An Object-Oriented Component Model for Heterogeneous Nets ⋆

Einar Broch Johnsen, Olaf Owe, Joakim Bjørk and Marcel Kyas

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

Abstract. Many distributed applications can be understood in terms of components interacting in an open environment. This interaction is not always uniform as the network may consist of subnets with different quality: Some components are tightly connected with order preservation of communicated messages, whereas others are more loosely connected such that overtaking of messages and even message loss may occur. Furthermore, certain components may communicate over wireless networks, where sending and receiving must be synchronized, since the wireless medium cannot buffer messages. This paper proposes a formal framework for such systems, which allows high-level modeling and formal analysis of distributed systems where interaction is managed by a variety of nets, including wireless ones. We introduce a simple modeling language for object-oriented components, extending the Creol language. An operational semantics for the language is defined in rewriting logic, which directly provides an executable implementation in Maude.

1 Introduction

Object-oriented modeling languages [3, 11, 22] aim for a high level of abstraction, and typically capture systems in a *platform independent* manner, as advocated by *model-driven architecture* [24]. Consequently, these languages abstract from low-level communication details such as, e.g., the specific properties of the communication medium components use. However modern distributed applications often require a certain *quality of service*, which cannot be modeled when perfect channels are assumed; e.g., a maximum latency or a minimum throughput. In practice, the properties of a specific connection may even evolve during execution. In particular, connections to other components may appear or disappear, and network components may be shared between several applications with different requirements. Consequently, the quality of a connection between two components may vary over time, and connections to components with the same functional interface may vary significantly in bandwidth or robustness. In many cases, the behavioral properties of the modeled system depend on the specific properties of the net. For such systems, it is desirable to enable *cross-layer designs* [23] by reflecting aspects of the (low-level) connectivity in the high-level modeling language.

In this paper, we develop a light-weight, timed component model with an executable semantics. We present a kernel language, extending our previous work on Creol

⋆ This research is in the context of the EU project IST-33826 *CREDO: Modeling and analysis of evolutionary structures for distributed services* (http://credo.cwi.nl).

[11, 12]. Creol is a modeling language for distributed concurrent objects communicating by means of asynchronous method calls. However, this previous work did not address the communication medium in which the concurrent objects live. In this paper, we introduce language primitives to reflect links with different qualities. This allows us to model communication in *heterogeneous nets*; these are nets in which some components may be tightly and reliably connected whereas others may be loosely connected through unreliable or wireless links. This allows certain aspects of the connection quality between components to be taken into account during the analysis of a model. Furthermore, a component may decide on its actions depending on how it is connected to other components. In particular we consider radio communication as well as multicast and broadcast communication, in order to integrate object-oriented modeling with wireless networks. The language abstracts from many implementation details; e.g., it uses a functional sublanguage for side-effect free expressions and execution may be highly non-deterministic. The language has an operational semantics defined in rewriting logic [15] and it is executable on the Maude platform [5], which supports various forms of analysis such as simulation and breadth-first search through the execution space. To illustrate the language and analysis using Maude's simulation support, an example of a simple sensor network is given.

The paper is structured as follows: Section 2 discusses modeling of network aspects, Sect. 3 presents the modeling language, and Sect. 4 provides an example based on wireless medical sensors. Sect. 5 defines the operational semantics of the language, Sect. 6 discusses related and future work, and Sect. 7 concludes the paper.

2 Modeling of Network Information

This paper considers modeling of distributed systems that communicate over different *links*, and introduces a novel framework where such systems may be modeled and simulated, and where system properties may be subjected to formal analysis. When modeling a distributed system, the model should not only describe the components and their behaviors, but also how the different communication media involved are composed, since media properties often affect the overall properties of the system. However, it is not desirable to address all aspects of the network and communication details in a high-level model . We will here focus on safety properties such as "the sender is aware of sent messages that have arrived", but also certain liveness properties such as "a message will arrive at its destination in at most n hops". This means that certain aspects of the communication and network media must be formalized, for instance whether communication preserves the order, whether communication is immediate or delayed, and whether message may get lost. Such factors typically affect overall system properties.

In particular, we consider here *tight*, *loose*, and *wireless* links. A *tight link* between two nodes provides a reliable communication channel that guarantees FIFO ordering of messages sent between the linked objects. A typical example is a serial line link between the two nodes, or a TCP/IP connection. A *loose link* between two nodes is still reliable but it does not guarantee the FIFO ordering; rather some messages take more time than others. A *wireless link* provides synchronous transmission of messages, but simultaneous messages may be lost if they are within reach of each other (message

Name		**Description and Examples**	**Simple Model**
Host Layers	7. Application	Web browser, file transfer, mail transfer	3. Application
	6. Presentation	Data representation (MIME, XML) and encryption	2. Transport
	5. Session	Interhost communication (RPC, iSCSI)	
	4. Transport	End-to-end connections & reliability (TCP, UDP)	
Media Layers	3. Network	Path determination & logical addressing (IP)	1. Media
	2. Data link	Physical addressing (802.3 (Ethernet), 802.11a/b/g MAC/LLC (Wireless))	
	1. Physical	Media (100BASE-TX (Ethernet), IEEE 802.11a/b/g PHY (Wireless)), signal, binary coding	

Fig. 1. Network layering model.

collision). A message is received if the sender sends, the message does not collide in transmission, and the receiver receives at the same time; otherwise the message is lost. (Remark that the models considered in this paper are non-deterministic but not probabilistic.) A network built from tight links is called a *tight network*. Analogously, we define *loose networks* and *wireless networks.* A network may have parts that are loose, tight, and wireless. A loose network may have parts that are tight, but not vice versa. We say that a tight link is *better than* any other link.

Our model is based on the *Open System Interconnection Basic Reference Model* (ISO/IEC 7498) [26], OSI model for short. This is considered as one of the standard models for describing networks and applications. It allows application level programming without knowledge of the underlying network protocols. However, the abstractions provided by the OSI model sometimes make it difficult to exploit the capabilities of the underlying network and protocols. For example, applications in wireless networks often have specific requirements on memory and energy use and still need to guarantee their service with a certain quality. Such applications call for *cross-layer designs,* where the abstractions of the OSI model are weakened with APIs that enable the control of aspects of the lower layers (cf. [23] and below). In addition, by using different protocols over the same net, one may obtain different network qualities, such as lossy and fast versus non-lossy and slow, or FIFO and slow versus reordering and fast. In both cases a model design for a given application may benefit from some low-level network (protocol) knowledge, and possibly also from the reprogramming abilities of certain network related aspects. By using the same language for the application level modeling and the network related aspects, such as the programming of network protocols and wireless radio controllers, one may obtain a uniform model with desired properties.

This paper considers a simplified version of the OSI model, which we compare to the original ISO/OSI model (see Fig. 1). Our model allows application level programming as well as the programming of network protocols and wireless radio controllers. Details that are not relevant for high-level modeling and analysis are abstracted away, while other details are included, such as the presence of communication buffers, order-

Syntactic categories.
C, I, m in Names
n in Network
t in Label
g in Guard
p in MtdCall
s in Stm
x in Var
e in Expr
o in ObjExpr
b in BoolExpr

Definitions.

$$
\begin{array}{rcl}
IF & ::= & \texttt{interface}\ I\,\{\overline{x:I}\}\,\{\texttt{inherits}\ \overline{I}\} \\
 & & \qquad \texttt{begin}\ \{\texttt{with}\ I\ \overline{Sg}\}\ \texttt{end} \\
CL & ::= & \texttt{class}\ C\,\{\overline{x:I}\}\,\{\texttt{inherits}\ \overline{C}\}\,\{\texttt{implements}\ \overline{I}\} \\
 & & \qquad \texttt{begin}\ \{\texttt{var}\ \overline{x:I\{=e\}}\}\ \overline{\{\texttt{with}\ I\}\ \overline{M}}\ \texttt{end} \\
M & ::= & Sg == \{\texttt{var}\ \overline{x:I\{=e\}};\}\ \overline{s} \\
Sg & ::= & \texttt{op}\ m\ (\{\texttt{in}\ \overline{x:I}\}\{\texttt{out}\ \overline{x:I}\}) \\
n & ::= & \texttt{loose} \mid \texttt{tight} \mid \texttt{wless} \\
g & ::= & b \mid t? \mid g \wedge g \mid g \vee g \\
p & ::= & m(\overline{e}) \mid o.m(\overline{e}) \\
s & ::= & (s) \mid s;s \mid x := e \mid x := t.\texttt{get} \mid x := \texttt{new}\ C\{(\overline{e})\}\{\texttt{in}\ o\} \mid \texttt{tick}(n) \\
 & & \mid \texttt{if}\ b\ \texttt{then}\ \overline{s}\ \{\texttt{else}\ \overline{s}\}\ \texttt{fi} \mid \texttt{while}\ b\ \texttt{do}\ \overline{s}\ \texttt{od} \mid \texttt{await}\ g \\
 & & \mid \{t :=\}!p \mid \{x :=\}p \mid \texttt{await}\ x := p \mid !\overline{o}.m(\overline{e}) \mid !\texttt{all} : I.m(\overline{e}) \\
s_{wless} & ::= & \texttt{send} \mid \texttt{receive} \\
s_{net} & ::= & \texttt{link}\ \overline{o}\ n\ \overline{o} \mid \texttt{unlink}\ \overline{o}\ n\ \overline{o}
\end{array}
$$

Fig. 2. The language syntax. Overlined terms such as $\overline{e}$, $\overline{x}$, and $\overline{s}$, denote lists over the corresponding syntactic categories and curly brackets denote optional elements. Additional constructs for low-level wireless programming are given by s_{wless}, and for network connections by s_{net}.

ing properties, immediateness of transmission, radio transmission synchronization and messages collision of wireless messages. Our model consists of three levels:

1. the *media layer* is represented by rules in the operational semantics, formalizing the transport of messages in the different nets and which is partly "programmable" in that **link** and **unlink** statements allow to establish and sever links between components, as well as synchronization of radio sending and receiving,
2. the *transport layer* is partly represented by rules of the operational semantics (formalizing the meaning of "tight" and "loose") and partly programmable by language primitives allowing; e.g., the programming of routing in wireless systems,
3. the *application level* represents top level programs. At this level the actual underlying net is invisible, in the sense that one may use the same high-level communication primitives, including broadcast and multicast primitives regardless of the actually used network.

