# Early Prototyping of Wireless Sensor Network Algorithms in PVS*

Cinzia Bernardeschi[1], Paolo Masci[1], and Holger Pfeifer[2]

[1] Department of Information Engineering, University of Pisa, Italy
{cinzia.bernardeschi,paolo.masci}@iet.unipi.it
[2] Institute of Artificial Intelligence, Ulm University, Germany
holger.pfeifer@uni-ulm.de

**Abstract.** We describe an approach of using the evaluation mechanism of the specification and verification system *PVS* to support formal design exploration of WSN algorithms at the early stages of their development. The specification of the algorithm is expressed with an extensible set of programming primitives, and properties of interest are evaluated with ad hoc network simulators automatically generated from the formal specification. In particular, we build on the *PVSio* package as the core base for the network simulator. According to requirements, properties of interest can be simulated at different levels of abstraction. We illustrate our approach by specifying and simulating a standard routing algorithm for wireless sensor networks.

**Keywords:** WSN algorithms, simulation, PVS.

## 1 Introduction and Motivation

Wireless Sensor Networks (WSNs) are distributed systems consisting of a large number of spatially distributed, autonomous and cooperating nodes. The nodes of the network, referred to as *sensor nodes*, are battery-operated devices which provide limited computation capabilities, low-rate and low-range wireless communication, and are equipped with a number of sensors and actuators to monitor physical or environmental conditions. The most characterising aspect of WSNs is that they are deeply embedded in the real world, and provide unattended operation for long periods of time without infrastructural support. Due to their small size, sensor nodes can be placed in close proximity to the subject to be monitored, thus enabling *in situ* monitoring of physical phenomena. A sensor network normally constitutes a wireless *ad hoc* network, in which communication is multi-hop: due to the limited operating distance of the wireless radio compared to the physical extension of WSNs, sensor nodes must coordinate communication to forward data to a distant receiver. WSNs are highly dynamic networks: even if nodes are placed in fixed positions, node failures (e.g., due to software

* This work was partially supported by the European Commission through the Network of Excellence ReSIST (IST-026764).

M.D. Harrison and M.-A. Sujan (Eds.): SAFECOMP 2008, LNCS 5219, pp. 346–359, 2008.

bugs or battery exhaustion), or environmental factors that are difficult to predict or avoid (e.g., physical obstacles, or humidity) may unexpectedly alter the connectivity of the network. For some application scenarios, mobile nodes may be involved in communication as well. Initially developed for military purposes such as battlefield surveillance, applications of WSNs today cover a wide spectrum of scenarios, including many safety-critical domains. For instance, WSNs have been deployed in critical infrastructures monitoring [1] to assess structural health of buildings, such as a pedestrian footbridge [2], or roads. In the area of traffic monitoring and control, a distributed application built on top of a WSN has been developed [3] to monitor a railway network for accidental or malicious system failures so as to prevent derailment of trains or even collisions. Wireless sensor networks are also deployed in health-care applications, for example to monitor vital signs of patients through tiny wearable sensor nodes [4], or as support and emergency systems for elderly people [5].

Large-scale networks are difficult to test, and the characteristics of wireless sensor networks only compound the problem. Hence, simulation plays a central role in current development processes of WSN applications. Software-based simulators are used to provide controlled environments in which experiments are to yield reproducible results. During the early stages of development, applications are commonly analysed with ad hoc simulators built as extensions of existing network simulators, such as *ns2*, or distributed system simulators, such as *ptolemy*. Once the application logic becomes consolidated, the software is evaluated in dedicated network simulators that provide emulation of real WSN hardware, such as *TOSSim* and *Avrora*. As of yet there is, however, no established standard simulation framework for WSN applications, and extensions to network simulators can usually only be accomplished by users who are familiar with the tool [6].

Inherently, simulations can only approximate real-world computation, and the challenge is to develop models that capture the behaviour of the environment in which WSNs are going to be deployed as accurately as possible. Recent studies have evidenced that for wireless networks simulations there is high risk of misleading or incorrect results because of assumptions hidden in the underlying network simulator [7]. Even in the presence of positive simulation results, failures may still occur when the system is deployed. As with any other system, remedying defects at this stage is costly at best, for WSNs it can be even impossible. Consequently, support for analysing WSN algorithms at early stages of development is essential. With a view to the reliability and safety requirements of applications of WSNs as the ones mentioned above, more rigorous analytical techniques are also desirable.

