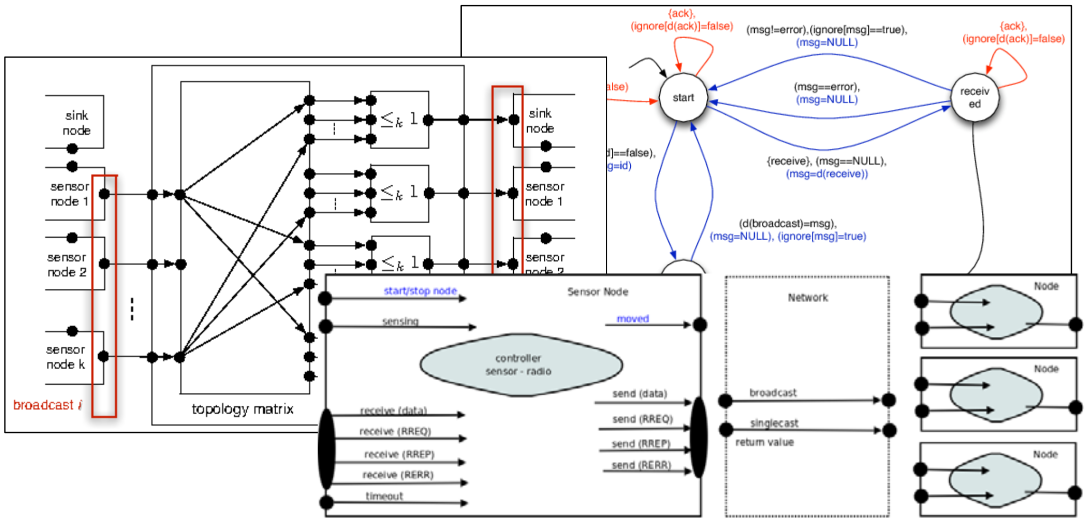
Norsk Regnesentral
NORWEGIAN COMPUTING CENTER

Report

Modelling of Biomedical Sensor Networks using the Creol Tools



CREDO

Report no 1022

Authors Wolfgang Leister, Xuedong Liang, Sascha Klüppelholz, Joachim Klein, Olaf Owe, Fatemeh Kazemeyni, Joakim Bjørk, Bjarte M. Østvold

Date 31th July 2009

ISBN 978-82-539-0532-7

About the authors

Wolfgang Leister is Chief Research Scientist at the Norwegian Computing Center. Bjarte M. Østvold is Senior Research Scientist at the Norwegian Computing Center. Sascha Klüppelholz and Joachim Klein are PhD students at the Faculty of Computer Science at the Technische Universität Dresden. Xuedong Liang and Fatemeh Kazemeyni are PhD students at the Rikshospitalet University Hospital (RRHF) and at the Institute of Informatics at the University of Oslo. Joakim Bjørk is PhD student at the Institute of Informatics at the University of Oslo. Olaf Owe is professor at the Institute of Informatics at the University of Oslo.

Norsk Regnesentral

Norsk Regnesentral
Norwegian Computing Center
Postboks 114, Blindern
NO-0314 Oslo, Norway

Besøksadresse
Office address
Gaustadalléen 23
NO-0373 Oslo, Norway

Telefon · telephone
(+47) 22 85 25 00
Telefaks · telefax
(+47) 22 69 76 60

Internett · internet
www.nr.no
E-post · e-mail
nr@nr.no

Title	**Modelling of Biomedical Sensor Networks using the Creol Tools**
Authors	**Wolfgang Leister (NR), Xuedong Liang (UiO/RRHF), Sascha Klüppelholz (TUD), Joachim Klein (TUD), Olaf Owe (UiO), Fatemeh Kazemeyni (UiO/RRHF), Joakim Bjørk (UiO), Bjarte M. Østvold (NR)**
Quality assurance	Einar Broch Johnsen (UiO), Frank de Boer (CWI)
Date	31th July 2009
Year	2009
ISBN	978-82-539-0532-7
Publication number	NR-Report No. 1022

Abstract

This document is submitted as Annex 6.3.3 for the Deliverable D6.3 (Final Modelling of Biomedical Sensor Networks using the Creol Tools) of the EU project IST-33826 CREDO: *Modeling and analysis of evolutionary structures for distributed services*. We describe how to model biomedical sensor networks using the CREDO methodology and the Creol Tools, specifically Creol, Vereofy, and UPPAAL. Properties from forwarding and routing as well as from the link- and network layers are put into focus. Modelling the AODV algorithm is presented in-depth in this document. The outcome of this work will be used for the validation phase of CREDO to judge the suitability of the Creol Tools.

Keywords	Creol, formal methods, modelling, biomedical sensor networks
Target group	Researchers, Developers of Creol Models
Availability	open
Project number	CREDO - 320362
Research field	Formal Methods
Number of pages	37
© Copyright	Norsk Regnesentral

Final Modelling of Biomedical Sensor Networks using the Creol Tools

Wolfgang Leister[1], Xuedong Liang[2,4], Sascha Klüppelholz[3], Joachim Klein[3], Olaf Owe[4], Fatemeh Kazemeyni[4], Joakim Bjørk[4], and Bjarte Østvold[1]

[1] Norsk Regnesentral, Oslo, Norway
[2] Rikshospitalet University Hospital, Norway
[3] Fakultät Informatik, TU Dresden, Germany
[4] Institute of Informatics, University of Oslo, Norway

1 Introduction

The generic architecture of a biomedical sensor network (BSN) is shown in Fig. 1, where each shaded element corresponds to one sensor node. The node n_1 reveals its internal structure, which consists of a radio object r, a controller object c, and (in our example) two sensor objects s_1 and s_2.

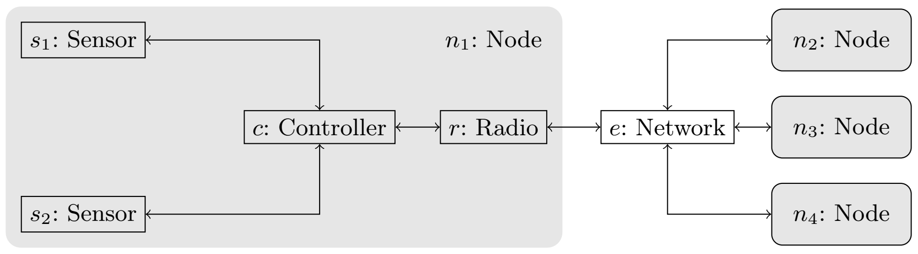


Fig. 1. Architecture of a sensor node and its relation to other nodes

The controller object c maintains the main activity of a node n_i. c reads data from the sensors, collects these readings into packets. The controller object c also receives messages from the radio and processes these. The processed packets are forwarded to the radio r for transmission via the Network e. Forwarding and routing behaviour of a node is modelled in the controller c.

The *Network* e between the sensor nodes models different aspects of (wireless) networks. In this object the communication properties between the nodes n_i are modelled. In our model the network contains a connection matrix which defines which note can reach which other node in a broadcast operation, i.e., the next hop for each node.

In order to forward packets or messages in a BSN from the source node to the sink node different strategies can be used. In the following we will look closer into models of *flooding* and the *routing protocol* AODV. Routing protocols are used to build up routing tables that are used by the nodes to forward messages to the next hop.

2 Modelling of Flooding

Flooding is a simple forwarding strategy where each node that wants to forward a message broadcasts this message to its neighbours, i.e., to the nodes that are reachable in one hop. Each message carries an unique identification, which is used to check whether this message already has been handled on this node. Messages that have not been seen on a node before are broadcast further to all neighbours, while the others are dropped. When a message reaches its destination this event is registered.

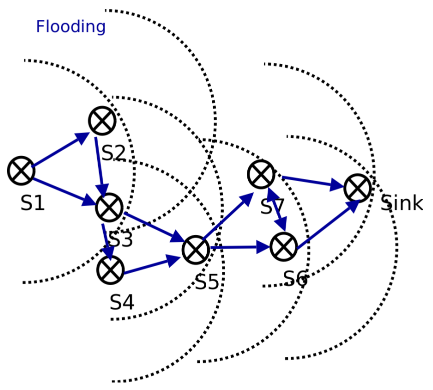


Fig. 2. Example of a sensor network with eight nodes showing the possible paths from $S1$ to the Sink.

Models of the flooding strategy are presented in the following. We use these models as a basis for further modelling efforts for more advanced protocols, like AODV described in Section 5. As an illustration we show an example of a network of sensor nodes in Figure 2 with the possible pathes of a message sent from Sensor $S1$ to the Sink.

2.1 Flooding modelled in Creol

We modelled the flooding strategy in Creol using the interfaces which are shown simplified in Fig. 3. The objects used in this model are besides the *Main*-object several *Node*s connected through one *Network*.