The chosen primitives enable *cross-layer* designs of wireless network applications, which arise from the necessity to adapt properties of lower layers to the applications under design [23]. Such designs allow to adapt the network for better quality of service [19] or to optimize its energy consumption [10]. The framework may be adjusted to cater for other communication properties, such as message loss and packet size.

3 A Modeling Language for Components in Heterogeneous Nets

We introduce an executable modeling language for components in heterogeneous networks, based on the object-oriented language Creol [11, 12]. Creol proposes imperative programming constructs for distributed concurrent objects, based on asynchronous

method calls and processor release points. Asynchronous method calls may be seen as triggers of concurrent activity, resulting in new processes in the called object. Objects are dynamically created instances of classes, which are organized in an inheritance hierarchy. Concurrent objects encapsulate an execution thread and an internal process queue. Active behavior, triggered by a *run* method, is interleaved with passive behavior by means of the processor release points. The modeling language includes a standard expression language for values of basic data types, which will not be explained in detail. Objects have unique identities (names); communication takes place between named objects, and object identities may be exchanged between objects. Object variables are typed by interfaces. The language is strongly typed: invoked methods are supported by the called object (when not *null*), such that formal and actual parameters match.

In contrast, the modeling language considered in this paper targets network components in heterogeneous networks. It combines object-oriented components and definitions of actual networks; including tight, loose and wireless networks, and dynamic network changes. Technically, we extend the concurrent object communication model of Creol with representations of components and networks, incorporate a notion of (local and global) time, and introduce multicasts and forms of broadcast. Finally we define primitives for dealing with the special needs of wireless networks inside this model.

Network components. In order to model the units of the heterogeneous network, we introduce a light-weight notion of multi-object *network components*. The objects inside a component are tightly connected and communicate directly with each other. A component supports all interfaces supported by its objects; thus the caller may a call a method on a component if the called method supported by some object in that component. In case several objects in the component support the called method, one of these is chosen non-deterministically. However, if the caller knows the identity of a preferred object inside the component, the caller may call that object directly.

Component creation has the syntax $x := \mathbf{new}\ C(\overline{e})$ where C is the class name and $\overline{e}$ the list of actual class parameters, if any. In fact, this statement creates a network component consisting of a single object. Both components and objects have identity: for an object identity o, the expression $\mathrm{component}(o)$ gives the component identity of o (for a component identity c, $\mathrm{component}(c) = c$). The statement $x := \mathbf{new}\ C(\overline{e})\ \mathbf{in}\ o$ creates a new (object) instance of C *inside* the component o.

Basic statements. The basic language syntax is given in Fig. 2. A program consists of interface and class declarations. Classes CL contain definitions of attributes x (with initial values) and methods M. A method contains a list of statements s, which may access class attributes, locally defined variables, and the formal parameters of the method (given by the keywords **in** and **out**). An interface IF contains method signatures Sg associated with a *cointerface* I, denoting the (minimal) type of a client of IF (given by a **with** clause). Both classes and interfaces may also contain parameters and inherit other classes and interfaces, respectively. Finally, a class implements a list of interfaces. In order to allow type correct call-backs, a method may use the implicit *caller* parameter, which supports the cointerface of the method. Input parameters, as well as the self-reference *this*, are ready-only. Note that remote attribute access is not permitted, so method interaction is the only means of communication in the language. Assignment

and **if**- and **while**-constructs are standard. We assume that purely local operations take no time; local delays may be captured by the statement **tick**(n), for n time units.

In the statement **await** g, the guard g is used to control processor release and may consist of Boolean conditions and return tests (see below). If g evaluates to false in the active process, the process is *suspended* and the execution thread becomes idle. When the execution thread is idle, any enabled process may be chosen from the local process queue. Therefore explicit signaling is not part of the language. The *run* method of an object is called upon creation, and initiates active behavior. Release points in the run method allow processes in the process queue to be handled.

Communication. After making an asynchronous method call $t := !o.m(\overline{e})$, the caller may proceed with its execution without waiting for the method reply. Here o is an object expression and $\overline{e}$ are (data value or object) expressions. The tag t will be assigned a unique tag value identifying the call (relative to the current object), which may later be used to refer to that call in two different ways. First the guard **await** $t?$ suspends the active process unless a return to the call associated with t has arrived. Second the return values are accessed by the blocking *reply statement* $x := t.$**get**, once a return has arrived. We identify certain special cases of these communication primitives: For local calls the dot-notation and o is omitted; e.g., $t := !m(\overline{e})$. If no return value is desired by the caller, the tag may be omitted; e.g., $!o.m(\overline{e})$. The sequence $t := !o.m(\overline{e})$; $x := t.$**get** gives a *blocking call*, abbreviated $x := o.m(\overline{e})$, whereas the call sequence $t := !o.m(\overline{e})$; **await** $t?$; $x := t.$**get** gives a *non-blocking call*, abbreviated **await** $x := o.m(\overline{e})$. A *multicast* $!\overline{o}.m(\overline{e})$ is an asynchronous method call with a set of target objects. The multicast is sent simultaneously to all callees. A *broadcast* **all** $: I.m(\overline{e})$ is an asynchronous method invocation which targets all objects of a certain interface I. For strong typing, I must provide a declaration of the invoked method. The use of these communication primitives is illustrated below and in Sect. 4.

Heterogeneous nets. In a given model, the network connecting the components need not be uniform. Actual nets are defined by means of a number of direct *links* between components, which may have different characteristics. In this paper, we consider three basic forms of links: *wireless*, *loose*, and *tight*. In order to model the heterogeneous net, links are declared by the statements **link** $\overline{o}$ **wless** $\overline{u}$, **link** $\overline{o}$ **loose** $\overline{u}$, and **link** $\overline{o}$ **tight** $\overline{u}$ (for sets of component expressions $\overline{o}$ and $\overline{u}$). These statements respectively add **wless**, **loose**, or **tight** links from each each component in $\overline{o}$ to each component in $\overline{u}$. Correspondingly, links are explicitly broken by, e.g., the statement **unlink** $\overline{o}$ **wless** $\overline{u}$. Remark that (wired) loose or tight links go both ways, but this is not generally the case for wireless links. Furthermore, links to self are redundant.

Example 1. Initial links may be made inside the `run` method of a class `System` from which the initial components are created.

```
op run ==
  var a,b,c,d : Any;
  a:= new Class1; b:= new Class2; c:= new Class3; ...
  link a wless b,c;
  link b,c,d wless a,b,c,d;
```

Here a must use b or c to communicate with d, but d may communicate directly with a.

Wireless communication. A wireless component needs radio functionality to handle the sending and receiving of messages. In order to control the timing of this sending and receiving precisely, we model the radio functionality in a separate radio object in the wireless component. Therefore a component acting in wireless media will consist of at least two active objects; the main processing unit and the radio object. It is the task of the radio object to make wireless messages available for regular processing by the objects inside the component. The objects themselvesact as if they work in a wired network, using the standard primitives for communication. A component acting in a wireless net must be able to wait for and to send a message in a given time interval. We use two explicit non-blocking primitives to capture wireless sending and receiving: **`receive`** to receive a wireless message and make it available to the component's objects, and **`send`** to send the first pending wireless message.

Example 2. A cycle of the radio unit of a wireless component could be to receive, then send, receive again, and finally sleep. Instead of defining a controller in the sensor's central processing unit, we use the radio's *run* method to control the cycle.

```
class Radio(sendtime:Nat, sleeptime:Nat, cycle:Nat, sync:Nat)
  implements Controllable
begin var on:Bool := true, timer: Nat := 0
  op run == while on do
    await (clock - sync) rem cycle = 0; *** synchronize
    timer:= clock;
    while clock < timer + sleeptime do
     if clock = timer + sendtime then send else receive fi od od
with Any
  op turnoff == on := false
  op turnon == on := true
  op reset (in time: Nat) == sync := time
  op setSend (in time: Nat) ==
         if time < sleeptime then sendtime := time fi
  op setSleep(in time:Nat) ==
         if sendtime<time<cycle then sleeptime := time fi
end
```

When the radio is turned on, the cycle consists of an active phase where the radio is sending in a specified interval (here of length 1 time unit) and otherwise receiving, followed by a sleeping phase. In addition there are methods to turn the radio on and off, to adjust the sending and sleeping intervals, and for synchronizing the radio cycle. These methods form an interface, *Controllable*, allowing external control of the radio. When sleeping, the processor is released and invocations of the radio methods may be processed. For simplicity, we here assume a fixed cycle length (set at creation time).

