Simulation of Ad hoc Networks in ReactiveML

Farid Benbadis Louis Mandel Marc Pouzet Ludovic Samper

December 15, 2006

Abstract

This paper presents a programming experiment of complex network routing protocols for mobile ad hoc networks within the reactive language ReactiveML.

Mobile ad hoc networks are highly dynamic networks characterized by the absence of any physical infrastructure. In such networks, nodes move, evolve concurrently, and synchronize continuously with their neighbors. Due to mobility, connections in the network can change dynamically and nodes can join or leave at any time. All these characteristics — concurrency with many communications and the need of complex data-structure — combined to our routing protocol specifications make the use of standard simulation tools (*e.g.*, NS-2, OPNET) inadequate. Moreover network protocols appear to be hard to program efficiently in conventional programming languages.

In this paper, we show that the *synchronous reactive model*, as introduced in the pioneering work of Boussinot, matters for programming such systems. This model provides adequate programming constructs — namely synchronous parallel composition, broadcast communication and dynamic creation — which allow a natural implementation of the hard part of the simulation. This proposition is supported by two concrete examples: the first one is a routing protocol in mobile ad hoc networks where the simulation focuses only on the network layer. The second one is the power consumption in sensor networks. Here, every single layer is faithfully simulated (hardware, MAC, and network layers). More importantly, the physical environment (*e.g.*, clouds) has also been integrated into the simulation using the tool Lucky.

The implementation has been done in ReactiveML, an embedding of the reactive model inside the statically typed and strict functional language OCaml. ReactiveML provides reactive programming constructs together with most of the features of OCaml. Moreover, it provides an efficient execution scheme for reactive constructs which made the simulation of real-size examples (with thousands of nodes) feasible.

1 Introduction

Ad hoc networks are highly dynamic networks characterized by the absence of any physical infrastructure. In this paper, we study two kinds of ad hoc networks: mobile and sensor networks.

Mobile ad hoc networks consist of autonomous nodes that evolve concurrently. Nodes have to synchronize continuously between each other. Among existing routing systems, age and position based protocols have recently emerged because of their relatively simple and efficient policies: no location service is required, the destination position discovery is achieved during the packets forwarding step where nodes take elementary forwarding decisions based solely on the coordinates of their direct neighbors and the destination [19]. This avoids the need for topology knowledge beyond one-hop.

A sensor networks is a constrained ad hoc network. It is composed of a large number of sensors (several thousands). Those nodes are designed to be as small and cheap as possible. Sensor networks can be deployed in situation with difficult access and/or no available energy. Thus, nodes are power-constrained. The network has to achieve a certain service for the longest time possible. Because there is no or very few infrastructure, and because of the size of the network, nodes running out of energy are not replaced. We can notice that every element of a network has some influence on power consumption: nodes architecture, radio access functionalities, communication protocols, application, and even network environment which stimulates sensors.

These networks are typical examples of *complex dynamic systems*, that is, dynamic systems where both the state of system and its internal structure evolve during the execution. Ensuring a correct behavior of such a network is challenging, and the best way to tackle this problem is to build *models* which can be simulated.

Characteristics of these networks — concurrency with many synchronizations and the need of complex data-structures — make the use of standard simulation tools such as NS-2 [1] or OPNET [29] inappropriate. Indeed, NS-2 has been originally designed for wired networks and does not correctly handle wireless networks. In particular, it can only simulate small networks (1000 nodes networks seems to be barely conceivable) whereas we consider large scale networks.

In this paper, we show that the *synchronous reactive model* introduced by Boussinot [10, 11, 37] is an appropriate model for programming those systems. We argue that this model provides appropriate programming constructs — synchronous parallel composition with a common global time scale, broadcast communication, and dynamic creation — making the implementation of the hard part of the network surprisingly simple and efficient. We can remark that the reactive synchronous model is not contradictory with the asynchronous aspect of these networks. Synchrony only gives the ability to all nodes to react in a fair way like it could be done in an imperative implementation.

The model provides *language concurrency* as opposed to *run-time concurrency*: reactive parallel programs are translated into conventional single-thread, yet efficient programs [2, 9, 13, 40]. Eventhough a similar formulation is possible in any conventional programming language using one run-time thread per node, it would not allow to simulate large networks for clear efficiency reasons.

Network simulators detailed below have been written in ReactiveML (RML for short), an embedding of the reactive model inside a statically typed, strict functional language [27, 25]. [1] ReactiveML provides reactive programming constructs with most of the features of OCaml [24]. Reactive constructs give a powerful way to describe the dynamic part of the system whereas the host language OCaml provides data-structures for programming the algorithmic part. Moreover, ReactiveML also provides an efficient execution scheme for reactive constructs which makes the simulation of real-size examples feasible.

The purpose of this paper is to convince of the adequacy of the reactive model for real-size simulation problems. Ad hoc networks are such systems. As a side-effect, these systems are good examples to compare the various implementations of the reactive model [2, 13, 40].

The remainder of this paper is organized as follows. Section 2 discusses the adequacy of the programming model on which ReactiveML is based for programming network simulators. Section 3 presents briefly a routing protocol and its ReactiveML implementation. This section is to be considered as a tutorial introduction to the language through an example. In order to ease the presentation, this section provides a survival kit which can easily be skipped and we only give samples of the code. People interested in the whole implementation should follow hyperlinks. Section 4 presents the simulation of a complete protocol stack for sensor networks, taking the physical environment into account. We discuss related works in section 5, and conclude in section 6.

2 Why ReactiveML matters for programming simulators?

One first observation is that even if there exists many different network simulators, people continue sometimes to develop their own simulator. Why? Creating your own simulator guaranties that this simulator will perfectly fit your needs. Indeed, even if some simulators provide several levels of detail (see section 5), a custom simulator can exactly address the faced problem.

The main point in our approach is to be able to quickly develop a problem specific simulator which is efficient enough to obtain simulation results.

High level reactive language Writing a simulator from scratch is time consuming but avoids the cost of learning another simulator. In order to reduce the time and effort needed to write his own simulator,

[1] The distribution can be accessed at: `http://ReactiveML.org`.

a high level language is required. ReactiveML, an ML-language, is a high level and formally defined programming language. Since ReactiveML is an extension of OCaml, it takes advantages of the host language strengths:[2] a powerful type system, user-definable algebraic data types and pattern matching and automatic memory management. The expressiveness of the host language is important to manipulate complex data structures such as nodes or packets.

Moreover, nodes of a network are components which run in parallel. Thus a language that already contains that primitive of running in parallel is very useful. It is an advantage that ReactiveML have above the classical programming languages or event driven libraries.

The synchronous approach Network simulators are typical examples of concurrent systems with a lot of parallel processes and communications between processes. The synchronous reactive model is designed for this kind of applications. So languages based on this model provide appropriate constructs to the programming of these systems, hence they have an efficient execution.

For example, the global time scale allows an easy way to model the network. First, it gives the ability to all nodes to react in a fair way. This reflects reality because nodes work independently thus during an instant that corresponds to a real unit of time, they can all do something. Secondly, the programmer can use instants to implement time in his simulator.

We can also notice that the synchronous nature of the model is not contradictory with the asynchronous aspects of networks. For instance, for the simulator presented in the section 4, it was easy to simulate clock drift of nodes.

Dynamic creation An ad hoc network is a highly dynamic computer system. Nodes can join or leave the network. These changes in the network topology may depend on the simulation such that the size of the network cannot be statically computed.

ReactiveML provides high level constructs to kill processes and it gives to the programmer the opportunity to create during the executions new processes that naturally interact with the other processes.

We present in section 3.4 examples of dynamic aspects. We define a process that dynamically creates new nodes. We run a simulation that dynamically delete processes and print the memory usage (see fig. 7). It shows that the OCaml garbage collector works well with ReactiveML programs and thus memory deallocation is not a worry for the programmer.

Synchronous observers To give information about the behavior of a network simulation, the simulator has to give outputs to the user. Those outputs can be trace files or graphic windows, for instance. Of course, printing whatever kind of statistics in a file is easy to achieve in OCaml and thus in ReactiveML. As well, a graphic library exists in OCaml and can be used in ReactiveML. But a key issue is, will the add of an observer modify the behavior of the simulation? The synchronous model enables to run the observers in parallel of the simulation and guaranties that adding one or several observers will not modify the execution of the simulation.

Example: Viewer Authors of [12] insist on the importance of having a visualization tool that helps users understand the complex behavior in network simulation. In our opinion, a visualization tool in a network simulator is essential. It is not only useful in order to give intuition about protocols to develop but also to aid in debugging both the simulator and the protocol stack.

It would be possible in a ReactiveML simulator to generate a trace file that is compatible with a visualization tool like Nam [17] (Network Animator, the visualization tool of NS-2). However, the graphic output of OCaml is very easy to use. It allows a dynamic display of the network.