In this paper we report on our work towards developing a simulation and analysis framework for WSN algorithms within a theorem prover. Specifically, we use the *Prototype Verification System (PVS)* [8, 9] to specify and simulate WSN communication protocols in the very early stages of their design. The distinguishing characteristics of PVS are its expressive specification language and its powerful theorem prover. A less often used component is its *ground*

*evaluator* [10] that can be used to animate functional specifications. Although PVS's specification language is based on higher-order logic and features a rich type system, a surprisingly large subset of it is executable. The ground evaluator translates the executable constructs of PVS into efficient *Lisp* code which can then be executed. The evaluation environment consists of a *read-eval-print*-loop that reads PVS expressions from the user and returns the result of their evaluation. The additional package *PVSio* [11] enhances PVS's specification language with built-in constructs for string manipulation, floating-point arithmetic and input/output operations. Thus, for certain types of applications, PVS can effectively serve as a functional programming language.

We employ the combination of PVS's rich specification language and the ground evaluator for early prototyping of a class of WSN algorithms. To this end, we introduce a series of general formal PVS models that can be refined to describe various WSN communication protocols. Specifically, we provide an extensible set of executable communication primitives to enable rapid and easy specification of protocols at different levels of detail. From these formal algorithm specifications, efficient Lisp code can automatically be generated using the ground evaluator and the PVSio extension. This implementation is suitable for simulation and allows to test and evaluate the algorithm from different perspectives. Finally, once the simulation experiments give sufficient confidence in the correctness of the algorithm, the PVS models can serve as the basis for the formal verification of the desired properties using the PVS theorem prover.

To demonstrate the effectiveness of our approach, we describe its application to the *Surge* routing protocol, which is used in various WSN systems, including prototypes in a safety-critical domain [4]. We analysed the *Surge* protocol under different aspects. During our tests of robustness of the protocol with respect to topology changes, we were able to detect a potential problem of routing loops that has gone unnoticed so far and can indeed be reproduced with one of the implementations of *Surge* provided in the library of the widely-used WSN operating system *TinyOS*.

## 2 PVS and PVSio

PVS is a specification and verification system which combines an expressive specification language with an interactive proof checker. It has been used for formal reasoning in several application domains (see [8] for an overview).

The PVS specification language builds on classical typed higher-order logic with the usual base types, `bool`, `nat`, `integer`, `real`, among others, and the function type constructor `[A -> B]`. Predicates are simply functions with range type `bool`. The type system of PVS also includes record types, dependent types, and abstract data types. The most powerful concept are *predicate subtypes*; e.g., the type `below(n:nat) : TYPE = {s: nat | s < n}` denotes the type of natural numbers less than a given bound `n`. Usage of predicate subtypes ranges from checking for violations such as division by zero, to expressing complex consistency requirements.

PVS specifications are packaged as *theories* that can be parametric in types and constants. A built-in prelude and loadable libraries provide standard specifications and proved facts for a large number of theories. A theory can use the definitions and theorems of another theory by importing it. For instance, consider the following theory `execution`:

```
execution [State : TYPE] : THEORY
BEGIN
  trans : VAR [State -> State]
  execute(trans)(n:nat) : RECURSIVE [State -> State] =
    LAMBDA (s:State): IF n = 0 THEN s
                        ELSE LET s_new = trans(s) IN
                               execute(trans)(n-1)(s_new)
                      ENDIF
      MEASURE n
END execution
```

The theory takes one type parameter, `State`, and defines a (higher-order) function `execute` that recursively applies `n` steps of a state-transition function `trans`, which is provided as a parameter. As all functions in PVS must be total, the termination of the recursion has to be demonstrated; the `MEASURE` part provides the information to the typechecker and prover to ensure this. By instantiating the theory parameter with a concrete value, the `execute` function can be imported into the context of a given algorithm specification:

```
simulation : THEORY
BEGIN
   ...   % -- concrete def'n of a state type omitted
   % -- definition of a single step of the algorithm
   algorithm_step(s:State): State = ... % -- omitted
   IMPORTING execution[State]
   % -- execution of 'steps' number of steps of the algorithm
   algorithm(steps:nat) : [State -> State] =
      execute(algorithm_step)(steps)
END simulation
```

Thus, the `execution` theory provides a generic mechanism to describe the execution of an algorithm, which can subsequently be used for simulation.