Node Interface. Each node has an interface for sensing, which may be implemented as an internal call when the node is an active object (which is the case

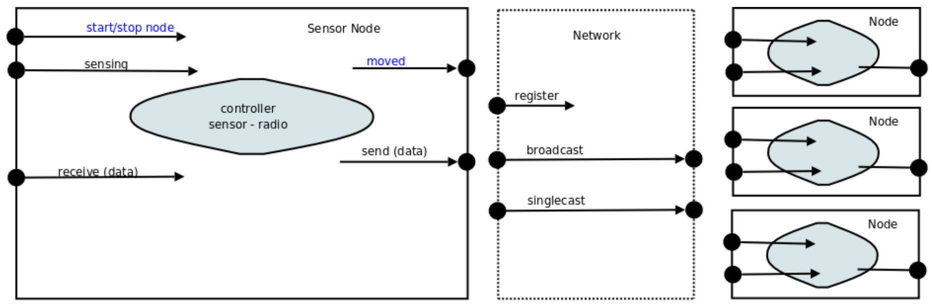


Fig. 3. Interfaces for flooding

in our model). For receiving messages from the network a call to receive broadcast messages is implemented which does not return its success. We also implemented a the reception of a singlecast message to demonstrate how to model whether the correct recipient receives a message. The sending operation of data is an internal call in the node; both broadcast and singlecast variants are implemented. Note, that for demonstration of the flooding strategy we only use the calls that implement broadcast. The external interface of a node looks as follows:

```
interface Node begin
  with Network
    op receiveBroadcast(in data: Message)
    op receiveSinglecast (in data: Message, rec: Int; out success: Bool)
end
```

Network Interface. The network includes interfaces for both broadcast and singlecast of messages. Additionally, for the setup of the direct connections between nodes of this network the method *register* is defined. Thus, the external interface of the network is as follows:

```
interface Node begin
  with Network
    op receiveBroadcast(in data: Message)
    op receiveSinglecast (in data: Message, rec: Int; out success: Bool)
end
```

Node implementation. The following snippet shows how the broadcast method is implemented in the flooding model. Using the list of connections the message is sent to all directly connected nodes.

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

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

 while $\neg$isempty(recs) do
   rec := head(recs);
   recs := tail(recs);
   if rec $\neq$ caller then
      rec.receiveBroadcast(data;)
   end
 end
```

Note that in this model all messages will arrive at all connected notes. In reality, different circumstances, for instance electrical noise, could prevent messages from arriving. Therefore, for model checking indeterminism could be applied, which would result in the following implementation (only last part presented):

```
[...]
 while $\neg$isempty(recs) do
   rec := head(recs);
   recs := tail(recs);
   if rec $\neq$ caller then
      rec.receiveBroadcast(data;) $\Box$ skip
   end
 end
```

Internally, in a node the incoming messages are processed by counting them when a message has arrived at the right recipient, else storing them in a buffer belonging to the node. This buffer is then inspected by a process in the node and forwarded to other nodes. Processing an incoming message is implemented as follows:

```
op processMessage(in data: Message) ==
   var theMessageType :Int;
   var psrc: Int; var ppld: Int;
   var p: [Int,Int];
   var plmdata: PayloadMessage;

   data.getMessageType(;theMessageType);
   if theMessageType = 1 then
     data.getPayloadMessage(;plmdata);
     plmdata.getSrcNode(;psrc);
     plmdata.getPayload(;ppld);
     p := (psrc,ppld);
     if $\neg$(p in reced) then
       reced := reced $\vdash$ p;
       store(data;)
```

```
    end
  end
```

The active process in a node for handling sensing or forwarding the stored messages from the buffer is implemented as a choice as follows:

```
 op run ==
    while true do
      await seqNo <noSensings; sense(;)
      □
      await #(stored) >0; sendOrForward(;)
    end
```

Modelling of messages. Since a model of flooding only contains only one message type, namely the payload, we can represent the message by one integer number. This was done in the early versions of the model. However, in order to be able to extend the model, we needed a more flexible representation in order to include more complex information into the messages, as explained in Section 5. Following the object-oriented paradigm we chose objects without an internal behaviour, comparable to structs in C/C++. We discuss the impact of this decision and alternatives later in Section 5.

We define a generic message which is forwarded in the network object, which also contains information about the link layer, i.e., the sending and receiving node of the current hop. Within the node objects we need access to the information of each message type. To specialise to a specific message object, we use the method *getPayloadMessage* which implements a typecasting pattern. The interfaces for messages are as follows:

```
interface Message
begin with Node
    op getSrcNode(out srcNode: Int)
    op getDstNode(out dstNode: Int)
    op getMessageType(out mt: Int)
    op getPayloadMessage(out m: PayloadMessage)
end

interface PayloadMessage inherits Message
begin with Node
    op getPayload(out payload: Int)
end
```

2.2 Extensions of the Flooding Models

An extended version of the flooding protocol has been modelled in Creol. In this model, we added the notion of distance between the nodes as well as power consumption of sending and receiving of the messages to the previous flooding model. We added the concept of position (altitude and latitude) to each node. The distance between each two nodes is calculated using these positions. When a node broadcasts a message, only the nodes that are within the valid distance range can receive that.

To consider the power consumption of nodes, we add the concept of power to each node. After each sending and receiving operation the total power of the node will be decreased by a predefined value.

The following snippet shows our modifications of the flooding model:

```
op send(in data: [Int,Int],  x_sender: Int,y_sender: Int) ==
   var l: Label[ ];
   power:=power − broadcast_power;
   l!network.broadcast(data,x_sender,y_sender)

 with Network
   op deposit (in data: [Int,Int],x_sender: Int,y_sender: Int) ==
   distance := (x − x_sender)∗(x − x_sender)−(y − y_sender)∗(y − y_sender);
   if (distance < tr) then
      power:= power − recieve_power;
      if ¬(data in reced) then
         !send(data, x_sender, y_sender);
         reced := reced ⊢ data
      end
   end
```

2.3 Flooding modelled in Vereofy

In this section we provide a brief overview on the Vereofy model for flooding in the BSN case study.

Data domain. As an abstraction we assumed that the data which should be transferred to the sink node corresponds to the ID of the originating sensor node. The data received by a sensor node A may be corrupted when collisions occur.

As depicted in Figure 4 the received data at a sensor node SN_1 is corrupted whenever two sensor nodes SN_2 and SN_3 which are both in sending reach of SN_1 broadcast a message at the same time. Other sensor nodes which are only in range of one of the broadcasting nodes will receive the uncorrupted message.

Thus, the global data domain for the flooding in the Vereofy main program corresponds to the set of sensor node IDs together with the ERROR_MSG which indicates a collision.

```
CONST NR_OF_SENSOR_NODES = 8;
TYPE Data = int(0, NR_OF_SENSOR_NODES);
CONST ERROR_MSG = NR_OF_SENSOR_NODES;
```

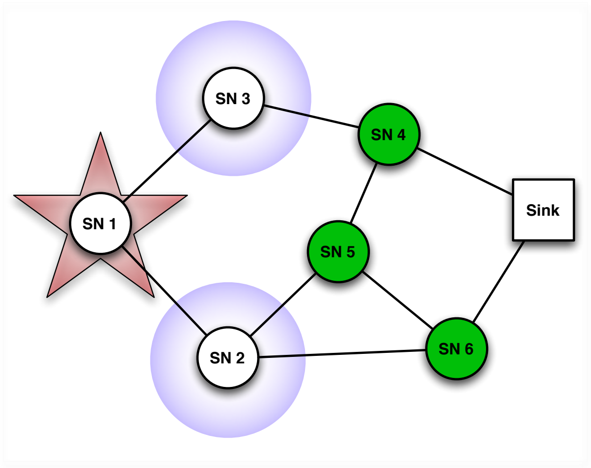


Fig. 4. Collisions

Sensor nodes. The prototype for a sensor node has two parameters. One for indicating the node ID and one for the encoding of corrupted messages. The sensor nodes consists of the sub-modules for sending and receiving messages. Both are modeled with the help of CARML. The resulting interface of a sensor node thus consists of a port for receiving messages, one port for broadcasting messages and a port for receiving acknowledgements from the sink.

```
MODULE sensor_node<id,error> {
 in: receive;
 in: ack;
 out: broadcast;

  // sub−modules for sending and receiving:
  ...
}
```

Sink node. Contrary to the sensor nodes the sink node does not send any data to other nodes. It simply receives messages via an input port and acknowledges the message via an output port.

```
MODULE sink_node{
  in: receive;
  out: send_ack;

  var: Data last_received := ERROR_MSG;
  var: enum{IDLE,BUSY} state :=IDLE;
  state==IDLE
    -[ {receive} & #receive!=ERROR_MSG ]→
  last_received:=#receive & state:=BUSY;

  state==BUSY
    -[ {send_ack} & #send_ack==last_received ]→
  state:=IDLE;
}
```

For the sink node we assume a reliable synchronous channel communication to each of the sensor nodes for sending acknowledgements. This is illustrated in Figure 5.

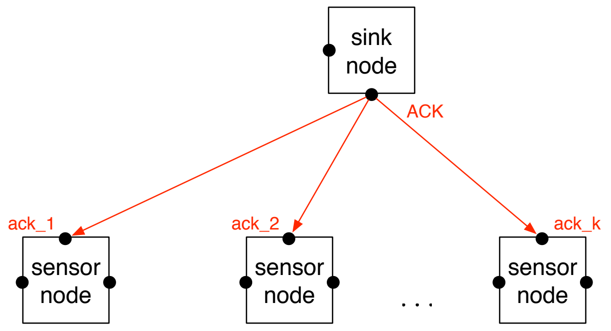


Fig. 5. The sink node with reliable communication

Thus, the sink node does not rely on the broadcast medium as it will be described in the next paragraph for sending the acknowledgements.

Broadcast medium. The network medium is modeled in RSL and composed out of several sub-networks; one for the topology which may dynamically change over time and one for the collision detection. Let from now on be k be the number of sensor nodes. Furthermore we identify the sink node with the sensor node with ID 0. Figure 6 shows how the broadcast medium is composed and how the sensor nodes are supposed to use the medium for sending and receiving messages. The sink node uses the medium for receiving only.

