Master's Thesis

Modelling and Verification of a Shortest Path Tree Protocol for Wireless Sensor Networks

Towards a Platform for Formal Verification Experiments

Wouter M. Everse

July 6, 2009

Master's Thesis

Modelling and Verification of a Shortest Path Tree Protocol for Wireless Sensor Networks

Towards a Platform for Formal Verification Experiments

Wouter M. Everse
eversewm@cs.utwente.nl
University of Twente
Computer Science
s0116521

Graduation Committee:
Dr. Ir. Rom Langerak (1st supervisor)
Dr. Mariëlle Stoelinga (2nd supervisor)
Ir. Leon Evers (3rd supervisor)

July 6, 2009

Abstract

Wireless Sensor Networks (WSNs) are ad-hoc wireless networks of typically hundreds or even thousands of small low-cost sensor nodes, that communicate in a wireless way. A sensor node is a small autonomous unit, often running on batteries, with hardware to sense environmental characteristics, a processor and a radio transceiver [33]. All nodes send their sensor data to a central gateway node for future analysis.

Existing network protocols are not suitable for the WSN setting, since they often require a lot of information exchange and bookkeeping. Therefore, dedicated WSN protocols are required and their correctness and robustness is essential. However, only few techniques are available to support the design of these protocols. One possibility is to mathematically prove that the design is correct, but this usually requires many assumptions and simplifications. Another possibility is simulation and testing, but this may not uncover all undesirable aspects of a protocol. We therefore formulated the following main research question: *Is formal verification (and more specifically: Model Checking) suitable for supporting the design of WSN protocols?*.

In order to find an answer to this question, we took a concept design of a WSN protocol (developed at the University of Twente). It is a routing protocol that attempts to build a *Shortest Path Tree (SPT)* in a distributed fashion. We first made this protocol explicit by specifying both an informal and formal description. Then we constructed models of it for the state-of-the-art model checkers UPPAAL, SPIN and PRISM. The main correctness properties that we checked for these models were deadlock freedom, correct parent selection and correct distance computation. Furthermore, we performed *verification experiments* with variants of the protocol.

The main results of this research project are an informal and formal specification of the given protocol, models for the different tools, together with correctness properties, a feasibility limit of 4 nodes, the idea of formal experiments and valuable insights in modelling and verification of this protocol and of WSN protocols in general. No errors were found in the protocol.

The answer to our research question has two sides: based on our research, the limitations found and on the current state of the art, the answer is negative. On the other hand, experimentation using formal verification turned out to be a powerful tool to support WSN protocol design. Further research is required to find suitable abstraction techniques to exploit the quantitative character of the verification.

Preface

'Things should be made as simple as possible, but not simpler.'
Albert Einstein (1879-1955)

Apparently, you were curious enough to open up this thesis and start reading. Now I am wondering whether that is because of the nice front page, the interesting title, the rather intimidating number of pages, or maybe just out of politeness to me? Nevertheless, it took me blood, sweat and tears but I am proud about the result and I thank you in advance for your interest.

This thesis is the result of the research project I carried out at the University of Twente, in order to earn my Master's degree. It took me almost twenty months to finish it, which is a considerable amount of time. A period in which I learned a lot, the primary thing being the field knowledge required to perform this research. Secondary, I learned a lot about doing research, setting scope and determining research approach as well. Moreover, I acquired the necessary LaTeX and 'MacBook' skills as well. More important however is that I am now enriched with valuable experience about myself, such as insights about my personality and about my view and interpretation of 'the big bad world'. I even think that it is not too much to say that it changed my perception positively.

Without the support, sympathy and understanding of the people near me, I would not have been able to successfully complete this project, this thesis and even this study. First of all, I would like to sincerely thank my girlfriend Marjolein ('Jootje') for her unlimited support, her patience and her ability to inspire and motivate me over and over again. I would also like to thank my supervisors Rom Langerak, Mariëlle Stoelinga and Leon Evers, for their support, feedback and guidance. A special thank to Rom (1st supervisor), his no-nonsense approach, sharp ideas and our meetings and discussions were very useful to me. I would also like to thank Stefan Blom and Theo Ruys for their useful support and feedback. Furthermore, I would like to thank my parents, brothers, parents-in-law, sister-in-law, friends, family, fellow students and of course my colleagues at the 'Hogeschool Zeeland'. I probably forgot to mention some people here: my apologies and thank you too!

Contents

CHAPTER 1

Introduction

This chapter will introduce you to my Master's project, which I carried out at the Formal Methods and Tools (FMT) group of the department of Computer Science (CS) at the University of Twente. First the project definition is given, followed by the project motivation. Subsequent sections elaborate on the research questions and the research approach. Finally, the structure of the rest of this thesis is sketched. The purpose of this chapter is to give an introduction rather than to elaborate on all concepts and terminology, as this is done in the next chapter.

1.1 Project Definition

This thesis is about my research project, entitled:

> *Modelling and verification of a Shortest Path Tree (SPT) protocol for Wireless Sensor Networks (WSNs).*

The *protocol* that is referred to is a concept network protocol, developed at the University of Twente. Tanenbaum [52] defines a *protocol* in computer networks to be 'an agreement between the communicating parties on how communication is to proceed'. A protocol is thus a set of rules and conventions that describes how to communicate. The protocol that we consider is implemented in MATLAB as proof of concept by my third supervisor Leon Evers of the Pervasive Systems (PS) group. It is a routing protocol for WSNs: it finds the shortest paths between nodes of the network. Detailed informal and formal descriptions of the protocol are given in chapter 3.

WSNs are ad-hoc wireless networks of typically hundreds or even thousands of small low-cost sensor nodes, that communicate in a wireless way. A sensor node – also known as 'mote' – is a small autonomous unit, often running on batteries, with hardware to sense environmental characteristics, a processor and a radio transceiver [33]. The software used on these nodes (e.g. for communication or for processing) should be as energy efficient as possible because of the limited battery life. Moreover, as opposed to conventional ad-hoc network nodes, WSN nodes are also restricted in their amount of memory and in their computational power. These tight restrictions together with the dynamically changing nature of WSNs form the main reason why the existing network protocols are not suitable for this setting [26].

The area of WSNs is, although relatively young, a very active research area, mainly because of the enormous application potential of such networks. This is a result of the fact that WSNs offer a data collection potential at spatial and temporal unprecedented scales, which is not feasible with other instrumentation [54]. Examples of these applications are climate monitoring in forests and natural reserves, intruder detection in large areas or buildings, flood detection, control and management of transport and logistic processes, etc. More examples are found in for instance [14, 28, 36, 46].

The third important term in the project title is *modelling and verification* (of the protocol). In a nutshell, this involves the construction of a model (of the protocol) in order to verify its functionality. We will elaborate more on each of these three terms in the sequel of this chapter and in the following chapter.

1.1.1 Problem Statement

WSNs are deployed more and more and the deployment itself already brings configuration problems, such as performance problems, short circuits, unknown software bugs, wrong sensor readings etc. [47, 49]. Therefore, the availability of correct WSN protocols is essential and, as a result, correctness proofs for these protocols are important. Correctness proofs also support protocol designers in their work by establishing confidence in the protocol design.

It turns out to be rather difficult and time consuming (if not infeasible) to prove WSN protocols mathematically correct. Thus most of the time, only strongly idealized or simplified protocols are mathematically proven to be correct. An example of such an idealization in the protocol we consider is that the amount of memory of the sensor nodes is assumed to be infinite, in order to be able to prove that the protocol converges to a stable situation. In reality, the amount of memory of a sensor node is of course far from infinite: it is relatively small. Another example of a simplification is the assumption that the topology is static. But what if the topology is dynamically changing? In general, one can state that 'provable' protocol versions are often not ready to be directly implemented in practice, due the assumptions and simplifications that were applied in order to deliver a mathematical proof.

Another way of proving correctness of a model is using a computer aided technique: formal verification. The behaviour of a protocol is modelled and every possible state of this model is automatically checked for errors. Among others, one problem of this approach is the modelling of the unreliable links in WSNs: messages might get lost which results in much more states of the system. A related problem is the infamous state space explosion problem which is inherent to model checking: the number of states to be checked grows exponentially in various components of the model, such as the number of variables or the number of parallel components in a concurrent system [3]. The problem is that a system consisting of too many states cannot be automatically checked due to time and memory limitations. An important characteristic of this approach is that the verification is only as good as the model [9]: the successful verification of a system property on a wrong or faulty model of a system is of course rather useless.

Both approaches suffer from various problems and this clearly obstructs us in our quest for proving correctness of WSN protocols. However, the model checking approach is more flexible and lends itself for *formal experiments*. Therefore, a combination of these approaches might lead us to a solution.

1.1.2 Motivation and Objective

The subtitle of this thesis is “Towards a Platform for Formal Verification Experiments”. This subtitle defines an eventual general objective of this research project: a platform or methodology to be able to perform verification experiments with WSN protocols. Such a platform would enable researchers to model a strongly idealized (provable) protocol and perform formal verification experiments with variants of it (more realistic versions) rather easily and quickly. This may eliminate the need for mathematically proving such a more realistic version, which is in general much harder. Moreover, such a platform could serve as part of a toolkit for supporting the design of WSN protocols and as such, simplify the work of protocol designers.

The objective of this project is to take the first essential steps that are required in the process of developing a platform or methodology for formal verification experiments with WSN protocols, to support the design of these protocols. We want to explore the boundaries, the possibilities and the problems encountered while modelling and verifying WSN protocols. These experiences then can be used in the development of such a platform or methodology.

1.2 Research Questions

The scope of this research project as described above is rather wide. We narrowed it by stating a main research question and dividing it into sub questions. The main question that arises from the problem statement and project motivation is:

> *Is formal verification (specifically: Model Checking) suitable for supporting the design of protocols for WSNs?*

In order to answer this question, we formulated the following sub questions:

1. What are the boundaries, problems and experiences of modelling the protocol?
2. What are the boundaries, problems and experiences of verifying the protocol?
3. What aspects are best modelled/verified using which verification tool?
4. How does the topology of a network influence the results?
5. How does the number of nodes influence the results?
6. Is the protocol under consideration correct?
7. What recommendations about a platform for formal verification experiments for WSN protocols can be given w.r.t. the gained experience?

These questions clearly specify the objectives and the research direction. We reconsider them in the concluding chapter (ch. 9) of this thesis.

1.3 Approach

The approach followed to reach the described goal and to find answers to the research questions is the following. We started with a literature study to become familiar with the concepts and terminology used in the field. Then the given protocol design was made explicit by specifying both an informal and a formal description of an idealized version of the protocol. This idealized protocol has been mathematically proven correct by my supervisors and it served as case study throughout our research project. We constructed models of this protocol and verified these models, whilst carefully documenting every useful detail of this process.

In order to further narrow the scope of our research, we selected three state-of-the-art model checking tools to model the (idealized) protocol for and to perform simulations and verifications with. The tools we selected are the Uppsala-Aalborg Model Checker (UPPAAL), the Simple Promela Interpreter (SPIN) and the Probabilistic Symbolic Model Checker (PRISM). The main motivation for using specifically these tools was their coverage of three different important paradigms within the area of model checking: respectively real-time model checking, distributed model checking and probabilistic model checking. Moreover, their popularity in this area also played an important role. After modelling and verification of the protocol using these tools, we performed some interesting experiments with more realistic variants of the protocol.

1.4 Main Results and Contributions

In this section we concisely summarize the main results and the main contributions of this research project. These are further explained in the concluding chapter of this thesis (ch. 9).

Main results:

- A clear informal and formal description of the protocol;
- Problematic aspects of modelling the protocol are link quality (probabilism), message broadcast and distance computation;
- The feasibility of a verification run depends heavily on the topology under consideration (especially the number of directed links and their quality);
- The process of verification of a model of the protocol has many input parameters, resulting in an instance explosion of possible verification runs.
- We created two UPPAAL models (V2 & V3), a SPIN model (with some variants) and a PRISM model, with associated correctness properties;
- Both UPPAAL (model V2) and SPIN are maximally capable to verify a completely connected 4-node topology with 10% links. The SPIN model has better probability approximation; Useful verification of the PRISM model turned out to be infeasible;
- We did not find any errors in the protocol.

Main contributions:

- Current state of the art formal verification techniques can handle only very restricted WSN configurations of maximally four or five nodes.

This induces the need for suitable abstraction techniques. These abstraction techniques should account for the *quantity* of the probabilistic link behaviour. The qualitative character of verification of WSN protocols changes and becomes more quantitative. More research is required to find suitable abstraction techniques that account for this quantitative aspect.

- We introduced the idea of a methodology or platform that supports the design of WSN protocols. This would be of great help for protocol designers. We propose a platform containing three stages:

  1. Mathematical proof of a (strongly) simplified protocol design;
  2. Matlab simulation to validate (possibly less simplified) protocol behaviour, in particular large scenarios;
  3. Formal verification experiments using model checking, to experiment with (more realistic, less simplified) variants of the protocol.

1.5 Thesis Structure

The next chapter contains the necessary background information of the research project: it thoroughly elaborates on wireless sensor networks and their applications, network protocols, graph theory, formal methods (and specifically model checking) and related work. In the third chapter we introduce the protocol by describing it both informally and formally. Then, the subsequent four chapters elaborate on modelling and verification of the protocol: one chapter about UPPAAL, two about SPIN and one about PRISM. Chapter 8 describes the formal experiments that we performed with our initial SPIN model and with variants of it. These variants implement more realistic aspects in the original protocol. Finally, the last chapter summarizes the main results, conclusions (answers to the research questions) and contributions. It also lists future work.

CHAPTER 2

Background

This chapter covers the necessary background for a thorough understanding of this thesis. It sketches the context of the research project and it introduces the terminology used. We first elaborate on Wireless Sensor Networks and their applications. Then we continue with a paragraph on network protocols. Next, the portion of graph theory required for the proper understanding of subsequent chapters is presented. In the subsequent section we consider formal methods and techniques for software verification, such as model checking. We finish this chapter with a discussion on related work.

2.1 Wireless Sensor Networks

As stated in the introduction, WSNs are networks of typically a large number of small low-cost sensor nodes, that communicate in a wireless way. Figure 2.1 shows us a representation of a WSN.

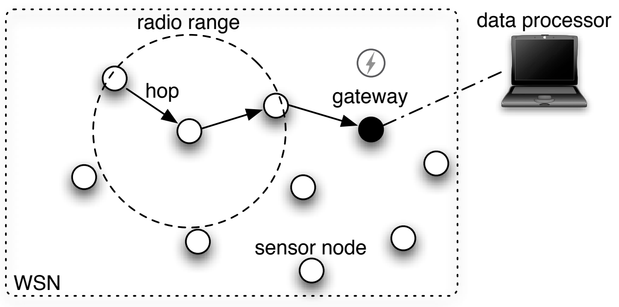
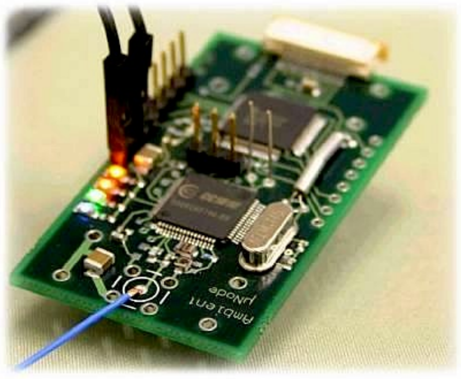


Figure 2.1: Wireless Sensor Network

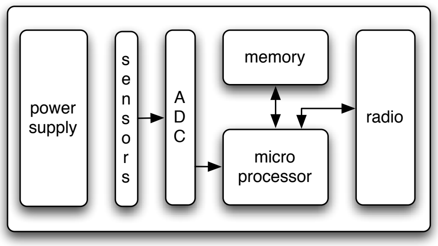
The elements that constitute any network are usually referred to as network nodes or simply *nodes*. The nodes that comprise a WSN are equipped with sensors to sense for instance temperature, vibrations, light, humidity, pressure etc. In figure 2.1 these sensor nodes are represented by the small empty circles. The bigger dashed circle denotes the wireless range of the radio transceiver of the node in its center. Below we elaborate on sensor nodes and on a special type of sensor node: *the gateway node* (the black circle in fig. 2.1).

2.1.1 Sensor Nodes

According to Leskovec et al. [33], a sensor node – also known as 'mote' – is a small autonomous device, often running on batteries, with hardware to sense environmental characteristics, a processor and a radio transceiver (to transfer measurement data to a central base station). As an example, figure 2.2(a) shows the Ambient μNode, which is developed at the University of Twente. Figure 2.2(b) depicts a typical high-level sensor node architecture.



(a) The Ambient μNode



(b) High-level Sensor Node Architecture

Figure 2.2: Sensor Node

Figure 2.2(b) shows us that a sensor node is intelligent since it contains a microprocessor. This intelligence is used to improve the quality of sensor readings (for example by discarding faulty readings). Additional motivation for adding intelligence is that the node should be able to operate autonomously: it should for instance be able to make decisions about routing when a neighbour dies. Moreover, adding intelligence allows for more efficient operation, implementation of new functionality and even accountability for business logic [7].

Besides the microprocessor, a sensor node consists of a limited amount of memory, a collection of sensors connected to one or more Analog-to-Digital

Converters (ADCs) and of course a wireless radio transceiver for communication. It is powered by a limited-lifetime power source, typically batteries. More concrete (i.e. lower level) architectures of sensor nodes are often layered modular systems [12, 20, 35, 39].