4 Example: A Model of a Wireless Sensor Network

A typical biomedical sensor network consists of a number of sensors, a sink, and users. The example in Fig. 3 has five sensors and one sink connected by wireless links. The sink sends signals which are sufficiently strong for all sensors to receive them. The sensors, which could be inside patients, run on battery, and save power by reducing their signal strength, which again limits their range. Hence some sensors are not directly connected with the sink and depend on other sensors to forward their messages. The sink

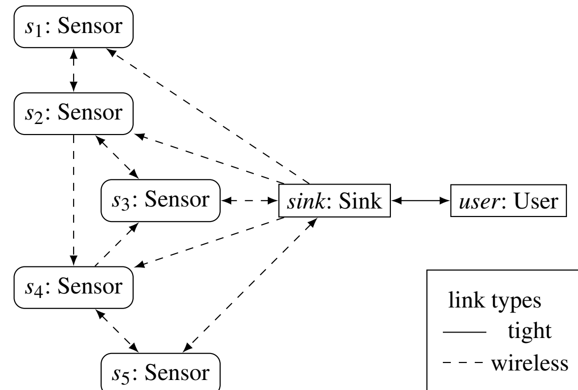

Fig. 3. Typical sensor network.

is connected to the end users by a tight network. The interfaces of the system components are given in Fig. 4. The `Forwarder` interface declares a forward operation, and is inherited by both the `Sink` and the `Sensor` interfaces. Methods of `Forwarder` have `Forwarder` as cointerface (given by the **`with`** clause) because a `Forwarder` object should communicate with another `Forwarder` object. The `Sensor` interface adds a method for updating the distance to the sink, which is accessible only to sink objects, as `Sink` is the cointerface. The `User` interface declares a `newData` method for receiving data from the sink.

The `run` method of the `System` class constructs the system model by creating all objects and setting up the initial network (not given here). A sensor component is created as in

```
s1:= new TempSensor(10); r1:= new Radio(t1,6,10,1) in component(s1);
```

and similarly for the sink component, ensuring that the sending intervals of the different radios are disjoint, by appropriate radio parameter values. It also reconfigures the network at runtime to simulate the patients' movements. For example, we express that sensor `s5` moves too far to reach the sink after 200 time units by severing the link:

```
await clock > 200; unlink s5 wless sink;
```

The sensors, sink, and users are described by the classes `TempSensor`, `Sink`, and `User` in Fig. 4. The sensors operate in cycles with a period given by the class parameter `interval`. In each cycle a sensor reads the current temperature by calling an instance of the `TempMeter` interface, using a non-blocking call. The `TempMeter` interface models access to the hardware.

After reading the temperature, the sensor sends a message with the temperature to all reachable components that implement the `Forwarder` interface by the statement

```
!all:Forwarder.forward(this, distToSink + 2, 1, clock, temp)
```

Here, *reachable* means that there is a direct link from the caller to the callee. If there is more than one link from the two components, the best link is selected. If the link is tight,

```
interface Forwarder begin with Forwarder op forward(...) end
interface Sensor inherits Forwarder
   begin with Sink op setDistToSink(...) end
interface Sink inherits Forwarder begin ... end
interface User begin with Sink op newData(...) end

class TempSensor(interval:Nat) implements Sensor
begin var forwarded:List[Oid⋆Nat] := emp, distToSink:Nat := 10,
      distUpdTime:Nat := 0, timer:Nat, temp:Int, tempm:TempMeter
  op run == tempm := new TempMeter in component(this);
    while true do timer := clock; await temp := tempm.getTemp();
     !all:Forwarder.forward(this, distToSink + 2, 1, clock, temp);
     await clock > timer+interval od
 with Sink
  op setDistToSink(in time:Nat, dist:Nat) ==
    if distUpdTime < time ∨(distUpdTime = time ∧ dist < distToSink)
    then distToSink := dist; distUpdTime := time fi
 with Forwarder
  op forward(in origin:Oid,htl:Nat,steps:Nat,timestamp:Nat,data:Int)==
    if not((origin, timestamp) in forwarded) ∧ origin ≠ this ∧
      htl > distToSink then if length(forwarded) > 9
         then forwarded := after(forwarded, length(forwarded)-9) fi;
     forwarded := forwarded ⊢(origin, timestamp);
     !all:Forwarder.forward(origin, htl-1, steps+1, timestamp, data) fi
end

class Sink() implements Sink
begin var forwarded:List[Oid⋆Nat⋆Int] = emp
 with Forwarder
  op forward(in origin:Oid, htl:Nat, steps:Nat,
                timestamp:Nat, data:Int) ==
    !origin.setDistToSink(timestamp, steps);
    if not((origin, timestamp) in forwarded) then
      if length(forwarded) > 9
       then forwarded := after(forwarded, length(forwarded)-9) fi;
      forwarded := forwarded ⊢(origin, timestamp);
      !all:User.newData(origin, timestamp, data) fi
end

class User(criticalLow:Nat, criticalHigh:Nat) implements User
begin var allData: List[Oid⋆Nat⋆Nat⋆Int] := emp;
  op alarm()...
 with Sink
  op newData(in origin:Oid, timestamp:Nat, data:Int) ==
     var i:Nat := 1;
     if data < criticalLow ∨data > criticalHigh then alarm() fi;
     while i <= length(allData)
          ∧ index(index(allData, i),1) ≠ origin do i := i+1 od;
     while i <= length(allData)
          ∧ index(index(allData, i),1) = origin
          ∧ index(index(allData, i),2) < timestamp do i := i+1 od;
     allData :=
       insertAtIndex(allData, i, (origin, timestamp, clock, data))
end
```

Fig. 4. Model of the temperature sensor, the sink, and a user. We here use $\vdash$ for list append.

the message will move directly from the caller's out-queue to the callee's in-queue. If the link is wireless then the radio unit transports the message.

In class `Sink`, the `forward` method has the following parameters: `origin` gives the identity of the sensor that has provided the data; `htl` (hops to live) gives the number of remaining hops the message should live; `steps` gives the number of hops this message has taken so far; `timestamp` stores the time when the origin sent this message; and `data` is the temperature measured in degrees centigrade. When a sensor receives a `forward` call, it adds the message to `forwarded` and forwards the call to all reachable forwarders, unless the message has already forwarded. The length of `forwarded` is limited to ten entries, a common limit of, e.g., biomedical sensors.

The distance to the sink from a sensor may be measured by the minimum number of hops needed. Because sensors may move, this distance may change. When the sink gets a message, it sends the number of hops taken by this message to the original sender, using `setDistToSink`. If the data are new to the sink, they are broadcasted to all users. The user stores data in a list `allData`, which is sorted by sensor name and sending time to simplify queries. Observe that none of the remote calls made in any class are blocking; consequently, the system is deadlock free (assuming that local calls always terminate).

5 Operational Semantics

The operational semantics of the language is defined using rewriting logic (RL) [15]. A *rewrite theory* is a 4-tuple $\mathscr{R} = (\Sigma, E, L, R)$, where the signature Σ defines the function symbols of the language, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. A state configuration in RL will be modeled as a multiset of terms representing local system states, of given types. These types are specified in (membership) equational logic (Σ, E), the functional sublanguage of RL which supports algebraic specification in the OBJ [9] style. RL extends algebraic specification techniques with transition rules: The dynamic behavior of a system is captured by rewrite rules, supplementing the equations which define the term language. Assuming that all terms can be reduced to normal form, rewrite rules transform terms modulo the equations of E. A rewrite rule $t \longrightarrow t'$ **if** c may be seen as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t', where the optional condition c is a conjunction of rewrites and equations which must hold for the main rule to apply. If several rules can be applied to distinct subconfigurations, they can be executed in a *concurrent rewrite step*. As a result, concurrency is implicit in rewriting logic semantics. Rules in RL may be formulated at a high-level of abstraction, similar to a compositional operational semantics. In fact, RL provides a semantic framework unifying equational and operational semantics [16]. Many concurrency models have been successfully represented in RL [5, 15]; including Petri nets, CCS, Actors, and Unity. RL also offers its own model of object orientation [5].

In RL, objects are commonly represented by terms $\langle o : C \,|\, a_1 : v_1, \ldots, a_n : v_n \rangle$ where o is the object's identity, C is its class, the a_i's are the names of the object's fields, and the v_i's are the corresponding values [5]. We adopt this form of presentation and define the elements of our semantics as RL objects. When auxiliary functions are needed,

these are defined in equational logic and evaluated in between transitions [15]. White-space is used as the associative and commutative constructor of multisets with identity element empty, whereas semicolon is used as the associative constructor of lists, also with identity element empty. Variables of the operational semantics are written in upper case letters, whereas variables of the modeling language (as well as auxiliary functions) are written in lower case letters. As before, variables for lists or multisets are written $\overline{M}$ for semantic construct M.

5.1 Configurations, Local and System Transitions

The modeling language considered in this paper depends on a notion of time. 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* [1] based on a global clock, which allows us to ignore clock synchronization between objects. In this clock model, the clock value is just a number that serves to group simultaneous events. 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 disappear as soon as time advances. Local clocks coordinate the objects' behavior with the global progress of time, enforcing the invariant that an object may only make a step when the local time is less than or equal to the global time. The global clock advances as soon as its value is less than the values of all local clocks.

A state *configuration*, of sort *Configuration*, is a multiset which consists of objects, classes, interfaces, queues, messages, and links. The empty configuration is denoted *empty*. A basic link is written $[O\ N\ O']$, where O and O' range over object identities and N over networks (i.e., wless, tight, and *lossy*). In order to capture the global clock in RL, we let a term $@C$ clock$(N)@$ of sort *System* include a configuration C with at least one object and a (global) clock, denoted *clock*(N), where the variable N ranges over natural numbers. There are three different kinds of rewrite rules:

- *Code execution rules* correspond to the different program statements;
- *Transport rules* move messages between objects, components, and the network;
- *System level rules* manage low-level activities such as global clock update and table lookup for classes and interfaces

Remark that code execution and transport rules apply to local configurations and allow concurrent execution, whereas system level rules apply to the whole system.