```
CIRCUIT topology_matrix<k>{
  // interface definition
  for (i=0;i<k;i=i+1){
    if (i>0){
      source[i-1] = NODE;
    }
    for (j=0;j<k;j=j+1){
```

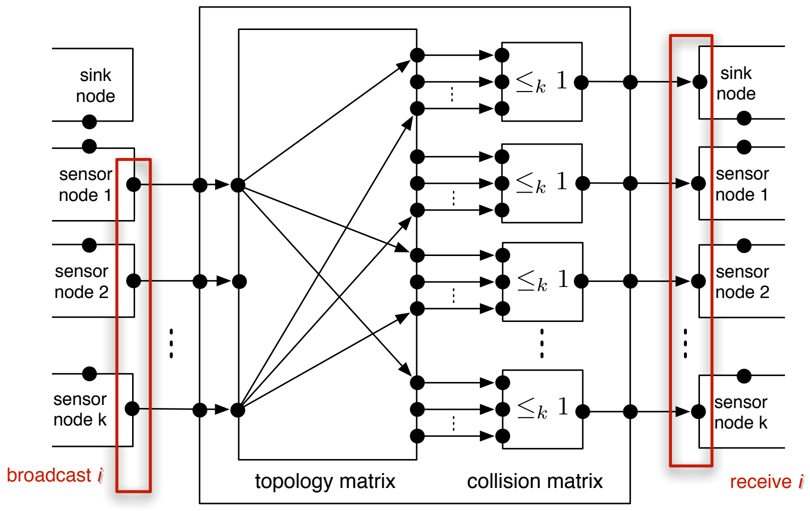


Fig. 6. The broadcast medium

```
      sink[(i∗(k)+j)]  = NODE;
    }
  }

  //  creating synchronous channels for each connection:
  ...
}
```

The *topology component* as it is composed by the RSL circuit above has k input ports and $(k+1)^2$ output ports. We require that there is a synchronous channel between input port i and the i-th input port of the collider j iff in the current topology sensor node i can reach sensor node j (including the sink). We used the same structure to model dynamically changing network topologies which may occur whenever nodes are added, power down, move to another location, or change their sending power. An example where the medium has two possible topologies is illustrated by the following RSL code.

```
  //  topology 0
  TOPO(0) = {
    for  (i=0;i<k;i=i+1){
      if  (i>1){ new SYNC(source[i];sink[(k)∗(i−2)+i]);  }
      if  (i>0){
        if  (i<k−2){
          new SYNC(source[i];sink[(k)∗(i+2)+i ]);
        }
      }
    }
    if (k>1){ new SYNC(source[2];sink[k+2]); }
    if (k>0){ new SYNC(source[1];sink[1]); }
  }

  // topology 1 has more connects:
  TOPO(1) = {
    if (k>2){ new SYNC(source[3];sink[(k*2)+3]);
      if (k>3){ new SYNC(source[3];sink[(k*4)+3]);
        if (k>5){ new SYNC(source[3];sink[(k*6)+3]); }
      }
    }
  }
```

The *collision matrix* consists of $k+1$ individual components; one for the sink and one for each sensor node. Each of the components behaves in the following way: Each collider always accepts data, i.e. the collision component is input enabled. If exactly one input is detected the data value will be passed to the output port. Whenever more than one output is detected the data item ERROR_MSG is written to its output port. The collision component for each sensor node can be composed of collision components of size 2 either by using linear or recursive composition. This is illustrated in Figure 7.

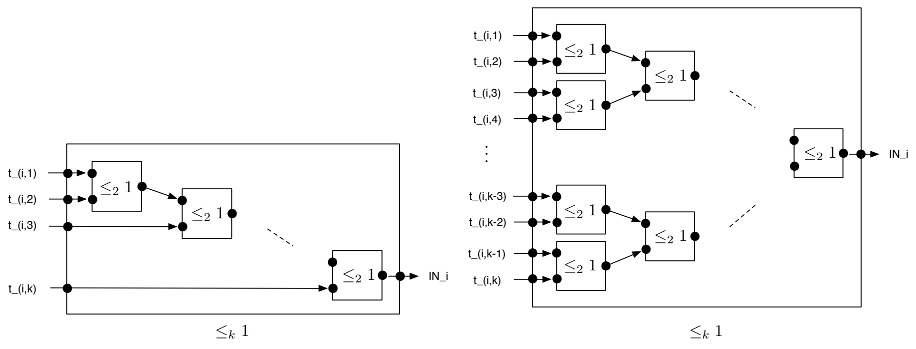


Fig. 7. Collision detection

Composite system. The composite system is then build using the following RSL script.