The size of sensor nodes largely varies and depends on the application area. The average size is around that of a box of matches. As with other technical devices, sensor nodes become smaller and smaller whilst technology advances. For example, Chee et al. [12] describe the architecture and implementation of the PicoCube, a 1cm^3 wireless sensing device powered by harvested energy. In the area of smart surroundings, there are even devices smaller than a few cubic millimeters that constitute networks known as Smart Dust [48].

The Gateway Node

As can be seen in figure 2.1, WSNs contain a dedicated node called the *gateway* (represented by the small black circle). It is typically connected to an external power supply rather than running on, for instance, battery power like the other nodes. The gateway is the central base station mentioned above, it functions as a sink: all nodes send their sensor data to it. Once at the gateway, these data are either analyzed locally or forwarded (for example via the internet) for future processing (the data processor in fig. 2.1). There may be several gateway nodes in a WSN.

The nodes in a sensor network need to send their sensor data wirelessly to a central gateway node. Therefore they need to be able to communicate via intermediate nodes, since the majority of the nodes will not be in the wireless range of the gateway. A direct link between two nodes is called a *hop* (fig. 2.1) and this type of communication is therefore called *multihop communication*.

2.1.2 Mobile Ad-hoc Networks

Mobile Ad-hoc Networks (MANETs) are multihop wireless networks of mobile nodes [40]. The communication between nodes takes place in an *ad-hoc* fashion, meaning that there is no fixed infrastructure for communication (in contrast to wired networks). Nodes rather spontaneously establish wireless communication channels between each other, based on information of neighbours. This self-configuration takes place dynamically and in a decentralized, distributed manner. In general, the nodes of a WSN are mobile and WSNs fall into the class of MANETs. A strongly related class of networks is that of mesh networks, the difference being that in mesh networks nodes are not mobile.

2.1.3 Applications

As already stated in the introduction, WSNs offer a data collection potential at spatial and temporal unprecedented scales, which is not feasible with other instrumentation [54]. In general, there are two typical application classes for WSNs: *data collection* and *event-detection*. The former class is the more conventional one of monitoring, which historically stems from military application (battlefield monitoring). Examples are systems that monitor all kinds of (environmental) properties. There are systems for (e.g. climate) monitoring in vineyards, forests, natural reserves and for monitoring pollution in the sea. A concrete example of these so-called 'habitat monitoring systems' is a WSN consisting of 32 nodes on Great Duck Island, off the coast of Maine, used to monitor seabirds nesting behaviour without human disturbance. [36].

An example of a WSN in its event-detection role is one that is used as an intruder detection system, either in a particular area or in a building. Forest fire detection or flood detection are also examples in this class. Data-collection and event-detection are often also combined, for example in WSNs that monitor seismic properties [54].

In general, WSNs can cover a large area and sense all kinds of properties at relatively high resolutions. Therefore, the potential of WSN application is enormous and still growing as it is more and more recognized. WSN application areas include environmental monitoring, medical monitoring, for instance as with Harvard's MoteTrack system [34], military applications such as battlefield monitoring, transportation (control and management of transport and logistics processes [14]), entertainment (intelligent light control, [43]), security and safety (e.g. person tracking, monitoring condition of buildings) etc. Clearly, there are numerous application scenarios for WSNs.

2.2 Network Protocols

To reduce network design complexity, networks are often organized in several stacked layers, following the (in computer science) well-known concepts of information hiding and abstraction [52]. A layer provides its services to the adjacent higher layer, while hiding the implementation details of these services, and it uses services of the adjacent lower layer. Layers at the top of the network stack provide more abstract services, layers at the bottom of the stack offer more concrete services. Layer n on one node communicates with layer n on another node according to a *layer n protocol* (figure 2.3). Tanenbaum [52] defines a protocol in computer networks to be 'an agreement between the communicating parties on how communication is to proceed'. A protocol is thus a set of rules and conventions that describe how to communicate. If the protocol is violated, the communication becomes much more complex, if not impossible. A list of protocols used by a certain system, one protocol per layer, is called a *protocol stack*.

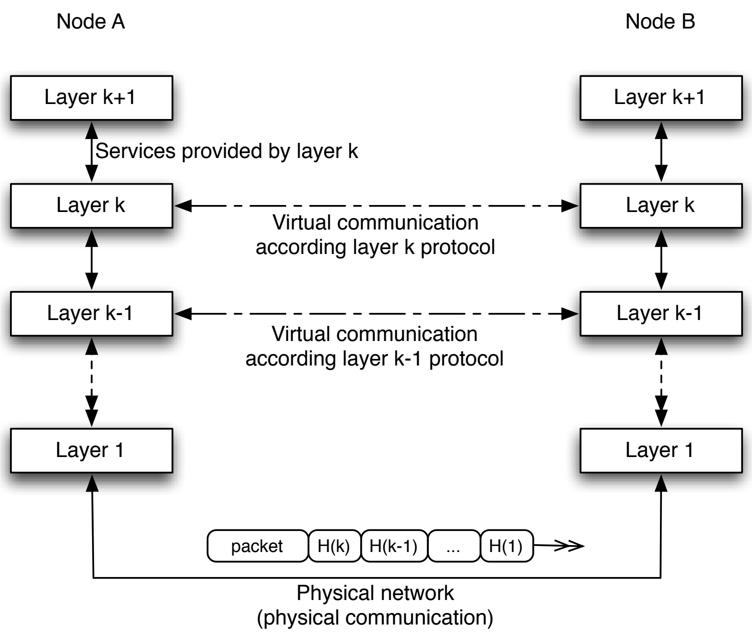

Node A
Node B
Layer k+1
Layer k+1
Services provided by layer k
Layer k
Layer k
Virtual communication according layer k protocol
Layer k-1
Layer k-1
Virtual communication according layer k-1 protocol
Layer 1
Layer 1
packet
H(k)
H(k-1)
...
H(1)
Physical network (physical communication)

Figure 2.3: Layered network principle (based on Tanenbaum [52])

Figure 2.3 is a representation of the abstract principle of layered networks. It makes clear that the direct communication between two layers k of different nodes, according to a layer k protocol, is only virtual. The actual physical communication happens of course over the physical network. For instance, when layer k of node A communicates with (i.e. sends a data packet to) layer k of node B, the packet travels from layer k downwards the stack of node A, via the physical connection (which might be any kind of connection, wired or wireless) and, once arrived at node B, upwards the stack to layer k of node B. Every layer of node A adds a header to the packet and every layer of node B removes and interprets the corresponding header. For the details we refer to the literature [52].

Well-known models that illustrate the layered network principle explained in the previous paragraph are the Open Systems Interconnection (OSI) Reference Model and its simplified version, the TCP/IP Reference Model[1]. For the details we refer again to Tanenbaum [52]. Both models have a Network Layer containing routing protocols, and it is this layer to which our SPT protocol belongs.

[1]Transmission Control Protocol (TCP), Internet Protocol (IP)

2.3 Graph Theory

This section presents the portion of graph theory that is required for a good understanding of the subsequent chapters.

2.3.1 Graphs and Trees

A *graph* is a mathematical structure that consists of a set of *vertices* and a set of *edges*. A real world analogy is for example a road map with a set of towns (the vertex set) and a set of roads (the edge set) that connects them. Another example is a network with nodes (vertices) and links between the nodes (edges). Of course roads and links can either be one-way or two-way. This translates to respectively a directed graph (containing directed edges) and an undirected graph, in which there is no notion of direction. In both cases the edge set is a set of pairs of vertices: ordered pairs in a directed graph and unordered pairs for the undirected case. It follows that the edge set is a binary relation on vertices. A convenient way of representing a graph is by using a picture. Figure 2.4 shows an example of a simple graph consisting of six vertices and eight edges.

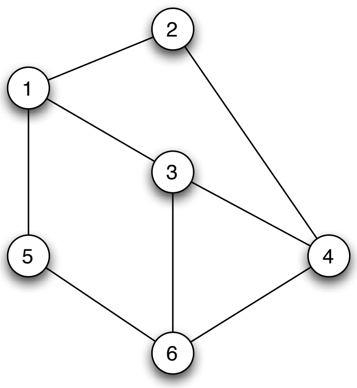


Figure 2.4: Example of a simple graph

The following definition is based on the definition of a graph of Grimaldi [18].

Definition 1 (Graph) *A* graph *G is a pair (V, E) consisting of a finite, non-empty set of* vertices *V and a set of* edges *$E \subseteq V \times V$. We write $G = (V, E)$ to denote such a graph. We call G a* directed *graph if E is a set of ordered pairs of V. If E is a set of unordered pairs, G is called an* undirected *graph.*

The example graph in figure 2.4 is an undirected graph because there is no notion of the direction of the edges. Therefore edge (2,4) cannot be distinguished from edge (4,2): they are considered equal. The vertices and edges of the graph in figure 2.4 are enumerated below:

- vertex set $V = \{1, 2, 3, 4, 5, 6\}$, and
- edge set $E = \{(1, 2), (1, 3), (1, 5), (2, 4), (3, 4), (3, 6), (4, 6), (5, 6)\}$.

A *path* in a graph is a sequence of adjacent vertices (i.e. vertices connected by edges), in which no vertex occurs more than once. A path in our example graph is for instance the sequence {2,1,3,6,4}. A *connected graph* is one in which there is a path between every pair of distinct vertices. A *complete graph* is one in which there is an edge between every pair of distinct vertices: it has $\frac{1}{2}n(n-1)$ undirected edges for n vertices. Another term we will use is a *cycle*: a path that starts and ends at the same vertex, for example {2,1,3,6,4,2} but also {3,6,4,3}.

A *weighted graph* is a graph of which every edge is associated with a certain weight, a positive number (fig. 2.5(a)). The weight of an edge for example may represent the distance between the two vertices, or the cost (e.g. fuel, hours, time, etc.) to get from one vertex to another. A path in a weighted graph has a certain *cost*, that is the sum of the weights associated to the edges that connect the vertices in the path. Now we define two more concepts from graph theory: a tree and a spanning tree.

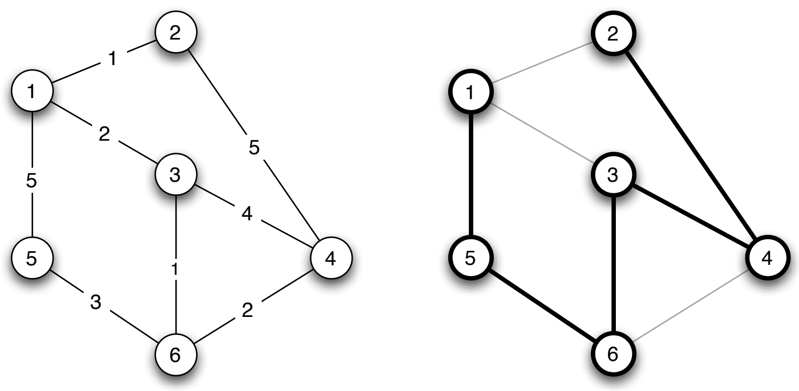
Definition 2 (Tree) *An undirected graph $G = (V, E)$ is called a* tree *if it is connected and contains no cycles.*

Definition 3 (Spanning Tree) *A tree $T = (V_T, E_T)$ is called a* spanning tree *of a graph $G = (V_G, E_G)$ if $V_T = V_G$ and $E_T \subseteq E_G$.*

An example of a spanning tree can be seen in figure 2.5(b). The thick lines together with the vertices form a spanning tree of the example graph we saw earlier. Of course other spanning trees are also possible in this graph. The cost of a weighted tree is the sum of its edges, so the tree of figure 2.5(b) would have a cost of $5 + 3 + 1 + 4 + 5 = 18$ in figure 2.5(a).

2.3.2 Dijkstra's Algorithm

The Shortest Path Tree protocol that will be described in chapter 3 enables all nodes of a network to find a shortest path to a certain root node (the gateway). In this section, a well-known algorithm for finding shortest paths (i.e. paths of minimal cost) in a graph is intuitively explained in order to provide the necessary background for our protocol. The algorithm for finding these shortest paths between a certain vertex and all of the other vertices of a connected weighted graph, is the Shortest Path Algorithm of



(a) a weighted graph

(b) a spanning tree of the example graph in figure 2.4

Figure 2.5: Weighted graph and Spanning Tree

the Dutch computer scientist Edsger Wybe Dijkstra (1930-2002) [2, 18]. As said, here the algorithm is explained intuitively and concise. For a detailed formal description, the reader is referred to the literature on which we based the rest of this section [2, 18].

Defining Notation

Given a weighted graph $G = (V, E, w)$ with $w = E \times \mathbb{N}$ a function that associates nonnegative integers to edges, for each edge $e = (x, y) \in E$, we interpret the associated weight $w(e)$ (sometimes for convenience also denoted as $w(x, y)$) as the distance between vertices x and y. If $(x, y) \notin E$, we define $w(x, y) = \infty$. The weight of a path $\pi = v_1 v_2 ... v_n$ of n vertices equals the sum of the weights of the edges that comprise the path, that is $w(\pi) = w(v_1, v_2) + w(v_2, v_3) + ... + w(v_{n-1}, v_n)$. If no path between v_1 and v_2 has weight less than $w(\pi)$, then π is called a *shortest path.* Following Grimaldi [18], we write $d(a, b)$ with $a, b \in V$ for the distance (weight) of a shortest path in G from a to b. If no such path exists, $d(a, b) = \infty$. For all $a \in V, d(a, a) = 0$.

The Problem

Given a connected weighted graph $G = (V, E, w)$ as described above, and a source vertex $s \in V$, the problem is to find a shortest path from s to every other vertex $v \in V$.

Dijkstra's Solution

Dijkstra's solution to this problem is a *greedy* algorithm, which means that it finds the best results (shortest paths) *globally* (for all vertices of the graph) by obtaining the best result *locally*. It starts with the given source vertex s and 'branches out' by selecting certain edges that lead to new vertices. The selection is done locally based on the weight of the edges: it always chooses an edge to a vertex that appears to be 'closest' to s [2]. This results in a tree, more specifically, in a *Shortest Path Tree (SPT)*.

Definition 4 (Shortest Path Tree (SPT)) *A Shortest Path Tree (SPT) of a connected weighted graph $G = (V, E, w)$ is a spanning tree of G, consisting of a vertex s, called the root node or root vertex, and a shortest path from s to every other vertex of G.*

The vertices of the graph are thought of as they were divided in three disjoint sets:

1. a set T of *tree* vertices that are in the SPT constructed so far,
2. a set F of *fringe* vertices that are adjacent to any vertex in T,
3. a set U of *unseen* vertices, containing all others.

Initially, all vertices are classified as *unseen*. Then the source vertex s is (re)classified as *tree* and all vertices adjacent to s as *fringe*. Now the algorithm proceeds as follows from figure 2.6.

```
  Data: $G = (V, E, w)$ and $s \in V$
  Result: $T$ is a SPT of $G$
1 $T := \{s\}, F := \{x | x \in V, (s, x) \in E\}, U := V - T - F$ ;      // init
2 while $|F| > 0$ do
3     $d(s, v) := \min_{t \in T, v \in F} \{d(s, t) + w(t, v)\}$;
4     $F := F - v$; $T := T + v$ ;                            // $v$ becomes tree
5     forall $(v, u) \in E$ with $u \in U$ do
6         $U := U - u$ and $F := F + u$ ;                  // $u$ becomes fringe
7     end
8 end
```

Figure 2.6: Dijkstra's Shortest Path Algorithm

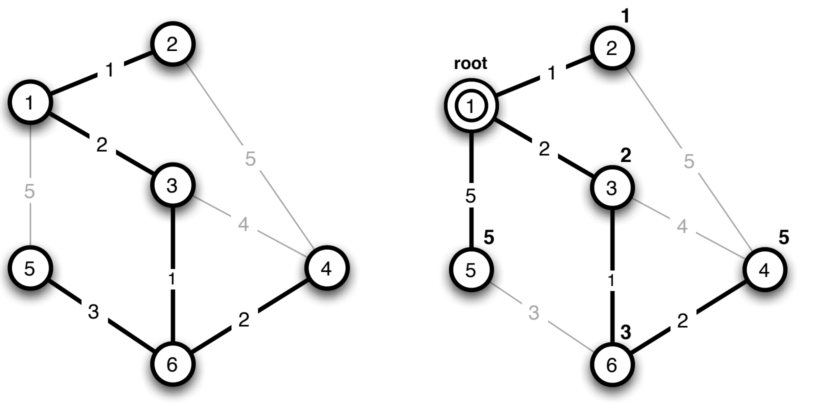
That is, while there are fringe vertices, select an edge between a tree vertex t and a fringe vertex v, such that $d(s, t) + w(t, v)$ is a minimum (line 3), and reclassify v as *tree* (add vertex v to the tree, line 4). Define the distance of a shortest path from s to v: $d(s, v) = d(s, t) + w(t, v)$ (line 3). Now reclassify all *unseen* vertices adjacent to v as *fringe* (lines 5-7) and iterate.

Note that there may be more than one shortest path from vertex a to vertex b in a graph. Therefore, a SPT need not be unique. The SPT should not be confused with the widely known Minimum Spanning Tree (MST):

Definition 5 (Minimum Spanning Tree (MST)) *Given a connected weighted graph $G = (V, E, w)$, a Minimum Spanning Tree (MST) is a spanning tree of G of minimal weight.*

The MST can be found by following for example Prim's MST algorithm, which actually has quite some aspects in common with Dijkstra's shortest path algorithm [2]. Further discussion of these algorithms is out of the scope of this project, more information can be found in the literature [2, 18].