Using the PVS ground evaluator one can compile the executable constructs of a specification, such as the `execute` function above, into efficient *Lisp* code. In order to still be able to simulate theories that also involve declarative specifications, the ground evaluator is augmented by so-called *semantic attachments*, through which the user can supply pieces of Lisp code and attach them to the declarative parts. Using this mechanism the *PVSio* package [11] extends the ground evaluator with a predefined library of imperative programming language features such as side effects, unbounded loops, and input/output operations, and also provides a high-level interface for writing user-defined semantic attachments. Thus, PVS specifications can conveniently be animated within the *read-eval-print*-loop of the ground evaluator.

## 3 Prototyping WSN Algorithms

In this section we present the basic aspects of the proposed approach. We show that prototyping of WSN algorithms can be performed with a collection of PVS models (theories), each of which represents a service installed on a sensor node (e.g., packet logger, clock), or structural properties of the network (e.g., the network graph), or communication functionalities (e.g., packet forwarding). For each theory, a number of different versions can be provided in order to specify and analyse WSN algorithms under several perspectives and at desired level of detail. The most abstract theory provides *i)* the declaration of types for a minimum set of mandatory attributes, *ii)* the declaration of interface functions. More detailed theories can be derived from the abstract definition by specifying the behaviour of interface functions, and by extending types. Abstractions enable users to create a model comprising only the parameters of interest at the desired levels of detail. Hence, lightweight models can be generated, and efficient code for simulations can be obtained.

**Network Connectivity.** Network connectivity is modelled with a directed graph without self-edges. We build type definition on top of directed graphs of the NASA library [12], in order to benefit from several useful lemmas and properties already proved in PVS. Custom network graphs can be generated. To simplify graph specification, we use an auxiliary topology function that identifies, for each node, the set of neighbouring nodes. Ideal and lossy links can be modelled, and topology changes can be used to model node mobility. Once the topology is given, the network graph can be instantiated with a specific interface function. Sensor nodes are identified by a unique natural number less that a given N. We developed a theory named `node_th` that provides a type definition for the node identifier. In the following, the theory which describes network connectivity is shown (`digraph[node_id]` is the NASA theory on directed graphs). An example of topology and the corresponding network graph are also included.

```
network_graph_th: THEORY
 BEGIN
IMPORTING node_th, digraphs[node_id]

   %-- network_graph: a directed graph without self-edges
   network_graph?(g: digraph[node_id]): bool =
       (FORALL (i: node_id): vert(g)(i)) AND
        (FORALL (i,j: node_id): edges(g)((i,j)) IMPLIES (i /= j))

   network_graph: TYPE = {g: digraph[node_id] | network_graph?(g)}
   topology: TYPE = [node_id -> finite_set[node_id]]
   new_network_graph(tp: topology): network_graph

   %-- instance of network graph
   fully_connected_network_graph: network_graph
      = new_network_graph(LAMBDA (i: node_id): {n: node_id | n /= i})
 END network_graph_th
```

**Services.** A service is identified by a unique name. A services `S` is associated to nodes by means of a function `[finite_set[node_id] -> S]`. Depending on the algorithm specification and on the property of interest, services can be installed on a single node, on a group of nodes, or on the entire network.

Currently, we have implemented the following services: *packet logger*, which stores statistics about sent and received packets, *receive buffer*, which models the buffer where packets sent by other nodes are stored, *energy consumption*, which evaluates the energy spent by nodes, *routing*, which provides the basic definitions for building routing tables, spanning trees and paths between nodes, and *node scheduler*, which gives the sequence of nodes that execute the algorithm (e.g., round robin, or random). As example of service, in the following we show the definition of energy consumption, where `energy` is the energy consumption of a sensor node, `network_consumption` is the function which associates the energy consumption to every node. Three interface functions are declared in the theory: two of them are used to compute energy consumption of senders and receivers, the other one to update consumption of the sender neighbours.

```
energy_th: THEORY
 BEGIN
   %-- type definition
   energy: TYPE = real
   network_consumption: TYPE = [node_id -> energy]
   %-- interface functions
   sender_consumption: energy
   receiver_consumption(g: network_graph, snd, rcv: node_id): energy
   update_network_consumption(ne: network_consumption,
                              g: network_graph,
                              snd: node_id): network_consumption
 END energy_th
```