```
CIRCUIT main{
  // create medium
  med = new medium<NR_OF_SENSOR_NODES+1>;
  // build the sink node (sensor[0])
  // and the other sensor nodes (sensor[i])
  for (i=0;i≤NR_OF_SENSOR_NODES;i=i+1){
    if (i==0){
      sensor[0] = new sink_node;
      ACK = sensor[0].sink[0];
      receive[0] = sensor[0].source[0];
      join(med.sink[0], receive[0]);
    } else{
      sensor[i] = new sensor_node<i−1, ERROR_MSG>;
      ACK = join(ACK, sensor[i].source[1]);
      receive[i] = sensor[i].source[0];
      broadcast[i] = sensor[i].sink;
      join(broadcast[i], med.source[i]);
      join(med.sink[i], receive[i]);
    }
  }
}
```

3 UPPAAL Model focusing on Link- and Network Layer

A timed automaton is a finite state automaton extended with real-time clocks. UPPAAL is a tool box for timed automata, which provides a modelling language, a simulator and a model checker. In UPPAAL, timed automata are further extended with data variables of types such as integer and array etc., and networks of timed automata, which are sets of automata communicating with synchronous channels or shared variables, to ease the modelling tasks. The modelling language allows to define templates to model components that have the same control structure, but different parameters, which is a perfect feature for modelling of sensor nodes.

In this section, we develop a UPPAAL model for a biomedical sensor network (BSN), as a network of timed automata where each automaton models a sensor node. As all sensor nodes are implemented with the same chip for wireless communication, running the same protocol, we use a template to model the node behaviour with open timing parameters to be fixed in the validation phase. The network topology is modelled using a matrix declared as an array of integers in UPPAAL. Elements in the matrix denotes the connectivity between pairs of nodes.

Modelling the Chipcon CC2420 Transceiver. To study the network performance, we need to model only the transceiver of a sensor node for wireless communication. We assume that all sensor nodes use the Chipcon CC2420 transceiver. We model the transceiver as a UPPAAL template based on the radio control state machine of the transceiver, described in its reference manual.

The modelled template is shown in Fig. 8. Most of the states are of the same name as the radio control states in the original state machine for the transceiver. The functionality of the transceiver is modelled by the state transitions according to the reference manual.

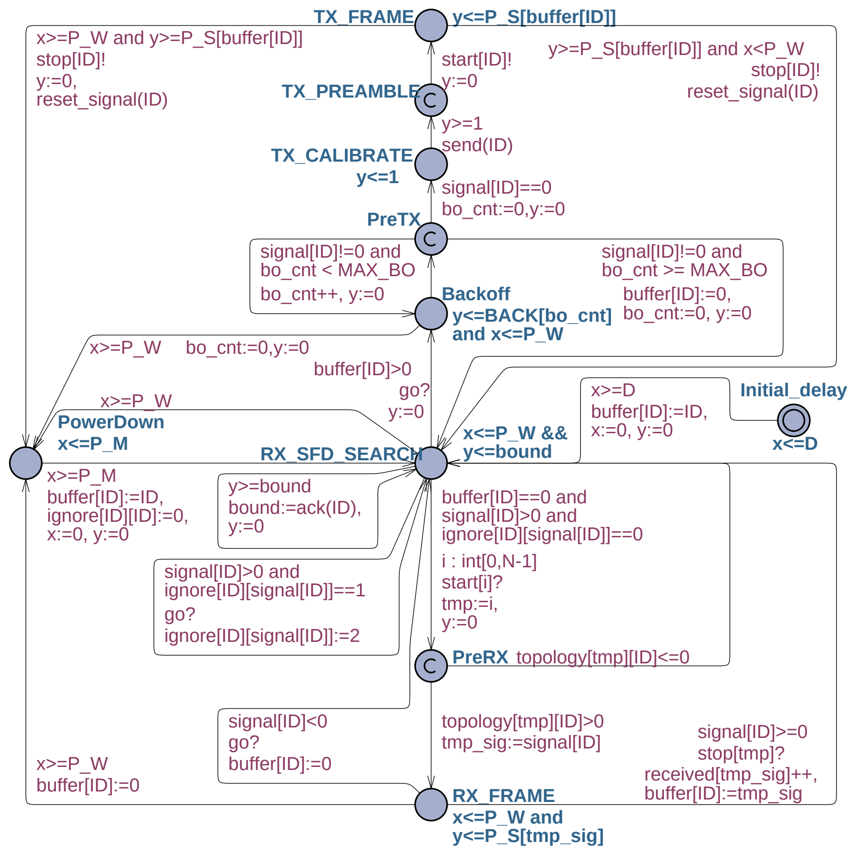


Fig. 8. A UPPAAL template for wireless sensor nodes based on the Chipcon CC2420 Transceiver

Modelling the Network and Communication. Now we describe how data packets are transferred between nodes and how errors are modelled, that may occur during packet transmission. The description is mainly on the global data variables used by the template automaton.

The network topology – the spatial distribution of the sensor nodes – represents the direct connections between the nodes. It is the task of the routing protocol to find a path for a packet from one node to the sink. We model the network topology using a matrix (`topology`) referred as topology matrix. The dimensions of this matrix correspond to the number of nodes in the network. Every element stands for the connectivity from one node (row index) to another (column index). If the matrix should map the topology, negative values can be used, for instance, to represent that a pair of nodes is not connected and positive values can reflect the distance or signal strength between the corresponding nodes. The matrix can also be used to store routing information. In this case, some values can stand for a connection, where a node is in range but not on a routing path.

Using the topology matrix, it is easy to model a fixed routing scheme. The matrix also allows us to model dynamic reconfigurations of the network topology due to the movement of a node or the change of routing information at runtime. To study dynamic reconfigurations, we have modelled controlled flooding which is a dynamic routing scheme. A node broadcasts a packet to all its neighbours and remembers every received packet to control this flooding. If a node receives a packet that has been forwarded earlier, it will be ignored, which avoids cyclic forwarding. The model contains a matrix (`ignore`) with which every node remembers the packets it has received so far. The same matrix is used to remember if an acknowledgement is expected or received. In addition to dynamic routing, the flooding scheme offers the opportunity for an implicit acknowledgement: when a node has transmitted a packet, it will most likely receive it again after a short while, because the receiver(s) will broadcast it again. When a defined time after transmission has passed, a node will call a function (`ack`) to check if a packet has to be retransmitted.

For simplicity, we abstract away from the contents of packets. Every node has an unique identifier and if a node emits a packet, it is named by the identifier of the node. The identifier is also used to determine the length of the packet (`P_S[ID]`). To transmit a packet, a node uses a function named `send`. The function walks through the topology matrix and updates the incoming signal of every node in range, where the incoming signals are modelled by an array named `signal`. Packet collisions that lead to packet losses are modelled with help of the signal array. If a node starts a transmission while another node in range is receiving a signal, the corresponding element in the signal array will be set to a negative value meaning that the packet is corrupted.

Based on the model, we have used UPPAAL to validate and tune the temporal configuration parameters of a BSN in order to meet desired QoS requirements on packet delivery ratio and network connectivity. The network studied allows dynamic reconfigurations of the network topology due to the physical movements of sensor nodes and also their temporally switching to the power-down mode for energy-saving.

Both the simulator and the model-checker of UPPAAL are used to analyse the average-case and worst-case behaviours. The simulator scales well; it can easily handle up to 50 nodes in our experiments. Even the model checker can handle networks with 5 up to 16 nodes depending on the properties to be checked; these are reasonable numbers for BSN applications in medical care.

Detailed information on the design of the timed automation model, simulation and verification environment settings and results can be found in [14,15].

4 An Object-Oriented Component Model for Heterogeneous Nets

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 synchronised, since the wireless medium cannot buffer messages. We proposed a formal framework for such systems, which allows high-level modelling and formal analysis of distributed systems where interaction is managed by a variety of nets, including wireless ones [10]. We introduce a simple modelling language for object-oriented components, extending the Creol language.

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 call a method on a component if the called method is supported by some object in that component. However, if the caller knows the identity of a preferred object inside the component, the caller may call that object directly.

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. We consider three basic forms of links: *wireless*, *loose*, and *tight*. In order to model the heterogeneous net, links are declared by special statements, for instance `link` $\overline{o}$ `wless` $\overline{u}$. This statement adds `wless` 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` $\overline{u}$. We do allow broadcast communication to all components supporting a given interface. The modelling language considered here depends on a notion of time. This is handled by a combination of global and local clocks.

Primitives are added for broadcast communication to objects supporting a given interface (this is useful to establish a connection between a wireless network and a new sensor moving into the network). The modelling language considered in this paper depends on a notion of time. This is handled by a combination of global and local clocks. An operational semantics for the language is defined in rewriting logic, which directly provides an executable implementation in Maude.

As a case study we have used this executable implementation to simulate a small sensor network. The sensors are connected to each other by wireless connections. Some of the sensors are also connected to a sink. The network is evolving which means that connections may be broken, and new ones may appear. The sensors measure some kind of data and tries to pass the measurements to the sink. Since not all of the sensors are connected to the sink, messages must be passed on by other sensors. The sink then passes the data to a user trough a wired network. The sensors and the sink are components consisting of a control object and a radio object. The executable program representing this whole system consists of approximately 70 lines of Creol code.

5 Modelling of AODV

In BSNs, the underlying communication networks must ensure that the data packets can be delivered to the medical center reliably and efficiently [9,4]. Data in a network are often forwarded by using a routing table, where the next hop in the communication chain can be retrieved. The routing table is built up by using a so-called routing protocol. The properties of routing protocols have a significant impact on the network performance. Many routing protocols have been proposed for wireless sensor networks in recent years, which can be classified into two main categories: proactive and reactive routing protocols [2,1].

We select the AODV [12] protocol as a study on how to model and evaluate protocols and algorithms in BSN. AODV has undergone model-checking before [11]. The entities to model are rather discrete, while timing and probabilistic properties can be abstracted away in a straight-forward manner. For protocols like AODV it is also possible to check invariants and other assertions in the model.

5.1 AODV

AODV is a routing protocol which can inherently handle the network dynamics, e.g., node mobility, varying wireless link qualities and the changing network topology. Besides, AODV is well documented and has been demonstrated to be effective in the application of distributed communication systems, e.g., wireless sensor networks, wireless AD hoc networks. Furthermore, AODV is a distributed routing protocol, i.e., sensor nodes can establish and maintain routes without the need of centralised control. Thus, AODV is selected as a suitable communication protocol to be modelled using Creol.

The *AODV* (Ad hoc On Demand Distance Vector) routing algorithm, designed for wireless networks, is a reactive routing protocol, i.e., routes are determined when they are needed. When a node has a packet to send, it will initiate a route discovery procedure by broadcasting RREQ (route request) messages. When a node receives a RREQ message, if it is the destination node or it has a route to the destination, the node will send a RREP (route reply) message to the node which originated the RREQ message; if not, the node will re-broadcast the RREQ message. This procedure continues until the RREQ message reaches the destination node or a node has a valid route to the destination node. The

RREP message is unicast to the source node through multi-hop communications, as the RREP message propagates, all the intermediate nodes setup routes to the destination. When the source node receives the RREP message, it can establish a route to the destination, and can begin to send data packets along the established route.

The most common metric used in AODV is the number of hops. Therefore, when multiple RREP messages are received by the source, the route with the minimum number of hops will be selected.[5]

If any of the links in the route breaks due to node mobility, wireless channel interferences, etc, the node which detects a broken link[6] can try to repair this locally, or inform the source node. For this it sends a RERR (route error) message along the reverse route, which will finally arrive at the source node. Thus the source node can know that the current route is broken and that it must initiate a new route discovery procedure. Detailed information on AODV routing protocol description, design, implementation and network performance measurement can be found elsewhere [12,6,3].

As an illustration, Figure 9 shows an example of a small BSN with eight nodes, where the potential RREQ messages are shown in blue, while the RREP messages are shown in read. Note that more or other pathes for the RREP messages are possible outcomes of this example.

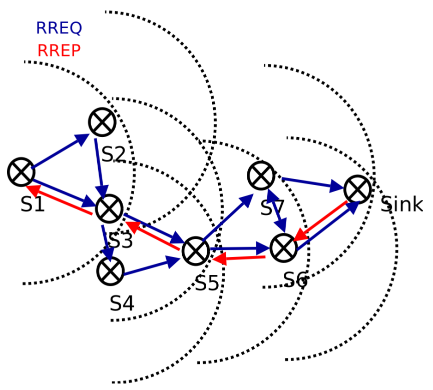


Fig. 9. Example of AODV messages in a BSN with eight nodes.

5.2 Model checking aspects of AODV

Models and implementations of AODV have been simulated and model-checked, and both specification and implementation errors have been found [11]. Both

[5] This is one of the properties that can be checked in the model.

[6] The Medium Access Aontrol (MAC) layer acknowledgement scheme can be used to detect link errors.

node-local (e.g., properties of the routing tables) and global properties (e.g., shortest route is selected; or the route contains no loops) can be checked.

Since the nodes in an AODV network do not have access to a global clock the protocol uses so-called sequence numbers to measure the freshness of routes. According to ([11], Section 4), there is an invariant property; for a route to d at a and b, b being the next hop to the destination, and using seq for the sequence number, and hcnt for the hop count:

$$(seq_a < seq_b) \vee (seq_a = seq_b \wedge \mathrm{h}cnt_a > \mathrm{h}cnt_b)$$

5.3 Modelling of AODV in Creol

The Creol model of AODV is derived from the Creol model of the flooding strategy. This final model contains code for both AODV and flooding in one model, which is facilitated by the object-oriented structure in the model of the messages. In the following we will present the implementation AODV model in Creol followed by a discussion of its features and alternatives.

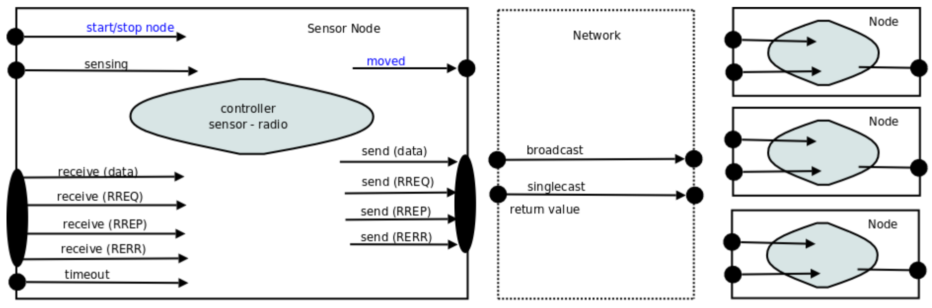


Fig. 10. Interfaces for AODV

Additionally to the features of the flooding model the AODV model needs to handle four different message types, for incoming and outgoing messages, and provide a storage for messages that need to wait for a route. Additionally time-outs (i.e., a RREQ message does not receive a RREP message) and the outcome of messages not delivered to the peer are modelled. A graphical presentation of the interfaces for the AODV model is shown in Figure 10. We recognise that this model is quite similar to the interfaces of the flooding strategy, except that several message types are to be handled.

Modelling of the messages. We modelled four different message types of AODV protocol: Payload messages, RREQ, RREP, RERR, including the handling routines in the nodes for incoming and outgoing messages. In the network the messages are not diversified, and thus only handled as a message where sender, and receiver between peers (i.e., link layer information) is available.

A suitable abstraction for this would be classes without an internal behaviour, not unlike a struct in C, exposing only access functions (i.e., get and set operators) in their interface. Additionally inheritance is suggested as a mechanism to diversify between different message types. This kind of implementation would be very close to a real implementation, and therefore reduce the modelling effort. However, there are several obstacles in Creol for such a solution, such as increasing the number of states when model checking, missing support for type casting in Creol, and the fact that access functions must be written as procedures. Nonetheless we chose to use Creol classes with inheritance to model messages. Alternatives to the use of Creol classes to model the messages are:

Parameter list: For each message type a separate routine is written for transmission in the *network* and for handling in the nodes, e.g., `transmitPayload`, `transmitRREQ`, etc. using the appropriate parameters for each type. However, this alternative would make it necessary to repeat code in the network class, instead of defining only routines for singlecast and broadcast. Note that a separate routine for each message type and the resulting repetition of code would make modelling of a different behaviour in the *network* object less flexible.

Common block: Instead of transporting all parameters explicitly in the call, one could use an unique identifier for each message. In order to access the single parameters for each message one could define a common block object that contains the necessary maps and tables to retrieve and set these values, e.g.,

op getMessageType(**in** idx: Int; **out** val: Int)

The *common block* object needs to be globally available in the nodes. From a practitioner's perspective this kind of modelling resembles somewhat to the programming practice of FORTRAN, where this technique is often used.

maps: The messages could also be modelled as maps, i.e., a collection of name and type. However, currently all elements of a map must have the same data type, e.g., Int. Therefore, all information in a message must be transformed to Int values, which is possible for all important values used in AODV.

pairs: An alternative is to use the build-in Pair type already provided by the Creol type system. By using nested Pairs one may then model parameter lists of any length and with elements of different types, as in

var parlist : [Int, [String, Bool]]

As in the model for flooding there is a generic class *Message* with the following main elements of interface[7]:

[7] Methods used for tracing and debugging purposes are omitted in this interface. In the real implementation we added a string variable where in the absence of printing commands in Maude, a trace of nodes is provided which were involved in processing this message.