We conclude this section with an example of both trees: figure 2.7 shows in thick lines both the MST and a SPT with root node 1, in the case of our example graph from figure 2.4. The MST has total weight 9, and there is no spanning tree that weighs less. The numbers to the upper right of each vertex in 2.7(b) denote the distance of the shortest path to the root (the vertex labelled 1). The reader is invited to apply Dijkstra's Algorithm to the example graph to determine the SPT with the source vertex labelled 6. It turns out to be equal to the MST. In other words, the MST of the example graph contains the shortest paths from source vertex 6 to each of the other vertices.



(a) The MST of the example graph   **(b)** SPT with root 1 in the example graph

Figure 2.7: Special spanning trees of the example graph

2.4 Formal Methods

The field of *Formal Methods* in the context of Computer Science (CS) covers all approaches for specification and verification of software systems, based on mathematical formalisms [50]. According to Ruys [50], "The aim is to establish system correctness with mathematical rigour. Using formal methods, system designs can be defined in terms of precise and unambiguous specifications that provide the basis for a systematic analysis.". This section will provide some background on formal methods, especially why they are needed and which common methods there are.

2.4.1 The Need for Formal Methods

Our lives are impregnated with all kinds of Information and Communication Technology (ICT) systems. For example, we rely for a large amount on the functioning of smart cards, handhelds, mobile phones, television systems, Digital Versatile Disc (DVD) recorders, the Internet and so on. Clearly, computers are ubiquitous nowadays. Moreover, we also rely heavily on more and more safety-critical systems, like for instance the control software in our cars, traffic control and alert systems and medical systems at home and in hospitals. Even the proper operation of chemical and nuclear plants relies vastly on software [3].

Without relying on such ICT systems, it is practically very hard to participate well in current society. Therefore, we are annoyed if something (for example our mobile phone) does not function properly. This malfunctioning is caused by software and/or hardware errors, which often have substantial (negative) financial consequences for manufacturers. For example, a bug in Intel's Pentium II processor (in its floating point division unit) was good for a loss of about $475 million plus a severely damaged reputation [3]. Correct ICT systems are essential for the survival of a company. Besides annoyance and financial impact, safety-critical systems that contain errors obviously may have a far more severe impact, involving one or more human lives. There is the notorious example of the Therac-25 radiation therapy machine, that caused the death of six cancer patients between 1985 and 1987. Here software errors caused the exposure of the patients to an overdose of radiation. Another example of a well-known fatal failure caused by a software error is the crash of an Ariane 5 rocket in 1996 (37 seconds after launch) [3].

In general, hardware and software systems are widely used in applications where failure is unacceptable [50]. On top of this, ICT systems still continue to grow in size and complexity. Therefore the probability that errors are getting introduced also increases. It will be clear that the reliability of ICT systems has become very important: it became a key issue in the system's design process.

2.4.2 Validation and Verification Techniques

Formal methods are concerned with the unambiguous specification and automated validation and/or verification of software systems based on mathematical formalisms. The terms *validation* and *verification* are often confused. A well-known trick to remember the difference originates from Boehm [6]: validation corresponds to the query "*are we building the right thing?*" whereas verification answers the question "*are we building the thing right?*". The former is thus related to a real-world user's perspective (compliance to user requirements), while the latter is concerned with compliance to the software development process.

Important validation and verification techniques based on formal methods are simulation, testing and formal verification (in particular model checking and theorem proving) [25, 50]. In the sequel of this section we shortly discuss each of these techniques.

Simulation

Simulation is a validation technique that is concerned with some executable model of the system under consideration. A software tool called a *simulator* executes the model following some scenarios (sets of possible system inputs) to determine the behaviour. This provides insight to the reactions of the system on certain inputs. The scenarios may be provided by the user or may be randomly generated. Simulation is typically useful for a quick assessment of a design, but not to show the presence of subtle errors as it is infeasible to simulate all possible scenarios [25].

Testing

Testing is the traditional way of validating the correctness of a design. In practice, it is probably the most frequently used validation technique. Testing is the process of stimulating the so called Implementation Under Test (IUT) with well-chosen input while observing its output. The observed output is then checked to conform to the required output that follows from the system specification. The input is in practice often obtained in a rather ad-hoc and heuristic manner and requires experience. Structured Testing, however, is concerned with more structural techniques for black box testing (e.g. equivalence partitioning, boundary value analysis) and white box testing (e.g. statement coverage, branch coverage).

As opposed to simulation that is based on a model of the system, tests are performed on a real implementation. Testing is similar to simulation in that it is incomplete: it is impossible to observe all possible outputs [25]. Testing provides insight in the quality of the implementation and helps assessing the risk of putting the implementation into operation.

Formal Verification

Formal verification techniques *prove* that a (model of a) system operates correctly, in contrast to testing and simulation. These techniques are based on the construction of a formal model (i.e. a mathematical model) of the system which represents the possible behaviour. The correctness requirements are stated as properties in a formal property specification language. Then it is checked whether the specification of the model (the system's possible behaviour) "contains" the desired behaviour (specified as correctness property). This can be unambiguously and explicitly checked since we are dealing with formal specifications [3]. It is important to note that *formal verification techniques are only as good as the model.*

As opposed to testing and simulation, formal verification techniques are capable to exhaustively check the behaviour of the system (model) under consideration. Two fundamental formal verification techniques are Model Checking and (Automatic) Theorem Proving. They are shortly discussed below.

Model Checking

In Baier and Katoen [3], model checking is defined to be an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for that model. Slightly more formally: given a finite-state model M and a property ϕ stated in some formal notation (e.g. temporal logic), model checking is the process of systematically checking the validity of the property, i.e. $M \models \phi$ [50]. This is called the verification question. Model checking is a process that is computer aided: given M and ϕ, a computer tool called a *model checker* performs the check. If the property does not hold for the given model (i.e. $M \not\models \phi$), a *counterexample* is provided that indicates how the error state of the model was reached. An example of a correctness property to check is that no deadlocks occur. In section 2.4.3 below, we will take a closer look at Model Checking.

Theorem Proving

Another prominent formal verification technique is Theorem Proving. Again we have the verification question $M \models \phi$ with M a system model and ϕ a correctness property. The system model is now expressed as a *formal proof system* existing of *axioms* and *inference rules*, expressed in some mathematical logic. Property ϕ is also expressed in that same logic. Theorem Proving is the process of showing that ϕ is a logical consequence of M, i.e. finding a derivation of ϕ by applying rules from M, starting with axiom(s) from M. A *theorem prover* is a computer tool that assists in this process of constructing a proof. Compared to a model checker, a theorem prover is less automatic since user interaction is typically required during proof construction. On the other hand, theorem provers are not restricted to finite state spaces.

Summarizing, given a model M and correctness property ϕ, simulation and testing check whether $M \models \phi$ for some executions of M. Model checking checks $M \models \phi$ exhaustively and systematically by checking that each reachable state in M satisfies ϕ. Theorem proving checks whether $M \models \phi$ by the construction of a formal proof of M, satisfying ϕ.

2.4.3 More on Model Checking

In this subsection we elaborate slightly on model checking, since this verification technique is used extensively during this research project. For technical details of model checking, search algorithms, property specification languages etc. we refer to the literature (for example the excellent book *Principles of Model Checking* by Baier and Katoen [3]).

Model Checking Phases

The model checking process can be divided into the following phases [3]:

- *Modelling phase* – in which the system under consideration is modelled using the model description language of the model checker of choice. During model construction, it is often possible to perform some simulations, as a first sanity check. The properties to be checked are also formalized in this phase, using the property specification language supported by the model checker.
- *Running phase* – in which the model checker systematically and exhaustively checks the validity of the given property for the constructed model.
- *Analysis phase* – in which the *result* of running the model checker is analyzed:
  - Property violated: analyze the generated counterexample by simulation and refine the model, the design or the property by iterating (restart the modelling phase).
  - Out of memory: try to reduce the model by iterating (restart the modelling phase).
  - Property satisfied: check the next property (if any) by iterating (restart the running phase with the next property).

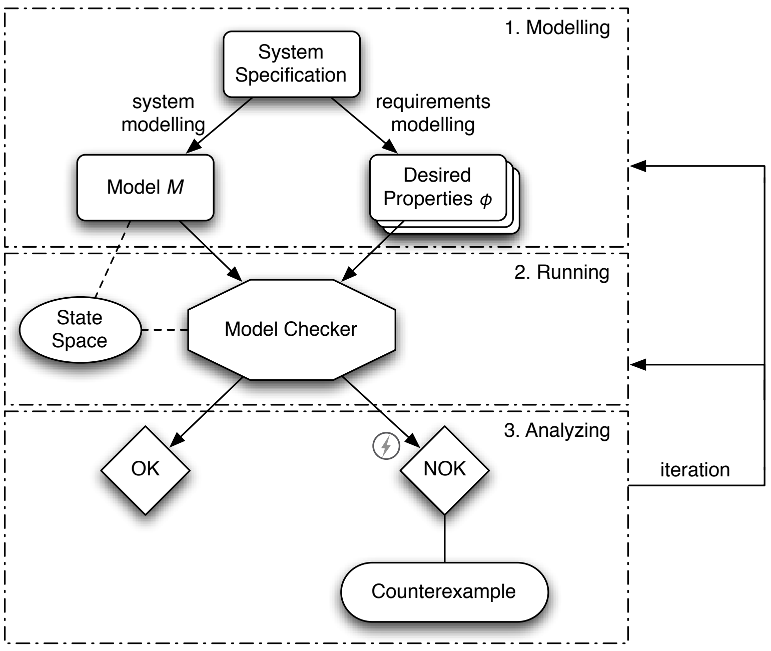


Figure 2.8: Schematic view of the Model Checking principle

These phases are also denoted in figure 2.8. The figure is a schematic view of the model checking principle. In fact, this is the classical view on model checking; a fast growing other view is that of *software model checking* in which the model to check is (either automated or not) abstracted from a system implementation (i.e. from program code). In this thesis, we focus on the classical view.

State Space Explosion

In the running phase in figure 2.8, there is an ellipse labelled "state space". The state space is the collection of reachable states of the model, generated by the model checker. Therefore it is connected (using dashed lines) to both the model and the model checker. The *state space explosion* is a notorious problem of the model checking approach: the number of states of a model is exponential in both the number of variables and the number of parallel components of the model. This causes an extremely large number of states to be generated for a model of a realistic system [3]. The state space of a practical model is often too big to fit in computer memory. This prevents the model checker from exploring every state to check the given property and the verification is aborted.

There are several optimization techniques to combat the state space explosion, such as model abstraction, partial order reduction and other reductions through equivalence, memory management techniques, symbolic techniques, etc. It is out of the scope of this thesis to discuss them here. Instead, we will refer to the literature, for example Baier and Katoen [3], Katoen [25], Ruys [50].

Benefits and Limitations

We conclude this section with some benefits and some limitations of model checking [3, 25].

Benefits:

- Model checking is based on a sound mathematical foundation.
- Unlike testing and simulation, it is not vulnerable to the likelihood that an error is exposed. This means that even rare errors will be found by model checking while they often remain undiscovered during testing and simulation.
- In case of property violation, it provides a counterexample (diagnostic information) which is very useful for debugging.
- Partial verification is possible: verification against a partial specification is possible since properties can be checked individually.

Model checking of course also comes with some limitations:

- Model checking is only as good as the model: the successful verification of a system property on a wrong or faulty model of a system is rather useless.
- It suffers from the notorious state space explosion problem described above.
- It may require some *expertise* in finding the right abstractions in order to reduce system models and to state correctness properties.
- In general, systems with an arbitrary number of components or parameterized systems cannot be checked. If, for instance, a protocol is verified to be correct for 2 and 3 processes it cannot provide an answer for the verification of the protocol for n processes (arbitrary n).

It can be concluded that model checking can provide a significant increase in the level of confidence of a system design, since it is an effective technique to expose potential design errors [3].

2.5 Related Work

This section provides an overview of relevant work in the field. Most of the many work on WSNs has been performed in the last decade. Therefore the demand for correct protocols and theoretical foundation also rapidly increased. Moreover, WSNs come with many network challenges (e.g. unreliability, mobility, efficiency) which motivated many research. This resulted in a huge amount of mostly separately developed WSN protocols, which have limited interoperability because of different assumptions about network architecture. In this light, a platform or methodology for formal verification experiments with WSN protocols would be very useful. However, in the literature there are only few reports on this topic. Kim, Kim, Lee, Ahn, Song, and Won [27] describe an automatic verification framework for the development of WSN protocols, but this is based on testing and test case generation, rather than on verification using model checking. There is no framework or platform that we know of for formal verification experiments with WSN protocols.

2.5.1 Formal Protocol Analysis

Many protocols (or portions thereof) have been formally verified: for example Kusy and Abdelwahed [30] describe the modelling and verification of a portion of the Flooding Time Synchronization Protocol (FTSP) using the SPIN model checker. They assumed ideal channels (no message loss) and an ideal network (no node failures) and were able to check the protocol up to and including 4 nodes. This illustrates a common problem with model checking protocols: due to the state space explosion, it is often only possible to verify just a few nodes, in this case only 4.

The work of Wibling, Parrow, and Pears [55] evaluates the model checkers SPIN and UPPAAL using the verification of a MANET routing protocol as a case study. This is closely related to our work since important modelling and verification issues are considered, such as modelling broadcast, connectivity and topology changes. Verification using either SPIN or UPPAAL was feasible for scenarios up to and including 4 nodes. In addition, UPPAAL was able to successfully verify one scenario of 5 nodes.

Câmara, Loureiro, and Filali [10] describe a methodology for formal verification of routing protocols for ad hoc networks. In contrast to most other work, their solution does not model any particular network topology, but rather focuses on the possible implications of a topology on the behaviour of the protocol: the so called topology abstraction. This methodology is useful for confirming the existence of functional problems but it does not help determining protocol limits. Another drawback is that error scenarios must be evaluated manually.

Interesting work with a different intent is that of McIver and Fehnker [38], in which the benefits of applying formal analysis to wireless networks are explored. It is concluded that formality makes assumptions explicit and that the exhaustive search of model checking can illustrate weaknesses in the system effectively and provide lower and upper bounds on quantitative behaviour, without the need for using large numbers of simulations. Fehnker, van Hoesel, and Mader [16] worked on the modelling and verification of a medium access control protocol (i.e. the Lightweight Medium Access Control (LMAC) protocol for WSNs) using the model checker UPPAAL. Their results are the improvement of the protocol and its description, and the discovery of some faults in the protocol. An example of formal verification using the probabilistic model checker PRISM, is the work of Fehnker and Gao on "formal verification and simulation for performance analysis for probabilistic broadcast protocols" [17]. This work describes the modelling and verification of a gossiping protocol. A model of 9 nodes was checked, more nodes exceeded the capabilities of PRISM. The results were validated in larger network settings using Monte Carlo simulation (implemented in Matlab). Formality making assumptions explicit was also experienced during their research. In work on "the graphical modelling for simulation and formal analysis of WSN protocols", Fehnker, Fruth, and McIver [15] argue that context-dependent details such as distance between nodes and node density militate against a clear and modular formal specification and therefore are difficult to add to a formal model. They argue that the simplest way of expressing the spatial relationships is graphically and translate a graphic model to a formal PRISM model with reception probabilities that account for the spatial details.

2.5.2 Routing in WSNs

Besides formal analysis of protocols for MANETs or WSNs, the work by Woo et al. [56] entitled "Taming the underlying challenges of reliable multihop routing in sensor networks" turned out to be very relevant. The paper describes the routing problem for WSNs and discusses the underlying design issues of routing protocols, such as routing metrics, link quality estimation and neighbourhood management. Different approaches to address these issues are explained and evaluated in an empirical study. This work inspired the development of our SPT protocol and provides useful insights in the internals of both the protocol and WSNs. Another contributor to these valuable insights is the work of Polastre et al. [45]. A unifying link protocol is proposed in order to improve the limited interoperability of the numerous protocols that recently have been developed for WSNs.

Another closely related work is the PhD thesis of Wu [57] which is a dissertation on "reliable routing protocols for dynamic wireless ad hoc and sensor networks". Although it does not focus on the verification of routing protocols, it provides valuable information about routing and related issues in MANETs and WSNs and about the routing protocols used in the field. It also provides an overview of protocol categories. We will elaborate on this in the next chapter. The main contributions of the thesis are a highly reliable, low traffic WSN routing protocol, a cross-layered approach to routing in high mobility sensor networks and a data-centric approach to get aggregated data from source to destination with high efficiency.

CHAPTER 3

The SPT Protocol for WSNs

This chapter is a detailed explanation of the Shortest Path Tree (SPT) protocol that served as our case study. We start with an introduction to routing and the routing problem in Wireless Sensor Networks (WSNs). Then we continue with an informal description of our protocol, after which we will present its formal specification.

3.1 Introduction to WSN Routing

An important task of the nodes in a WSN is to send the data obtained by their sensors through the network to a special node: the gateway G (of which there may be several). Once there, these data are either forwarded via for instance internet (for future processing) or directly processed (stored, analyzed et cetera). Since a WSN typically consists of many nodes and only one (or few) gateways, it is very plausible that the majority of the ordinary nodes are outside of the wireless range of a gateway node. Moreover, since WSNs fall in the class of Mobile Ad-hoc Networks (MANETs) there is no central infrastructure. Therefore, each node should be able to act as a *router*: on receive of messages for the gateway from a neighbouring node (i.e. any node within a node's wireless range), the node should be able to forward these messages such that they will eventually arrive at the gateway, via a (multi-hop) path that is as efficient as possible (based on some routing metric). The nodes must be able to determine such paths themselves.