Synchronous observer is a good design pattern to program a viewer. It does not modify the simulation: with or without the viewer, the simulations will remain the same.

[2] http://caml.inria.fr/about/index.en.html

Moreover, the visualization tool is also self made within the same language as the simulator which lets the user display just what he need on the graphic output. It can be the collisions, the emission of packets, or whatever.

Interactive simulations The dynamic viewer presented before allows to build interactive simulations. We can, for instance, stimulate a node with a mouse click that generates a packet to send.

Thanks to the dynamic aspects of ReactiveML, interactive simulations can also be used to add or remove nodes in a network. For example, a function which creates a new node in the network with a mouse-click is about ten lines long (see section 3.4).

The interaction between the user and the simulator can also be used for instance to dynamically manage the observers. Indeed, the graphic window could be removed during execution (to speed up the simulation) and then displayed again to monitor the simulation.

Simulating the environment: Lucky In wireless networks, data transmission impacts the behavior of a network. To generate messages, simulators can use statistic laws on nodes. Poisson processes for instance are often used. This environment modeling shows its limit, especially in the case of sensor networks. A more accurate modeling of the environment is sometimes needed. Most of classical network simulators do not integer a way to simulate the environment, in sensor networks a model of the environment appears to be essential to perform realistic simulations ([35]). Thus people begin to integer environment models in their simulators, see section 5.

It is possible to simulate the environment by a user interaction as we have seen before. Or it is possible to describe an environment model directly in ReactiveML. An environment process will then run in parallel with the rest. A better formalism to describe a non-deterministic environment is Lucky [23]. ReactiveML provides an interface with Lucky, thus a Lucky process works as an ReactiveML process.

Lucky is described in further detail in the sensor network example, section 4.1.4.

Scalability We would like to emphasis on efficiency. First, a simulator that simulates exactly what is needed is more efficient than a generic simulator. Second, simulator efficiency depends on the efficiency of the programming language. ReactiveML is compiled into plain OCaml. Thus, it takes advantage of its efficient native code compilers. The handling of the concurrency without run-time threads is also a reason for the language efficiency. More information about ReactiveML implementation can be found in [25]. Last, the efficiency relies on the algorithmic. In section 3.2.3, we illustrate how expressiveness of the reactive model allows to implement an efficient algorithm to compute the neighborhood of a node.

ReactiveML vs other synchronous reactive approach Compared to other implementations of the reactive model such as SugarCubes [11] or FairThreads [37] that propose a "library" approach, ReactiveML proposes a language approach. Reactive library gives access to all the features of the host language and is relatively light to implement. Nonetheless, this approach can lead to confusions between values from the host language used for programming the instant and reactive constructs. This can lead to re-entrance phenomena which are usually detected by run-time tests. Moreover, signals in the reactive model are subject to dynamic scoping rules, making the reasoning on programs hard.

Finally, even if the reactive model is based on the synchronous one as it can be found in Lustre [21], Signal [20] or Esterel [7], it would be difficult to implement a simulator in these languages. They are designed for the programming of safety critical applications. They must guaranty execution in bounded time and memory. So, dynamic creation is forbidden. This is a strong limitation for dynamic systems simulation such as ad hoc networks. Moreover the use of complex data structures that are shared between the reactive part and the computational one would be difficult.

We illustrate these points on two examples: a mobile ad hoc network [26] and a sensor network [36].

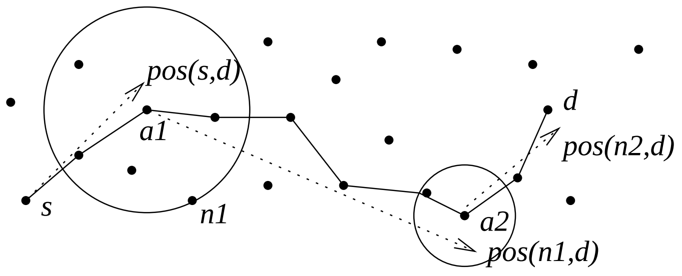


Figure 1: Routing a packet from s to d: anchor nodes a_1 and a_2 refine estimation of d's position.

3 Simulation of Mobile Ad hoc Networks

Our first example is a simulator made to evaluate dissemination methods for Age and Position Based (APB) routing protocols in mobile ad hoc networks.

3.1 Age and Position Based Routing

The main principle of APB routing protocols is that each node may have an information about each other node's location. This information is stored in a position table and associated to an *age* that represents the time elapsed since the last time the information has been updated. The position table is queried by a packet to estimate destination's position.

In this routing methods, destination location discovery is performed during packet transfer: a source node does not know the destination's location when it sends the packet, it only has an estimation about it. We describe the EASE (Exponential Age SEarch) routing method, where a source node s needs to communicate with a destination d, as follows:[3]

```
Set $i := 0$, $age := \infty$, $a_0 := s$ in
While $a_i \neq d$ do
    search around $a_i$ a node $n_i$ such that $age(n_i, d) \leq age/2$;
    $age := age(n_i, d)$;
    Set $m := a_i$ in
    While $m$ is not the closest node of $pos(n_i, d)$ do
        $m :=$ next neighbor toward $pos(n_i, d)$
    done;
    $i := i + 1$;
    $a_i := m$ (* the closest node of $pos(n_i, d)$ *)
done
```

where each a_i is an anchor node that searches for a better estimation of destination's position than the one included in the packet, $pos(n_1, n_2)$ is n_2's position as known by n_1, and $age(n_1, n_2)$ is the age of this information. An illustration of this algorithm is represented in Fig. 1.

Two different methods are used to update position tables in APB routing protocols. The first one, LE (for Last Encounter), introduced in [19], uses encounters between nodes. Each node remembers the location and time of its last encounter with other nodes. The second method, ELIP (Embedded Location Information Protocol), uses also encounters between nodes, but disseminates nodes locations in data packets [6]. In this method, a source node can include its current coordinates in every message it sends in such a way that all the nodes that participate to the forwarding procedure update their knowledge about the source. The source node does not always include its position in data packets to reduce the traffic overhead of this disseminating method.

[3]For more details about EASE, refer to [19]

To simulate these two protocols, we have to represent a set of nodes that evolve in parallel. All of them move, communicate, and update their local position tables, which contain estimations of all other nodes coordinates, at every simulation instant.

Our simulator has been conceived in order to compare two dissemination methods to be used in an APB ad hoc routing algorithm. We did not conceive a generic simulator which can be used for any routing protocol. Moreover, we do not focus on the routing efficiency of EASE, which has been proven in [19], but on the performance of ELIP and LE, two dissemination algorithms. The important point is that the two dissemination algorithms are evaluated in the same conditions. For this reason, we do not consider the physical and link layers and do not take into account the interferences and packets loss. We only focus on the network layer, and consider that when a node broadcasts a packet, all its direct neighbors receive it.

3.2 Implementation in ReactiveML

We present here the structure of the simulator and detail some key points. The full implementation is available at `http://ReactiveML.org/spe`.

3.2.1 Basic features

ReactiveML is built above OCaml. Every OCaml program (without objects, labels and functors) is a valid ReactiveML program and ReactiveML code can be linked to any OCaml library. In the following, we assume that the reader is familiar with functional programming in ML.

A program is a collection of type definitions and values. For example, the following program defines the type `position` of positions as a record, and a position `pos` of this type. Then, it defines the function `distance2` that computes the square of the Euclidean distance between two positions.

```
type position = { x: int; y: int }
let pos = { x = 4; y = 2 }

val pos : position

let distance2 p1 p2 =
  (p2.x - p1.x) * (p2.x - p1.x)
  + (p2.y - p1.y) * (p2.y - p1.y)

val distance2 : position -> position -> int
```

As for OCaml, the compiler automatically infer types (printed in *italic* font).

ReactiveML adds to this functional language, the definition of a *process*. Processes are state machines whose behavior can be executed through several instants as opposed to functions which are considered to be instantaneous. Consider, for example, the process `hello_world` that prints "hello" at the first instant and "world" at the second one (the `pause` statement suspends the execution until the next instant):

```
let process hello_world =
  print_string "hello␣";
  pause;
  print_string "world"

val hello_world : unit process
```

This process can be executed by typing: `run hello_world`.

The expression e_1`||`e_2 is the synchronous parallel composition. It executes the expressions e_1 and e_2 at each instant. So the execution of the process `hello_world_2` prints "hello hello " at the first instant and "worldword" at the second one.

```
let process hello_world_2 =
  run hello_world
  ||
  run hello_world
```

val hello_world_2 : unit process

The construct `for/dopar` is a parallel iterator. It executes the instances of its body in parallel. The process `hello_world_n` executes `n` instances of `hello_world` in parallel.

```
let process hello_world_n n =
  for i = 1 to n dopar
    run hello_world
  done
```

val hello_world_n : int -> unit process

Communication between parallel processes is made by broadcasting signals. A signal can be emitted (`emit`) and awaited (`await`). There is also suspension (`do/when`) and preemption (`do/until`) constructs that use signals. We illustrate these constructs with a `ping_pong` process that prints alternatively `ping` and `pong`.

```
let process ping_pong =
  signal s1, s2 in
  loop
    await s1;
    print_string "ping";
    emit s2
  end
  ||
  emit s1;
  loop
    await s2;
    print_string "pong";
    emit s1
  end
```
val ping_pong : unit process

The construct `signal/in` declares the two signals `s1` and `s2`. Then, two expressions are executed in parallel. The first one prints `ping` and the other one prints `pong`. Synchronizations are made through the signals `s1` and `s2`.