Services are wrapped together into an extensible structure called *network state*. The network state is described by the set of functions that specify the allocation of services to nodes. For instance, in the following theory, a network state is defined as the collection of two services (receive buffer and log):

```
network_th: THEORY
 BEGIN
   network_state: TYPE =
        [# net_receive_buffer: [node_id -> receive_buffer],
           net_log: [node_id -> log] #]
 END network_th
```

The network state maintains the state of all nodes. The state of a node consists of the state of the services installed on a node. The state of a node can be obtained by indexing the network state: given a node $x$ and a network state `ns`, the state of $x$ is `ns(x)`, and the state of the logging service of $x$ is `net_log(ns)(x)`.

**Communication Primitives.** Nodes can exchange packets. A packet is a structure with two mandatory fields (the sender and the destination), and a number

of optional fields. The sender is a single node, while destination is specified with a finite set of nodes. Broadcast address is represented with a special constant `bcast_addr`, which is the full set of nodes. In the following example, a packet consists of five fields (timestamp, source, sender, destination and payload):

```
packet_th: THEORY
 BEGIN
IMPORTING node_th, time_th
   bcast_addr: finite_set[node_id] = LAMBDA (i: node_id): node_id?(i)
   packet: TYPE =
          [# timestamp: time,
             source_addr: node_id,
             sender_addr: node_id,
             destination_addr: finite_set[node_id],
             payload: finite_sequence[int] #]
 END packet_th
```

We modelled three low level single-hop primitives in order to easy the specification of communication algorithms: inject, forward and drop. Additionally, nodes are also allowed to perform an idle transition.

**Inject** can be used to send out packets generated by nodes (e.g., packet generated by the application executed on nodes, or control packets generated by the routing service): the function takes a packet as parameter, and sends out such packet.

**Forward** is suitable to relay packets previously received by nodes (e.g., when multi-hop communication is needed to reach the destination): the function takes a packet as parameter, removes the packet from the receive buffer of the node, and sends out a packet with a sender address automatically updated with the identifier of the sending node.

**Drop** is used to discard received packets: the function takes a packet as parameter, and removes such packet from the receive buffer of the node.

**Idle** is useful to update state variables of nodes, such as energy consumption, when no operation on incoming/outgoing packets is performed.

The implemented primitives are suitable for unicast, multicast and broadcast communication. The side effect of sending out the packet is that neighbouring nodes of the sender receive the packet. The graph connectivity affects reception of packets: if node $x$ sends out a broadcast packet, it is received only by neighbours of $x$. A basic version of the forward primitive with the essential functionalities is the following:

```
network_th_A: THEORY
 BEGIN
   IMPORTING receive_buffer_th
   network_state: TYPE = [# net_receive_buffer: network_receive_buffer #]
```

```
   % packet pk is sent out by the forwarder node
 forward(pk: packet)(forwarder: node_id)
       (net: network_state, g: network_graph): network_state =
    LET fw_pk = pk WITH [sender_addr := forwarder],
       nrb0 = update_receivers_buffer(net_receive_buffer(net), g, fw_pk),
       nrb1 = update_sender_buffer(nrb0, pk)
     IN net WITH [net_receive_buffer := nrb1]
 END network_th_A
```

The sender address of the packet is updated, and `update_receivers_buffer` and `update_sender_buffer` functions are invoked to update the network state. A more detailed version of the above primitive could be obtained by adding, for example, energy consumption and packet logger services to nodes. In this case, `energy_th` and `log_th` theories must be imported and the functions to update energy and log must be invoked.