To fulfil this routing functionality, each node needs to select a neighbouring node to forward messages for the gateway to. Because of the energy constraints in a WSN, this selection should be done in such a way that the messages will arrive at the gateway as energy efficient as possible. Combining this with the fact that every message transmission in a WSN takes a relatively large amount of energy, results in a selection of a neighbouring node such that it is on a path to the gateway that minimizes the total number of message transmissions. But because of the extremely unreliable wireless links [11] between nodes in a WSN, the number of message transmissions along a path to the gateway is not simply the number of hops to the gateway.

3.1.1 Unsuitability of Existing Routing Protocols

Why can we not simply use the existing routing protocols that are used in conventional (wireless) networks? WSNs are a special type of MANETs but introduce additional challenges in the design of protocols and software. This is a result of their unique characteristics such as the usually large number of nodes and the severe resource limitations of the nodes, in combination with the dynamically changing topology, unreliable links and high probability for node failure. Especially, *routing* in WSNs is very challenging because traditional (wireless) network protocols are unsuitable due to the aforementioned characteristics [57]. Traditional network routing protocols as Open Shortest Path First (OSPF) and the Routing Information Protocol (RIP) are based on the two major routing algorithms Link State and Distance Vector respectively [29], which require a lot of information exchange and bookkeeping. This is infeasible in WSNs due to the resource constraints and the usually large number of nodes. Even most MANET protocols, optimized for mobile networks are not suitable for WSNs, as they still require too much computational effort, energy and bookkeeping. Moreover, these protocols often assume the layered convention of network architecture, that in WSNs might be be broken for efficiency reasons [57].

3.1.2 Routing Protocol Categories

This subsection provides a short overview of routing protocol categories based on the taxonomy described by Wu [57]. MANET routing protocols can be divided in to categories: table-driven protocols and on-demand protocols. The former category contains routing protocols that maintain a consistent and up-to-date view of the network, for example the Wireless Routing Protocol (WRP). Table-driven protocols are less suitable for dynamic WSNs as they converge very slowly after topology changes. The latter category comprises protocols that only create routes when required (on demand), thereby reducing control overhead. An example of an on-demand protocol is Ad hoc On demand Distance Vector (AODV). AODV is loop-free, self-starting and it scales well.

For WSN routing protocols, three categories can be distinguished: flat, location-based and hierarchical. Direct Diffusion (DD) falls into the first category, it is a flat routing protocol, meaning that all sensor nodes are peers of each other, they all have the same functionality. DD combines data coming from different sources along the route to the gateway, and by doing so, eliminates redundancy and minimizes the number of transmissions. Location-based protocols reduce control overhead by making routing decisions on cached geographical information (often from Global Positioning System (GPS)). An example is Geographic and Energy Aware Routing (GEAR). The third and last category is that of hierarchical routing protocols, of which Low Energy Adaptive Clustering Hierarchy (LEACH) is an example. Nodes are classified into hierarchical groups of functionality. This allows for example for using “the most healthy” nodes as “backbone” routers and giving less healthy nodes (e.g. slower links, lower energy) a less intensive task.

Armed with the general knowledge about routing in WSNs that was presented in this section, we now continue with the informal description of our SPT protocol that is referred to in the title of this thesis.

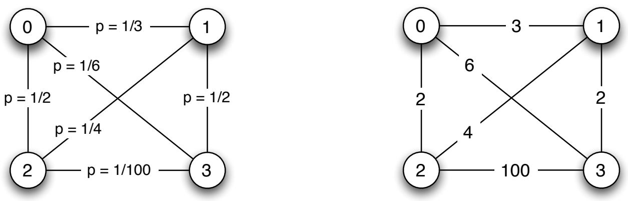
3.2 Informal Protocol Description

This section provides an intuitive description of the SPT protocol that we used as case study. The protocol was developed at the University of Twente and implemented rather abstract in Matlab by Leon Evers, as a proof of concept. It is a network-level routing protocol for WSNs that enables each node to select a neighbour node that is its parent in the global SPT rooted at the gateway. Recall from chapter 2 that a SPT is not necessarily unique for a graph: sometimes a node can make a selection out of several options.

3.2.1 The ETX Routing Metric

As defined in the section on Graph Theory in the previous chapter, a shortest path tree in a weighted graph is a spanning tree, containing the shortest paths from a selected root node to every other node of the graph. In order to apply this theory to WSNs, we need an abstract view on WSNs: namely that of a weighted graph.

If we abstract from the physical details of a WSN, the remaining structure can be viewed as a complete graph, as in figure 3.1. The unreliable wireless medium induces that a sent message arrives at its destination with a certain probability (which may equal zero). In our abstract WSN (i.e. the complete graph in figure 3.1(a)), there is a probability p_{ij} associated to every edge (i, j): a message sent by node i arrives at node j with probability p_{ij}. In other words: each link has a certain link quality. We assume this link quality to be symmetric, that is $p_{ij} = p_{ji}$. In figure 3.1(a), the link between node 2 and node 3 for example, has the associated probability $\frac{1}{100}$ and is thus expected to deliver only 1 out of 100 messages: it is highly unreliable.



(a) Link Qualities **(b)** ETXs (weights)

Figure 3.1: An abstract model of a WSN of 4 nodes: a WSN corresponds to a complete graph

Given this view on WSNs, the routing metric that our protocol uses to base its calculation of shortest paths on, is *the expected number of message transmissions needed to get the message across.* If the successful transmission of a message over a link (i, j) has probability p_{ij}, then the expected number of transmissions necessary to be sure of at least one successful transmission (the message gets across the link) equals the inverse of this link probability (i.e. $\frac{1}{p_{ij}}$).

This metric is in fact a simplified version of the Expected Transmission Count (ETX) routing metric, invented by De Couto [13] in 2004 and meanwhile a known metric in the field [11, 24, 45]. De Couto [13] defines the ETX to be $\frac{1}{p_f \times p_r}$ with p_f and p_r being the forward and reverse probability of a link respectively. This is a consequence from the fact that message reception is normally acknowledged: p_f corresponds to the transmission of a message and p_r corresponds to the returned acknowledgement (of reception of the message) from the receiver, thus in reversed direction. In our protocol we do not account for acknowledgements: we simply rule out their influence.

The ETX values can be considered the weights in our complete graph (fig. 3.1(b)) as they are a measure for the cost of the links. So now we are able to construct an ETX-based Shortest Path Tree. Figure 3.2 shows the construction of a SPT rooted at the gateway. Unless stated otherwise, throughout this work, the gateway will have id and label 0.

Note that the SPT may and will change dynamically due to fluctuating link qualities, as a result of the unreliable wireless medium and possibly also as a result of physical movement of the nodes.

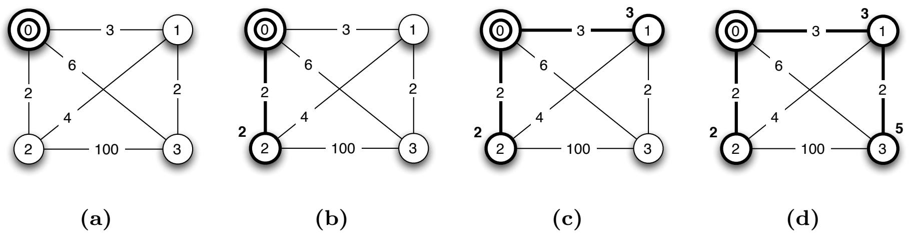


Figure 3.2: ETX-based SPT construction

3.2.2 Unknown ETXs

As shown before, the SPT can be easily built-up when having global information of the network (all ETX values). However, the (distributed) protocol runs locally on each node and thus only can obtain local information of the network. Moreover, the ETX values are unknown initially.

To solve the problem of unknown ETX values, a node needs to *measure* the link quality of the links to its neighbours[1]. By computing the inverse of the measured link quality, an approximation of the ETX is obtained. This measuring of the link quality is accomplished by a periodical broadcast of a probe message containing a node identification number. In other words: time is divided in slots (which we call *message rounds*) and every message round, each node broadcasts a probe message containing its id.

The link quality of a link from node i to node j according to node i (perceived by node i after M message rounds) is the fraction of M probe messages that is sent by j and received by node i. Observe that strictly spoken, a node thus uses the quality of the incoming link for the quality of the outgoing link.

3.2.3 Distributed Operation

To enable a node to select a correct neighbour as its parent in the SPT, the perceived link qualities to its neighbours do not suffice. It will also need the ETX values of the rest of the network to determine which neighbour is on a shortest path to the gateway. This problem is solved by including the *ETX-to-gateway* G (as perceived so far) in each probe message. We prefer to speak about the *distance-to-*G: in the sequel of this thesis we will mainly use the more intuitive term *distance* wherever we mean ETX. To see how the inclusion of the perceived distance to the gateway solves the problem of distributed shortest path finding, consider the following example, illustrated by figure 3.3.

[1]A node is a neighbour of another node if the latter 'can hear' the former.

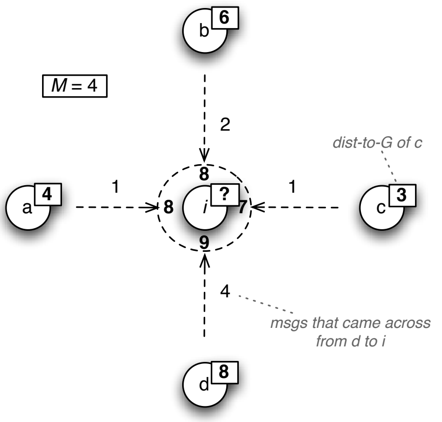


Figure 3.3: The principle of the protocol: the question mark will be 7 in this case (the minimum of the computed distances-to-G: 8, 8, 7, 9).

The figure depicts the situation after four message rounds ($M = 4$). In each of these four message rounds, nodes a, b, c, d and i broadcasted probe messages with their id and their perceived distance to the gateway. At the end of each message round, the nodes determine their shortest distance to the gateway, based on the received information. We assume that nodes a, b, c and d already found that their shortest distance-to-G is resp. 4, 6, 3 and 8 in earlier message rounds. We now focus on node i: let us see how it determines its shortest distance to the gateway. Each node maintains a message round counter and counters for the received number of messages per neighbour. This enables node i to compute its own distance-to-G via each of its neighbours, and to select the neighbour for which a minimum is found.

In the situation of the figure, node i's counter for the number of received probe messages from node a equals 1, so the link to a is considered to have a quality of $\frac{1}{4}$ and the corresponding ETX (distance) is 4. We assume that the message that node i received from node a also contained a's distance-to-G of 4, and therefore node i computes its distance-to-G via a to be $4+4 = 8$. In a similar way, it computes the distances via nodes b, c and d to be respectively 8, 7 and 9 (as indicated within the dashed circle). Consequently, node i now selects node c as its parent because this results in a minimum distance to the gateway of 7.

Note that there is an *important assumption* about the way of sending *data messages* (i.e. sensor data) to the gateway: node i selected parent c and transmits the same data message 4 times, since it estimates the link to c of quality $\frac{1}{4}$ (in this case). Once at node c, it is forwarded in a similar way.

3.2.4 The Gateway and the SPT

All nodes start with their distance-to-G initialized to infinity, except for the gateway. Obviously, the gateway's distance to the gateway is invariantly equal to zero. Therefore, it does not have to compute anything (and it thus does not have to listen to probe messages). Except for this reduced behaviour, the gateway is the same as other nodes: it just periodically sends its messages, saying "I am G and my distance-to-G is 0". In other words: the gateway is the initiator (and thus the root) of the SPT building process.

The shortest path tree at a certain point in time is never explicitly present, so why is it called an 'shortest path tree' protocol? The answer lies in the fact that the spanning tree at any point in time can be constructed from the edges obtained by 'asking' a node to which of its neighbours it currently would send a message that is to be forwarded to the gateway (i.e. "who is your selected parent?"). That will be the node for which it found the minimum distance (at that point in time). The tree found in this way is an approximation of the SPT. As longer the protocol runs, the better the approximation (in a static topology).

3.2.5 Matlab Implementation

As stated at the beginning of this section, a very abstract version of the protocol was implemented in the mathematical software Matlab by Leon Evers. This implementation is used to perform simulations with. This subsection is a short explanation of the implementation, which is contained in the appendices of this thesis (appendix A).

Matlab stands for *Matrix Laboratory* and it is a high-performance language for technical computing. It integrates computation, visualization and programming in an intuitive environment, using familiar mathematical notation. Typical uses include math and computation, algorithm development, modelling, simulation and prototyping, data analysis and visualization, scientific and engineering graphics and even application development. The basic data element is a matrix that does not require dimensioning [37].

Matlab allows for efficient specification of our protocol by means of matrices. The implementation randomly generates node positions for a fixed number of nodes. Based on these positions, it builds a distance matrix that contains distances between every pair of nodes. Using these distances it calculates the receive probability between all node pairs (based on an s-curve of distance vs. receive probability, derived from the empirical study in [56]). The implementation then plots the optimal SPT tree based on these data. It continues with iteratively (per message round) executing the protocol, while plotting the chosen tree (or forest, i.e. a collection of trees) each message round (this allows for visual comparison to the optimal tree). After a pre-set number of message rounds the node positions are shuffled randomly (and distance, receive probability and the optimal tree are re-computed), while the protocol continues operation. This allows for observing the reaction to a topology change: does it converge to the new tree and how fast does it converge? After a fixed number of message rounds it finishes, while plotting statistics on the average quality of the chosen tree and the percentage of correct parents.

This proof of concept provides useful insight in the protocol's internals and dynamics. Moreover, confidence in the correctness of the (implementation of the) protocol is established. It is also useful for experimentation, to improve performance by tuning parameters etc. The source of this implementation can be found in appendix A, together with some screen shots of the generated plots.

3.3 Formal Protocol Specification

My supervisor Mariëlle Stoelinga initiated a draft version of the formal specification of the SPT protocol. The specification (which is in a more definitive form in this section) comprises the protocol in pseudo code, together with a sound recursive characterization. We start with table 3.1, which introduces the reader to the notation used.

G :	The gateway node
N :	The number of nodes in the network
M :	Counter for the number of message rounds (time units)
i, j, f :	Index variables denoting arbitrary nodes, $1 \leq i, j, f \leq N$
d :	Auxiliary variable denoting the distance in a message
p_{ij} :	The probability that a message from node i arrives at node j, we assume $p_{ij} = p_{ji}$.
$R_i^M[j]$:	The number of messages from node j received by node i. Note that this defines the link quality of the link between nodes i and j, as perceived by i after M time units (i.e. the fraction $\frac{R_i^M[j]}{M}$ of messages sent by j that has been received by i). The inverse of the link quality is the expected number of transmissions needed for a message to get across.
$D_i^M[j]$:	Distance (total ETX) from node j to G, according to node i's information after M time units. $D_i^M[i]$ is the distance-to-G from i itself after M time units.
$msg_i^M[j]$:	Message that node i sends to j in round M. Message loss is denoted by $\perp$ (i.e. $msg_i^M[j] = \perp$ if the message gets lost).
$parent$:	Denotes the currently chosen neighbour to forward messages for the gateway to.

Table 3.1: Notation used in the formal specification of the protocol

3.3.1 Pseudo Code

De pseudo code of the protocol is divided in code for the gateway node and code for any other node present in the network. The code for the gateway node (denoted G) simply represents a single thread, which sends periodically a message, containing the gateway's identification and its distance-to-G (our synonym for the total ETX to the gateway), which of course is invariantly equal to zero:

G's Active Thread

```
1 for ever do
2     Send(G, 0);                    // broadcast probe msg
3     Wait(1);                       // wait for 1 time unit
4 end
```

Figure 3.4: Pseudo code of the gateway's active thread

The code for node i, $i \neq G$, is divided in three parts: the initialization, a passive thread and an active thread. Observe that the subscript index i and the superscript M used in the notation in table 3.1 (for example in $D_i^M[j]$) are not necessary here and thus omitted. However, their presence in the table will be justified in the recursive characterization of the next subsection.

Node i's initialization

```
1 $M := 0$;
2 $\forall j : D[j] := \infty$;
3 $\forall j : R[j] := 0$;
```

Figure 3.5: Pseudo code of node i's initialization ($i \neq G$).

Node i's Passive Thread

```
1 for ever do
2     $(j, d) \leftarrow$ Receive();          // wait until receive
3     $R[j]++$;                      // count received msg of j
4     $D[j] := d$;                   // record dist-to-G of j
5 end
```

Figure 3.6: Pseudo code of node i's passive thread ($i \neq G$).

The function call on line 2 listens to the wireless channel for probe messages of other nodes. If such a probe message is received, it delivers the message as a tuple (j, d) with j the id of the sender of the message and d the distance-to-G of the sender (as perceived so far by the sender).

Node i's Active Thread

```
1 for ever do
2     Send($i$, $D[i]$);                                  // broadcast probe msg
3     $M := M + 1$;                                      // update msg round number
4     Wait(1);                                           // wait for 1 time unit
5     $D[i] := \min_{1 \le f \le N, i \ne f} \left\{ \frac{M}{R[f]} + D[f] \right\}$;    // compute dist-to-G
6     $parent := f_{min}$;                               // set parent
7 end
```

Figure 3.7: Pseudo code of node i's active thread ($i \neq G$).