Components consist of tightly connected objects, and are represented by a naming discipline: a component name may be extracted from every object identity by a function *component* : $Oid \rightarrow Cid$, where the component name is of sort *Cid* (a subsort of *Oid*). The objects with the same component name form a component. From outside a component, one may then refer to an object by its identity or by its component name. A component O has one *in-queue* object $\langle O$: InQu | EvQ: $\overline{M}\rangle$, where the queue $\overline{M}$ is FIFO ordered, and one *out-queue* object $\langle O$: OutQu | EvQ: $\overline{M'}$, Tag: $K\rangle$, where $\overline{M'}$ has a simple form of priority ordering and the tag K (together with the object identity) is used to uniquely identify outgoing messages. (Other forms of priority queues could be

considered in more specialized settings; e.g., LIFO out-queues would give priority to fresh messages.) The queues have the same name as the component, and provide (a controlled form of) shared data structures for the component's objects. The queues may interact with the net at the same time as internal actions inside the component's objects. The specific message processing depends on the different networks linked to an object. Remark that by using the same component name for all objects in a component, we need no further encapsulation syntax for components. In many cases, including the examples in this paper, full object names are not needed and component names suffice.

A *concurrent object* is represented by $\langle O : C \mid \text{Pr: } Q, \text{PrQ: } \overline{Q}, \text{Att: } \overline{V}\rangle$, where O is the object identity, of sort Oid, C the class name, Pr the active process (which includes code and local variables), PrQ a multiset of suspended processes with unspecified queue ordering, and Att the object state variables, including some predefined system variables such as $clock$ which represents the *local clock* of the object. A *process* is modeled as a pair consisting of code and local state, $(\overline{S}, \overline{W})$, where $\overline{S}$ is a statement list and $\overline{W}$ is a state mapping from (local) variable names to values, using + for concatenation (and overwriting) and $_ \mapsto _$ for constructing variable-to-value associations. The suspended processes in the process queue represent remaining parts of method activations. Programs have read-only access to the clock, so the programmer may not assign to the clock variable. Let $[\![E]\!]_{\overline{V}}$ denote the evaluation of an expression E in the state $\overline{V}$.

Example. A wireless sensor may have one radio object $O2$ responsible of sending and receiving wireless messages in interaction with the network, together with a main object $O1$ doing the main computations. Such a component may have the form:

$\langle O : \text{InQu} \mid \text{EvQ: } \overline{M}\rangle \; \langle O : \text{OutQu} \mid \text{EvQ: } \overline{M}', \text{Tag: } K\rangle$
$\langle O1 : C \mid \text{Pr: } Q_1, \text{PrQ: } \overline{Q}_1, \text{Att: } \overline{V}_1\rangle \; \langle O2\text{: Radio} \mid \text{Pr: } Q_2, \text{PrQ: } \overline{Q}_2, \text{Att: } \overline{V}_2\rangle$

where component(O1) = component(O2) = O, and a class $Radio$ defines active behavior controlling the wireless sending and receiving of messages (see Section 4).

In a *class* $\langle C : \text{Cl} \mid \text{Ifc: } \overline{I}, \text{Inh: } \overline{C}, \text{Par: } \overline{Y}, \text{Att: } \overline{V}, \text{Mtds: } \overline{P}\rangle$, C is the class name, Ifc is the list of interfaces supported by the class, Inh is the list of superclasses, Par the list of class parameters, Att a list of attributes with initial values, and $Mtds$ a multiset of methods (including the initialization method $init$ and a method run defining active object behavior). The attributes include a system variable $token$ used for unique naming of generated objects. When an object needs a method, it is loaded from the $Mtds$ multiset of the object's class. Similarly, in an *interface* $\langle I : \text{Ifc} \mid \text{Inh: } \overline{I}\rangle$ I is the name, $\overline{I}$ the inherited interfaces. The inheritance list is used for broadcasts at run-time to determine all (connected) objects of a given super-interface. Method and cointerface declarations in an interface are used for type checking purposes and may be ignored at run-time.

Heterogeneous networks are represented by sets of links. We let $Link$ be a subsort of $Config$, thereby allowing (multi)sets of links directly in the configuration, with terms $[O\ N\ O']$ (where N denotes a net; i.e., either wless, tight, or loose). We assume that the transmission strength in a wireless link may vary, in contrast to a wired link. Consequently, wireless links are directed and not symmetric, whereas wired connections are both symmetric and transitive. In the multiset of links, duplicates as well as links to self are ignored (i.e., we have the equations $CN\ CN = CN$ and $[O\ N\ O]\ \overline{U} = \overline{U}$, where CN denotes some link and O a component). The link statements described in Sect. 3 result in changes of link configurations.

We define a function bestcon : Oid Oid Configuration $\rightarrow$ Net^{+} to identify the best connection between two objects, exploiting the transitivity and symmetry of wired networks. The sort Net^{+} extends the sort *Net* with the constant noNet and we define bestcon$(O, O', \overline{U})$ = noNet if there is no connection path from (the component of) O to (the component of) O'. Otherwise, the connection between the two objects is tight if there is a connection path from O to O' consisting of tight direct connections only; loose if there is one or more loose direct connections in the path; and wless if there is a wireless connection between O and O'. Objects in the same component are always tightly connected:

$$\text{bestcon}(O, O', \overline{U}) = \text{tight } \textbf{if } \text{component}(O) = \text{component}(O')$$

For example, bestcon(o1, o3, [o1 wless o3][o1 tight o2][o3 loose o2]) = loose, whereas bestcon(o1, o3, [o1 wless o3][o1 tight o2]) = wless.

Messages. There are three different kinds of message bodies *MB*: these have the form *invoc* m(*par*) for *invocation messages*, where m is the name of the called method and *par* are actual parameters; *comp* (*par*) for *completion messages*; and *error* (*name*) for *error messages*, capturing network errors or other kinds of errors. The actual parameters include system generated parameters such as the *caller* identity and tag value. With full header information, a message has the form *MB* **from** O **to** $\overline{O}$ **by** *NET*, where O is the sender object, $\overline{O}$ the destination (either a single object or a list of objects), and *NET* is the network to be used: loose, tight, wless, or noNet. For simplicity, we omit sender information from messages inside out-queues and keep only message bodies inside in-queues. The network information of a message is determined when the message is placed in the out-queue (by means of an equation taking the total network into consideration). Remark that messages by noNet cannot be sent.

5.2 The Rewrite Rules

The operational semantics ensures that clock values increase, and that the global clock is less than or equal to each local clock. The global clock is updated by the rule (CLOCK) in Fig. 6, in which the variable $\overline{U}$ ranges over configurations, clockmin gives the smallest local clock value in $\overline{U}$, and *refresh* removes any remaining receive statements and wireless messages from the configuration (since such messages are not persistent). The function clockmin is defined by the following equations (where *OB* denotes an object):

$$\begin{aligned} \text{clockmin}(OB\ OB'\ \overline{U}) &= \min(\text{clockval}(OB), \text{clockmin}(OB'\ \overline{U})) \\ \text{clockmin}(OB\ \overline{U}) &= \text{clockval}(OB)\ \textbf{otherwise} \\ \text{clockval}(\langle O : C \mid \ldots \text{ Att: } \overline{V}\rangle) &= [\![\text{clock}]\!]_{\overline{V}} \end{aligned}$$

Note that in RL, equations marked by **otherwise** only apply when no other equations are applicable [5]. In general, an object may only compute when its local clock value equals the global clock. Thus a rule modeling object behavior typically has the form

$$\text{object}\ \ \text{clock}(T) \longrightarrow \text{object}'\ \text{clock}(T)\ \textbf{if}\ \text{clockval}(\text{object}) = T$$

where *object* is a pattern representing an object (possibly with its associated in-queue) and *object'* is the resulting object state, typically with local time increased. In particular

$$@\ \langle O:C\ |\text{PrQ: }\overline{Q}\rangle\ \langle O:\text{InQu}\ |\text{Ev: invoc m}(\overline{E});\overline{M}\ \rangle\ \overline{U}\ @$$
$$\text{(BIND)}\quad =@\ \langle O:C\ |\text{PrQ: }\overline{Q};\ \text{bind}(\text{m},\overline{E},\text{C},\overline{U})\ \rangle\ \langle O:\text{InQu}\ |\text{Ev: }\overline{M}\rangle\ \overline{U}\ @$$
$$\textbf{if}\ \ \text{supports}(\text{C},\ \text{m},\ \overline{U})$$

$$\langle O:\text{C}\ |\text{Pr: (await G ; }\ \overline{S},\overline{W}),\ \text{PrQ: }\overline{Q},\ \text{Att: }\overline{V}\ \rangle\ \langle O:\text{InQu}\ |\text{Ev: }\overline{M}\ \rangle$$
$$\text{(GUARD)}\quad \longrightarrow\langle O:\text{C}\ |\text{Pr: }(\overline{S},\overline{W}),\ \text{PrQ: }\overline{Q},\ \text{Att: }\overline{V}\ \rangle\ \langle O:\text{InQu}\ |\text{Ev: }\overline{M}\ \rangle$$
$$\textbf{if}\ \ \text{enabled}(\text{G},\ (\overline{V}+\overline{W}),\overline{M})$$

$$\langle O:\text{C}\ |\text{Pr: }(\overline{S},\overline{W}),\ \text{PrQ: }\overline{Q},\ \text{Att: }\overline{V}\ \rangle\ \langle O:\text{InQu}\ |\text{Ev: }\overline{M}\ \rangle$$
$$\text{(SUSPEND)}\quad \longrightarrow\langle O:\text{C}\ |\text{Pr: idle},\ \ \text{PrQ: }\overline{Q};(\overline{S},\overline{W}),\ \text{Att: }\overline{V}\ \rangle\ \langle O:\text{InQu}\ |\text{Ev: }\overline{M}\ \rangle$$
$$\textbf{if}\ \ \textbf{not}\ \ \text{enabled}(\overline{S},(\overline{V}+\overline{W}),\overline{M})$$