```
interface Message begin with Any
    op getSndNode(out seno: Int) // link layer / peer−to−peer
    op setSndNode(in seno: Int)  // link layer / peer−to−peer
    op getRecNode(out reno: Int) // link layer − negative for broadcast
    op setRecNode(in reno: Int)  // link layer − negative for broadcast

    op getMessageType(out mt: Int)
    op getPayloadMessage(out m: PayloadMessage)
    op getRREQMessage(out m: RREQMessage)
    op getRREPMessage(out m: RREPMessage)
    op getRERRMessage(out m: RERRMessage)
    op getFloodingMessage(out m: FloodingMessage)
end
```

The elements *SndNode* and *RecNode* denote the sender and receiver on the link-layer, i.e., between two peers. The second part shows the methods for the typecasting mechanism necessary for diversifying the messages according to their type.

As an example, the RREQ message has the following interface which provides read access to all message content; for the other message types we refer to the code.

```
interface RREQMessage inherits Message begin
  with Any
    op getHopCount(out hc: Int)
    op setHopCount(in hc:  Int)
    op getDstNode (out dsn: Int)
    op getDstSeqNo(out dsno: Int)
    op getOrgNode (out osn: Int)
    op getOrgSeqNo(out osno: Int)
    op getUFlag(out uflag:  Bool)
    op getRREQID(out theRREQID: Int)
end
```

The definition of the Message class and its implementation, including the typecasting mechanism and cloning messages takes about 300 lines of code.

Modelling the network. The interface and implementation of the *Network* is similar to the flooding case, but the fact that AODV needs both the *broadcast* and *singlecast* methods.

```
interface Network begin
  with Any
    op register(in node: Node, connections: List[Node])
  with Node
    op broadcast(in data: Message)
    op singlecast(in data: Message, rec: Int; out success: Bool)
end
```

Modelling the nodes. The interface of the *Node* is similar to the model in flooding. The interface includes receive-routines for broadcast and singlecast; the latter provides a result parameter. Note that all nodes in the vicinity of a sender will handle incoming singlecast-messages, since their radio part will be occupied during this operation. Each node will ignore all incoming singlecast messages that are bound for other nodes than itself. An extra method is the *raiseTimeout* method which models possible timeouts that may occur when an outgoing RREQ message is not responded with a suitable RREP message. The classes *RoutingLogic* and *CacheLogic* contain data structures needed internally in the node, that is one instance of the routing table, and a cache that is suited to detect duplicate messages.

```
interface Node inherits RoutingLogic inherits CacheLogic begin
  with Network
    op receiveBroadcast(in data: Message)
    op receiveSinglecast(in data: Message, rec: Int; out success: Bool)
  with Any
    op start
    op raiseTimeout(in timeoutID: Int)
end
```

The behaviour of a node is similar to the behaviour of a flooding node. the *storedQ* implements a queue of messages that are yet to be transmitted through the radio of a node. The internal behaviour of a node is implemented by the following methods:

```
op sendOrForward ==
   var processResult: Bool;
   processOutgoingMessage(head(storedQ);processResult);
   storedQ := tail (storedQ)

op run ==
   await start;
   while true do
     await seqNo <noSensings; sense(;)
     □
     await #(storedQ) >0; sendOrForward(;)
   end
```

The processing logic for messages is impelemented in the methods *processIncomingMessage* and *processOutgoingMessage*. Both of these call the specific routine suitable for the message to be handled. That is, all incoming messages are received through this one method, which retrieves the message type, and calls the appropriate handling routine, e.g., *processIncomingRREQMessage*. The outgoing messages are handled similarly.

For incoming messages with a payload the method *processIncomingPayloadMessage* first checks whether the current node is the receiver. For messages to be forwarded further the node checks whether an entry to the destination exists;

if so the message is put into the *storeQ*, else the message is moved into the *waitingQ* and an RREQ message is generated. The following code snippet illustrates this further:

```
(this as RoutingLogic).existsValidRouteTo(pdst;existRouteToDst);
if existRouteToDst then
   store(data;)                         // put into storedQ
else
   waitingQ  :=  waitingQ ⊢plmdata; // put into waitingQ
   couma.getNextMessageId(;newmid);
   ownSequenceNumber :=ownSequenceNumber +1;
   newRREQmsg :=new RREQMessage(id,−1,newmid,
                     id,ownSequenceNumber,pdst,0,false,ownRREQID);
   ownRREQID :=ownRREQID +1;
   newmsg    :=  newRREQmsg;
   store(newmsg;)
end
```

In the nodes there are three larger data structures defined as classes, that are inherited into a node; these are the *RoutingLogic* (implementing the routing table), two different *CacheLogic* classes, which remember already incoming RREQ and RERR messages. While we implemented the *Cache Logic* functionality, it depends on the properties to check whether these classes are needed. Note that the code will not work without these classes, since we derived the model from the RFC specification.

The DeusExMachina object. Our model supports several possiblities of indeterminism introduced in the real world due to properties of the outer environment, resulting in, e.g., lost messages which could cause timeouts. Even when these incidents occur, the invariants while running the model must be fulfilled. To model such behaviour of the model we introduce the object *DeusExMachina* that can make decisions, and that could have an internal behaviour to generate timeouts. At the moment, *DeusExMachina* is implemented as an oracle that tells whether a message will arive, or whether a timeout occurs. In principle this functionality also could be implemented in the *Network*.

The following snippet shows how indeterminism whether messages arrive are implemented. This snippet also shows the implementation of timeouts, that can occur between a RREQ is broadcast and the corresponding RREP is received. In our model, timeouts potentially occur (somewhen) before a message arrives. In this model we avoid to model time explicitely, and instead abstract the time intervals between two incoming messages as one possibility for a timeout to occur. Note also that for simulation arbitrary decisions are made, while for model-checking all possiblities are checked. In the AODV model the *raiseTimout* method implements to resend RREQ messages up to a specified number of times.