**Algorithms.** An algorithm is specified as a cyclic procedure executed on a generic node. For instance, let us consider the flooding algorithm [13], which is designed to deliver packets to all nodes in the network. Flooding is typically used for dynamic route discovery, reconfiguration/reprogramming and to request specific data from sensors. A simple variant of flooding behaves as follows: whenever a node receives a packet, the packet is forwarded to neighbouring nodes if it is received for the first time, otherwise it is dropped. The algorithm can be specified as follows:

```
flooding_th: THEORY
 BEGIN
IMPORTING network_th, log_th
   flooding(x: node_id)(net: network_state, g: network_graph):
         network_state =
       IF empty?(net_receive_buffer(net)(x)) THEN idle(x)(net, g)
       ELSE LET pk = getpacket(net_receive_buffer(net)(x)) IN
              IF empty?(net_log(net)(x)(fw)) THEN forward(pk)(x)(net, g)
              ELSE drop(pk)(x)(net, g)
              ENDIF
       ENDIF
END flooding_th
```

Flooding can be analysed at different level of abstractions by importing specific theories and leaving the specification of the `flooding` function unmodified. This way, different properties of the algorithm can be analysed and different implementations can be evaluated. For instance, energy consumption can be analysed in the above theory by importing a different theory for the network state. In order to discover problems of the algorithm, the underlying services can be assumed to behave correctly. Conversely, the algorithm can be also analysed by modelling malfunctions of the underlying layers in order to evaluate, for instance, service degradation.

## 4 A Case Study: Surge

In this section we analyse *Surge*, a popular routing protocol for WSNs. Surge is currently part of the TinyOS distribution, and it has been used as routing service during the evaluation of several WSN-based systems, including prototypes for safety-critical systems [4]. We introduce Surge with an excerpt from [14]:

> The Surge protocol forms a dynamic spanning tree, rooted at a single node (*the base station*). Nodes route packets to the root. Nodes select a new parent when the link quality falls below a certain threshold. Surge suppresses cycles in the routing by dropping packets that revisit their origin.

**Modelling Surge.** The Surge algorithm can be decomposed into a forwarding service built upon a dynamic spanning tree service which, in turn, relies on lower level services for single-hop communication. In order to discover bugs in the specification, such services can be analysed separately, assuming that the underlying layers behave properly. Suppose that we are interested in analysing the forwarding service. The spanning tree service can be assumed correct, i.e., it provides a correct routing table `rt` to the forwarding service. Hence, the forwarding algorithm of Surge applied by a single node to a received packet can be formally specified in PVS as follows:

```
surge_th: THEORY
 BEGIN
IMPORTING network_th, routing_th

surge(x: node_id)(net: network_state, g: network_graph)
     (base_station: node_id, rt: routing_table): network_state =
    IF empty?(net_receive_buffer(net)(x)) THEN idle(x)(net, g)
    ELSE LET received_pk = getpacket(net_receive_buffer(net)(x)),
             source_addr = source_addr(received_pk),
             sender_addr = sender_addr(received_pk),
             next_hop = next_hop(x,base_station)(g,rt)
          IN IF source_addr /= x
             THEN forward(received_pk
                       WITH [destination_addr := next_hop])(x)(net, g)
             ELSE drop(received_pk)(x)(net, g)
             ENDIF
    ENDIF
 END surge_th
```

**Analysing Surge.** To analyse Surge, a WSN application must be specified in PVS. In the following, we will explore examples of analyses.

*Receive queue size.* We report the results of a simulation to evaluate receive queue size of sensor nodes in a monitoring scenario, in which sensor nodes periodically send packets to report data to the base station. The routing table is

assumed to be correct, but it may change. A scheduler that selects the nodes that execute the algorithm must be specified. Every time a node is selected by the scheduler, such node sends out a new packet and relays all packets of other nodes. The scheduler guarantees fairness of execution between nodes. The PVS theory for the monitoring application is the following, where node 0 is the base station. Two recursive functions are defined: `surge_rec`, which relays all received packets, and `surge_app`, which invokes the scheduler (`round_robin`) to select a node. The selected node, if different from the base station, sends out a packet and relays packets of the other nodes.

```
surge_app_th: THEORY
 BEGIN
IMPORTING surge_th
base_station: node_id = 0

surge_rec(x: node_id)(net: network_state, g: network_graph)
         (base_station: node_id, rt: routing_table): RECURSIVE
      network_state =
  IF empty?(net_receive_buffer(net)(x)) THEN idle(x)(net, g)
  ELSE LET net_prime = surge(x)(net, g)(base_station, rt)
         IN surge_rec(x)(net_prime, g)(base_station, rt)
  ENDIF
   MEASURE size(net_receive_buffer(net)(x))

surge_app(ti, tf: nat)
         (net: network_state, g: network_graph,
          rt: routing_table): RECURSIVE network_state =
 IF ti >= tf THEN net
 ELSE LET sender = round_robin(ti),
          dst = next_hop(sender, base_station)(g, rt)
     IN IF sender = base_station
        THEN surge_app(ti + 1, tf)(net, g, rt)
        ELSE
          LET net_inj = inject(new_packet(sender, dst))(sender)(net, g),
              net_prime = surge_rec(sender)(net_inj, g)(base_station, rt)
           IN surge_app(ti + 1, tf)(net_prime, g, rt)
         ENDIF
    ENDIF
  MEASURE tf - ti
 END surge_app_th
```