$$\langle O:\text{C}\ |\text{Pr: idle},\ \ \text{PrQ: }\ (\overline{S},\overline{W});\overline{Q},\ \text{Att: }\overline{V}\ \rangle\ \langle O:\text{InQu}\ |\text{Ev: }\overline{M}\ \rangle$$
$$\text{(PRQ-READY)}\quad \longrightarrow\langle O:\text{C}\ |\text{Pr: }(\overline{S},\overline{W}),\ \text{PrQ: }\overline{Q},\ \text{Att: }\overline{V}\ \rangle\ \langle O:\text{InQu}\ |\text{Ev: }\overline{M}\ \rangle$$
$$\textbf{if}\ \ \text{enabled}(\overline{S},(\overline{V}+\overline{W}),\overline{M})$$

$$\langle O:\text{C}\ |\text{Pr: idle},\ \ \text{PrQ: }\overline{Q},\ \text{Att: }\overline{V}\ \rangle\ \langle O:\text{InQu}\ |\text{Ev: }\overline{M}\ \rangle\ \text{clock(T)}$$
$$\text{(IDLESTEP)}\quad \longrightarrow\langle O:\text{C}\ |\text{Pr: idle, PrQ: }\overline{Q},\ \text{Att: advance}(\overline{V})\ \rangle\ \langle O:\text{InQu}\ |\text{Ev: }\overline{M}\ \rangle\ \text{clock(T)}$$
$$\textbf{if}\ \ \textbf{not}\ \ \text{enabled}(\overline{Q},\overline{V},\overline{M})\ \textbf{and}\ [\![\text{clock}]\!]_{\overline{V}}\ =\text{T}$$

Fig. 5. Rules for process queue handling. In the rules we omit object fields not relevant for the rule. Note that matching is modulo associativity, commutativity, and identity for the multiset constructor, and modulo associativity and identity for the list constructor.

the rule (IDLESTEP) in Fig. 5 increases the local time of objects that are *idle*; i.e., objects with no active process in which no processes in the process queue are enabled. Similarly, the rule (NO REPLY) in Fig. 6 increases local time when the active process is blocked (i.e., `x:=t.get`) with no matching label value in the in-queue. In addition, all communication statements increase the local clock (see Fig. 6); e.g., send, receive, method calls, return, link, and unlink.

The principle that local computation requires the local and global clock values to be the same, may be relaxed for *internal* object actions; i.e., by allowing local actions of object which do not involve any interaction (affected by or affecting other objects). For example, assignments to local and state variables do not need to depend on the global clock. Thus, the assignment rule, which may be given by

$$\langle O:C\ |\text{Pr: }(X{:=}E;\overline{S},\overline{W}),\ \text{Att: }\overline{V}\rangle$$
$$\text{(ASSIGN)}\quad \longrightarrow\ \ \textbf{if}\ \ X\ \textbf{in}\ \overline{V}\ \textbf{then}\ \langle O:C\ |\text{Pr: }(\overline{S},\overline{W}),\ \text{Att: }\overline{V}+(X\mapsto[\![E]\!]_{(\overline{W}+\overline{V})})\rangle$$
$$\textbf{else}\ \langle O:C\ |\text{Pr: }(\overline{S},\overline{W}+(X\mapsto[\![E]\!]_{(\overline{W}+\overline{V})})),\ \text{Att: }\overline{V}\rangle\ \textbf{fi}$$

does not increase the local time. The **`tick`**(n) statement evaluates as an assignment on the local clock; i.e., $clock := clock + n$. The rules for **`if`** and **`while`** (omitted here), as well as guards and process queue handling do not involve clocks, except (IDLESTEP) (given in Fig. 5). The (BIND) rule additionally uses the class hierarchy to bind methods, and the supports function checks if a class supports a method in a given class hierarchy.

To simplify, object names in the rules are abstracted to component names. This allows a direct matching by names ($O = O'$) rather than matching by component name (as in component(O) = component(O')). Note that method binding based on component names is non-deterministic when the component has several objects supporting the

Rule	Equation / rule
(CLOCK)	$@\ \text{clock}(T)\ \overline{U}\ @ \longrightarrow @\ \text{clock}(\text{clockmin}(\overline{U}))\ \text{refresh}(\text{T}, \overline{U})\ @$ $\textbf{if}\ \ T < \text{clockmin}(\overline{U})$
(NET)	$@\ \langle O : \text{OutQu}\ \vert\text{Ev: } \overline{M};\ (MB\ \textbf{to}\ O');\ \overline{M'}\ \rangle\ \overline{U}\ @$ $= @\ \langle O : \text{OutQu}\ \vert\text{Ev: } \overline{M};\ (MB\ \textbf{to}\ O'\ \textbf{by}\ \text{bestcon}(O,O',\overline{U}));\ \overline{M'}\rangle\ \overline{U}\ @$
(NONET)	$\langle O : \text{OutQu}\ \vert\text{Ev: } \overline{M};\ (MB\ \textbf{to}\ O'\ \textbf{by}\ \text{noNet})\rangle$ $= \langle O : \text{OutQu}\ \vert\text{Ev: } \overline{M};\ (\text{error(“noNet”)}\ \textbf{to}\ O\ \textbf{by}\ \text{tight})\rangle$
(MULTIMSG1)	$\langle O : \text{OutQu}\ \vert\text{Ev: } \overline{M};\ (MB\ \textbf{to}\ (O'\ ;\ \overline{O}))\rangle$ $= \langle O : \text{OutQu}\ \vert\text{Ev: } \overline{M};\ (MB\ \textbf{to}\ O');\ (MB\ \textbf{to}\ \overline{O})\rangle$
(MULTIMSG2)	$\langle O : \text{OutQu}\ \vert\text{Ev: } \overline{M};\ (MB\ \textbf{to}\ \text{empty})\rangle\ = \langle O : \text{OutQu}\ \vert\text{Ev: } \overline{M}\ \rangle$
(MULTIMSG3)	$@\ \langle O : \text{OutQu}\ \vert\text{Ev: } \overline{M};\ (MB\ \textbf{to}\ \text{all: } I)\rangle\ \overline{U}\ @$ $= @\ \langle O : \text{OutQu}\ \vert\text{Ev: } \overline{M};\ (MB\ \textbf{to}\ \text{all}(O,I,\overline{U}))\rangle\ \overline{U}\ @$
(MULTICAST)	$\langle O : C\ \vert\text{Pr: } (!\overline{O}.m(\overline{E})\ ;\ \overline{S},\overline{W}),\ \text{Att: } \overline{V}\ \rangle\ \langle O : \text{OutQu}\ \vert\text{Ev: } \overline{M},\ \text{Tag: } K\ \rangle\ \text{clock}(T)$ $\longrightarrow \langle O\text{: C}\ \vert\text{Pr: }\ (\overline{S},\overline{W}),\ \text{Att: advance}(\overline{V})\ \rangle$ $\langle O : \text{OutQu}\ \vert\text{Ev: } \overline{M};\ \text{invoc m}(O,K,[\![\overline{E}]\!]_{(\overline{V}+\overline{W})})\ \textbf{to}\ [\![\overline{O}]\!]_{(\overline{V}+\overline{W})}),\ \text{Tag: } K{+}1\ \rangle$ $\text{clock}(T)\ \textbf{if}\ \ [\![\text{clock}]\!]_{\overline{V}}\ = T$
(UNICAST)	$\langle O : C\ \vert\text{Pr: } (L{:=}!O.\text{m}(\overline{E})\ ;\ \overline{S},\overline{W}),\ \text{Att: } \overline{V}\ \rangle\ \langle O : \text{OutQu}\ \vert\text{Ev: } \overline{M},\ \text{Tag: } K\ \rangle\ \text{clock}(T)$ $\longrightarrow \langle O : C\ \vert\text{Pr: } (L := K;\ \overline{S},\overline{W}),\ \text{Att: advance}(\overline{V})\ \rangle$ $\langle O : \text{OutQu}\ \vert\text{Ev: } \overline{M};\ \text{invoc m}(O,K,[\![\overline{E}]\!]_{\overline{V}+\overline{W}})\ \textbf{to}\ [\![O]\!]_{\overline{V}+\overline{W}})),\text{Tag: } K{+}1\ \rangle\ \text{clock}(T)$ $\textbf{if}\ \ [\![\text{clock}]\!]_{\overline{V}}\ = T$
(BROADCAST)	$\langle O : C\ \vert\text{Pr: } (!\ \text{all: }\ I.\text{m}(\overline{E})\ ;\ \overline{S},\overline{W}),\ \text{Att: } \overline{V}\ \rangle\ \langle O : \text{OutQu}\ \vert\text{Ev: } \overline{M},\ \text{Tag: } K\ \rangle$ $\text{clock}(T)$ $\longrightarrow \langle O : C\ \vert\text{Pr: } (\overline{S},\overline{W}),\ \text{Att: advance}(\overline{V})\ \rangle$ $\langle O : \text{OutQu}\ \vert\text{Ev: } \overline{M};\ \text{invoc m}(O,K,[\![\overline{E}]\!]_{(\overline{V}+\overline{W})})\ \textbf{to}\ \text{all: } I,\ \text{Tag: } K{+}1\ \rangle\ \text{clock}(T)$ $\textbf{if}\ \ [\![\text{clock}]\!]_{\overline{V}}\ = T$
(RETURN)	$\langle O : C\ \vert\text{Pr: } (\text{return}(\overline{E})\ ;\ \overline{S},\overline{W}),\ \text{Att: } \overline{V}\rangle\ \text{clock}(T)\ \langle O : \text{OutQu}\ \vert\text{Ev: } \overline{M},\ \text{Tag: } K\ \rangle$ $\longrightarrow \langle O : C\ \vert\text{Pr: } (\overline{S},\overline{W}),\ \text{Att: advance}(\overline{V})\ \rangle\ \text{clock}(T)$ $\langle O : \text{OutQu}\ \vert\text{Ev: } \overline{M}\ ;\ \text{comp}\ ([\![(\text{label},\overline{E})]\!]_{\overline{V}+\overline{W}})\ \textbf{to}\ [\![\text{caller}]\!]_{\overline{W}}),\ \text{Tag: } K\ \rangle$ $\textbf{if}\ \ [\![\text{clock}]\!]_{\overline{V}}\ = T$
(REPLY)	$\langle O : C\ \vert\text{Pr: } (\overline{X} := L?\ ;\ \overline{S},\overline{W}),\ \text{Att: } \overline{V}\ \rangle\ \text{clock}(T)$ $\langle O : \text{InQu}\ \vert\text{Ev: } \overline{M};\ \text{comp}(K,\overline{E});\ \overline{M'}\ \rangle$ $\longrightarrow \langle O : C\ \vert\text{Pr: } (\overline{X} := \overline{E};\ \overline{S},\overline{W}),\ \text{Att: advance}(\overline{V})\ \rangle\ \text{clock}(T)$ $\langle O : \text{InQu}\ \vert\text{Ev: } \overline{M};\ \overline{M'}\ \rangle$ $\textbf{if}\ \ [\![\text{clock}]\!]_{\overline{V}}\ = T\ \textbf{and}\ K = [\![L]\!]_{(\overline{V}+\overline{W})})$
(NO REPLY)	$\langle O : C\ \vert\text{Pr: } (\overline{X} := L?\ ;\ \overline{S},\overline{W}),\ \text{Att: } \overline{V}\ \rangle\ \text{clock}(T)\ \langle O : \text{InQu}\ \vert\text{Ev: } \overline{M}\ \rangle$ $\longrightarrow \langle O : C\ \vert\text{Pr: } (\overline{X} := L?\ ;\ \overline{S},\overline{W}),\ \text{Att: advance}(\overline{V})\ \rangle\ \text{clock}(T)\ \langle O : \text{InQu}\ \vert\text{Ev: } \overline{M}\ \rangle$ $\textbf{if}\ \ [\![\text{clock}]\!]_{\overline{V}}\ = T\ \textbf{and not}\ \text{inqueue}([\![L]\!]_{(\overline{V}+\overline{W})},\overline{M})$

