pp. 150–169.

[15] Marcel Kyas & Einar Broch Johnsen (2009): *A Real-Time Extension of Creol for Modelling Biomedical Sensors*. In: Frank S. de Boer, Marcello M. Bonsangue & Eric Madelaine, editors: *Formal Methods for Components and Objects, 7th International Symposium, FMCO 2008, Sophia Antipolis, France, October 21-23, 2008, Revised Lectures*, *Lecture Notes in Computer Science* 5751, Springer-Verlag, pp. 42–60.

[16] Wolfgang Leister, Xuedong Liang, Sascha Klüppelholz, Joachim Klein, Olaf Owe, Fatemeh Kazemeyni, Joakim Bjørk & Bjarte M. Østvold (2009): *Modelling of Biomedical Sensor Networks using the Creol Tools*. Technical Report 1022, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway.

[17] José Meseguer (1992): *Conditional rewriting logic as a unified model of concurrency*. *Theoretical Computer Science* 96, pp. 73–155.

[18] P. C. Ölveczky & J. Meseguer (2002): *Specification of real-time and hybrid systems in rewriting logic*. *Theoretical Computer Science* 285, pp. 359–405.

[19] Peter Csaba Ölveczky & José Meseguer (2007): *Semantics and pragmatics of Real-Time Maude*. *Higher-Order and Symbolic Computation* 20(1–2), pp. 161–196.

[20] Peter Csaba Ölveczky & Stian Thorvaldsen (2009): *Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude*. *Theoretical Computer Science* 410(2-3), pp. 254–280.

[21] Simon Tschirner, Liang Xuedong & Wang Yi (2008): *Model-based validation of QoS properties of biomedical sensor networks*. In: Luca de Alfaro & Jens Palsberg, editors: *Proceedings of the 8th ACM & IEEE International conference on Embedded software (EMSOFT'08)*, ACM, pp. 69–78.