A simulation has been performed with grid networks of different size. Networks of hundreds of nodes can be simulated. Results for a 25 node network are shown in Figure 1. For each node (except for the base station) we have evaluated the maximum number of packets in the receive buffer. The application has been simulated several times and for different number of steps. For high number of steps, the queue size almost stabilised. As expected, the maximum number of packets in the receive queue is bigger for nodes closer to the base station, because they have to relay packets for a larger number of nodes.

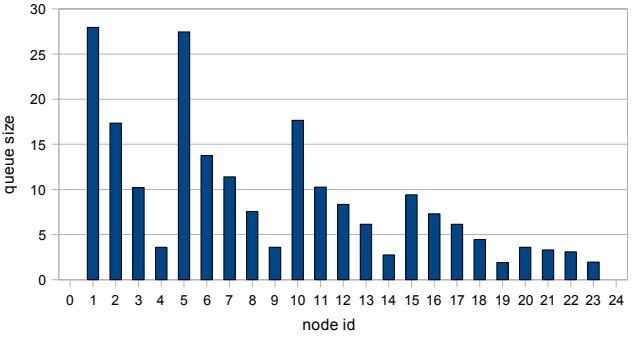
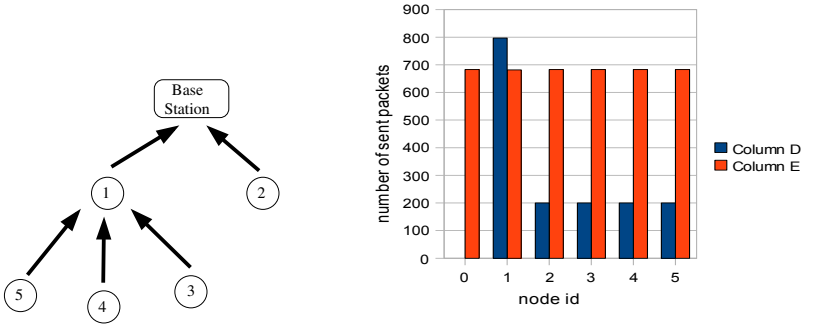



**Fig. 1.** Receive queue size for each node




(a) Network topology (b) Sent packets (D) and normalised energy (E)

**Fig. 2.** Topology and results of the simulation for energy consumption

*Energy consumption.* We report the results of a simulation to evaluate energy consumption for the previous monitoring application. To evaluate energy consumption, we used a theory providing an analytical model for idle consumption[1]. The PVS specification of both Surge and the monitoring application are left unchanged. The simulation can be executed by simply importing appropriate PVS theories. We compared our analysis with that described in [15], for a network of six nodes topology shown in Figure 2(a). In Figure 2(b), for each node, the total amount of sent packets and the energy consumption is shown. The application has been simulated for about one thousand simulation steps. Because we modelled idle energy, the evaluated energy consumption reflects real energy consumption. We obtained coherent results with respect to those of [15].

*Robustness to topology changes.* In order to test robustness of the protocol to topology changes, we consider a monitoring application in which only one node (node 5) periodically sends packets to the base station. The other nodes relay packets according to the Surge algorithm. We were able to detect a potential

[1] Idle consumption is the consumption of a sensor node during idle behaviour.