Fig. 6. Rewrite equations and rules for message processing.

method. With this simplification, all remote calls are made to components. For many practical purposes, including the sensor example, this simplification works well.

(Tight1)	$\langle O : \text{OutQu} \mid \text{Ev: } (MB\ \textbf{to}\ O'\ \textbf{by}\ \text{tight}); \overline{M}\rangle\ \langle O' : \text{InQu} \mid \text{Ev: } \overline{M'}\rangle$ $= \langle O : \text{OutQu} \mid \text{Ev: } \overline{M}\rangle\ \langle O' : \text{InQu} \mid \text{Ev: } \overline{M'}; MB\rangle$
(Tight2)	$(M1\ \textbf{to}\ \overline{O1}\ \textbf{by}\ \text{wless}); (M2\ \textbf{to}\ \overline{O2}\ \textbf{by}\ N)$ $= (M2\ \textbf{to}\ \overline{O2}\ \textbf{by}\ N); (M1\ \textbf{to}\ \overline{O1}\ \textbf{by}\ \text{wless})\ \textbf{if not}\ N\ = \text{wless}$
(Loose1)	$\langle O : \text{OutQu} \mid \text{Ev: } (MB\ \textbf{to}\ O'\ \textbf{by}\ \text{loose}); \overline{M}\rangle$ $= \langle O : \text{OutQu} \mid \text{Ev: } \overline{M}\rangle\ (MB\ \textbf{from}\ O\ \textbf{to}\ O'\ \textbf{by}\ \text{loose})$
(Loose2)	$(MB\ \textbf{from}\ O\ \textbf{to}\ O'\ \textbf{by}\ \text{loose})\ \langle O' : \text{InQu} \mid \text{Ev: } \overline{M}\rangle \longrightarrow \langle O' : \text{InQu} \mid \text{Ev: } \overline{M}; MB\rangle$

Fig. 7. Tight and loose networks

Basic Statements and Creation. The rules for basic statements, such as **`skip`**, **`if`**, **`while`**, **`link`**, and **`unlink`**, are straightforward (see Fig. 9). The rule for object creation creates a new object and associated queues. In case of a new object in a given component, the component identity is reused for the new object and no further queues are created. Local calls are defined by remote calls to self (by the obvious equation). For simplicity, the rules for reentrance are ignored in this paper.

Network processing. The rewrite rules for network processing are given in Fig. 6. In the equation (Net), the network determines how to send messages. Notice that the equation is on the total system, such that all possible connections are considered. The equation represents network actions, realized by hardware or the operating system. Equation (noNet) reflects that messages to noNet cannot be sent. Such messages may represent serious communication failures and should be communicated to the caller. In our framework this is indicated by an *error* message. Message sending to multiple destinations is defined by means of messages to single destinations in equation (MultiMsg1), and by ignoring messages sent to empty destination lists in (MultiMsg2). The rules for *tight* and *loose networks* are given in Fig. 7. A tight network from o to o' is defined by a transport equation (Tight1) which takes the first message from the out-queue of o marked by "tight", directly into the in-queue of o'. We let wired messages in out-queues have priority over wireless ones, as stated in equation (Tight2). Loose networks are modeled by using an equation to move messages from an out-queue into the configuration, and by a (nondeterministic) rule taking a message from the configuration into the appropriate in-queue ((Loose2)). Messages over loose nets are automatically sent to the network (i.e., placed in the configuration multiset) by the equation (Loose1).

Communication. Messages in the out-queue are created by call and return statements in the associated objects. We consider uni-cast, multicast, and broadcast; the rules are given in Fig. 6. Of these, only *labeled uni-casts* allow the caller to request a result of the call. We have seen that blocking (synchronous) methods calls are understood in terms of labeled asynchronous method calls: $t := !o.m(\overline{e})$ where o is an object expression. The label value provides a way of identifying the call and the reply. In rule (Unicast), a labeled call has only one callee, which ensures that there is a unique reply. A call's result value is communicated in rule (Return) as a completion message, caused by a *return* statement in the callee, where *caller* and *label* are the implicit local parameters identifying the caller and the tag. A blocking reply statement is captured by the rule

(WSEND1)	$\langle O : C \mid$Pr: (send ; $\overline{S}, \overline{W}$), Att: $\overline{V}\rangle$ clock(T) $\langle O$: OutQu $\mid$Ev: (MB **to** O' **by** wless) ; $\overline{M}\rangle$ $= \langle O : C \mid$Pr: ($\overline{S}, \overline{W}$), Att: advance($\overline{V}$) $\rangle$ clock(T) $\langle O$: OutQu $\mid$Ev: $\overline{M}\rangle$ (MB **from** O **to** O' **by** wless) **if** $[\![\text{clock}]\!]_{\overline{V}} = T$
(WSEND2)	$\langle O : C \mid$Pr: (send ; $\overline{S}, \overline{W}$), Att: $\overline{V}\rangle$ clock(T) $\langle O$: OutQu $\mid$Ev: empty $\rangle$ $\longrightarrow \langle O : C \mid$Pr: ($\overline{S}, \overline{W}$), Att: advance($\overline{V}$) $\rangle$ clock(T) $\langle O$: OutQu $\mid$Ev: empty $\rangle$ **if** $[\![\text{clock}]\!]_{\overline{V}} = T$
(COLLIDE)	($M1$ **from** $O1$ **to** O **by** wless) ($M2$ **from** $O2$ **to** O **by** wless) = (error(“collision”) **from** null **to** O **by** wless)
(RECEIVE)	@ $\langle O : C \mid$Pr: (receive ; $\overline{S}, \overline{W}$), Att: $\overline{V}\rangle$ $\langle O$: InQu $\mid$Ev: $\overline{M}\rangle$ (M **from** O' **to** O **by** wless) [O’ wless O] clock(T) $\overline{U}$ @ = @ $\langle O : C \mid$Pr: ($\overline{S}, \overline{W}$), Att: advance($\overline{V}$) $\rangle$ $\langle O$: InQu $\mid$Ev: $\overline{M}$; $M\rangle$ [O’ wless O] clock(T) $\overline{U}$ @ **if** $[\![\text{clock}]\!]_{\overline{V}} = T$ **otherwise**
(REFRESH1)	refresh (T, (MB **from** O **to** O' **by** wless) $\overline{U}$) = refresh (T, $\overline{U}$)
(REFRESH2)	refresh (T,$\langle O : C \mid$Pr: (receive; $\overline{S}, \overline{W}$), Att: $\overline{V}\rangle$ $\overline{U}$) $= \langle O : C \mid$Pr: ($\overline{S}, \overline{W}$), Att: advance($\overline{V}$)$\rangle$ refresh($T, \overline{U}$) **if** $[\![\text{clock}]\!]_{\overline{V}} = T$
(REFRESH3)	refresh (T, $\overline{U}$) = $\overline{U}$ **otherwise**

Fig. 8. Wireless communication.

(REPLY), when the corresponding completion has arrived. Otherwise, the rule (NO REPLY) ensures that the local clock progresses. A similar rule ensures the progress of the local clock when an object is idle without any enabled process in the queue.