As it is the case in Esterel, a signal may carry some value and this call for a particular treatment in case of multi-emission. For example, what is the value of `x` in the following example where the values `1` and `2` are emitted during the same instant?

```
  emit s 1 || emit s 2 || await s(x) in ...
```

Several answers are possible. When a valued signal is declared, we have to define how to combine values in the case of multi-emission on a signal during the same instant. This is achieved with the construct:

`signal` *name* `default` *value* `gather` *function* `in` *expr*

Thus, if we want to define the signal `s` such that it computes the sum of the emitted values, we can write:

```
  signal s default 0 gather (+) in
  emit s 1 || emit s 2 || await s(x) in print_int x
  (* s : (int, int) event *)
```

The expression `await s(x) in print_int x` awaits the first instant in which `s` is emitted and then, at the next instant, prints `3` which is the sum of the emitted values. The type `(int, int) event` of the signal `s` states that the emitted values and the combined values are integers.

The type of emitted values on a signal `s` and the type of the combined value are not necessarily the same. If τ_1 is the type of the value emitted on `s` and τ_2 the type of the combined value, then `s` is of type `(`τ_1`,`τ_2`) event`. In this case, the default value must have type τ_2 and the gathering function must have type $\tau_1 \rightarrow \tau_2 \rightarrow \tau_2$.

In the following example, the signal `s` collects all the values emitted during the instant:

```
signal s default [] gather fun x y -> x :: y in
emit s 1 || emit s 2 || await s(x) in ...
(* s : (int, int list) event *)
```

Here, the default value is the empty list and the gathering function builds the list of emitted values. So the value of `x` is the list `[2; 1]`.[4] The notation `signal s in ...` is a shortcut for this gathering function.

We stop this short introduction to ReactiveML here. Various examples of programs can be found at `http://ReactiveML.org`.

3.2.2 Data structures

We consider a node n. In order to use an age and position based routing protocol, n needs to know its position. *Global Positioning System* (GPS) is the simplest way for a node to discover its location. Moreover, n needs to continuously store its neighbors' coordinates. For this purpose, it uses a local position table. Each entry in this position table looks like this:

$$[ID_a, pos(n,a), date(n,a)]$$

This entry concerns a node a. $pos(n,a)$ is an estimation of a's position, and $date(n,a)$ indicates when n has got this information. We assume here that n knows, thanks to it MAC layer, its immediate neighborhood which is represented by the set of all the nodes under its radio range.

We then define the type of a node as a record:

```
type node =
  { id: int;
    mutable pos: position;
    mutable neighbors: node list;
    mutable date: int;
    pos_tbl_le: Pos_tbl.t;
    pos_tbl_elip: Pos_tbl.t; }
```

where `id` is the unique identifier of a node, `pos` is its current position, represented by its coordinates on a grid with squares of one meter square, `neighbors` the list of nodes that are under its coverage range, and `date` is the current local date of the node, essentially used to compute the age of other nodes position information. `pos_tbl_le` and `pos_tbl_elip` are the position tables used to simulate the LE and ELIP dissemination protocols.

The record contains mutable fields which can be modified, and non-mutable fields which are fixed at the creation of the concerned record. `pos_tbl_le` and `pos_tbl_elip` are not mutable because we implement them as imperative structures in the module `Pos_tbl`.

Packets for age and position based routing protocols contain the following fields: the source and destination identifiers, an estimation of destination position, the age of this information, and data to be transmitted. When using ELIP, the packets also contain source node location.

In the simulator, packets do not contain data but contain other information used for statistics computation. This information is also useful for the graphical interface.

[4]The order of the elements of the list associated to `s` is not specified.

```
type packet =
  { header: packet_header;
    src_id: int;
    dest_id: int;
    mutable dest_pos: position;
    mutable dest_pos_age: int;
    (* to compute statistics *)
    mutable route: node list;
    mutable anchors: node list; }
```

`src_id`, `dest_id`, `dest_pos` and `dest_pos_age` are used for routing. `route` is the list of nodes the packet traveled through, and `anchors` is the list of anchor nodes. `header` indicates if the packet is a LER or an ELIP packet.

```
type packet_header =
  | H_LE
  | H_ELIP of position option
```

The constructor `H_ELIP` is associated to a value of type `position option` such that ELIP packets can contain the position of the source node or not. The protocol does not always include the position of the sender in the packet to reduce the overload.

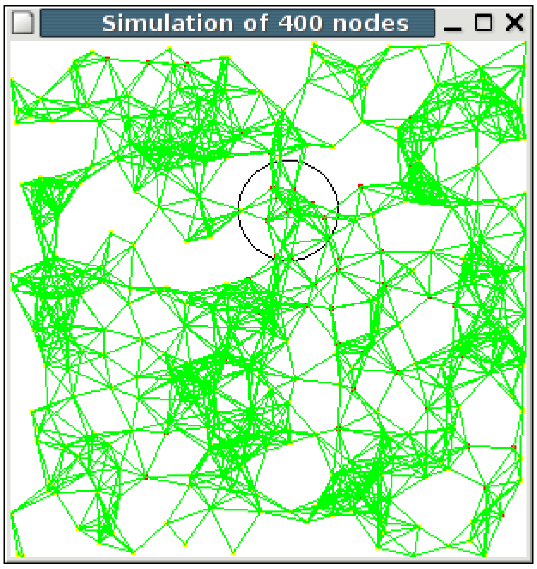
3.2.3 The behavior of a node

The simulator engine executes all the nodes in parallel. The behavior of each node is composed of three steps. All the nodes execute the same step at the same instant. A node (1) moves, (2) discovers its neighborhood, and (3) routes packets. These steps are combined in a process `node` which is parameterized by the initial position of the node `pos_init`, a function `move` that computes its next position, and a function `make_msg` that creates a list of destinations to reach.[5]

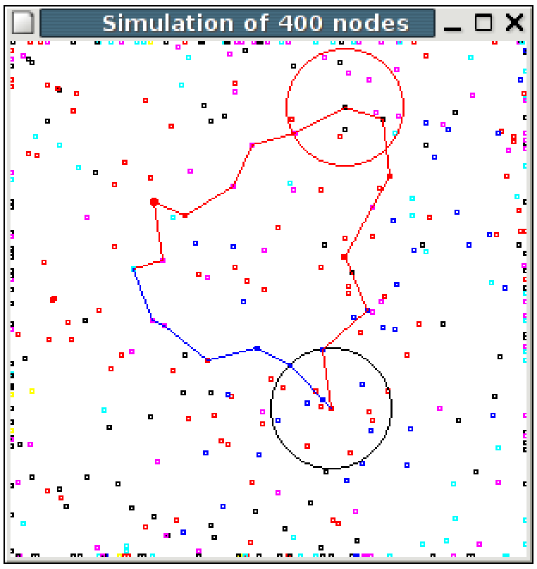
```
let process node pos_init move make_msg =
  let self = make_node pos_init in
  await immediate start;
  loop
    self.date <- self.date + 1;

    (* Moving *)
    self.pos <- move self.pos;
    emit draw self;

    (* Neighborhood discovering *)
    ...
    update_pos_tbl self self.neighbors;

    (* Routing *)
    pause;
    let msg = make_msg self in
    ...
    pause;
  end
```

This process creates a record of type `node` that represents the internal state of the node. Then, it awaits for the global signal `start` to be synchronised with the other nodes. When the signal `start` is present, the node

[5] http://ReactiveML.org/spe/elip/node.rml.html



(a) Topology connectivity. Each green line represents two neighbor nodes, while the black circle represents one node coverage region.



(b) An example of routing paths using ELIP (blue) and LE (red) dissemination methods. The red circle represents the search performed by the anchor node when using LE.

Figure 2: Screen-shots of the simulator graphical interface.

enters in the permanent behavior which is executed through three instants. In the first one, a node updates the local date, moves and emits its new position on the global signal `draw` for the graphical interface (a screen-shot is given in Fig. 2). At the end of the first and during the second instant, the new neighborhood is computed and the position tables are updated using encounters between nodes. The third and last instant is for routing. By enclosing this part between two `pause` statements, we guarantee that topology changes are not possible. We detail now the main steps of the process.