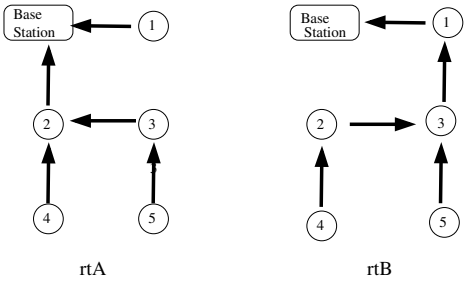



**Fig. 3.** Routing tables used for the robustness analysis of Surge

problem of infinite loops of routed packets in the algorithm specification. There are situations in which a packet may travel indefinitely in the network, because the routing table may change in response to topology changes. We evidenced such issue in a simulated grid network of 6 nodes by using routing tables $rtA$ and $rtB$ (see Figure 3). The critical situation is the following: assume that the routing table is $rtA$ and that node 3 forwards a packet to node 2. Suppose that the routing table changes to $rtB$ just before node 2 forwards the packet. Hence, the packet returns to node 3, which does not drop the packet because it is not the source. Just before node 3 forwards the packet, the routing table may change to $rtB$ again, and so on. Such a pathological case is a real problem, because routing tables may actually change during Surge operations. The specification of Surge gives no constraints on routing table changes: the only assumption is that there is an underlying service which provides a correct spanning tree rooted at the base station. The design consideration discussed in [16] allows to conclude that such a bug is indeed possible. With such an assumption, loop detection based on the source address of the packet is not sufficient to avoid the problem. Infinite loops of packets may overload the network and in safety critical applications the service provided by the network could be downgraded to an unacceptable level.

## 5 Related Work and Conclusions

In this paper we propose a simulation and analysis framework for WSN algorithms within a theorem prover. The need for formal modelling and analysis of WSN algorithms has been pointed out in many papers. In [17], basic properties of the Reverse Path Forwarding algorithm have been analysed with FDR and Alloy Analyser. Scalability is the main problem of such approach: only very simple and small network configurations were analysed, and a proof by hand was used to prove correctness of the algorithm under specific hypotheses. In [18], TinyOS is modelled as a hybrid automaton and a sensor network is specified as a network of hybrid automata. The proposed analysis is only oriented to evaluate energy consumption of sensor nodes. Moreover, in [19] model checking is applied to a TinyOS application. In [20], Lamport's Temporal Logic of Actions is used to model and simulate diffusion protocols for discovering routing trees for gathering

and disseminating data. The analysis focuses on performance variation of push and pull phases of the diffusion protocol for routing trees with different shapes, however without the objective of algorithm design evaluation. In [21] a formal model, called *Space Time Petri Nets*, has been presented to model WSNs. Time Petri Nets are augmented by adding location information to every place, and modelling broadcast transmission with a special transition. The formalism lacks both flexibility, as nodes cannot be modelled at different levels of abstraction, and scalability with respect to the generation of the reachability graph. In [22], Real-Time Maude has been applied to the OGDC density control algorithm and networks of several hundred nodes can be analysed. The specification models a node as an object, and the communication primitives are broadcast and unicast. The approach is capable of modelling the algorithm at high levels of detail, and results can be more accurate compared to other network simulators, such as *ns2*.

The framework proposed in this paper allows developers to formalise the WSN at different levels of abstraction, and it can be applied at the early stage of the development process to consolidate the algorithm design. We have used this approach to specify and simulate the Surge routing protocol for a number of networks with different topologies and number of nodes. During our analyses of Surge, we were able to detect a potential problem of routing loops due to topology changes, which has gone unnoticed so far. Further work includes the use of the theorem prover of *PVS* to verify correctness properties of WSN algorithms.