Unlabeled calls may have multiple destinations, given by a list $\overline{O}$ of object expressions. These are captured by rule (MULTICAST), where *advance*($\overline{V}$) is defined by $\overline{V}$ + (clock $\mapsto [\![\text{clock}]\!]_{\overline{V}} + 1$). We let semicolon denote both concatenation and append, for sequences of statements as well as of messages. Notice that the caller O and tag-value K are added as implicit parameters. *Broadcast*, captured by the rule (BROADCAST), is restricted to all objects of a given interface, using the notation *all* : I. This restriction is needed to maintain strong typing. A broadcast message MB **to** *all* : I should arrive (in a single copy) to all I-objects in the system which are connected to O. The *all* : I expression is expanded by means of the equation (MULTIMSG3) on the total system, using a function *all* to collect all objects of interface I (or better) in a configuration:

all ($O, I, \overline{U}$ $\langle O' : C \mid ...\rangle$) = O'; all($O, I, \overline{U}$)
  if supports ($C, I, \overline{U}$) **and** bestcon($O, O', \overline{U}$) $\neq$ noNet
all ($O, I, \overline{U}$) = empty **otherwise**

Here, supports ($C, I, \overline{U}$) checks whether class C implements interface I, or a subinterface of I, in the configuration $\overline{U}$.

5.3 Wireless Sending and Receiving

The two language primitives **`send`** and **`receive`** model synchronized wireless communication. Their operational semantics, given in Fig. 8, depends on the refresh func-

tion, used in the (CLOCK) rule, which erases wireless messages and receive statements as time passes. The equation (WSEND1) defines the semantics of sending; the first wireless message in the out-queue is sent, making the **from**-part of the message header explicit. When there is no message to send, a send message is skipped in rule (WSEND2). Recall that wireless messages in a network (i.e., configuration) disappear when global time advances. This is captured by an equation (REFRESH1) on the *refresh* function used in the global clock rule. Two wireless messages with the same destination which occur in the configuration at the same time, cause a collision and destroy each other. We model this by an equation (COLLIDE), which results in an *error* message. The equation detects collisions of two or more messages. The receiving itself is modeled by an equation (RECEIVE). Here, the **otherwise**-clause implies that the equation should have lower priority than (COLLIDE). Therefore, the left hand side considers the system state and includes all possible matches of the (COLLIDE) rule. The rule also checks that there is a wless connection, since it may have been disconnected after the message was sent. By advancing the local clock, we ensure that two wireless messages cannot be received at the same time.

Notice that the condition on the clocks implies that the receiving of a message happens at the same time as its sending, thereby modeling *synchronous transmission*. Recall that *refresh* applies whenever the global local is advanced, which should also happen when the local time of receiving objects is equal to the global time. We therefore add an equation for this case

$$\text{clockval}\,(\langle O : C \mid \text{Pr: (receive ; } \overline{S}, \overline{W}\text{), Att: } \overline{V}\rangle) \;=\; [\![\text{clock}]\!]_{\overline{V}} + 1$$

and otherwise use the previous equation for clockval.

5.4 Simulation and search

The operational semantics outlined above is executable on the RL platform Maude [5], and thus form the basis for an analysis tool for the modeling language of Sect. 3. We illustrate this by showing how to perform some simple simulations of the wireless sensor network example of Sect. 4 in Maude.

The model of Sect. 4 focuses on the behavior of the components. A special class `System` is used to set up the system model by creating components and establishing the links between components. The initial `System` object can later be used to modify these links, thus changing the topology of the network.

Interesting behaviour of the communication medium may be investigated by simulating the system given in Fig. 4; e.g., can message overtaking occur in this net? By message overtaking, we mean that a message arrives at the user before an older one from the same sender. This may happen, if an additional link from the sensor s_1 to the `sink` in the network of Fig. 3 is introduced at runtime and the interval between two measurements is sufficiently short. To exhibit this behaviour, the following code is added to the `run` method of the `System` class:

```
await (clock > int(25)); connect s1 wless sink;
```

Now the link between s_1 and the `sink` is added when the local clock has reached 25 time units. Simulating the system will then show that a later message may overtake an

(LINK)

$$\langle \text{O : C} \mid \text{Pr: } (\overline{W}, (\text{link E NW E'}); \overline{S}), \text{Att: } \overline{V}\rangle$$
$$\longrightarrow \langle \text{O :C} \mid \text{Pr: } (\overline{W}, \overline{S}), \text{Att: } \overline{V}\rangle$$
$$\text{component}([\![E]\!]_{\overline{V}+\overline{W}}) \text{ NW component}([\![E']\!]_{\overline{V}+\overline{W}})$$

(UNLINK)

$$\langle \text{O : C} \mid \text{Pr: } (\overline{W}, (\text{unlink E NW E'}); \overline{S}), \text{Att: } \overline{V}\rangle \text{ [O' NW O'']}$$
$$\longrightarrow \langle \text{O :C} \mid \text{Pr: } (\overline{W}, \overline{S}), \text{Att: } \overline{V}\rangle$$
$$\textbf{if } \text{component}([\![E]\!]_{\overline{V}+\overline{W}}) = \text{ O' } \textbf{and} \text{ component}([\![E']\!]_{\overline{V}+\overline{W}}) = \text{ O''}$$

(IF-EL)

$$\langle \text{O : C} \mid \text{Pr: } (\overline{W}, \text{if E then } \overline{S1} \text{ else } \overline{S2} \text{ fi ; } \overline{S}), \text{Att: } \overline{V}\rangle$$
$$\longrightarrow \textbf{if } [\![E]\!]_{\overline{V}+\overline{W}})) \textbf{ then } \langle \text{O :C} \mid \text{Pr: } (\overline{W}, \overline{S1} \text{ ; } \overline{S}), \text{Att: } \overline{V}\rangle$$
$$\textbf{else } \langle \text{O : C} \mid \text{Pr: } (\overline{W}, \overline{S2} \text{ ; } \overline{S}), \text{Att: } \overline{V}\rangle \textbf{ fi}$$

(WHILE)

$$\langle \text{O : C} \mid \text{Pr: } (\overline{W}, \text{while E do } \overline{S1} \text{ od ; } \overline{S2})\rangle$$
$$\longrightarrow \langle \text{O :C} \mid \text{Pr: } (\overline{W}, (\text{if E then}$$
$$(\overline{S1} \text{ ; while E do } \overline{S1} \text{ od) else skip fi}); \overline{S2})\rangle$$

(NEW1)

$$@ \langle \text{O :C} \mid \text{Pr: } (\overline{W}, (\text{X :=new C' } (\overline{E}); \overline{S})), \text{Att: } \overline{V}\rangle\ \overline{U}\ @$$
$$\longrightarrow @ \langle \text{O :C} \mid \text{Pr: } (\overline{W}, (\text{X :=O'}); \overline{S}), \text{Att: } \overline{V} + (\text{token} \mapsto [\![\text{token}]\!]_{\overline{V}} + 1)\rangle\ \overline{U}$$
$$\langle \text{O' : C'} \mid \text{Pr: init (C' } [\![\overline{E}]\!]_{\overline{V}+\overline{W}}, \overline{U}), \text{PrQ: empty,}$$
$$\text{Att: } (\text{clock} \mapsto [\![\text{clock}]\!]_{\overline{V}}) + (\text{this} \mapsto \text{O'}) + \text{inherit(C' } ([\![\overline{E}]\!]_{\overline{V}+\overline{W}}, \overline{U}))\rangle$$
$$\langle \text{O' : InQu} \mid \text{Ev: noMsg}\rangle \langle \text{O' :OutQu} \mid \text{Tag: 1, Ev: noMsg}\rangle\ @$$
$$\textbf{if } \text{O' := newId}(O, [\![\text{token}]\!]_{\overline{V}})$$

(NEW2)

$$@ \langle \text{O :C} \mid \text{Pr: } (\overline{W}, (\text{X :=new C' } (\overline{E}) \textbf{ in } \text{E}); \overline{S}), \text{Att: } \overline{V}\rangle\ \overline{U}\ @$$
$$\longrightarrow @ \langle \text{O :C} \mid \text{Pr: } (\overline{W}, (\text{X :=O'}); \overline{S}), \text{Att: } \overline{V} + (\text{token} \mapsto [\![\text{token}]\!]_{\overline{V}} + 1)\rangle\ \overline{U}$$
$$\langle \text{O' : C'} \mid \text{Pr: init (C' } [\![\overline{E}]\!]_{\overline{V}+\overline{W}}, \overline{U}), \text{PrQ: empty,}$$
$$\text{Att: } (\text{clock} \mapsto [\![\text{clock}]\!]_{\overline{V}}) + (\text{this} \mapsto \text{O'}) + \text{inherit(C' } ([\![\overline{E}]\!]_{\overline{V}+\overline{W}}, \overline{U}))\rangle\ @$$
$$\textbf{if } \text{O' := component}([\![E]\!]_{\overline{V}+\overline{W}})$$

Fig. 9. Rules concerning basic statements and object creation. The function newId is used to create new object identities (composed by the parent object and a counter). The auxiliary functions inherit and init are used to define multiple inheritance and initial code. An if-clause of the form $O' := E$ represents a let expression.

earlier message, because the earlier message is waiting to be forwarded at sensor s_2, while the later message is sent directly to the sink via the newly established link.

Maude also allows to search in a breadth-first manner through all possible executions from a given initial state. The state space of concurrent and distributed systems is huge, usually growing exponentially in the number of components. In order to make searching feasible, abstractions that reduce the state space are needed, preferably by eliminating components. For example, objects of class `TempSensor` may be replaced by equations for forwarding messages, moving these from in-queues to out-queues while updating the "step" and the "hops to live" attributes. This way, simple searches about communication patterns may be performed while abstracting from the internal functionality of the sensors.

6 Related and Future Work

Our approach is based on modelling with active objects. Active objects have been used to model mobile ad-hoc networks, which are similar to our biomedical sensor networks, in [7]. However, in contrast to our work, cross-layer design is not considered, because no means for reasoning about the network are provided.