Mobility Nodes movements are parameterized by a mobility function `move`. This function computes the new position of a node according to the current position. The `move` function must have the following signature:

```
val move : position -> position
```

We can implement very simple mobility functions like random moves where a node can move to one of its eight adjacent positions.

```
let random pos = translate pos (Random.int 8)
```

```
val random : position -> position
```

`(Random.int 8)` is the call of the function `Random.int` of the OCaml standard library and `translate` which is a function that returns a new position.

We can also implement more realistic mobility models like the random way-point one. With this mobility model, a point is chosen randomly in the simulation area and the node moves up to this point. When it reaches this point, a new one is chosen. This function is interesting because it needs to keep an internal state.

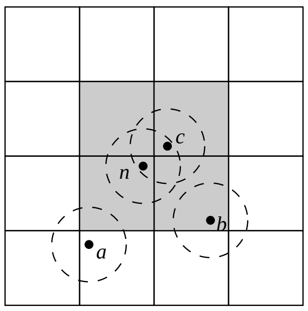


Figure 3: Topology split into multiple squares. Node n emits its position on the gray squares, while it listens on the one it is located.

```
let random_waypoint pos_init =
  let waypoint = ref pos_init in
  fun pos ->
    if pos = !waypoint
    then waypoint := random_pos();
    (* move in the direction of !waypoint *)
    ...

val random_waypoint :
  position -> position -> position
```

The partial application of this function with only one parameter:

```
  random_waypoint (random_pos())
```

returns a mobility function that can be given as an argument to a node.

Neighborhood In real networks, the neighborhood of a node is obtained thanks to the link layer. By contrast, in the simulator it has to be computed. Neighborhood discovery is the key point of the efficiency of the simulator. We first give a simple method to compute the neighbors of a node, then we explain how it can be improved.

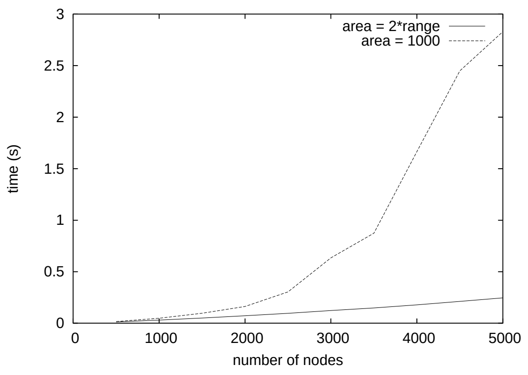
To compute its neighborhood, a node needs to know the position of other nodes. In this first method, we use a signal `hello` to gather all nodes coordinates. Each node emits its position on `hello` such that the value associated to the signal is the list of all nodes. Thus the code of a node looks like the following (`self` is the internal state of the node):

```
  emit hello self;
  await hello(all) in
  self.node_neighbors <- get_neighbors self all;
```

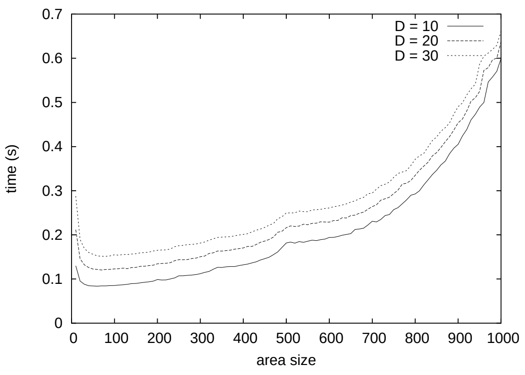
The function `get_neighbors` returns the `all`'s sublist that contains the nodes under the coverage range of `self`.

This neighborhood discovery method is very simple but its drawback is that each node has to compute its distance with all other nodes leading to a quadratic complexity in the number of nodes. To improve this method, we split the simulation area in small areas and associate a `hello` signal to each area. That way, a node has only to compute its distance with the nodes in the areas in which nodes could be under its range. This is the areas that intersect its covering range.

We consider node n in Fig. 3. A `hello` signal is associated to each square. Node n sends its position on the signals associated to the four squares touched by its radio transmission (the four gray squares in this



(a) Mean time for the simulation of one second of the network in terms of the number of nodes for two neighborhood discovery methods.



(b) Mean time for the simulation of one second of the network in terms of the areas size for the improved method.

Figure 4: Simulation times for neighborhood discovery.

figure). In the same way, nodes a, b, and c emit their position on the signals associated to the squares that intersect their coverage range. So, nodes a and c transmit their positions on the signal associated to the square where n is. n receives the positions of a and c. Using this information, n computes its distance from a and c and concludes that c is a neighbor while a is not. n does not consider node b because this node does not emit its position on the signal associated to the square where n is located.

All the `hello` signals are stored in a two dimensional array `hello_array`. We define a function `get_areas` that returns the area of a node and the list of neighbor areas that are under its range.

```
val get_areas :
  position -> (int * int) * (int * int) list
```

Now the behavior of a node is to emit its position in all the areas under its range and to compute its distance with all the nodes which have emitted their positions in its area. So the code of the neighborhood discovery becomes:

```
(* Compute areas under the coverage range *)
let (i,j) as local_area, neighbor_areas =
  get_areas self.pos.x self.pos.y
in
(* Emit the position on each of these areas *)
List.iter
  (fun (i,j) -> emit hello_array.(i).(j) self)
  (local_area::neighbor_areas);
(* Get the nodes that emits their position *)
await hello_array.(i).(j) (all) in
self.neighbors <- get_neighbors self all;
```

Fig. 4 shows the effect of the area split on execution time. In Fig. 4(a), we compare the first method, where all the nodes emit and listen on the same signal, to the second one, where each nodes emits only on the areas under its radio range. Because, in the first method, each node computes its distance to every other node, the neighborhood discovery procedure spends much more time than in the second method, where each node computes its distance to the nodes that emit on its adjacent areas only. We observe that for the simulation of 1500 nodes the second method is twice faster than the first one. Then for 2500 nodes it is 5 times faster and for 5000 nodes it is more than 10 times faster.

We focus now on the second method, which is more appropriate. As we can see in Fig. 4(b), the execution time depends heavily on the area size. This figure represents the time required for the simulation of a 3000 nodes topology using three different densities (average number of neighbors per node). We observe that dividing the topology in too many squares is not efficient. In this case, each node emits its position on a large number of signals, which requires resources. On the other hand, dividing the topology in large squares implies that a node receives a large number of nodes positions on its signal. It spends then long time to compute distances with nodes placed far from it. Simulation results show that 2-ranges-sided squares seems to be a good compromise for the three densities simulated.

Routing The last step in a node execution is the packets routing, which is described in section 3.1[6] The important point is that we assume that routing is instantaneous, which means that the topology is fixed during routing. This scenario is realistic because we assume that nodes move at human speed, while packets travel at radio waves speed. Topology is then supposed to change at time scale of seconds or longer, while packets spend at most tens of milliseconds from source to destination. We can then use OCaml functions, which are supposed instantaneous, to implement the routing protocols.

In the simulator, we compare two location dissemination methods, both of them are combined with the same forwarding algorithm. This algorithm computes the next node which will receive the packet. We use a greedy geographical method. The packet is forwarded to the neighbor that is the nearest (for the Euclidean distance) of the destination. The interesting point in the implementation of the forwarding algorithm (function `forward`) is that a node can access without locks or mutex to the internal state of other nodes executed in parallel to compute the distance between the neighbors to the destination. This concurrent access to share memory is not a problem in ReactiveML (compare to the preemptive thread model) because the forward function is instantaneous and nodes do not move during the routing step.

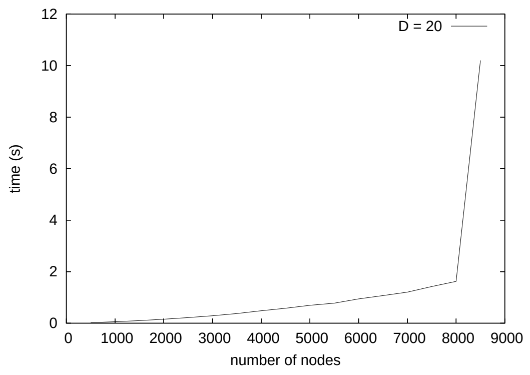
3.2.4 The main process

The main process, which executes the simulation, starts with an initialization part to define simulation parameters. Then it executes `n` nodes in parallel (`for/dopar` is a parallel iterator), the graphical interface and others synchronous observers.