The distance computation on line 5 is important here. One might think that it is more efficient to do this on message reception. However, the distance must be updated each message round, regardless of whether a message is received: that way the distance decreases if messages arrive and it increases as long as no messages are received. The variable f is used to range over all node ids and f_{min} in line 6 is the node id for which the minimum is reached in line 5.

3.3.2 Recursive Characterization

A more abstract way of formally specifying a protocol, is to characterize the behaviour mathematically. Our SPT protocol is conveniently specified recursively, because of its periodicity (i.e. message rounds). This results in a concise specification of mathematical rigour, that abstracts away from implementation details such as the location of functionality (e.g. different threads). A recursive definition consists of a non-inductive basis and one or more inductive clauses (that induce the recursion).

The basis of this recursive characterization corresponds to the initialization part of the pseudo code above. The inductive clauses correspond to the active and passive threads of a node. Below we explain the inductive clauses in more detail.

Basis

$$R_i^0[j] = 0 \tag{3.1}$$

$$D_i^0[j] = \begin{cases} 0 & \text{if } i = G \\ \infty & \text{otherwise} \end{cases} \tag{3.2}$$

Inductive clauses

$$msg_i^{M+1}[j] \quad = \quad \begin{cases} d = D_i^M[i] & \text{with probability } p_{ij} \\ \perp & \text{with probability } 1 - p_{ij} \end{cases} \tag{3.3}$$

$$R_i^{M+1}[j] \quad = \quad \begin{cases} R_i^M[j] & \text{if } msg_j^{M+1}[i] = \perp \\ R_i^M[j] + 1 & \text{if } msg_j^{M+1}[i] = d \end{cases} \tag{3.4}$$

$$D_i^{M+1}[j] \quad = \quad \begin{cases} D_i^M[j] & \text{if } i \neq j \text{ and } msg_j^{M+1}[i] = \perp \\ d & \text{if } i \neq j \text{ and } msg_j^{M+1}[i] = d \end{cases} \tag{3.5}$$

$$D_i^{M+1}[i] \quad = \quad \min_{1 \leq f \leq N, i \neq f} \left\{ \frac{M+1}{R_i^{M+1}[f]} + D_i^{M+1}[f] \right\} \tag{3.6}$$

The variable $msg_i^{M+1}[j]$ (3.3) introduces the probability of message arrival p_{ij}. This is essential as we deal with an unreliable wireless medium. The message sent by node i to node j will get across with probability p_{ij} and it will get lost with probability $1 - p_{ij}$. Note that in equations 3.4 and 3.5, we need $msg_j^{M+1}[i]$ instead of $msg_i^{M+1}[j]$ (recall that we assume $p_{ij} = p_{ji}$).

As stated in table 3.1, the variable $R_i^{M+1}[j]$ of clause 3.4 is a message counter for determining the link quality. The link quality of the link between nodes i and j (as perceived by i and after M message rounds) is defined to be $\frac{R_i^M[j]}{M}$.

The distance is computed as follows: if node i did not receive anything from node j in a message round, then the distance of j to G as perceived by i does not change (first line of equation 3.5). If it did receive a message then the distance of j to G as perceived by i is set to the distance d contained in the message (second line of equation 3.5).

What remains for node i is to determine its own distance to the gateway. This is the distance of a shortest path to the gateway starting from node i itself. Such a path is a combination of the distance of a path from a neighbour to the gateway and the expected number of messages based on the measured quality of the link to that neighbour (equation 3.6). Note that the latter is the inverse ($\frac{M+1}{R_i^{M+1}[n]}$) of the link quality ($\frac{R_i^{M+1}[n]}{M+1}$).

CHAPTER 4

Verification using UPPAAL

In the previous chapters, we discussed the background of this research project and the details of the Shortest Path Tree (SPT) protocol, which functions as our case study. We learned from chapter 3 that we are given an idealized routing protocol description, which has been mathematically proven correct by my supervisors and implemented and simulated using Matlab. Therefore, there is already considerable confidence in its correctness. We are now going to formally verify this protocol, on the one hand to increase (or decrease) our confidence in its correctness, and on the other hand, to document experiences and boundaries of the used tools. The focus is mainly on the latter, as the ultimate goal is to explore the possibilities for developing a platform for formal verification experiments.

This chapter discusses our modelling and verification activities and the results, using the model checking tool UPPAAL. Chapters 5 and 7 do the same for resp. SPIN and PRISM. Each of these chapters introduces the tool and motivates why it has been chosen. This tool introduction is then followed by an elaboration on modelling our protocol using the considered tool, which in turn is followed by a section about the verification experiences and results. Each of these three chapters ends with a summarizing section that reports the modelling and verification experiences and (sub)conclusions.

4.1 Tool Introduction

The Uppsala-Aalborg Model Checker (UPPAAL) is a tool suite for modelling, validation and automatic verification of real-time systems. The tool is developed in collaboration between the Department of Information Technology at Uppsala University, Sweden and the Department of Computer Science at Aalborg University in Denmark. UPPAAL's first version was released in 1995 [5, 32] and it has been in constant development since. The current official release is UPPAAL 4.0.7 (Nov. 24, 2008), which is a mature product. It is freely available at http://www.uppaal.com for non-commercial applications.

4.1.1 Underlying Theory

The underlying theory is that of *timed automata*. In general, an UPPAAL model is a network consisting of timed automata composed in parallel, which can be constructed, simulated and verified [4]. A *timed automaton* is a finite-state machine extended with clocks: it is a sextuple (L, l_0, C, A, E, I) with a set of locations L, an initial location l_0, a set of clocks C, a set A of actions, co-actions and the internal τ-action, a set E of edges between locations and a function I to assign invariants to locations.

The UPPAAL modelling language extends traditional timed automata with a rich set of extra features. Among these are constants, bounded integer variables, binary synchronization, broadcast channels, urgent locations and urgent synchronization, committed locations and array variables [4]. These additional features have proven to be very useful for model construction. To specify correctness properties, UPPAAL uses a query language that is a subset of the temporal logic Computation Tree Logic (CTL). It allows for specification of path formulae and state formulae, but nesting of path formulae is not possible.

We refer to the literature for further details about and explanation of the UPPAAL modelling language, syntax and semantics of timed automata, UPPAAL's query language, et cetera [4].

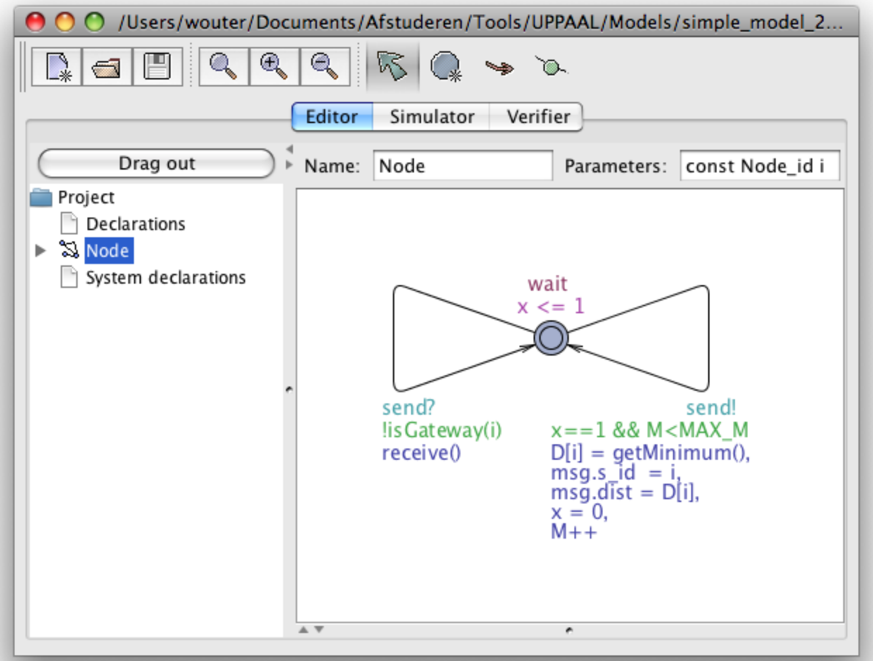
4.1.2 Tool Motivation

The choice for UPPAAL is mainly made based on the fact that it is a very well-documented, mature, state-of-the-art tool with an intuitive Graphical User Interface (GUI) and a not so steep learning curve. It enables the user rather easily to graphically define timed automata that represent the different processes to be modelled.

An additional part of the motivation is that UPPAAL has been used successfully for several industrial case studies [4], e.g. Bang & Olufsen Audio/Video Protocol, Bounded Retransmission Protocol, Philips Audio Protocol, Root Contention Protocol [51], etc. Moreover, UPPAAL is used in several studies mentioned in the related work section of chapter 2 (e.g. [16, 55]).

At first, an additional reason was that UPPAAL is a real-time model checker and our protocol has certain timing aspects (message rounds). However, as we proceed with this chapter, it will become clear that we hardly exploited any of the timing features of UPPAAL because we wanted our model to be comparable to the models for the other tools (i.e. SPIN, PRISM).

We used UPPAAL version 4.06 (rev. 2987), released March 2007. Figure 4.1 depicts UPPAAL's GUI, to get a first impression of the tool. The hardware used is a 13.3" MacBook (Intel Core2Duo processor on 2 GHz, 2 GB RAM) running Mac OS X 10.5 Leopard.



Name: Node   Parameters: const Node_id i

```
wait
x <= 1

send?
!isGateway(i)
receive()

send!
x==1 && M<MAX_M
D[i] = getMinimum(),
msg.s_id = i,
msg.dist = D[i],
x = 0,
M++
```

Figure 4.1: The GUI of UPPAAL 4.0.6 in edit mode

4.2 Model Construction

In this section we elaborate on the process of constructing a model of our SPT protocol using UPPAAL. The difficulties and challenges encountered are pointed out and the models we constructed are explained in detail. We took an iterative approach to modelling the protocol: we started with a simple model that lacked a lot of of important functionality, and iteratively increased complexity by adding functionality. The upcoming three subsections illustrate that approach. The fourth subsection introduces a common problem with communication protocol models: the execution order of the nodes causes a blow-up of the number of states. It also presents our pragmatic solution to this problem.

Unlike the subsequent chapters, this chapter rather extensively describes the modelling process. This allows for becoming familiar with modelling in general and with the protocol and its 'ins and outs'. We continuously validated our results and ideas using UPPAAL's built-in simulator. Appendix B contains an overview of and insight into this simulation process.

Before moving to the model construction process, let us first explain some additional terminology of UPPAAL. A model of any system in UPPAAL is a network of extended timed automata that represent the different processes of the system. They are graphically specified in so called *templates*. A template is in fact a *class* of processes (extended timed automata): a process is an instance of a template. Template instances (processes) are created with a *process declaration*. These processes can then be composed (in parallel) into a system: a *system declaration*. Templates may be parameterized: each formal parameter in the template is substituted for its actual value (which is passed as an argument in the process declaration). Variables used in templates must be declared either globally or locally: *global declarations* hold for all templates, *local declarations* are local to a certain template.

4.2.1 Protocol Model V1

To become familiar with a tool and its possibilities and restrictions, it is often a good idea to start with the construction of some naive try-out models. Such try-outs lack important behaviour but they introduce the model constructor to the various aspects and difficulties that he or she will have to find solutions for, sooner or later. After experimenting with UPPAAL and building some try-out models, we made the following observations, on which we based the first version of our SPT protocol model.

1. A straightforward approach to start modelling the protocol is to simply map each single thread to a single process. This would result in a process for the gateway's active thread (page 35) and separate processes for both the passive and the active threads of a node (p. 35 and 36). This would be a bad start w.r.t. the state space explosion problem: a model of only two nodes and the gateway, already results in a parallel composition of five processes! It turned out that it is much more intuitive, more efficient and more practical to model a single node as a single process that mimics the behaviour of both the passive and the active thread.

2. The gateway should not be modelled separately: it is in principle a normal node with some restrictions on its behaviour (unlike ordinary nodes it does not have a passive thread and it does not need initialization).

3. The arrays for distance-to-G and receive counters should be modelled as local arrays (local to each node) instead of globally. Further, a node should have a local round number and a local clock.

4. UPPAAL provides support for the primitive data types `integer` and `boolean`. The protocol however requires us to model probabilities (link quality) which are fractions (floating point numbers) rather than integers. We therefore decided to model link qualities as percentages.

5. There should be a global constant number to represent infinite distance and some mechanism that initializes the entries of the local distance array of each node to this number.

6. There should be a global constant to set the maximum number of message rounds to be executed, otherwise verification is infinite.

These observations led to a first serious protocol model that can be simulated and verified, although it is simplified in several ways: every node can hear every other node with 100% link quality.

Listings 4.1 – 4.3 together with figure 4.2 show the entire UPPAAL model. This comprises global declarations, local declarations, the node template and the system declaration (the entire source of the UPPAAL models can also be found in appendix B). Declarations are specified textually in UPPAAL's editor, which supports code colouring. Templates are specified graphically in the editor, by drawing locations and edges. In the sequel we walk through the elements of the model and elaborate on the details of it.

Global Declarations

Variables and functions that are declared globally are available for all templates (instances) in the system. This is useful for defining general parameters of the protocol (such as the number of nodes), but also for modelling the message channel and the message itself. The global declarations are printed in listing 4.1. The C-style block and line comments speak for theirselves. Pay some attention to the type definition of type `Node_id` (line 20), which is defined to be the integer interval $[0, N-1]$. This interval consists of N values that are used as node identification numbers (ids). The node template is (as we will see soon) parameterized with a variable of this type and that results in a system with N node processes (instances of the node template) with the respective node ids $i = 0$, $i = 1$, $i = 2$,. . . , $i = (N-1)$.

```
 1  /*****
 2  Shortest Path Tree protocol for Wireless Sensor Networks
 3  Author: W.M. Everse
 4
 5  Simple model
 6      All nodes can hear each other with link quality of 100%
 7      Nodes operate synchronously
 8  *****/
 9
10  // Number of nodes, including gateway(s) G
11  const int N = 4;
12
13  // Maximum number of message rounds
14  const int MAX_M = 10;
15
16  // Big number to represent 'infinite' distance
17  const int MAX_DIST = 10000;
18
19  // Define type Node_id, parameter of Node template
20  typedef int[0,N-1] Node_id;
21
22  // Synchronization channel to model message sending/receiving
23  broadcast chan send;
24
25  // Model a message as a struct
26  // msg.s_id = sender id and msg.dist = distance
27  meta struct{
28      Node_id s_id;
29      int dist;
30  } msg;
31
32  // Determine whether the given node is a gateway node
33  bool isGateway(Node_id node) {
34      return node == 0;
35  }
```

Listing 4.1: UPPAAL Model V1 - Global Declarations

Another interesting global declaration is the broadcast channel (line 23), a useful feature of UPPAAL. This is a synchronization channel but unlike conventional binary synchronization, it synchronizes with all processes available for synchronization (instead of just one). If there is no process to synchronize with, it can still be executed (i.e. broadcast sync is non-blocking). In this model, the channel `send` is used for synchronized broadcasting of messages. A sending node performs a transition with label `send!` and all receiving nodes at the same time perform a transition `send?` (as we will see below in the node template).

The probe messages of the protocol are modelled as a `C`-like `struct`, a structure consisting of a node id (to denote the sender) and a distance (lines 27–30). A node that is going to broadcast a probe message sets the `msg.s_id` field to its own id and the `msg.dist` field to its perceived dist-to-G (`D[i]`) and starts broadcasting. Note that keyword `meta` precedes the message definition, which means that we do not want UPPAAL to consider message content as part of the states of our system (because it is duplicate information).

Finally, the global function `isGateway(Node_id node)` (l. 33) is a *user-defined function* (introduced in UPPAAL version 4.0) and can be directly called from automaton transitions and other user-defined functions. It takes one argument being of type `Node_id` and returns `true` if it equals 0. In other words: this model contains exactly one gateway node, which is the node with id $i = 0$.

Local Declarations

Before we move to the node template itself, we will first discuss the corresponding local declarations. These are declarations of items used in the template (they are local to the template thus cannot be used by for instance another template). The local declarations of our node template are printed in listing 4.2. The node template is parameterized but UPPAAL shows parameters separately from the local declarations. In fact, parameters are special local declarations so we added them here, entirely at the top in commented code, for illustrative purposes. In this case there is one parameter: a constant i of type `Node_id`. This enables us to create an instance (a process) with a specific value (of type `Node_id`) for i. Moreover, UPPAAL enables us to create an instance for each possible value of type `Node_id`, by not specifying a value for i at all. We return to this later.

```
/* <parameter>const Node_id i</parameter> */    //explained in text

// Node clock
clock x;

// Local round number
int M;

// Dist-to-G per node (D[y] = dist-to-G from y, perceived by this node)
int D[Node_id] = {MAX_DIST, MAX_DIST, MAX_DIST, MAX_DIST};

// Msg counters per node (R[y] = #messages received from node y)
int R[Node_id];

// The selected parent
meta Node_id parent;

// Function to determine the minimum distance
int getMinimum(){
    meta int minval = MAX_DIST;      // To hold the min. found so far
    meta int try;                    // To hold the next value
    if( isGateway(i) ) return 0;     // Minimum dist-to-G of G is 0
    if( M == 0 ) return MAX_DIST;    // First round return MAX_DIST
    for(j : Node_id){
        if( R[j] > 0 && j != i && D[j] < MAX_DIST ){
            try = M/R[j] + D[j];
            if( (M % R[j]) >= (R[j] / 2) ) try++; // Round to nearest int
            if( try <= minval){
                minval = try;
                parent = j;
            }
        }
    }
    return minval;                   // Return the min. found
}

// Function executed on message reception
void receive(){
    R[msg.s_id]++;                   // Increase msg counter of sender
    D[msg.s_id] = msg.dist;          // Update Dist-to-G of sender
}
```

Listing 4.2: UPPAAL Model V1 - Local Node Declarations

Besides its id i, a node has a local clock and a local round number. It also maintains local arrays containing a dist-to-G (initialized to `MAX_DIST`) and a message counter for each node, and a (`meta`) field for the currently selected parent. Furthermore, a node has two local functions. The function `getMinimum()` computes and returns the minimum distance of this node to the gateway (implementing equation 3.6 on page 37). Function `receive()` is called on message reception and saves the message content and updates the message counter (which corresponds to line 3 and 4 of the passive thread in figure 3.6 on page 35).