## References

1. Xu, N., Rangwala, S., Chintalapudi, K., Ganesan, D., Broad, A., Govindan, R., Estrin, D.: A wireless sensor network for structural monitoring. In: Proc. Intl. Conf. on Embedded Networked Sensor Systems, pp. 13–24. ACM, New York (2004)
2. Kim, S., Pakzad, S., Culler, D., Demmel, J., Fenves, G., Glaser, S., Turon, M.: Wireless sensor networks for structural health monitoring. In: Proc. Intl. Conf. on Embedded Networked Sensor Systems, pp. 427–428. ACM, New York (2006)
3. Aboelela, E., Edberg, W., Papakonstantinou, C., Vokkarane, V.: Wireless sensor network based model for secure railway operations. In: Intl. Workshop on eSafety and Convergence of Heterogeneous Wireless Networks, pp. 623–628 (2006)
4. Lorincz, K., Malan, D.J., Fulford-Jones, T.R.F., Nawoj, A., Clavel, A., Shnayder, V., Mainland, G., Welsh, M., Moulton, S.: Sensor networks for emergency response: Challenges and opportunities. IEEE Pervasive Computing 3(4), 16–23 (2004)
5. Stanford, V.: Using pervasive computing to deliver elder care. IEEE Pervasive Computing 1(1), 10–13 (2002)
6. Chen, G., Branch, J., Pflug, J., Zhu, L., Szymanski, B.: Sense: A sensor network simulator. Advances in Pervasive Computing and Networking, 249–267 (2004)
7. Pawlikowski, K., Jeong, H., Lee, J.: On credibility of simulation studies of telecommunication networks. IEEE Communications Magazine 40(1), 132–139 (2002)
8. Owre, S., Rushby, J., Shankar, N., von Henke, F.: Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. IEEE Trans. on Software Engineering 21(2), 107–125 (1995)
9. Owre, S., Rushby, J., Shankar, N., Stringer-Calvert, D.: PVS: an experience report. In: Applied Formal Methods. LNCS, vol. 1641, pp. 338–345. Springer, Heidelberg (1998)

10. Crow, J., Owre, S., Rushby, J., Shankar, N., Stringer-Calvert, D.: Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA (2001)
11. Muñoz, C.: Rapid prototyping in PVS. Technical Report NIA Report No. 2003-03, NASA/CR-2003-212418, National Institute of Aerospace, Hampton, VA (2003)
12. Butler, R., Sjogren, J.: A pvs graph theory library. Nasa technical memorandum 1998-206923, NASA Langley Research Center, Hampton, Virginia (1998)
13. Heinzelman, W., Kulik, J., Balakrishnan, H.: Adaptive protocols for information dissemination in wireless sensor networks. In: Proc. Intl. Conf. on Mobile Computing and Networking, pp. 174–185. ACM, New York (1999)
14. Levis, P., Lee, N., Welsh, M., Culler, D.: TOSSim: accurate and scalable simulation of entire TinyOS applications. In: Proc. Intl. Conf. on Embedded Networked Sensor Systems, pp. 126–137. ACM Press, New York (2003)
15. Shnayder, V., Hempstead, M., Chen, B., Allen, G., Welsh, M.: Simulating the power consumption of large-scale sensor network applications. In: Proc. Intl. Conf. on Embedded Networked Sensor Systems, pp. 188–200. ACM, New York (2004)
16. Woo, A., Tong, T., Culler, D.: Taming the underlying challenges of reliable multihop routing in sensor networks. In: SenSys 2003, pp. 14–27. ACM Press, New York (2003)
17. Bolton, C., Lowe, G.: Analyses of the reverse path forwarding routing algorithm. In: Proc. Intl. Conf. on Dependable Systems and Networks, pp. 485–494. IEEE Computer Society, Los Alamitos (2004)
18. Coleri, S., Ergen, M., Koo, T.J.: Lifetime analysis of a sensor network with hybrid automata modelling. In: Proc. Intl. Workshop on Wireless Sensor Networks and Applications, pp. 98–104. ACM, New York (2002)
19. Xie, F., Browne, J.C.: Verified systems by composition from verified components. SIGSOFT Softw. Eng. Notes 28(5), 277–286 (2003)
20. Nair, S., Cardell-Oliver, R.: Formal specification and analysis of performance variation in sensor network diffusion protocols. In: Proc. Symp. on Modeling, Analysis and Simulation of Wireless and Mobile Systems, pp. 170–173. ACM, New York (2004)
21. Luo, Y., Tsai, J.J.P.: A graphical simulation system for modeling and analysis of sensor networks. In: ISM 2005: Proceedings of the Seventh IEEE International Symposium on Multimedia, pp. 474–482. IEEE Computer Society, Washington (2005)
22. Ölveczky, P., Thorvaldsen, S.: Formal modeling and analysis of the ogdc wireless sensor network algorithm in real-time maude. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 122–140. Springer, Heidelberg (2007)