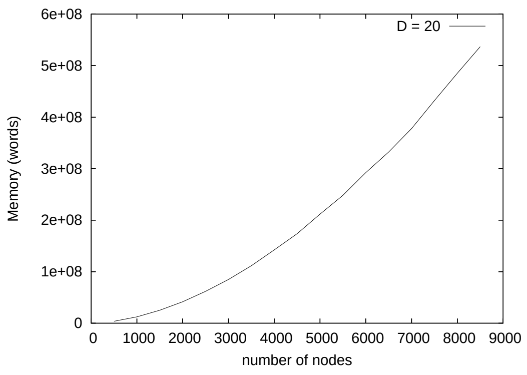
```
let process main =
  (* Initialization part *)
  ...

  (* Main part *)
  begin
    for i = 1 to n dopar
      let pos = random_pos() in
      run (node pos (Move.random_waypoint pos)
                Msg.make)
    done
    ||
    run (draw_simul draw)
    ||
    ...
  end
val main : unit process
```

The structure of this process is the classical structure of the main process of a simulator.

[6] http://ReactiveML.org/spe/elip/routing.rml.html



(a) Simulation times.



(b) Memory usage in terms of the number of nodes. For a given number of nodes, the memory need is constant for the whole simulation.

Figure 5: Simulations depending on the number of nodes with a topology density D=20.

3.3 Analysis

The simulation speed depends on the parameters: number of nodes, coverage range, number of emitted packets, simulation area size, etc. These parameters are linked through the relative density, given by the number of nodes per coverage zone, in order to get a realistic simulation environment.

The simulations have been done on the following computer:

PC Dual-PIV 3.2Ghz, RAM 2GB
running Debian Linux 3.1

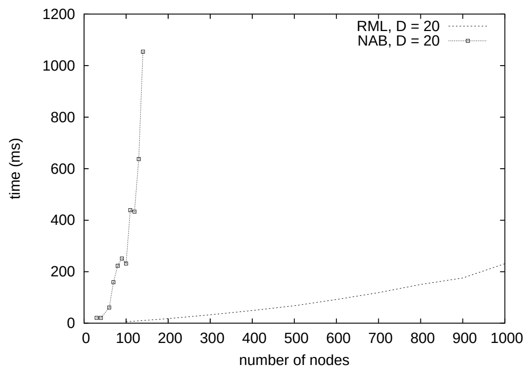
First, we analyze the ability of our program to simulate large networks. Fig. 5(a) represents the mean time for the simulation of one second of the life of the network in terms of the number of nodes. We observe that at about 8000 nodes the execution time becomes suddenly more important. This is due to memory usage. The memory needed increases with the number of nodes. When there is no more memory available, the computer swaps. In Fig. 5(b), the memory usage looks like being quadratic in the number of nodes. This result is natural because each node has a position table that contains positions of all other nodes. To overcome this limitation, we can limit the number of destination nodes such that only a subset of nodes have to be in the position tables.

Now, we compare our simulator with NAB (Network in A Box) [16]. NAB is a simulator developed by the authors of EASE. It is designed to simulate wireless ad hoc and sensor networks. A distinguishing feature of NAB is that it is written in OCaml.

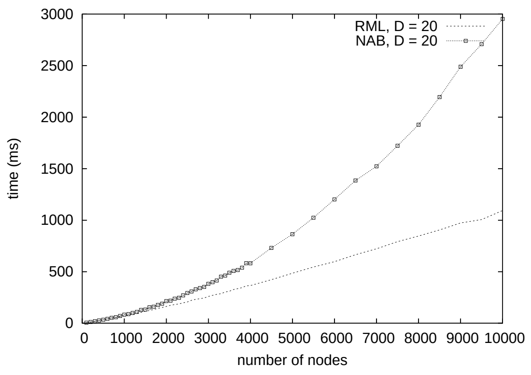
Fig. 6(a) represents the execution time of one second of the network life. In this simulated network, each node moves, and emits one packet per second.

Simulation times are longer with NAB than the ReactiveML implementation but this comparison is unfair. Indeed NAB simulates the MAC layer such that routing a packet is much more time consuming than in our simulator. Because neighborhood discovery is time consuming (about 25% of the simulation time with the optimized version), an interesting comparison with NAB is, thus, the packet-free simulations. In this case, we compare only the neighborhood discovery. The MAC layer does not affect the simulation such that, the two simulators have to do exactly the same thing. The execution time is given in Fig. 6(b).

Moreover, our simulator uses less memory than NAB. For example, the memory usage for the simulation of a 5000 nodes network without packet emission is about 550 MB in ReactiveML and 650 MB in NAB.



(a) With packets emission.



(b) Without packet emission.

Figure 6: Comparison of simulation times, between NAB and ReactiveML simulator, depending on the number of nodes with a topology density D=20.

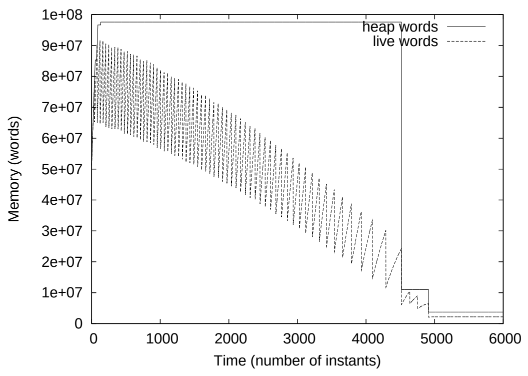


Figure 7: Memory usage of a simulation where a node is removed at each instant.

3.4 Dynamic Extension

In ad hoc networks, protocols must be robust to topology changes, which includes nodes join and leave. Thus, nodes can be added or removed dynamically.

Preemptible nodes are defined using the construct `do/until` that executes its body until a signal is emitted:

```
let process preemptible_node pos_init move make_msg kill =
  do
    run (node pos_init move make_msg)
  until kill done
```

Here, when the signal `kill` is emitted, the node is removed from the simulation.

Fig. 7 gives the memory usage of a simulation where one node is removed at each instant. It shows that the garbage collector works well and deallocate the memory used by the removed processes.

A more interesting point is the dynamic creation of processes. In ReactiveML, dynamic creation is made through recursion. We define the recursive process `add` that creates new nodes as follows:

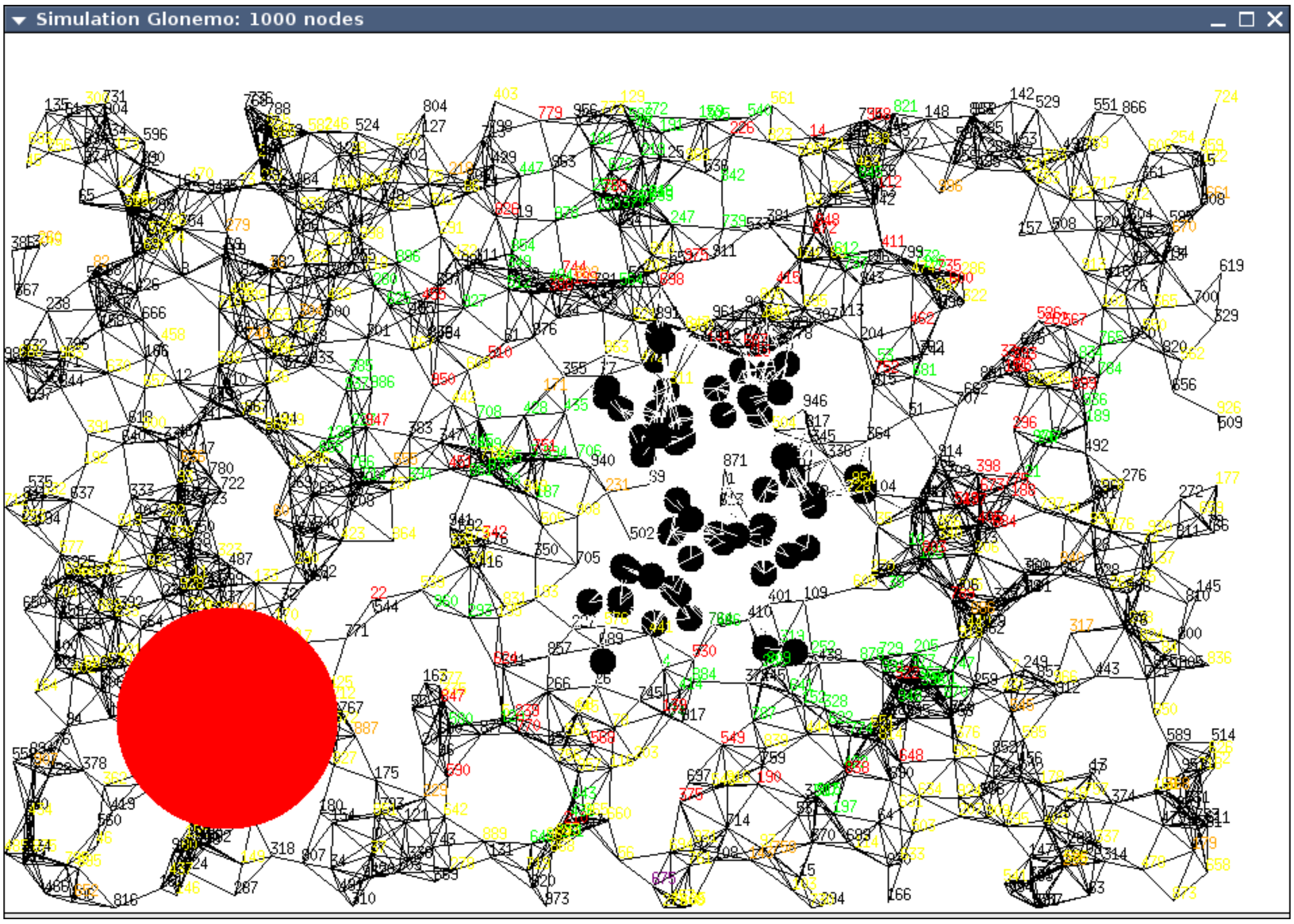


Figure 8: Screen-shots of the simulator graphical interface. The red disk represents a toxic cloud. Black disks are nodes without energy.

```
let rec process add new_node =
  await new_node (pos) in
  run (add new_node)
  ||
  run (node pos
           (random_waypoint (random_pos()))
           make_msg)
```

This process is parametrized by the signal `new_node`. `new_node` is emitted (with an initial position) when a new node has to be created.