```
op receiveBroadcast(in data: Message) ==
var doesArrive:  Bool;
var doesTimeout: Bool;
couma.doesRiseTimeout(id,intrID;doesTimeout);
if doesTimeout then
   this.raiseTimeout(intrID;)
end;
data.getSndNode(;fromNode);
couma.doesArriveBroadcastMessage(fromNode,id;doesArrive);
if doesArrive then
   processIncomingMessage(data;theSuccess)
end
```

Initialising the model. As an example how to set up and initialise the AODV model we show the *Main* class. As an extension the setup can be defined as a method for more advanced network topologies.

```
class Main begin
  var nw: Network
  var n1, n2, n3, n4, sn: Node
  var couma: DeusExMachina

  op run ==
    var none: List[Node] := nil;
    var noNodes: Int := 5;
    couma  := new DeusExMachina;
    nw := new BroadcastNetwork(noNodes,couma);
    n1 := new SensorNode(1, nw, couma);
    n2 := new SensorNode(2, nw, couma);
    n3 := new SensorNode(3, nw, couma);
    n4 := new SensorNode(4, nw, couma);
    sn := new SinkNode(nw, couma);
    //  Network topology:
    //
    //  N1 <->N2 <->SN
    //  ^     ^     ^
    //  |     |     |
    //  V     V     |
    //  N3    N4<----+
    nw.register(n1, [n2,n3];); nw.register(n2, [n1,n4,sn];);
    nw.register(n3, [n1];);    nw.register(n4, [n2,sn];);
    nw.register(sn, [n2,n4];);
    !n1.start(); !n2.start(); !n3.start(); !n4.start(); !sn.start()
end
```

About the Creol Code. The model of AODV in Creol is about 1400 lines of code, which is rather large. However, different real world implementations of AODV are of about 5000-9000 lines of code [11]. The fact that our Creol model was to some extent developed from more low level implementations, only partially explains the length of the model.

For the TinyAODV [13], which is a tailored version of the standard AODV adapting to the resource-constrained wireless sensor networks, the core AODV processing file has about 1500 lines of code, which is comparable to our Creol AODV model. Besides this, there are about 2000 lines in the non-core files in TinyAODV which define the underlying communication protocols. In total, around 3500 lines of code are used to in the TinyAODV implementation.

While the length of our model is comparable to the implementation of TinyAODV, we must admit that we model less functionality. There are other reasons for the lengthyness of our model, like the fact that the definition of messages alone takes 300 lines of code, while the much simpler alternative model of Section 5.4 contains the same number of lines. We will explain more reasons for the lengthyness of our Creol model later in Section 6.1.

Using Creol Modules and Maude Functions. An alternative to use Creol is defining Creol Modules. An example for defining messages is as follows:

```
fmod CREOL−AODV−MESSAGE is

 extending CREOL−DATA−SIG .
 protecting CREOL−DATATYPES .

∗ ∗ ∗ record−like messages (with a trace)
op Msg :Expr Expr Expr Expr →Expr .
op Msg :Data Data Data Data →Data [ctor] .

vars sndNode recNode msgId theTrace :Expr .
vars sndNode' recNode' msgId' theTrace' val' : Expr .
∗ ∗ ∗ get functions for selecting a field
eq "getsndNode" (Msg(sndNode, recNode, msgId, theTrace)) = sndNode .
eq "getrecNode" (Msg(sndNode, recNode, msgId, theTrace)) = recNode .
eq "getmsgId"  (Msg(sndNode, recNode, msgId, theTrace)) = msgId .
eq "gettheTrace"(Msg(sndNode, recNode, msgId, theTrace)) = theTrace .
∗ ∗ ∗ set functions for setting a field of a record
∗ ∗ ∗ e.g. setsndNode(message, newsndNodevalue) give updated message
eq "setsndNode" (Msg(sndNode, recNode, msgId, theTrace):: sndNode') =
      Msg(sndNode', recNode, msgId, theTrace) .
eq "setrecNode" (Msg(sndNode, recNode, msgId, theTrace):: recNode') =
      Msg(sndNode, recNode', msgId, theTrace) .
eq "setmsgId" (Msg(sndNode, recNode, msgId, theTrace):: msgId') =
      Msg(sndNode, recNode, msgId', theTrace) .
eq "settheTrace" (Msg(sndNode, recNode, msgId, theTrace):: theTrace') =
      Msg(sndNode, recNode, msgId, theTrace') .
∗ ∗ ∗ add the trace: adds a value to the trace
```

```
eq "apptheTrace" (Msg(sndNode, recNode, msgId, theTrace):: val') =
       Msg(sndNode, recNode, msgId, "⊢"(theTrace ::val')) .
endfm
```

Note that using this model would imply some (trivial) changes to the Creol compiler, in order to make applications of the defined functions syntactically acceptable.

5.4 Extensions of the AODV Models

The AODV model. We modelled an alternative, simplified model of AODV using Creol and in Spin [8,7], to verify some properties which may form the basis of a comparison between these two frameworks. Both models represent simplified models of AODV in Creol and in Promela, the modelling language of Spin [5]. The Creol and Promela models have about the same length of about 250 lines of code.

The first property we checked is the absence of a deadlock, which means the protocol always can find the routing path if there is one. The second property we checked is to evaluate if the path is the shortest possible path to destination.

These prober ties are important in the AODV protocol, since it is designed to always find a path if there is one, and to find the shortest path. In the model we abstract away some details of the protocol that are not related to these properties. We use the assumptions in our model that no message is lost in the protocol; the the topology of network consists of a pre-defined number of nodes; and the messages do not expire. Figure 11 shows a simplified graph of our model. The nodes in this graph correspond to the labels in the Spin node process and also correspond to the methods in the Creol *Node* class.

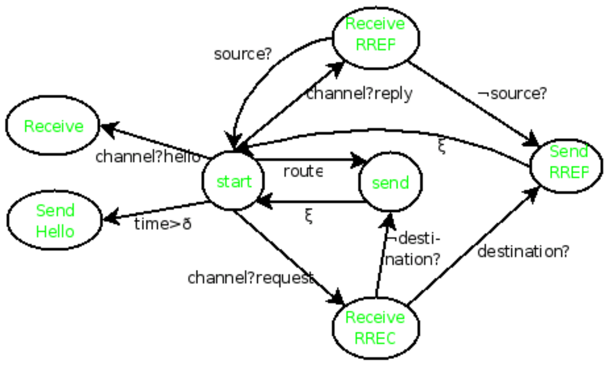


Fig. 11. Simplified graph of extended AODV model

In this[8] model we abstract away *Hello messages*. Instead, we check all the possible neighbourhoods of nodes by initialising the neighbours in each run non-deterministically.

Model Information Our model has a fixed number of nodes, that is three nodes. The connection between nodes are chosen non-deterministically at each run. Messages consist of these parts: type of message, source, destination, message ID, previous node in path, and number of hops.

Each node has its own routing table that stores the path to each destination; the information in the routing table for each destination is the next hop (node) that the path to destination starts with it, and number of hops of that path. When a node finds a shorter path to a destination (with less number of hops), it updates its routing table and replace the older path with the shorter one.

Verification We checked whether the algorithm can find a routing path between two nodes, and whether this path is the shortest possible one. In order to verify this property in Spin, we use the *accept*-label. At first, we put an accept-label in the position that shows that the desired routing path has been found. Then we use the LTL property manager of the xspin tool to verify these properties. To formulate the properties we define the following:

1. q is defined to express the number of hops in the path (length of path) which should always be less than three (number of nodes).
2. p is defined to express a path to the destination has been found.
3. Path is defined to express the possible topologies that a path exists between source and destination.

The LTL formulae of properties are:[9]

$$\mathrm{P}ath- ><> p$$

$$\mathrm{P}ath- > pUq$$

These properties have been shown correctly by the SPIN model checker.

5.5 Modelling AODV with Vereofy

In this section we provide a brief overview on the AODV modeling on the level of the interfaces using Vereofy's input languages CARML and RSL. The model is very close to RFC3561 [12], except we abstract from the lifetime of routes (TTL). For our model we also abstract from collisions and assume reliable connections such that messages are not lost. Contrary to the Vereofy flooding model, the AODV model is able to handle messages to arbitrary destination nodes; not only to one designated sink node.

[8] Internal reviewer note: unknown what "this" refers to . . . ; it is also unclear what a Hello message is, and why it is abstracted away. Fatemeh, Olaf: This must be re-written. – Reviewer: wvl

[9] Internal reviewer note: The entire section is out of context. I am not sure whether this should be here; especially the formulae. Verification should possibly be moved to the next deliverable D6.4. – Reviewer: wvl

Data domain. The data domain consists of two types of messages; the actual data which should be transferred to the sink and the AODV messages (RREQ, RREP, and RERR) forming the routing protocol. The data messages are sent via unicast while the AODV messages are broadcasted in some cases and unicasted in others. All messages are encapsulated into an address frame consisting of the ID of the sending node (from_ip) and an address of the target node or the broadcast address (to_ip). The to_id field corresponds to a node ID in case of unicast and the broadcast address in case of broadcasting messages.