The function `getMinimum()` (lines 19–35) needs some more explanation. As said, it computes and returns the minimum distance of a path to the gateway, based on the perceived information so far. To achieve this, the `for`-loop ranges over all entries of the local node arrays. That is: for each node, the distance of a path via that node to the gateway is computed and the minimum is returned. This computation (`try = M/R[j]+D[j]`) involves a fractional: the inverse of the link quality (i.e. the expected number of transmissions for a message to come across). Since UPPAAL does not support floating point numbers, this fractional results in just the integer part. This is always the nearest *smaller* integer while we rather want it to be rounded to the nearest integer. Therefore line 27 is added: if the remainder of the division is greater or equal than half the denominator, then the result should be increased with one. Subsequently it is checked whether the result is a minimum (so far) and if so, the parent field is set to the node for which the minimum is reached (corresponding to f_{min} in fig. 3.7, p. 36).

Node Template

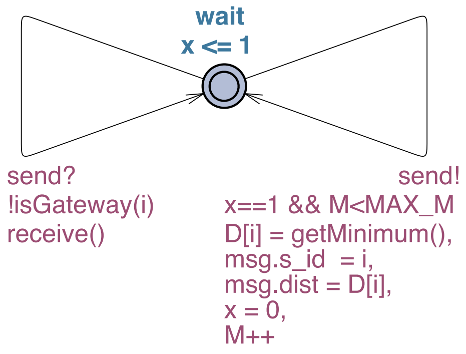


Figure 4.2: UPPAAL Model V1 - Node Template

Figure 4.2 depicts the node template. It consists of just one location and two edges, one for each thread. The left edge corresponds to the passive thread and the right one to the active thread. Edges may be annotated with *selections*, *guards*, *synchronizations* and *updates*. Selections can be used to non-deterministically bind a value of a given type to a given identifier. We did not use such a selection on any of the edges in the automaton of fig 4.2.

The Passive Thread Edge

As mentioned above, the instance of the node template with id $i = 0$ is defined to be the gateway. The different behaviour of the gateway is achieved by using a *guard* on the receive edge in the automaton. A guard is a boolean expression and must evaluate to `true` for the associated edge to be enabled. The guard on the right edge (in the form of the boolean function `isGateway(Node_id node)`) is only true for ordinary nodes with $i > 0$. The receive edge in the automaton of the gateway node (i.e. the node with $i = 0$) is thus always disabled. In other words: a gateway never receives probe messages of this protocol.

This passive thread edge also bears a synchronization label and an update label. The latter contains a call to `receive()`, we already saw that this function updates the receive counter and records the message content. The former (`send?`) is to synchronize with another process containing an edge bearing the label `send!`, where "synchronize" means that both edges are fired at the same time (if both are enabled). According to the UPPAAL manual, "The intuition is that two processes can synchronize on enabled edges annotated with complementary synchronization labels, i.e. two edges in different processes can synchronize if the guards of both edges are satisfied, and they have synchronization labels `e1?` and `e2!` respectively, where `e1` and `e2` evaluate to the same channel. (...) The update expression on an edge synchronizing on `e1!` is executed before the update expression on an edge synchronizing on `e2?`" [4]. Our global broadcast channel `send` thus allows us to synchronize a sending node (`send!`) with all receiving nodes (`send?`). Doing so, probe message broadcast is simulated.

The Active Thread Edge

The guard on the right edge (the active thread) ensures that this edge is only enabled whenever local clock `x` equals 1 and the local message round number is less than the maximum number of message rounds `MAX_M` (a global constant to bound verification and random simulations, as we will see later). Further, this edge is annotated with the synchronization label `send!`, the complement of the one on the passive thread edge. The update label, finally, contains 5 update expressions to update respectively the node's own distance-to-G, the message contents, the local clock `x` (reset) and the local message round counter `M`.

Location Invariants

The location in this template is labelled “`wait`” and has the location invariant `x<=1`. This means that a process of this template can only be in this location whenever the value of local clock `x` is smaller or equal to 1. A process whose local clock equals 1 must take the right edge of the automaton due to the guard `x==1` on this edge. This way message rounds are simulated. Note that we thus (mis)use time in UPPAAL, the model is synchronous and discrete. We will return on this later on.

System Declaration

```
// This instantiates template Node(const Node_id i)
// for all values in Node_id.
system Node;
```

Listing 4.3: UPPAAL Model V1 - System Declaration

The model that UPPAAL will use during simulation and verification is the model that is specified in the system declaration (listing 4.3). The processes to be composed (in parallel) into a system are simply listed behind the `system` keyword. Processes are named instances of templates: the process declaration `p1=Node(1);` creates a process named `p1`, which is an instance of the `Node` template with $i = 1$. A system is thus a parallel composition of concurrent processes (e.g. `p1||p2||p3`) together with global and local variables and channels. However, in listing 4.3 we use a feature of UPPAAL: as said before, specifying a parameterized template without actual values will result in all possible instances of the template w.r.t. the type of the parameter. In our case, we only have one template to list (i.e. `Node`), for which we do not specify an actual value for the parameter.

Note that we therefore can control the number of processes that will be composed into a system simply by changing our global variable `N`. For example, setting `N = 4` results in the type `Node_id` containing the values 0, 1, 2 and 3. Composing the node template into a system (the way it is done in listing 4.3) will now result in an instance for every possible value of the template's parameter `i`: we thus always obtain `N` processes.

4.2.2 Protocol Model V2

In our first version of the model, every node can hear every other node with link qualities of 100%. This does not properly reflect the characteristics of a Wireless Sensor Network (WSN). We therefore created a second version of our model. This subsection explains the extensions w.r.t. version 1 of our model. The source of the entire model is contained in appendix B.

Version 2 of our model allows for specification of connectivity between the nodes in a *connectivity matrix*: a $N \times N$ matrix `C` that defines the link quality between all node pairs as a percentage. This matrix is symmetric as the protocol assumes the link quality from A to B to be the same as the quality from B to A. We declared a global constant 2-dimensional array of integers (from 0 to 100) with dimensions `Node_id` $\times$ `Node_id`. This rare form of dimensioning is possible in UPPAAL and results in the values of the type to be used for the indices of the array (i.e. `int[0,N-1]`). An example of such a global declaration of array `C` (with initializer) is contained in listing 4.4.

```
// Symmetric connectivity matrix with percentages
// if C[x][y]==100 then x can hear y with quality of 100%
const int[0,100] C[Node_id][Node_id] = {
    { 0, 15, 10, 90},
    {15,  0,  0, 33},
    {10,  0,  0,  0},
    {90, 33,  0,  0}
};
```

Listing 4.4: UPPAAL Model V2 - Global Connectivity Matrix

Since the array is declared as a constant, it must be initialized. We already saw an example of an *array initializer* in the local declarations: it initializes all entries of the dist-to-G array with the value of the global constant `MAX_DIST`. In the case of listing 4.4, the initializer specifies a 4×4 array, the corresponding number of nodes `N` equals 4 and both dimensions of the array are indexed 0, 1, 2, 3. The example topology specified by this initializer is depicted in figure 4.3. Note that whenever we change the number of nodes `N` in our model, we should update this array initializer to reflect our connectivity wishes and to match the new number of nodes. We should also update the initializer of the local dist-to-G array to match the new number of nodes.

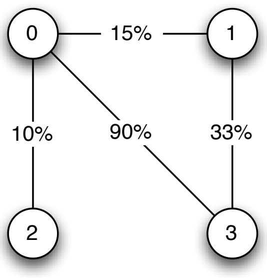


Figure 4.3: 4-node topology specified by the array initializer in listing 4.4

It is now possible to add connectivity information to the model, but what still remains is that it should be taken into account somewhere. That is, either on message sending or on message reception. We found the latter more intuitive and easier to implement, since message reception is modelled using the dedicated node function `receive()`. For each link, the model should ensure a certain percentage of message loss (i.e. 100% minus the given percentage in the connectivity matrix). In order to realize this, we took a “message loss balancing approach”: the model itself balances the actual message loss ratio around the implied value. The modified node function `receive()` is listed in listing 4.5.

```
 1  // This counter is used to realize link qualities, see receive() below
 2  meta int lost_from[Node_id];
 3
 4  // Function executed on message reception
 5  // It also enforces the given link qualities (from connection matrix)
 6  void receive(){
 7      // Compute the current enforced lost ratio
 8      int lost_ratio = 100 * lost_from[msg.s_id] / (M+1);
 9      if(lost_ratio < (100 - C[i][msg.s_id]) )
10          lost_from[msg.s_id]++;      // The message got lost
11      else
12      {   // Message received
13          R[msg.s_id]++;              // Increase msg counter of sender
14          D[msg.s_id] = msg.dist;     // Update Dist-to-G of sender
15      }
16  }
```

Listing 4.5: UPPAAL Model V2 - Message Loss Balancing

How does this ‘message loss balancing’ work? To determine the actual message loss ratio it is necessary to keep track of the total number of messages that is sent (which corresponds to the message round number `M`) and the number of messages that have been discarded in the past. This is implemented as follows. We added an array `lost_from[Node_id]` to the local node declarations, to keep track of the number of lost messages so far (`Node(i).lost_from[j] == 20` means that node `i` lost 20 messages from node `j`). Every time a message comes in, the current loss ratio is determined (line 8). This is compared to the ratio implied by the connectivity matrix (i.e. 100% minus the given link quality for that link). If the actual loss ratio is smaller than what it should be (the derived loss ratio), the message is discarded (and the loss ratio thus increases, line 10). Otherwise the message is passed (‘received’) and the message counter and distance corresponding to the sender of the message are updated (lines 13 and 14).

4.2.3 Protocol Model V3

Version 2 of our protocol model ‘balances’ the loss rate between every node pair according to the connectivity matrix. This rather “artificial” method simulates the probability p_{ij} that a message arrives at its destination. This works, but the method is very predictable and deterministic (we know in advance which message gets lost). This in contrast to the real stochastic nature of message loss on wireless channels. Therefore we created a third version of our model (appendix B).

Version 3 of our model attempts to model the message loss unpredictably by defining an explicit *non-deterministic* process for each link. The only way of introducing unpredictability in UPPAAL is using *non-determinism*. The automata in the models presented so far were *deterministic*: in a single automaton is no situation in which two or more edges are enabled at the same time. If there is such a situation, unpredictable behaviour is introduced since the edge to be fired will be chosen non-deterministically by UPPAAL.

In the following we will explain the modelling of explicit links (i.e. a separate process for each concrete link) using a simple non-deterministic link template. We further propose a general template to avoid a lot of manual labour.

Explicit Link Processes

In the previous models, links between nodes are modelled implicitly using a broadcast synchronization channel. In order to introduce non-determinism in message delivery, our approach was to model the links explicitly, by constructing a non-deterministic process for each link. That would enable the following example scenario: if a link has a given quality of 75%, its message loss should be 25%: 1 out of 4 messages sent over this link should get lost. This specific link process should thus non-deterministically decide which one of the four messages to discard. The result is that it is not predictable in advance anymore which message gets lost.

Explicit links need parameters `src` and `tgt` of type `Node_id` that indicate the id of the source and target nodes of the link. Note that these links are thus unidirectional. The link is an intermediary process in the parallel composition: `src||link||tgt`. On one end, the sending node (with id `i=src`) synchronizes with a link process by executing a `send[i]!` action. Now the message is "in" the link. If the link (non-deterministically) decides to deliver the message, it synchronizes with the receiving node by executing a `recv[tgt]!` action. Note that we now need a broadcast synchronization channel for each node for both sending and receiving (i.e. we declare two global arrays: `broadcast chan send[Node_id]`, `broadcast chan recv[Node_id]`).

An example of a UPPAAL template for such a non-deterministic link process is depicted in figure 4.4. It represents a link of quality 50%: it should discard 50% of the message that are sent over it. Starting from the initial location in the center, there is a choice for either the left or the right branch. In the left branch, the first of two messages is lost (the `send[src]?` action

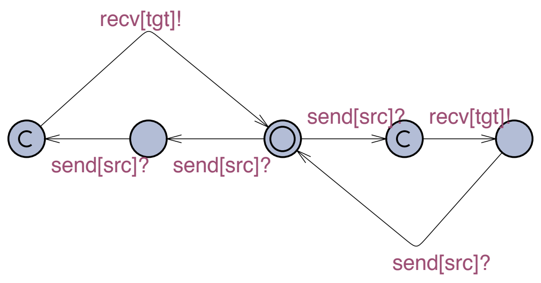


Figure 4.4: Non-deterministic template for a link with quality of 50%

is not followed by a `recv[tgt]!` action), the second is delivered properly. The right branch does the inverse. Note the locations marked with a `C`, these are *committed locations* and ensure that message passing stays atomic. Committed locations are an additional feature of UPPAAL. According to the manual ([4]), a state of the composed UPPAAL system is committed if any of the locations in the state is committed. A committed state cannot delay and the next transition must involve an outgoing edge of at least one of the committed locations.

We can now for example declare an UPPAAL system by composing a source node process, a link process and a target node process:

```
source = Node(0);
link01 = Link50(0,1);
target = Node(1);
system source, link01, target;
```

Listing 4.6: Example System Declaration

In the system above, the template `Link50` from figure 4.4 is instantiated with actual values 0 and 1 for the parameters `src` and `tgt`.

Generalized non-deterministic link templates

Building the link templates as depicted in figure 4.4 is rather naive and inefficient: it requires us to build a template for all possible kinds of links. Moreover, specific link qualities result in templates with many locations and branches: for example a template representing a link of $66\frac{2}{3}$ discards one out of three messages, which results in already 3 branches and 13 locations. Therefore we searched for a generalization.

The approach followed above to construct non-deterministic link templates can be generalized using two more parameters u and v for the link template. It then should ensure that u out of v messages are properly *delivered* and that the remaining $v - u$ messages get lost. Figure 4.5 depicts our solution template, the corresponding local declarations are listed in listing 4.7.

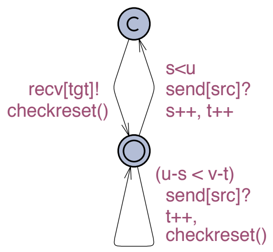


Figure 4.5: Generalized non-deterministic template: u of v messages gets across

Starting in the initial location, the template consists of two branches. The upper branch delivers an incoming message properly (`send[src]?` followed by `recv[tgt]!`). The lower branch causes a message to get lost. The generalized link ensures that the upper branch is taken u times and the lower branch thus $v - u$ times. This is realized with guards, using two local counters (lst. 4.7): `s` counts the number of messages that are properly delivered (that came across the link) and `t` counts the total number of attempts (to send a message over this link), modulo v. This *modulo v* is realized using the local function `checkreset()`, which is called from both branches.

```
/* <parameter>const Node_id src, const Node_id tgt, const int u, const int v</parameter> */
/*****
LinkAccross

Nondeterministic process that represents an unidirectional link
    between two nodes (parameters 'src' and 'tgt').

The quality of this link is determined by parameters u and v:
    u on v messages get across. Which ones is decided
    nondeterministically.
*****/

// counts number of messages that came across
int[0,u] s = 0;

// counts period
int[0,v] t = 0;

// resets counters if v msgs passed
void checkreset(){
    if(t==v){
        s=0;
        t=0;
    }
}
```

Listing 4.7: Local declarations of link template of model V3

So far, we have two functional UPPAAL models (V2 and V3), both based on model V1. Model V2 allows for specification of link qualities through a rather deterministic method using a connectivity matrix, while in model V3 this is done using explicit non-deterministic link processes for each (unidirectional) link. We continuously validated the behaviour of our ideas using the simulator of UPPAAL, which showed us that the models indeed behave as expected (see appendix B for simulation results).

4.2.4 Adding Execution Order

While simulating our models, we noticed that small topologies already result in many possible states of the model. This is what is called the state space explosion problem which we described in chapter 2. The number of states of the composed system is exponential in both the number of variables and the number of parallel processes. This is a problem since most automatic verification techniques require the generation, exploration and storage of all possible states, while resources (memory, time) are limited.

Our models have to deal with many parallel processes since the bigger the topology, the more parallel processes need to be composed into a system. In order to be able to simulate (and verify) a network that is as large as possible, we thus need to reduce the number of states as much as possible. Current model checking tools by default apply automatic state space reduction techniques such as Partial Order Reduction (POR) and so does UPPAAL[4]. The tools also contribute by using efficient data structures and by applying memory reductions. In addition, the user of of a tool can of course manually contribute to a minimal state space by minimizing the number of variables, variable ranges and parallel processes. We already account for these things during construction and as a consequence we cannot gain much by further minimization. There is however yet another possibility to rather drastically reduce the number of states of our models.