4 Simulation of Sensor Networks

In this part, we detail the programming of another simulator in ReactiveML. This simulator is dedicated to sensor networks and, this time, we design a finer simulation, where all the network layers are modelised. We call it Glonemo (for **glo**bal **ne**twork **mo**del). See [36].

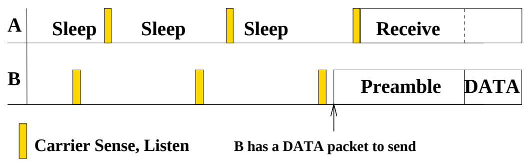


Figure 9: Medium Access Control: the preamble sampling technique.

In our example, the network has to warn when a toxic cloud is detected and the goal is to have a global sensor network simulator that give an accurate estimation of the energy consumption.[7]

4.1 Structure of the Simulator

4.1.1 Hardware Model

In order to obtain an accurate energy consumption evaluation, a hardware model is needed. Indeed, without this modeling, the energy would have to be evaluated using other observations (like the number of emitted packets) and abstractions. The accurate model of the hardware is easily described in ReactiveML, it contains several automata, one for each consuming part: radio, CPU, and memory.

4.1.2 Medium Access Control

The radio module is an important source of consumption for sensor nodes. To reduce that consumption, there exists specific Medium Access Control (MAC) protocols for sensor networks. Those protocols usually minimize the time the radio is alight [31, 15]. Thus, to analyze the energy consumption of a sensor network, the MAC layer cannot be omitted.

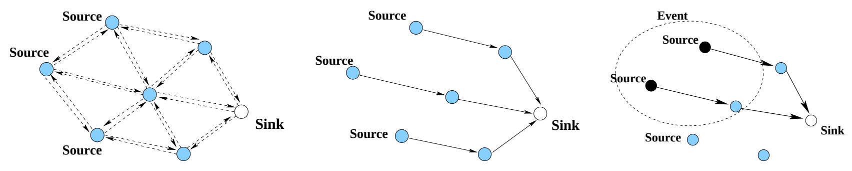
The sensor networks MAC protocol that has been implemented here is a Preamble Sampling MAC protocol (see Fig. 9 for details), like WiseMAC [15] and BMAC [31]. In the preamble sampling technique, a preamble precedes each data packet for alerting the receiving node. All nodes in the network sample the medium with a common period, but their relative schedule offsets are independent. If a node finds the medium busy after it wakes up and samples the medium, it continues to listen until it receives a data packet or the medium becomes idle again. The size of the preamble is set to be equal to the preamble sampling period.

4.1.3 Routing

As for the MAC layer, the routing protocols are also specific in sensor networks. Two of these are flooding and Directed Diffusion [22]. We implemented both. In flooding, each node receiving a packet repeats it by broadcasting unless it had previously sent this packet. This mechanism is useful for the network management, since some messages have to reach the whole network.

Directed diffusion is a data-centric routing used to collect data in sensor networks. In sensor networks, one (possibly several) node, called “sink”, is usually used or dedicated to collect data from other nodes. This routing protocol has three steps: (a) first, the sink floods an interest message, which is a task description to the whole network, (b) the sensors set up gradients, and (c) when a source has data for the interest, it sends the packet to the sink along the interest's gradient path. Fig. 10 illustrates this routing protocol on an example.

[7]Screen-shot Fig. 8



(a) Broadcast interest (b) Setup gradient (c) Send alarm

Figure 10: Routing: An example of directed diffusion.

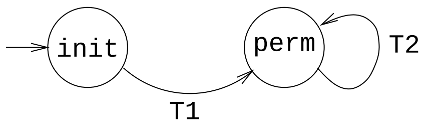


Figure 11: The automaton described by Lucky.

4.1.4 Application

A sensor network is dedicated to a particular application. The whole protocol stack depends on this application. In our example, the network that we simulate has to send an alarm to the sink in case of a danger (here that a toxic cloud is detected).

The environment is the source of (almost) all the activity that occurs in the network. It is not realistic to have independent stimuli that activate the sensors [35]. A sensor network simulator has to include a model of the environment. The model used here has been implemented using Lucky.

Lucky [23] is a programming language for the description of non deterministic reactive systems. It is a part of the Lurette [33] tool box, an automatic testing tool for reactive programs.

A Lucky program defines a set of input variables, a set of output variables, and an automaton with constraints on transitions. The outputs generated respect the constraints that may involve the inputs and the previous values of the outputs. The execution of a Lucky program is a synchronous system. At each step, the Lucky process reads the inputs, takes a transition where constraints can be satisfied, and generates random outputs that satisfy the constraints.

Let us point out that the description of the whole network is in the same formalism. Indeed, Lucky is actually a reactive language as ReactiveML.

In Glonemo, a toxic cloud moves according to the direction and speed of the wind. The model consists in two processes, one for a two-dimensional wind, which does not vary a lot and another for a cloud.

The Lucky code for the `wind` process is the following:

```
inputs { }
outputs {
    wind_x : float;
    wind_y : float;
}
start_node { init }
transitions {
 init -> perm // transition T1
   ~cond
      wind_x = 0.0 and wind_y = 0.0;

 perm -> perm // transition T2
```

```
   ~cond
      abs (wind_x - pre wind_x) < 1.0 and
      abs (wind_y - pre wind_y) < 1.0 and
      abs wind_x < 5.0 and abs wind_y < 5.0
}
```

This LUCKY program defines a two states automaton (see Fig. 11) with two output variables `wind_x` and `wind_y`. The constraints on the outputs are defined at the transitions. For those conditions, the keyword of the language in LUCKY is `~cond`. Here, transition `T0` sets the initial values of `wind_x` and `wind_y` to `0.0`. The transition `T1` guarantees that, at each activation, the values of the output variables are closed to their previous values.

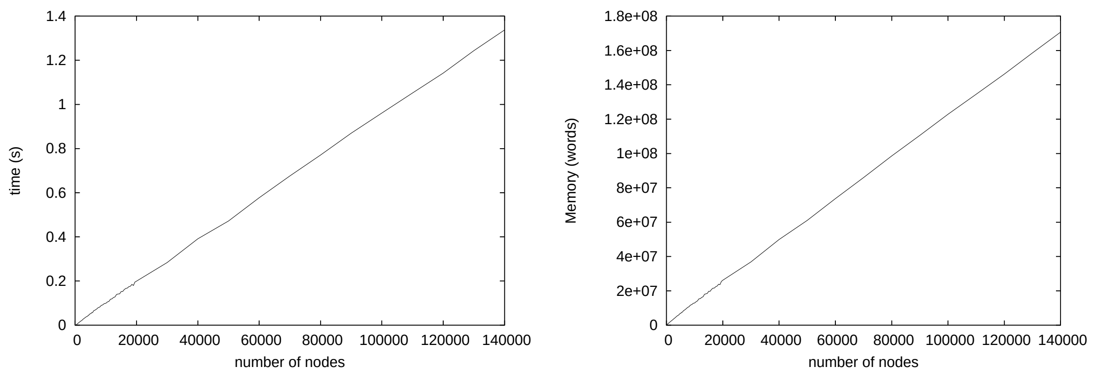
The cloud is a disk whose center has the coordinates `cloud_x` and `cloud_y`. Similarly, it is defined by an automaton where `wind_x` and `wind_y` are the inputs and `cloud_x` and `cloud_y` the outputs.

```
inputs {
  wind_x : float;
  wind_y : float;
}
outputs {
  cloud_x: float;
  cloud_y: float;
}
start_node { init }
transitions {
 init -> perm // transition T0
   ~cond
      cloud_x = 0.0 and cloud_y = 0.0;

 perm -> perm // transition T1
   ~cond
      (if wind_x >= 0.0
       then ((cloud_x - pre cloud_x) >= 0.0
         and (cloud_x - pre cloud_x) <= wind_x)
       else ((cloud_x - pre cloud_x) <= 0.0
         and (cloud_x - pre cloud_x) >= wind_x))
    and
      (if wind_y >= 0.0
       then ((cloud_y - pre cloud_y) >= 0.0
         and (cloud_y - pre cloud_y) <= wind_y)
       else ((cloud_y - pre cloud_y) <= 0.0
         and (cloud_y - pre cloud_y) >= wind_y))
}
```

These LUCKY programs can be imported into REACTIVEML and turned into processes parameterized by their input and output variables. Parameters become REACTIVEML signals. The behavior of the process is to read the value associated to the input signals and to emit the value computed by LUCKY on the output signal at each step.