```
TYPE message_type_t = enum {RREQ,RREP,RERR,DATA};
TYPE address_t      = int(0,nodes);
TYPE id_t           = int(0,nodes−1);
TYPE data_type_t    = enum {data0,data1};

TYPE message_t = struct {
   // determine the type and destination
  message_type_t     message_type;
  id_t               dest_id;

   // encapsulation: sender id and receiver id / broadcast
  address_t          to_ip;
  id_t               from_ip;

   // case 1: sending aodv messages
  hop_counter_t      hop_count;
  seq_no_t           dest_seq_no;
  id_t               orig_id;
  seq_no_t           orig_seq_no;
  Bool               unknown_seq_no;

   // case 2: for sending data messages
  data_type_t        the_data;
};

TYPE Data = message_t;
```

Sensor nodes. As in the case of the flooding protocol the prototype description of a sensor node is parametrized. The first parameter determines the node id while the second indicates the number of nodes in the composite system. The sensor node consists of the sub-modules for sending and receiving AODV and data messages. Both are modeled with the help of CARML. The resulting interface of a sensor node thus consists of one port for receiving both, data and AODV messages, one port for sending both, data and AODV messages, and a port for receiving link failure messages which may occur when links break at runtime.

```
MODULE sensor_node<id,k>{
  in:  receive ;
  in:  failure ;
  out: send;
   ...
}
```

A sensor node consist of the sub-modules for sensing data, receiving messages, and sending messages. The distinct modules synchronize their activity via shared variables. Each sensor nodes has its own routing table and buffers for data and AODV messages.

```
  / /  message buffers
  var: Data               message;
  var: Data               AODVMessage;

  / /  routing  table
  var: seq_no_t[k]        rt_dest_seq_no       :=  0;
  var: Bool[k]            rt_valid_dest_seq_no :=FALSE;
  var: Bool[k]            rt_route_valid       :=  FALSE;
  var: hop_counter_t[k] rt_hop_count           :=  k;
  var: id_t[k]            rt_next_hop          :=  id;
```

While the receiving and sensing of messages accords to writing messages into its corresponding buffer, the handling of AODV and data messages is rather complex. Figure 12 gives a brief overview on the different cases. E.g., one can see that receiving an request for a route (RREQ) may trigger forwarding new the request or create an routing reply answer package (RREP) which is then being broadcasted.

For each of the different cases in the sub-module handling received messages we introduced control locations in the model. The flowchart for a refined version of the schema from Figure 12 is depicted in Figure 13. This flowchart contains also the behavior of the sub-modules for sensing data and receiving messages.

Sink node. Since the protocol is flexible enough to deal with arbitrary destination nodes instead of just one designated sink node, the behavior of the sink agrees with the behavior of the other sensor nodes except that the sink has no sensor and sends AODV messages (RREP) only.

Omnicast medium. As for the flooding the medium represents the topology in which the sensor nodes are arranged. Since we abstract from collisions and message loss the medium for AODV consists of the topology matrix only and can now deal with unicast and broadcast communication. Thus, we used a broadcast filter channel instead of synchronous channels to model connections between neighbored nodes. These broadcast filter channels forward a message if the to_ip corresponds either to the intended receiving node or the broadcast address (b). For indicating link failures we add unicast filters back to the sender of the message, such that whenever a node tries to send to an unreachable neighbor, the

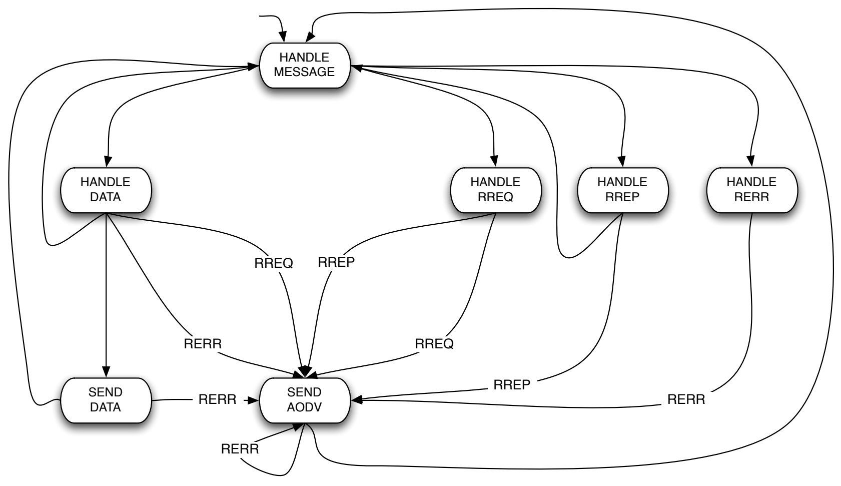


Fig. 12. The AODV schema

node will get feedback via its failure port. Thus for k nodes the topology matrix has k input and $2 \cdot k$ output ports. Figure 14 shows the structure of an omnicast medium for the links of node 1 only. The connections for the other nodes are set up in the same way.

Whenever the first sensor node sends a unicast message to either sensor node 2 or sensor node 4 the message will be delivered to the corresponding neighbor. In case node 1 uses broadcast communication the sensor nodes 2 and 4 will receive the message. Sending a unicast message to any other node $(0, 1,$ or $3)$ will lead to a link failure which is fed back to the failure port of the sending node via a unicast filter. The RSL code for a chain topology matrix looks as follows.

```
CIRCUIT topology_matrix<k>{
  // interface declaration
  for (i=0;i<k;i=i+1){
    // one input
    source[i] = NODE;

    // one output
    sink[(2 * i)] = NODE;

    // and the error feedback
    sink[(2 * i)+1] = NODE;
  }
   ...
}
```

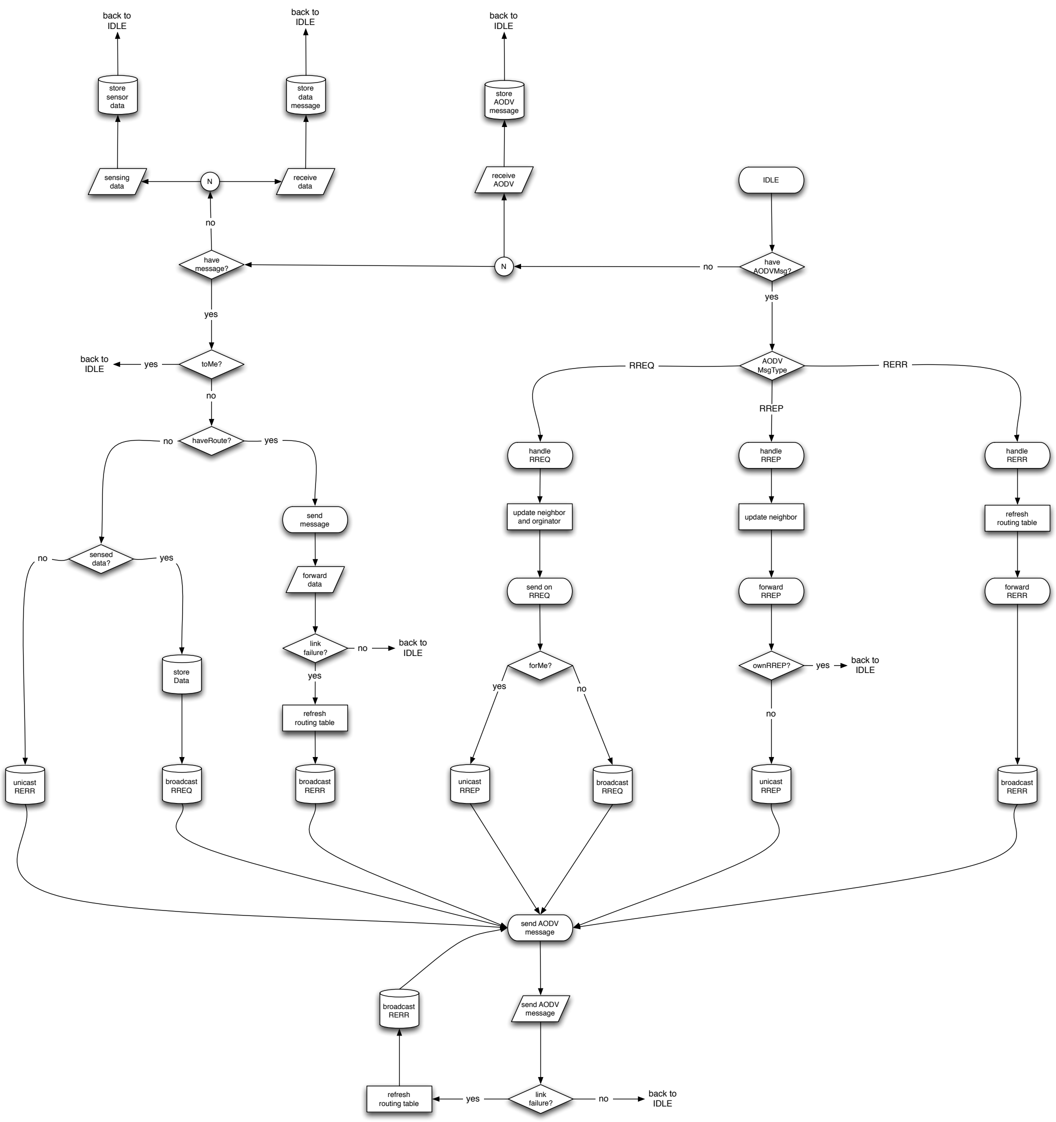


Fig. 13. The full AODV flowchart schema

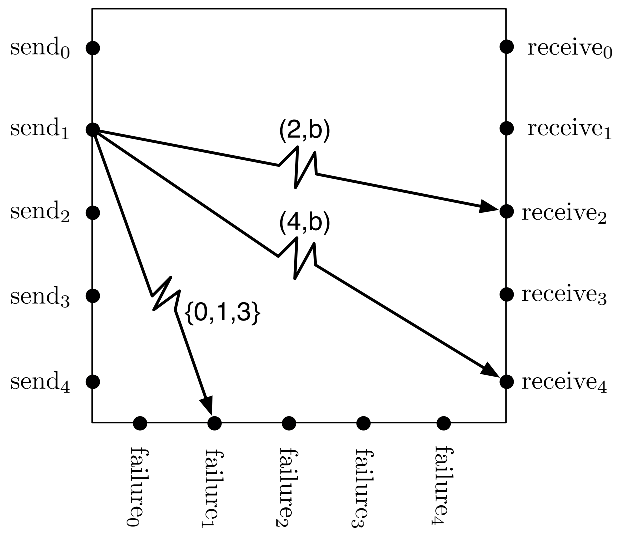


Fig. 14. Omnicast medium for AODV