Models that consist of many similar parallel processes often exhibit much symmetry in the state space ([19]), and so do our models: all node processes are very similar, they only differ in their identifier `i`. This similarity induces *behavioural equivalent execution orders*: executing process A followed by B is equivalent to executing B followed by A, i.e. both executions result in equivalent states of the model! In fact, this requires us to prove the existence of an equivalence relation. We postpone this to future work.

In our models, a message round is just a period of time in which every node broadcasts a probe message exactly once (executes the active thread). Within one message round, the order in which the nodes broadcast is not fixed: all different orders are possible. This corresponds to all permutations of the node identifiers: there are $N!$ different possibilities per message round for N nodes. The order in which the nodes broadcast does however not matter: all permutations are behavioural equivalent! This allows for executing only one of all possibilities each message round, causing a drastic reduction of the number of states.

POR is a state space reduction technique that exploits (partial) equivalence of all possible interleavings of actions of parallel processes [3]. In contrast, the kind of reduction we described above is of a higher abstraction level: it is concerned with interleaving of parallel processes rather than with the interleaving of single actions within parallel processes. POR is a fully automatic technique implemented in model checkers, but it can not detect the higher level behavioural equivalence. It is possible however for model checker tools to account for this kind of symmetry but the modeller needs to specify the equivalent behaviour by using *scalar sets*[19]. Scalar sets are sets of integer-like elements with a limited number of operations (assignment and identity testing). We tried to update our models by using these scalar sets, but using them breaks important functionality (distance recording, computation and message counting). Building a complete new protocol model using scalar sets might be interesting, we added it to future work.

Instead, we followed a pragmatic approach to solve this problem. We manually modified our models in order to fix the execution order within each message round. As can be seen in appendix B we added a global variable `turn` and a global constant boolean `NOTURN` to toggle between fixed and unfixed execution order. If `NOTURN==false`, the process to execute is the process with identifier equal to `turn`. After execution, `turn` is increased (modulo `N`) such that the next process can execute. The initial value of `turn` determines the node process to execute first. The effect of fixed order execution can be seen in appendix B, and is discussed later on.

4.3 Verification

This section describes the verification process using UPPAAL. We verified models V2 (connectivity matrix) and V3 (non-deterministic links) using several correctness properties. The detailed results are contained in appendix B. Here we provide a global overview of the results, point out noteworthy results and discuss them.

UPPAAL's Java GUI is connected to its model checking engine via TCP/IP (implementing a client-server architecture). Therefore, the two components may also be run on different machines. The engine runs locally by default and there is a stand-alone version for the command-line [4].

A model is verified against a requirement specification. This specification is expressed as a property in a simplified version of the temporal logic CTL. CTL consists of state and path formulae but UPPAAL does not allow nesting of path formulae. A state formula is an expression that can be evaluated for an individual state. Path formulae quantify over paths in the model. Path formulae can be classified in reachability (does there exist a path such that . . .), safety (something bad never happens or, equivalently, something good is invariantly true) and liveness properties (something will eventually happen) [4].

We used UPPAAL version 4.06 (rev. 2987), released March 2007. The hardware used is a 13.3" MacBook (Intel Core2Duo processor on 2 GHz, 2 GB RAM) running Mac OS X 10.5 Leopard. We ran the verification engine locally, but on the command line since this allows for easily timing the verification and getting informative statistics about the verification. Further, all possible verification settings are the defaults, unless stated otherwise.

An example of a command line verification using UPPAAL:

```
time verifyta -u chain_10p_cm.xml noDeadlock.q
```

where:

- `time` is a UNIX utility for measuring the duration of the execution of a particular command (in this case `verifyta`). When the command finishes, it outputs the real time, user time and system time. We reported the real time measured by `time` in our results.
- `verifyta` is the stand-alone version of the model checking engine of UPPAAL (version as mentioned above).
- `-u` is an option of the verifier to "Show summary after verification (incorrect for liveness properties)". The summary provides information about the number of states stored and explored.
- `chain_10p_cm.xml` is a file that contains the UPPAAL model to be verified.
- `noDeadlock.q` is a property file, that contains one or more requirement specifications in UPPAAL's query language. `verifyta` will check all properties in this file.

4.3.1 Verification Parameters

The verification process depends on a number of parameter values defined in our models as well as some external factors (e.g. different models, topologies, properties). This causes an instance explosion of potential models to verify, since we can vary endlessly with the values of these inputs. This subsection defines the input parameters of our verification process. Since it is infeasible to verify all possible instances, we indicate which instances we verified by defining for which values of the input parameters we did a verification.

The parameters of the verification process are enumerated below, together with the selected values for verification.

1. *Models* – The models to verify. Since protocol model V1 lacks important functionality we verified models V2 and V3.

2. *Properties* – Besides some incidental other properties (on which we report later on), we structurally verified the following properties: deadlock freedom, correct parent selection and correct distance computation.

3. *Number of nodes* – We are interested in boundaries so we start with `N=2` nodes and increase this parameter until the verification becomes infeasible.

4. *Topologies* – The kind of topology, i.e. how the nodes are arranged and connected (node configuration and link qualities). We came up with five interesting topologies which are depicted in figure 4.6.

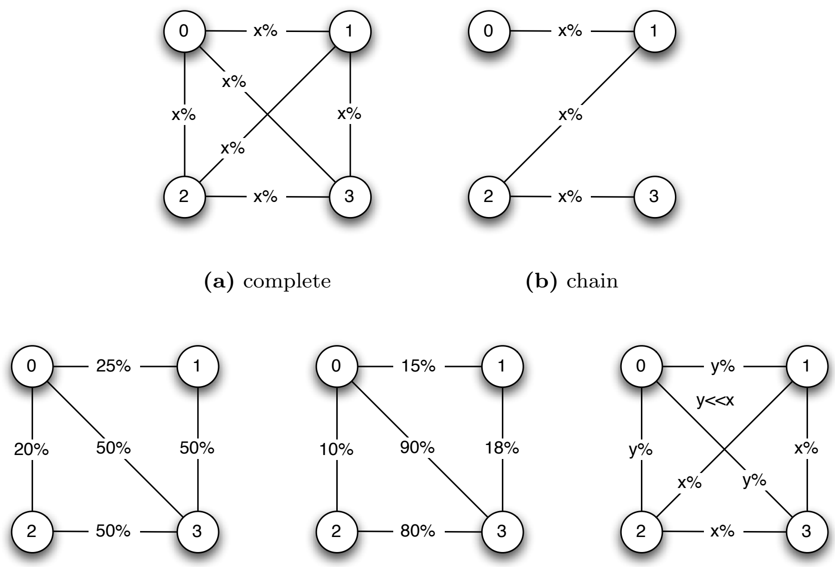
5. *Maximum number of message rounds* – Without the global constant `MAX_M` in our models, verification would never end. The value denotes the maximum number of message rounds to be executed. We verified instances with `MAX_M` equal to 10, 20, 50 and 100.

6. *Precision* – The global constant `ACCURACY` of our models is used to regulate the precision of the distance computation in the receive function, since UPPAAL does not support floating point numbers. We used values 1 and 10 (the former results in integer precision, the latter results in a precision of 1 position behind the decimal point).

7. *Execution order* – As described at the end of the model construction section, fixing the order of execution of nodes within a message round should (in theory) drastically decrease the number of states. We compared verification runs with and without a fixed execution order.

The types of topologies we verified are depicted in figure 4.6 using four nodes. It is rather simple to extend these to another number of nodes, while ensuring that the characteristics of the type are maintained. 4.6(a) corresponds to a complete graph. For n nodes it contains $\frac{1}{2}n(n-1)$ undirected edges since there is a link between each pair of nodes. 4.6(b) is a topology type in which the nodes are connected 'chain-wise' ($n-1$ edges). 4.6(c) and 4.6(d) are specific topologies of 4 nodes in which two SPTs with root node 0 might be found. The latter's link qualities are wider spread and might induce precision problems since the distance computation involves fractionals. Finally, 4.6(e) is a topology in which node 0 (which is typically the gateway) has relatively bad links to the rest of the nodes, which are mutually well-connected.

In the sequel we will provide an overview of the verification results we obtained. The detailed results are contained in appendix B. Here we attempt to



(a) complete **(b)** chain

(c) multiple SPTs **(d)** multiple SPTs **(e)** bad gateway

Figure 4.6: Interesting topologies for verification

describe the results more globally, based on interpretations of these detailed verification results. We point out trends, give explanations and formulate preliminary conclusions.

4.3.2 Deadlock Freedom

A deadlock is a situation in which a complete system model is in a terminal state, although at least one process is not in its local terminal location. A typical deadlock example originated by Dijkstra is called *The Dining Philisophers*: five philosophers (whose lives are all about eating and thinking) are sitting at a round table with a bowl of rice in the middle and a single chopstick in between two neighbouring philosophers. To eat rice, two chopsticks are needed thus at any time, only one of two neighbours can eat. Now a deadlock occurs when all philosophers possess a single chopstick: no one can eat! The solution is a fair ‘eating protocol’ such that all philosophers can eat and think infinitely often [3].

Translated to UPPAAL models, a deadlock is a state of the composed system in which there are no outgoing action transitions, neither from the state itself or any of its delay successors (clock delay). UPPAAL’s query language defines a special formula consisting of the keyword `deadlock`, that is satisfied in a deadlocked state [4].

When verifying a new model it is often a good idea to start with a check for *deadlocks*. This is a good sanity check for the model as deadlocks that are found often indicate model design errors or errors in the system that is modelled: the user is forced to investigate the source of the deadlock. Moreover, all states of the model must be checked in order to give a verdict about deadlock presence or absence: the total number of reachable states of the model is determined.

Our models actually do have a deadlock state (as defined by the UPPAAL tutorial [4]). It is the state in which all nodes reached `MAX_M`. So we specify the property: the only deadlock state that occurs is the one in which all nodes have reached the maximum number of message rounds `MAX_M`:

```
A[] deadlock imply ( forall(i:Node_id) Node(i).M == MAX_M )
```

This can be read as follows: on all paths, it is always the case that if a deadlock state occurs, it is the state in which for all nodes hold that their local message round counter `M` equals `MAX_M`. Note that:

- The phrase *On all paths, it is always the case that...* corresponds to `A[]` and is about every reachable state: the semantics of a composed system in UPPAAL (a network of timed automata) is defined as a transition system. The property must hold in every state on every path in the transition system.
- This property is called a *safety* property, meaning that something bad does never happen or, equivalently: something good is invariantly true.
- UPPAAL allows us to use expressions like `forall()` such that we can easily quantify over all nodes.

Protocol Model V2

Tables B.1 – B.4 in appendix B display the results of the verification runs we performed on protocol model V2, considering this deadlock freedom property. The property is satisfied for each run, i.e. the only deadlock scenario that occurred is the one in which `MAX_M` is reached. The tables show the number of states of the model that the verifier of UPPAAL stored and explored in order to determine this outcome, together with the time it took on our hardware. Tables B.1 and B.2 show the results for the complete topology of figure 4.6(a), with all links 100% and all links 10% respectively. Tables B.3 and B.4 show the results for the chain topology of figure 4.6(b), also with all links 100% and all links 10% respectively.

The fact that the property is satisfied for all runs that ended normally is of course good news and increases our confidence in this model. However, it is much more interesting to look at the relations between the results of different runs.

If we take a close look at table B.1, the following observations can be made:

- For 50 message rounds, verifying 4 nodes results in 1277 states and verifying 8 nodes results in 146117 states of the model. These runs took approximately 0.105 and 26.7 seconds respectively.
- The number of states grows exponentially in the number of nodes.
- The number of states grows proportionally in the number of message rounds.
- Verifying with the global constant `ACCURACY` set to 1 instead of 10 causes a decrease of the number of states: a verification with `N=6` and `MAX_M=50` results in 3753 instead of 11957 states. The difference in the amount of states grows fast in the number of nodes, but is independent of `MAX_M`.
- Fixing execution order results in a tremendous gain in the number of states: an ordered run with `N=8` and `MAX_M=50` results in 401 states instead of 146117 states.
- Results of verification runs with fixed execution order are independent of the value of `ACCURACY`.
- Number of states for fixed execution order $s^O = N \cdot MAX\_M + 1$

The observations about execution order and accuracy indicate that fixing the execution order in this deterministic model results in a rather trivial verification. When fixing execution order, there is only one process that can execute at any time: there is no branching in the state space! It consists just of a sole path of states: an initial state and `MAX_M` times `N` successive states.

Table B.2 shows what happens if the link quality is low (complete topology and all links 10%): the message loss balancing approach combined with the low link qualities causes the number of states to grow exponentially (instead of proportionally in the previous case) for increasing `MAX_M`. Compared to the previous topology, this is what could be expected because many new model states are generated because of lost messages.

Table B.3 displays the results for chain topologies with links of quality 100%. We expected that these verification runs would result in less states of the model compared to the complete 100% case since there are less links. The results however show the contrary: there is a fixed amount of states *more* compared to the completely connected case (of table B.1). The difference in the number of states depends on `N`. This unexpected result is a consequence of our message loss balancing approach: in the complete 100% case, message are never lost. As soon as messages do get lost, additional states are generated. Our expectation about a relation between number of links and number of states is right for the 10% complete and chain cases, as becomes clear from B.4.

Protocol Model V3

Tables B.5 – B.8 in appendix B display the results of the verification runs we performed on our non-deterministic protocol model V3. The property for deadlock freedom is again satisfied for each run, i.e. the only deadlock scenario that occurred is the one in which `MAX_M` is reached. The tables again show the number of states and the time taken by the corresponding verification run, for the same topologies as verified with model V2.

An instance of model V3 contains more processes compared to model V2 for the same topology and number of nodes, because of the presence of the explicit link processes. Therefore we expected that the number of nodes that can be verified using this V3 model would be less. This expectation is confirmed by the results.

We observed that a verification run of model V2 with fixed process execution order is rather trivial, due to the absence of branching in the state space. This is however not the case with model V3, as (besides execution orders) the non-determinism in the link processes causes additional branching. Moreover, the observation that verifying a chain topology results in more model states than verifying the corresponding complete topology also does not hold for model V3. In model V3 this results in less states, as we already expected to be the case with model V2.

4.3.3 Correct Parent and Distance

As we explained in section 2.3, a SPT need not be unique since there may be several paths (from a node to the gateway) of the same distance (total ETX). For particular topologies it is thus possible that a node has several options for parent selection. To check whether all nodes found a correct parent we use a global function `isCorrectParent(i,j)` that returns `true` if the given node index `j` is a correct parent of the given node index `i`. Of course, the correct parents (i.e. node ids for which the function should return `true`) are to be set manually by the user before verification, according to the topology that is to be verified. We can call this function from a correctness property and, by doing so, check for correct parent selection. Note that checking such a property will result in a certain confidence level w.r.t the correct operation of the protocol, but also w.r.t the correctness of the models.

Another way to determine correct operation of the protocol (c.q the model) is to check if the nodes do find their correct distance to the gateway. This way we abstract from which parent (and whether this is a correct one) is chosen. To be able to easily check for the correct distance to the gateway, we make use of a function `isCorrectDistanceToG(i,d)` that returns `true` if the given `d` is the correct dist-to-G of the given node index `i`. Again, the distances that are correct for the topology to be verified are to be set manually by the user before verification. Moreover, the function enables us to specify an upper and a lower epsilon ϵ_l and ϵ_u such that it returns true if the given `d` is between $d_c - \epsilon_l$ and $d_c + \epsilon_u$, where d_c is the shortest distance w.r.t the topology verified. This tolerance is very useful since distances computed by the nodes will fluctuate more or less as the computation is based on perceived link qualities.

Please note that the manually assigned distances depend on the value of global constant `ACCURACY` since floating point numbers are not supported. If the minimum distance for a node in a certain topology for example evaluates to 2.303 then it should be set to 2 for `ACCURACY=1` and to 23 for `ACCURACY=10`.

The source code of these functions can be found in the global declarations of both model V2 (page 159) and V3 (page 161) in appendix B.

We will now describe the properties that make use of these functions. First of all, it is of course interesting to check whether the nodes manage to select a correct parent (or compute the correct distance) at all, during a verification run. We can check this by verifying the following *liveness* properties:

```
A<> forall(i:Node_id) isCorrectParent(i, Node(i).parent)
```

```
A<> forall(i:Node_id) isCorrectDistanceToG(i, Node(i).D[i])
```

This can be read as follows: along all paths (`A`), eventually (`<>`) a state is reached in which for each node holds that its parent is a correct parent (or its computed dist-to-G falls in the given distance interval). In other words: each node eventually selects a correct parent or computes the correct distance (with a certain tolerance). Note that these properties may not hold for models with small values for `MAX_M`, since the protocol needs some time (i.e. message rounds) to sufficiently approximate the link qualities which are needed for distance computation and parent selection. This type of liveness (something good will eventually happen) is called *bounded liveness* since each verification run is artificially terminated after `MAX_M` message rounds have passed by.