We illustrate it with the following cloud example:

```
external.luc cloud_lucky
  {wind_x : float; wind_y : float;}
  {cloud_x: float; cloud_y: float;} = ["cloud.luc"]
val cloud_lucky :
  ('a, float) event * ('b, float) event ->
```



(a) Simulation times of one logical instant.

(b) Memory usage.

Figure 12: Simulation times and memory as a function of number of nodes.

```
  (float, 'c) event * (float, 'd) event ->
  unit process
```

Here, we create a process named `cloud_lucky`. The inputs `wind_x` and `wind_y` must be signals of type *('a, float) event* such that the value associated to the signals are floats. The outputs `cloud_x` and `cloud_y` have type *(float, 'a) event* since the process emits values of type *float*. In the same way, we can create the process `wind_lucky`.

To observe particular behaviors for the cloud without having to program them, it is useful for the user to be able to modify the cloud position during the simulation.

When the simulator is executed with a graphical interface, the `fan` process reads keyboard inputs and generates a particular wind. This process has the following interface:

```
val fan :
  (float, 'a) event * (float, 'b) event ->
  unit process
```

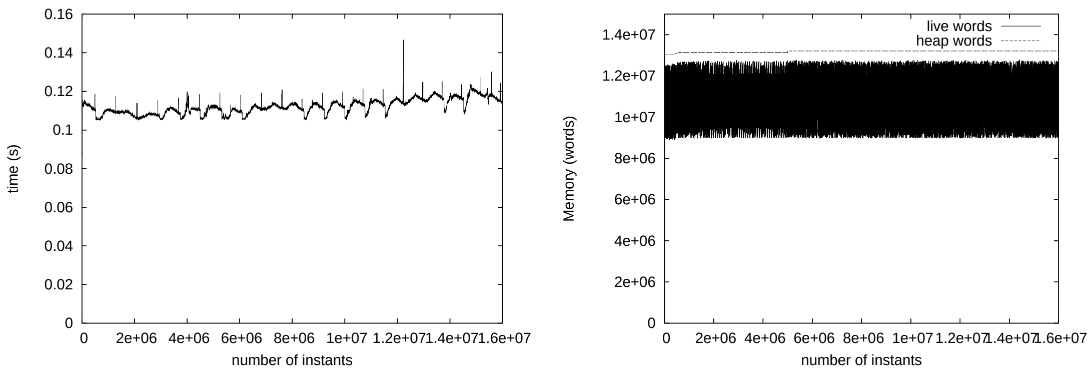
So, interactive simulations can simply be done by the parallel composition of the processes `wind_lucky` and `fan`. Winds produced by the Lucky process and the `fan` are combined through the signals `wind_x` and `wind_y`:

```
  signal wind_x default 0.0 gather (+.) in
  signal wind_y default 0.0 gather (+.) in
  run (wind_lucky () (wind_x,wind_y))
  ||
  run (fan (wind_x,wind_y))
```

The keyword `gather` at the declaration of a signal defines how to combine multiple emissions on a signal at the same instant. Here `(+.)` means that the wind produces by the process `wind_lucky` and the wind of the fan, `fan`, will be added.

4.2 Benchmarks and Scalability

Sensor networks are huge systems composed by thousands or even millions of nodes. Thus, a simulator dedicated to sensor networks must be able to simulate such a high number of elements. In this section, we



(a) Simulation times of one logical instant. (b) Memory usage.

Figure 13: Simulation of 10000 nodes during 20 days.

discuss the capacity of GLONEMO to execute such networks. We measure both the time of the simulation and the memory usage.

GLONEMO focuses on the energy consumption. That is why a fine grain simulation is needed. Indeed, to model the energy consumption in an accurate way, a model of the hardware is introduced in the execution of the simulator. The time scale for that is really small when compared to the time involved at the network layers. In GLONEMO, the execution of one logical instant represents 10^{-2} seconds. Thus, to simulate the behavior of a network during one hour, we need 360000 instants. For a 10000 nodes network, such a simulation takes about 11 hours. On Fig. 12(a), we print execution time of one single instant in terms of number of nodes. This time appears to be linear with the number of nodes. For a 140000 nodes network, the execution time of one instant takes about 1.4 second. This is a long simulation but regarding the memory (Fig. 12(b)), this simulation takes only 700.

On Fig. 13, we plot the speed and memory of a given simulation with a fixed number of nodes. The memory needed to run the simulation is constant (see Fig. 13(b)). This ensures that a simulation will not swap during an execution. Moreover, the time spent to execute one instant is constant during the whole simulation (see Fig. 13(a)). GLONEMO, written in REACTIVEML, is able to simulate in an accurate way more than 100000 nodes.

5 Related Works

Network simulators are extensively used in the network community research thus many relevant simulators have been developed. We describe now the distinguishing features of some of them.

Network simulators In 2000, Breslau et al. [12] defend the need for a single simulator for the research community. This was the VINT project leading to the NS-2 simulator [1]. Indeed, NS-2 is one of the most popular simulator in the research community. It is a packet-level simulator, initially designed for wired networks. NS-2 is a discrete event simulator. The interest of having one single simulator is to enable comparison between different protocols without the need to implement the protocol we want to compare with. Indeed, NS-2 offers a large protocol library. However, even if NS-2 provides several levels of abstraction (four according to [12]), it is more effective to implement the needed exact level of abstraction. This is why some people still write stand-alone simulators. Moreover, NS-2 is not really scalable and is convenient for simulating a few hundred of nodes only.

To overcome the scalability limitation of NS-2, people propose *Parallel Discrete Event Simulation* [18] where the simulator is distributed among several machines. GTNetS [34] is developed with this paradigm. This is a complementary approach to a centralized implementation as provided in ReactiveML. The two dedicated simulators implemented in ReactiveML appear to be scalable enough to run on a single machine.

NAB [16] is a network simulator written in ML. The arguments for using ML instead of C are the same as ours, but ReactiveML provides parallelism as a primitive construct.

Network simulators for sensor networks Sensor networks are a new kind of ad hoc networks that interest the research community. Those networks have different characteristics and new constraints, thus new simulators are needed. Because one of the key issue in sensor networks is the power consumption, people began to develop simulators that take into account the energy consumption. Avrora [41] and Atemu [32] are cycle-accurate simulators (RTL level). With that level of details, scalability is probably hopeless.

Environment simulators for wireless sensor networks The classical network simulator do not have realistic environment models. People from the field of sensor network simulation are faced with the issue of having a model of the physical environment that activates the sensors. We present in this section the existing approach.

Sridharan et al [39] propose to link a Matlab environment simulator with the sensor network simulator TOSSIM. Their approach suits their problem: the example taken is the monitoring of the health of a building structure. This is modeled as a state-space system which follows a differential equation. Matlab is appropriate for that application because it is dedicated to solve differential equations.

Another approach is to do an analogy between the sensing phenomenon and the radio waves. In several sensor network simulators (SensorSim [30], J-Sim [38] and an extension of NS-2 by Downard [14]), another channel like the radio channel is created but it is a *sensor* channel. On this channel phenomena propagate. Some approaches define new propagation models: Seismic and Acoustic in J-Sim. Others use the radio propagation model of their simulator to model the propagation of the phenomenon.

The analogy between the radio wave propagation and the propagation of a phenomenon is restrictive. It implies that the phenomenon propagates. In our approach, with Lucky, we can be more general, we manage to model different kind of environment.

Synchronous Languages Finally, it would have been difficult to implement our two simulators in a synchronous language like Lustre [21], Esterel [7] or Signal [20] for at least two reasons: the use of complex data structures that are shared between the reactive part and the computational one, and the dynamic creation which is not allowed in these languages.

6 Conclusion and Perspectives

From the observation that generic network simulators are not always satisfactory and that users still develop their own simulators from scratch, we propose the use of the reactive model to program them. This model is dedicated to the programming of systems with a large number of parallel processes and communications. This is typically the case of network simulators.