Formal automata models have been used analyse protocols and channels. The properties of communication media are usually modelled as automata, too. For example, Nancy Lynch models communication media by processes in [14]. A lossy channel is modeled by a process that randomly drops messages. In contrast to these approaches, which apply ad-hoc techniques to model various kinds of links and networks, our modelling language fully integrates into the modelling language a set of primitives to describe dynamically evolving network topologies.

TinyOS [6] is a popular operating system for wireless sensor nodes. The associated programming language nesC [8] takes an approach similar to ours: Programs in nesC are structured in components. However, the number of components in nes C is statically fixed and each component ressembles a single Creol objects. In contrast, our components may be created dynamically and contain a number of concurrent objects. In nesC tasks correspond to our processes and are cooperatively scheduled, because sensor nodes usually do not permit dynamic scheduling. In contrast, our approach abstracts from particular scheduling schemes; in fact, our models could be refined with application-specific schedulers (see [21]). This may be starting point for a development technique for applications which target TinyOS. We are currently investigating the relationship between our models and nesC programs in more detail.

For the analysis of networks, the current state of the art focuses on discrete event simulation software, such as OMNet++ [25], that defines accurate models of wireless communication networks and channels. These simulators target the *quantitative* aspects of the model, such as throughput figures, whereas we are mainly concerned with *functional* aspects, e.g., the correctness of the deployed protocol and sensor functionality.

Verisim [2] is a simulator similar to OMNet++, which is used to validate functional properties of wireless networks. Verisim allows models from discrete event simulations to be used directly, and integrates well with established design methods. Monte-Carlo simulation is used to record traces of events, which may be queried using a special language. In contrast, our approach is based on a simple, high-level modeling language with a formal semantics. Furthermore, the integration with Maude makes it possible to customize the simulation strategy, as well as to apply search techniques to the models.

Ölveczky and Thorvaldsen [18] have shown how Real-Time Maude [17] can be applied to model and analyse advanced wireless sensor network algorithms, using, e.g., Monte Carlo simulations for performance evaluation for networks with up to 800 nodes. Rodriguez [20] has similarly used Real-Time Maude to analyse flooding in WSN protocols. These papers focus on the modeling and analysis of protocol algorithms. Our work complements this approach by emphasising sensor functionality and behavior, as well as heterogeneous media. However, we intend to investigate how their techniques for simulation may apply in our setting.

Compared to earlier work on Creol [11, 12], the main contribution of this paper is the extension to heterogeneous networks and the introduction of language abstractions

suitable for a unified modeling of network components and different kinds of networks. In particular, the extension consists of the notion of *network components* with tightly connected objects sharing in- and out-queues, specification of different and dynamic networks architectures, including *wireless networks* and *radio programming*, multi- and broadcasts, as well as the extension to a *timed semantics*. The proposed primitives are useful to establish connections at the network level but may also be exploited at the application level, for instance in service-oriented architectures [4].

Since our approach allows the radio level to be programmed inside the modeling language, we may experiment with different radio solutions. For instance, the active radio object used in the example of Sect. 4 may be replaced by a passive radio model. This can be done by letting the sensor class inherit a passive radio class, and letting the sensor object control the radio sender and receiver. Furthermore our approach may be adjusted to allow more realistic models of wireless communication by considering factors like battery capacity, power consumption of sending and receiving, signal strength, and location. Stochastic modeling, however, is less trivial, but might be addressed using the probabilistic Maude tool [13] as a basis for the operational semantics.

7 Concluding Remarks

The main contribution of this paper is a modeling framework for heterogeneous networks. The framework allows the unified modeling of network components in different kinds of networks, as well as network changes. In particular, we consider wireless networks and radio communication, as well as loosely and tightly connected wired networks. Our approach extends the object-oriented paradigm by suggesting novel language abstractions related to heterogeneous networks, using Creol as an underlying language for concurrent and distributed objects. The extended language may be used for high-level application programming (without knowledge of the particular network available) as well as for network-aware programming such as radio controllers. Our framework is based on formal methods and may serve as a basis for reasoning about system properties and semantical analysis. The formal semantics of the language is presented through a high-level operational semantics, defined in rewriting logic. The operational semantics is executable, allowing simulation and formal analysis by means of the Maude tool. The language is demonstrated through an example of a wireless sensor network, and some initial simulations and analysis have been performed.

The value of a formal framework as a basis for intuitive understanding of a language and towards practical modeling and reasoning, depends crucially on the simplicity of the semantics. The presented operational semantics consists of 32 rules or equations, apart from auxiliary function definitions. This covers the semantics of basic statements and object-oriented issues such as object creation, inheritance, late binding, and (asynchronous) method calls and replies, as well as extensions for heterogeneous networks, including broad- and multicast, timing (with global and local clocks), programming and re-programming of network links, and primitives for wireless receiving and sending. The semantical simplicity may also be taken as an argument for the appropriateness of the proposed abstractions and their integration within the object-oriented paradigm.

The long term goal of this work is to adapt the object-oriented paradigm to the setting of modern distributed systems by exploring suitable language abstractions and constructs that at the same time support simplicity both in reasoning and in semantics. The present paper may be seen as a first step in the direction of high-level, object-oriented, and formal modeling of heterogeneous systems where properties of the different networks are directly modeled.

Acknowledgments. We are grateful for comments by Wolfgang Leister and Xuedong Liang on sensor network modeling and analysis.

References

1. R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In J. W. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice, REX Workshop 1991*, *LNCS* 600, pages 74–106. Springer, 1992.
2. K. Bhargavan, C. A. Gunter, M. Kim, I. Lee, O. Sokolsky, and M. Viswanathan. Verisim: Formal analysis of network simulations. *IEEE Transaction on Software Engineering*, 28(2):129–145, Feb. 2002.
3. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
4. D. Clarke, E. B. Johnsen, and O. Owe. Concurrent Objects à la Carte, *Correctness, Concurrency, and Components: Festschrift for Willem-Paul de Roever*. To appear in LNCS, Springer, 2008.
5. 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.
6. D. E. Culler, J. L. Hill, P. Buonadonna, R. Szewczyk, and A. Woo. A network-centric approach to embedded software for tiny devices. In T. A. Henzinger and C. M. Kirsch, editors, *Proc. 1st Intl. Workshop on Embedded Software (EMSOFT'01)*, *LNCS* 2211, pages 114–130. Springer, 2001.
7. J. Dedecker and W. V. Belle. Actors for mobile ad-hoc networks. In L. T. Yang, M. Guo, G. R. Gao, and N. K. Jha, editors, *Proc. Intl. Conf. on Embedded and Ubiquitous Computing (EUC'04)*, *LNCS* 3207, pages 482–494. Springer, 2004.
8. D. Gay, P. Levis, J. R. von Behren, M. Welsh, E. A. Brewer, and D. E. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. Conf. on Programming Language Design and Implementation (PLDI'03)*, pages 1–11. ACM, 2003.
9. J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, Advances in Formal Methods, chapter 1, pages 3–167. Kluwer Academic Publishers, 2000.
10. A. J. Goldsmith and S. B. Wicker. Design challenges for energy-constrained ad hoc wireless networks. *IEEE Wireless Communications*, 9(4):8–27, Aug. 2002.
11. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
12. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.
13. N. Kumar, K. Sen, J. Meseguer, and G. Agha. A rewriting based model for probabilistic distributed object systems. In E. Najm, U. Nestmann, and P. Stevens, editors, *Proc. 6th IFIP Intl. Conf. on Formal Methods for Open Object-based Distributed Systems (FMOODS'03)*, *LNCS* 2884, pages 32–46. Springer, Nov. 2003.

14. N. A. Lynch. *Distributed Algorithms*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, Inc., 1996.
15. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
16. J. Meseguer and G. Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In D. A. Basin and M. Rusinowitch, editors, *Proc. 2nd Intl. Joint Conf. on Automated Reasoning (IJCAR 2004)*, *LNCS* 3097, pages 1–44. Springer, 2004.
17. P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285(2):359–405, Aug. 2002.
18. P. C. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude. *Theoretical Computer Science*, 2008. To appear.
19. V. T. Raisinghani and S. Iyer. Cross-layer design optimizations in wireless protocol stacks. *Computer Communications*, 27(8):720–724, May 2004.
20. D. E. Rodríguez. On modelling sensor networks in Maude. In G. Denker and C. Talcott, editors, *Proc. 6th Intl. Workshop on Rewriting Logic and its Applications (WRLA 2006)*, volume 176 of *Electronic Notes in Theoretical Computer Science*, pages 199–213. Elsevier, July 2007.
21. R. Schlatte, B. Aichernig, F. de Boer, A. Griesmayer, and E. B. Johnsen. Testing concurrent objects with application-specific schedulers. In J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigun, editors, *Proc. 5th Intl. Colloquium on Theoretical Aspects of Computing (ICTAC'08)*, *LNCS* 5060, pages 319–333. Springer, Aug. 2008.
22. G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.
23. V. Srivastava and M. Motani. Cross-layer design: A survey and the road ahead. *IEEE Communications Magazine*, 43(12):112–119, Dec. 2005.
24. A. U. Stephen J. Mellor, Kendall Scott and D. Weise. Model-driven architecture. In J.-M. Bruel and Z. Bellahsene, editors, *Advances in Object-Oriented Information Systems*, *LNCS* 2426, pages 233–239, 2002.
25. A. Varga. Omnet++. *IEEE Network Interactive*, 16(4), July 2002.
26. H. Zimmermann. OSI reference model—the ISO model of architecture for open system interconnection. *IEEE Transactions on Communication*, 28(4):425–432, Apr. 1980.