```
  // connect up
  for (i=0;i<k;i=i+1){
    for (j=0;j<k;j=j+1){
      if (j+1==i | i+1==j){
        // connected
        new broadcast_filter<j,k>(source[i];sink[2 * j ]);
      } else{
        // unconnected
        new unicast_filter<j>(source[i];sink[(2 * i) + 1]);
      }
    }
  }
}
```

Composite system. The composite system is then build using the following RSL script.

```
CIRCUIT main{
  TOPO_MATRIX_OF_SIZE[nodes] = new topology_matrix<nodes>;

  for (i=0;i<nodes;i=i+1){
    send[i]      = TOPO_MATRIX_OF_SIZE[nodes].source[i];
    receive [i]  = TOPO_MATRIX_OF_SIZE[nodes].sink[2*i];
    failure [i]  = TOPO_MATRIX_OF_SIZE[nodes].sink[(2*i)+1];
    node[i]      = new sensor_node<i,nodes>(receive[i],failure[i];send[i ]);
  }
}
```

6 Experiences

6.1 Experiences with Creol

While modelling we found it rather easy to start with modelling in Creol once the run-time system and compiler were installed on the computer. We found a reasonable selection of language constructs suitable for the modelling task. When selecting how express the model in Creol we got some guidance by the developers. However, since there is no best practice established yet we had to try and possibly fail in some occasions.

The Creol model of both flooding and AODV were implemented like a program in a real world application. This has the advantage that the content of messages can be kept together in structures, but had the disadvantage of rather lengthy code. Besides that, from a developer's perspective modelling in Creol can be compared with programming in many other programming languages. Creol offers many of the programming paradigms and abstractions programmers are used to.

Of course, being a modelling language under development, there are several oddities in the language, flaws in the implementation, and features not implemented. We discuss some of these. Note that some of our comments refer to the syntax and are independent of the Creol runtime system.

One feature of the Creol syntax is its use of the ";" symbol which only can be used to separate commands. While this might be nice for the beauty of the language, it is not practical from a programmer's perspective. This feature causes many unnecessary error messages, and thus increases the number of compiler runs!

In the absence of input- and output-facilities in the language we needed to find a way to debug code. One practical solution is to put string variables with a characteristic name into all objects, and append debug information into these strings. Since objects are persistent in Creol this debug information can be accessed while inspecting the objects.

The absence of local scoping of variables is another disadvantage, especially in combination with the non-availability of access functions to values in objects. When methods get larger many local variables must be defined at the beginning of the method, which is responsible for the modellers loosing overview.

As mentioned before, our AODV model got quite lengthy, and several reasons contribute to that. The non-availability of access functions (which are access-methods in the current implementation) increases the code length. For each in-parameter of a method that is to be retrieved from a class object we need to define a temporary variable and to retrieve the value by a method call; thus for example, cloning a RREQ message (9 parameters) by calling a *new* operator with all parameters retrieved from another RREQ message would result in extra 9 temporary variables, and extra 18 lines of code. Note that this does not (necessarily) influence the length of the compiled code in Maude.

Creol is based on a object-oriented paradigm. However, while developing the model we observed that there is no sufficient support for typecasting and multiple inheritance. While technical reasons might be the reason for that, some design decisions while modelling would have been made different, had we been aware of this fact. The implementation of a simple pattern for typecasting (poor-man's typecasting) caused the definition of *Messages* to get quite lengthy; about 300 lines of code. There is also a technical limitation when using multiple inheritance, since method names in all classes must be distinct. Failure to obey this unwritten rule will cause unforeseen results.

We would also propose that behaviour-less objects should be introduced into Creol. While we implemented these using ordinary classes, their use may have an impact on the state space, and we should consider alternatives. On the other hand the use of behaviour-less objects was a blessing for debugging, since all the message objects created ever are stored in the state and can be inspected. This also allowed us to implement a trace in a string variable, which was a very helpful feature.

6.2 Experiences with Vereofy

We found it rather straight-forward to model flooding and AODV using Vereofy's input languages CARML and RSL for the sensor nodes. Abstract description formalisms, such as control flow charts, could immediately be translated into CARML modules. The composite approach facilitated the modelling procedure a lot. For modelling the broadcast and unicast media we depended on specialist knowledge about REO and some practice on composing channels in the right way, to end up with the intended behaviour. While modelling we identified some language features that were not implemented, like complex data structures for the data domain or support for user defined functions. The language features that were required during the modelling phase were added to Vereofy which furthered its development.

6.3 Experiences with UPPAAL

In modelling the CC2420 radio and the controlled flooding protocol using the model checker UPPAAL, we found it very intuitive to model the distributed communication system using timed-automata, simulate it, and then verify properties on it. The features of using templates to define and instantiate processes in a complex system, the C-like modelling language, and the integrated toolbox make UPPAAL being a convenient validation tool for wireless sensor networks. For instance, we modelled a medium size biomedical sensor network and validated the QoS and network connectivity properties with reasonable effort.

From a practical perspecive the use of pointers in the modelling language of UPPAAL could ease modelling of complex communication protocols, since many implementations, like the Tiny-AODV, use these low-level features in their implementations.

References

1. K. Akkaya and M. Younis. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks*, 3:325–349, May 2005.
2. J. N. Al-Karaki and A. E. Kamal. Routing techniques in wireless sensor networks: a survey. *IEEE Wireless Communications*, 11:6–28, Dec. 2004.
3. I. Chakeres and E. Belding-Royer. AODV routing protocol implementation design. In *Proc. the 24th International Conference on Distributed Computing Systems Workshops(ICDCS'04)*, pages 698– 703, Tokyo, Japan, Mar. 2004.
4. D. Chen and P. K. Varshney. QoS support in wireless sensor networks: A survey. In *Proc. of the 2004 International Conference on Wireless Networks (ICWN'04)*, pages 227–233, Las Vegas, Nevada, USA, june 2004.
5. R. Gerth. Concise promela reference. fulltext, available with SPIN, 1997.
6. C. Gomez, F. Salvatella, O. Alonso, and J. Paradells. Adapting AODV for IEEE 802.15.4 mesh sensor networks: Theoretical discussion and performance evaluation in a real environment. In *Proc. IEEE the 7th International Symposium on World of Wireless, Mobile and Multimedia Networks (WOWMOM'06)*, pages 159–170, Niagara-Falls, Buffalo-NY, USA, June 2006.
7. G. J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.
8. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003. ISBN 0-321-22862-6.
9. L. H. II and R. TAFAZOLLI. A survey of QoS routing solutions for mobile ad hoc networks. *IEEE Communications Surveys & Tutorials*, 9(2):50–70, July 2007.
10. E. B. Johnsen, O. Owe, J. Bjørk, and M. Kyas. An object-oriented component model for heterogeneous nets. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 5382 of *Lecture Notes in Computer Science*, pages 257–279. Springer, 2007.
11. M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: a pragmatic approach to model checking real code, 2002.
12. C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental), July 2003.
13. N. Pham, J. Youn, and C. Won. A comparison of wireless sensor network routing protocols on an experimental testbed. In *Proc. IEEE the 2006 International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'06)*, pages 276–281, Taichung, Taiwan, June 2006.
14. S. Tschirner, X. Liang, and W. Yi. Model-based validation of QoS properties of biomedical sensor networks. In *Proc. The International Conference on Embedded Software (EMSOFT2008)*, pages 69–78, Atlanta, Georgia, USA, Oct. 2008.
15. S. Tschirner and W. Yi. Validating QoS properties in biomedical sensor networks. In *Proc. The 19th Nordic Workshop on Programming Theory (NWPT'07)*, pages 11–15, Oslo, Norway, Oct. 2007.