Two different simulators have been considered: a coarse-grained one (ELIP) and a fine-grained one (Glonemo). Both simulators, with the graphical interface, were defined in less than 2000 lines of ReactiveML. It is easy to define data structures describing nodes and packets. Moreover, the reactive model appeared to be well adapted for both the description of mobility in ELIP and to the modular description of the different network layers in Glonemo. Finally, the underlined model of concurrency of reactive programs states that every node of the network reacts synchronously during an instant. This makes the correspondence between the logical time and the simulation time.

The link between ReactiveML and Lucky allowed to simulate the physical external environment in Glonemo. This point is particularly important for sensor networks since a naive model of the environment does not give relevant simulation results.

Simulators were efficient enough and robust to obtain useful simulation metrics [4, 5, 6]. It is clearly possible to develop more efficient simulators than ELIP and GLONEMO but it appears that there was a good compromise between the development time and the simulators efficiency.

This work offers several perspectives, some concerning the simulators by themselves and some concerning REACTIVEML. For the GLONEMO simulator, it would be interesting to have several levels of simulation: a fine-grained simulator in order to obtain an accurate estimation of the energy consumption, and a faster simulator which gives information about higher layers. Understanding how to write such a multi-level simulator is a challenging direction.

Another direction is the use of formal validation techniques and tools for reactive programs. Technically, this means extracting models in a form usable by the validation tools. In GLONEMO, for example, we would like to prove two kinds of properties. The first one is the validation of the *abstractions* that are needed for the model to be of a reasonable complexity. For instance, we think that we should never include a full description of the hardware to the model, at the abstraction level which is needed for precise energy evaluations, i.e., the RTL (Register Transfer Level) level. But if we include an abstraction of it, we should *prove* that: (a) an abstraction of the real hardware is required, and (b) the composition with the rest of the model preserves this abstraction. The second kind is the verification of *global* properties such as: *after time T, the system still has more than x % of nodes alive.* Verifying reactive programs with dynamic creation of processes is still largely an open problem. Establishing close relations between the reactive model and process algebra could give some useful insight [3].

The key perspective is to use REACTIVEML not only to simulate ad hoc networks but also other embedded systems. There is a first experiment with the simulation of a gyroscopic system. This example is taken from the avionic industry. It deals with the treatment of position variations of an airplane [28]. An interesting point here is how to extract the embedded (real-time) software from the REACTIVEML program.

References

[1] The Network Simulator - ns-2.

[2] Raúl Acosta-Bermejo. *Rejo - Langage d'Objets Réactifs et d'Agents.* PhD thesis, Ecole des Mines de Paris, 2003.

[3] R.M. Amadio and F. Dabrowski. Feasible reactivity for synchronous cooperative threads. In *Extended abstract presented at the workshop Expressiveness in Concurrency*, San Francisco, September 2005.

[4] F. Benbadis, M. Dias de Amorim, and S. Fdida. 3P: Packets for positions prediction. In *Proceedings of IEEE INFOCOM students workshop'05*, Miami, FL, USA, 2005.

[5] F. Benbadis, M. Dias de Amorim, and S. Fdida. Dissémination prédictive des coordonnées pour le routage géographique basé sur l'âge. In *Proceedings of CFIP 2005 Conference*, Bordeaux, France, 2005.

[6] F. Benbadis, M. Dias de Amorim, and S. Fdida. ELIP: Embedded location information protocol. In *Proceedings of IFIP Networking 2005 Conference*, Waterloo, Canada, 2005.

[7] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 425–454. MIT Press, 2000.

[8] F. Boussinot and L. Hazard. Reactive scripts. In *RTCSA '96: Proceedings of the Third International Workshop on Real-Time Computing Systems Application (RTCSA '96)*, page 270, Washington, DC, USA, 1996. IEEE Computer Society.

[9] Frédéric Boussinot. Concurrent programming with Fair Threads: The LOFT language, 2003.

[10] Frédéric Boussinot and Robert de Simone. The SL synchronous language. *Software Engineering*, 22(4):256–266, 1996.

[11] Frédéric Boussinot and Jean-Ferdy Susini. The SugarCubes tool box : A reactive java framework. *Software Practice and Experience*, 28(4):1531–1550, 1998.

[12] Lee Breslau, Deborah Estrin, Kevin R. Fall, Sally Floyd, John S. Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, 2000.

[13] Christian Brunette. *Construction et simulation graphiques de comportements: le modèle des Icobjs*. PhD thesis, Université de Nice-Sophia Antipolis, 2004.

[14] Ian Downard. Simulating sensor networks in ns-2, 2005.

[15] Christian C. Enz, Amre El-Hoiydi, Jean-Dominique Decotignie, and Vincent Peiris. Wisenet: An ultralow-power wireless sensor network solution. *IEEE Computer*, 37(8):62–70, 2004.

[16] EPFL. Network in A Box.

[17] Deborah Estrin, Mark Handley, John Heidemann, Steven McCanne, Ya Xu, and Haobo Yu. Network visualization with nam, the vint network animator. *Computer*, 33(11):63–68, 2000.

[18] Richard M. Fujimoto, Kalyan S. Perumalla, Alfred Park, Hao Wu, Mostafa H. Ammar, and George F. Riley. Large-scale network simulation: How big? how fast? In *MASCOTS*, page 116. IEEE Computer Society, 2003.

[19] Matthias Grossglauser and Martin Vetterli. Locating nodes with EASE: Last encounter routing in ad hoc networks through mobility diffusion. In *Proceedings of IEEE Infocom*, March 2003.

[20] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Lemaire. Programming real-time applications with signal. *Proc. of the IEEE*, 79(9):1321–1336, September 1991.

[21] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language lustre. *Proc. of the IEEE*, 79(9):1305–1320, September 1991.

[22] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MOBICOM*, pages 56–67, 2000.

[23] Erwan Jahier and Pascal Raymond. The lucky language reference manual. Technical report, Unité Mixte de Recherche 5104 CNRS - INPG - UJF, 2004.

[24] Xavier Leroy. The Objective Caml system release 3.09. Documentation and user's manual. Technical report, INRIA, 2006.

[25] Louis Mandel. *Conception, Sémantique et Implantation de ReactiveML : un langage à la ML pour la programmation réactive*. PhD thesis, Université Paris 6, 2006.

[26] Louis Mandel and Farid Benbadis. Simulation of mobile ad hoc network protocols in ReactiveML. In *Synchronous Languages, Applications, and Programming (SLAP'05)*, Edinburgh, Scotland, April 2005. ENTCS.

[27] Louis Mandel and Marc Pouzet. ReactiveML, a reactive extension to ML. In *ACM International conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005.

[28] Lionel Morel and Louis Mandel. Executable contracts for incremental prototypes of embedded systems. Submitted to publication, 2006.

[29] OPNET Modeler. `http://www.opnet.com`.

[30] Sung Park, Andreas Savvides, and Mani B. Srivastava. Sensorsim: a simulation framework for sensor networks. In *MSWIM '00: Proceedings of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 104–111, New York, NY, USA, 2000. ACM Press.

[31] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107, New York, NY, USA, 2004. ACM Press.

[32] Jonathan Polley, Dionysys Blazakis, Jonathan McGee, Dan Rusk, and John S. Baras. ATEMU: A Fine-grained Sensor Network Simulator. *Secon*, 2004.

[33] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[34] George F. Riley. The georgia tech network simulator. In *MoMeTools '03: Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, pages 5–12, New York, NY, USA, 2003. ACM Press.

[35] Ludovic Samper, Florence Maraninchi, Laurent Mounier, Erwan Jahier, and Pascal Raymond. On the importance of modeling the environment when analyzing sensor networks. In *Proceedings of International Workshop on Wireless Ad-Hoc Networks 2006 (IWWAN 2006)*, page 7, New York, United States, June 2006.

[36] Ludovic Samper, Florence Maraninchi, Laurent Mounier, and Louis Mandel. GLONEMO: Global and accurate formal models for the analysis of ad-hoc sensor networks. In *Proceedings of the First International Conference on Integrated Internet Ad hoc and Sensor Networks (InterSense'06)*, Nice, France, May 2006.

[37] Manuel Serrano, Frédéric Boussinot, and Bernard Serpette. Scheme fair threads. In *Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 203–214. ACM Press, 2004.

[38] Ahmed Sobeih, Wei-Peng Chen, Jennifer C. Hou, Lu-Chuan Kung, Ning Li, Hyuk Lim, Hung-Ying Tyan, and Honghai Zhang. J-sim: A simulation environment for wireless sensor networks. In *Annual Simulation Symposium*, pages 175–187, 2005.

[39] Avinash Sridharan, Marco Zuniga, and Bhaskar Krishnamachari. Integrating environment simulators with network simulators. Technical report, University of Southern California, 2004.

[40] Jean-Ferdinand Susini. *L'approche réactive au dessus de Java : sémantique et implémentation des SugarCubes et de Junior*. PhD thesis, Ecole des Mines de Paris, 2001.

[41] Ben L Titzer, Daniel K Lee, and Jens Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. *Proceedings of IPSN*, 2005.