The preceding properties indicate whether on each path in the state space a state is reached in which all nodes found a correct parent or distance. If this is the case, we do however not know at which point in time this occurred. Moreover, after such a state on a path, there may be many states in which the property does not hold though. In order to check whether all nodes selected a correct parent[1] at the end of each path (i.e. when all nodes reached MAX_M), the following property can be used.

```
A<> forall(i:Node_id)
    (Node(i).M == MAX_M) && isCorrectParent(i, Node(i).parent)
```

This property says as much as: along all paths eventually a state is reached in which each node selected a correct parent after MAX_M message rounds. The added condition thus enforces that there are no successor states of the states in which the property holds: it is the terminal state of each path. This enables us to experiment with the value of MAX_M and by doing so, to get some insights in the number of message rounds that are needed for the nodes to select correct parents.

Executing experiments with this property requires us to continuously change the model (the value of MAX_M), save the model and re-verify the model. We can improve this by converting the property to an almost equivalent *safety* property and subsequently use a variant of it. In words, the safety property says: along all paths it is always the case (i.e. in each state) that whenever all nodes reached MAX_M, they also all selected a correct parent.

```
A[] (forall(i:Node_id) Node(i).M == MAX_M) imply
    (forall(j:Node_id) isCorrectParent(j, Node(j).parent))
```

This property is almost equivalent to the preceding liveness property, except for the fact that in this case the property does not ensure that there are states in which each node reached MAX_M at all (as a result of the implication). A general difference between checking liveness properties and safety properties is that for the latter always the entire state space must be explored, while a positive answer to the former may be found after exploring it only partially.

A useful variant of the preceding safety property is the following:

```
A[] (forall(i:Node_id) Node(i).M >= x) imply
    (forall(j:Node_id) isCorrectParent(j, Node(j).parent))
```

[1] In the sequel of this discussion we only mention parent checking: we omit the properties for checking the correct distance, since they can be easily obtained by substituting the distance checking function for the parent checking function.

This property allows us to determine a *threshold* message round number M_T: along all paths it is always the case that if all nodes reached message round `x` (i.e. M_T), they all selected a correct parent and keep selecting a correct parent up to message round `MAX_M`.
As said, the properties for correct distance checking are similar but use the distance function. All properties are contained and numbered in section B.5.2 together with detailed results of verifying them. Verification was mostly done from the GUI, since it has a useful feature that allows to import an error trace into the simulator such that it can be investigated manually.

Verification Results

We verified model V2 and model V3 for three 4-node topologies. First we verified a rather arbitrary topology that can be obtained by substituting 33% for 18% in topology 4.6(d) on page 57. Second, we verified topology 4.6(c) which has two SPTs and simple link qualities. Third, we verified 4.6(d), which is also a topology with two SPTs, but with more complex link qualities resulting in fractions in distance computation. We refer to B.5.2 for the detailed results. Here we will shortly discuss the global results.

First of all, an important result is that the protocol finds correct parents and distances for each node. Of course this is only guaranteed for the topologies we verified, but it results in a higher level of confidence in the protocol and in both of our models V2 and V3. Moreover, the fact that multiple SPTs are present in the second and third topology does not really influence the verification results: the protocol handles this well.

The most interesting results are the ones obtained from checking for a message round threshold: if we compare the results of parent checking and distance checking, we see that the former requires less resources and is thus to prefer (tables B.13 shows that, for the single-SPT topology, only 10 message rounds are needed to select correct parents, whereas B.14 shows that as much as 128 rounds are needed to compute the correct distance with tolerance 1). This can be explained as follows: checking for correct distances requires a more accurate measurement of the link quality and thus takes more message rounds. For parent selection the distance computation can be less accurate.

Another important result is also established using the properties for determining a message round threshold, namely that the protocol stabilizes and not continuously builds a different tree. For example, with `MAX_M=50` and `ACCURACY=10`, we find that from 10 message rounds on, the protocol keeps selecting the correct parent in the single-SPT topology, up to message round 50. This indicates that the protocol found and maintained a stable SPT after ten message rounds until the end of the verification run.

Property 8, that checks the message round threshold for correct distances, shows that some fluctuation in distance computation occurs: the property cannot be satisfied for $\epsilon_l = \epsilon_u = 0$ in topologies 1 and 3. A simulation run did learn us that this is a result of the computation of the dist-to-G of node 2: the result keeps fluctuating between 23 and 24 (it should be $13 + 11 = 24$), even for values of `MAX_M` of 5000! This is caused by the 80% link, which induces a distance (ETX) of $\frac{1000}{80} = 12.5$. Apparently, the protocol (probably combined with the rounding) causes this to evaluate sometimes to 12 rather than 13. This is not a severe problem since parent selection is not influenced in this case. However, there might be topologies in which it will form a problem (e.g. topologies with many of such links).

As a final global result, we see that model V2 and V3 perform well in both parent selection and distance computation. A disadvantage of model V2 is that link probabilities are modelled rather predictable, in contrast to real probabilistic behaviour. Model V3 improves this using non-determinism, but at the price of more processes (resulting in less feasible verification runs).

4.3.4 Verification Cluster

The faculty Electrical Engineering, Mathematics and Computer Science (EEMCS) of the University of Twente owns a verification cluster consisting of several machines equipped with powerful hardware. As an attempt to move some boundaries in feasible verification, we did some verification runs on the server named "BIG1" in this cluster. It is equipped with Intel's Dual Quad Core Xeon 3.00GHz (64bit) with 1333FSB, 64GB RAM memory and 2 harddisks of 160GB.

Unfortunately, there is no distributed version of UPPAAL available to run on our cluster. Moreover, there isn't also a 64bit version of UPPAAL. Since 32bit UPPAAL can only address $2^{32} = 4$GB of the server's total memory amount of 64GB. We therefore did not expect great performance gain. The hardware on this verification server is however more powerful than our own hardware so we did a few verification runs to document the difference. We again used the stand-alone version of UPPAAL's verification engine `verifyta`.

The table B.2 on page 168 contains an entry saying '*oom*', which means 'out of memory'. The run corresponding to this entry was verified on BIG1 and reached the 4GB limit, which was indicated by an out of memory message. The two italicized rows of table B.4 contain results of verification runs that were also verified on server BIG1. A run (`N=4`, `MAX_M=50`) which is infeasible on our normal hardware was actually feasible on the verification server, it took about 15 minutes to verify over 26 million states. The same run, but with `ACCURACY` set to 1, was actually feasible on our hardware and took about 253 seconds to verify 4,33 million states. The same run on server BIG1 took

144 seconds. We gained 109 seconds, that is 43%. This is considerable but still not enough: the number of states of our models grows too fast, causing the 4GB memory limit to be reached. Most of the time, verification on BIG1 thus results in waiting to see an out of memory message. Therefore we did not do more verification runs on the server.

4.4 Conclusions

In this section we will summarize our experiences, problems and results encountered in this chapter. We will first enumerate our experiences with UPPAAL and its usability. Next, a number of conclusions about the UPPAAL models are enumerated together with the main results of their verification. We finalize this chapter with some concluding remarks about the protocol.

4.4.1 UPPAAL Experiences

Compared to many other academic tools, the quality of Uppsala-Aalborg Model Checker (UPPAAL) is relatively high. Its look and feel together with the completeness and documentation give a rather robust impression. It is relatively easy for a beginner to get started due to the intuitive GUI and the very complete help function. The underlying theory of timed automata is rather complex but only few knowledge of it is required. Below we enumerated our findings about UPPAAL for model construction, simulation and verification.

Model Construction

- An UPPAAL model consists of global declarations, process templates with local declarations and a system definition.
- The use of parameterizable process templates allows for creation of many similar processes and for flexibility in process instantiation (using parameters and even partial instantiation).
- UPPAAL's modelling language is a quite expressive C++/Java-like language with powerful user-defined functions.
- Only integer and Boolean data types are supported, floating point numbers are not supported.
- It is rather difficult to model probabilistic behaviour.
- Message broadcast is easily modelled using the powerful and very useful language concept of broadcast synchronization channels.
- UPPAAL comes with a syntax checker that generates useful error descriptions.

- Models are saved in well-defined XML files (supporting potentially easy portability).

Simulation

- UPPAAL contains a powerful simulator that allows for validation of the model's behaviour.
- Random simulation with variable speed is supported.
- Generated simulation traces can be saved to and loaded from files.
- The simulator displays all or a selected set of variable values per system state.
- System states are also visually represented.
- Error traces of the verifier can be imported in the simulator for detailed examination.
- Message sequence charts graphically show the sequence of transitions.

Verification

- UPPAAL's Java GUI is connected to its model checking engine via TCP/IP (implementing a client-server architecture).
- Correctness properties are specified in a simplified version of CTL (no nesting of path formulæ).
- There is no 64-bit version of UPPAAL's verifier available.
- There is no distributed version of UPPAAL's verifier available, although the developers do mention[2] the existence of a distributed version on a cluster of Aalborg University.
- The GUI does not report on additional verification statistics after a verification run (such as the number of states, time elapsed etc.).
- The stand-alone command line version of the verifier `verifyta` does provide an option (`-u`) to show a (short) summary of verification statistics, and allows for timing the verification.

UPPAAL is a real-time model checker but we did not fully exploit its timing functionality. In fact we just misused a local clock variable since its value only alternates between 0 and 1.

[2] http://tech.groups.yahoo.com/group/uppaal/messages

4.4.2 The Models

The enumeration below contains observations and conclusions about our UPPAAL models:

- The concept of unreliable links having a certain link quality is not easily modelled in the UPPAAL modelling language. In model V2 the probabilism is imitated by a highly predictable message loss balancing approach, whereas in model V3 we used a less predictable non-deterministic approach.
- Since floating point numbers are not supported, link qualities were specified as percentages. A global constant `ACCURACY` was introduced to obtain a more precise distance computation.
- The asynchronous, continuous-time protocol was modelled as a synchronous, discrete-time system in an attempt to keep the state space as small as possible. We thus implicitly assumed absence of clock drift.

Results and conclusions about the verification of these models are collected below:

- Both models are deadlock free, besides the terminal state in which all nodes reached `MAX_M`.
- In both models, correct parents are selected (w.r.t. the SPT rooted at the gateway) for the verified topologies.
- In both models, correct distances to the gateway are computed (w.r.t. the SPT rooted at the gateway and w.r.t. certain tolerance bounds) for the verified topologies.
- The number of nodes that can be verified within reasonable time depends heavily on the topology under consideration and the link qualities therein. Comparing all results, we see that verifying a complete topology with 10% links is possible up to 4 nodes. This is however a rather difficult topo.
- A problem with verifying network protocols is the instance explosion of topologies and the infeasibility to verify all possibilities. A possible solution is to design a model that abstracts away from specific topologies [10].
- Decreasing the value of the global constant `ACCURACY` reduces the number of generated states at the price of a less accurate distance computation.

- Verification using more powerful hardware does not result in a significant gain in time.
- Using model V2 to verify the protocol results in a smaller state space compared to using model V3 for the same topology, since model V3 requires more parallel processes. Model V2 however uses a very poor and predictable mechanism for simulating link qualities.
- Checking for correct parent selection is preferable above checking for correct distance, since it requires less resources: a parent can be selected correctly without exactly computed distances. Therefore it is often the case that all nodes found a correct parent earlier then they all found the correct distance.

4.4.3 The SPT Protocol

We did not found severe errors in the SPT protocol, which increases confidence in its correctness. It does however not prove its correctness since we can only verify the protocol for certain interesting topologies, not for all possible ones.

Specification and modelling of the protocol resulted in an improvement of the protocol and its description (chapter 3). Originally, the passive thread was specified to save link qualities instead of just numbers of messages per node. This is however redundant information and causes unnecessary overhead: just counting the number of messages received from other nodes suffices.

Our verification process showed that there may be fluctuation in distance computation of the protocol. This often does not form a problem since it fluctuates around the correct value. If the fluctuation is relatively high and all link qualities are near to each other, this may influence the parent selection process. In that case errors might occur.

Note that we did not model any form of message collision. That is because the protocol does not account for collision. The philosophy is that it is inherent to WSNs that messages get lost, independent of the source of the message loss.

CHAPTER 5

Verification using SPIN

This chapter describes our modelling and verification activities using the model checker SPIN. The structure of this chapter is similar to the previous chapter: we will first introduce the reader to the tool and its underlying theory and motivate why it has been chosen. Then we elaborate on our final SPIN model and subsequently we report on the verification experiences and results. Finally, we summarize our findings in the concluding section.

The previous chapter extensively introduced many of the issues encountered when modelling and verifying the SPT protocol (e.g. modelling broadcast, execution order, accuracy of distance computation). We refer to these if necessary and focus here on modelling and verification with SPIN.

5.1 Tool Introduction

The Simple Promela Interpreter (SPIN) is an efficient model checker for distributed software systems, essentially a command line tool, written in American National Standards Institute (ANSI) standard C. It is developed mainly by Gerard J. Holzmann at Bell Labs in the eighties and nineties and still continues to evolve [22]. SPIN's first version was released in 1991, accompanying the first book about it, written by Holzmann [23]. In April 2002, the prestigious Software System Award 2001 of the Association for Computing Machinery (ACM) went to SPIN, assigning it the status of breakthrough software system (as UNIX, TeX, SmallTalk and TCP/IP). The current official release is SPIN 5.1.7 (Dec. 23, 2008), with default graphical front-end XSpin 5.1.0 (Oct. 9, 2008). It is freely available at http://spinroot.com.

5.1.1 Underlying Theory

The underlying theory is a variation of the theory of finite automata, known as the theory of *ω-automata* [22]. In general, a SPIN model is the parallel composition of a number of ω-automata. An *ω-automaton* is a finite state automaton with the acceptance conditions not only covering finite executions, but also infinite ones.

A *finite state automaton* is a quintuple (S, s_0, L, T, F) with a finite set of states S, a distinguished initial state $s_0 \in S$, a finite set of labels L, a set $T \subseteq (S \times L \times S)$ of transitions between states and a set $F \subseteq S$ of *final* states. A *run* of such an automaton is an ordered set of adjacent transitions. A run is called *accepting* if it is *finite* and its final transition ends in a state in F (i.e. is a *final* or *accepting* state). An ω-automaton has another acceptance condition: an *infinite run* or ω-run is called *accepting* if and only if some state in F is visited infinitely often (Büchi Acceptance). Additionally, finite runs are extended with infinitely many self loop transitions (at their final state) with label ε (called the null action which is added to the set of labels). This is called *stutter extension* and automata with Büchi acceptance conditions are called *Büchi Automata* [22].

The language to specify input models for SPIN in, is called Process Meta Language (PROMELA), which is a description language rather than an implementation language: the emphasis is on process modelling, not on computation. The building blocks are asynchronous processes, buffered and unbuffered message channels, synchronizing statements and structured data [22]. To specify correctness properties (or *correctness claims* in proper SPIN terminology), PROMELA provides basic assertions, several state labels and never claims. A *never claim* is a special type of PROMELA process describing undesired behaviour and it thus should never reach the end of its body. It may be written by hand or it may be automatically generated from Linear Temporal Logic (LTL) formulæ. LTL is a temporal logic like CTL (of which a subset is used by UPPAAL). The difference is that in the LTL view, time is linear: at each moment in time there is a single successor moment. CTL considers time as a branching, tree-like structure [3]. SPIN does not support full LTL: the temporal operators *next* (X) and *weak until*[1](U) are not supported [22].

We refer to the literature for further details about and explanation of the model specification language PROMELA, syntax and semantics of Büchi Automata, LTL, never claim generation, etcetera (for example [3, 22, 53]).

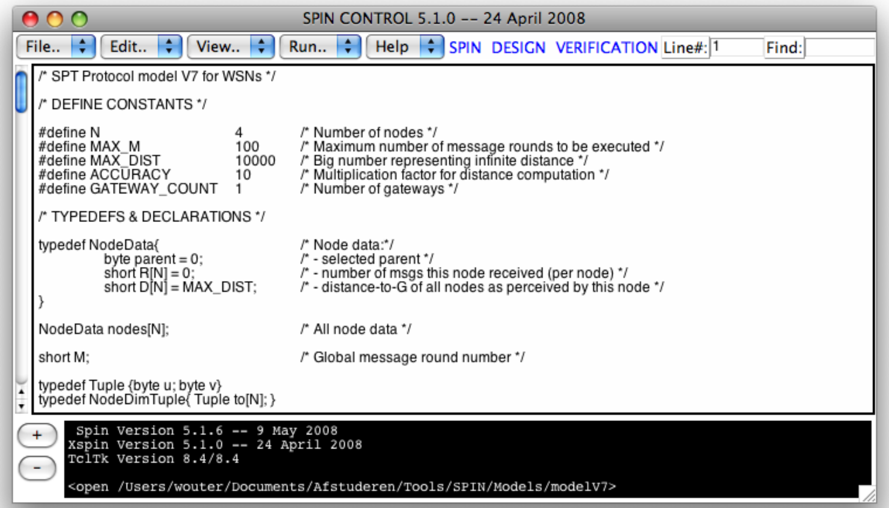
[1]The *weak until* differs from the *strong until* operator in that the former does not require sub formula q in p U q to become true, in contrast to the latter, which actually does define that q eventually becomes true.

5.1.2 Tool Motivation

We started the modelling and verification of our protocol using the real-time model checker UPPAAL (ch. 4). It turned out however, that we did not exploit the concept of time (we rather misused a clock in the node template to establish message rounds). UPPAAL is optimized for real-time model checking (e.g. optimized data structures for clock constraints, symbolic states, zones, regions). In our discrete-time model, these techniques may however cause much overhead, since we hardly exploit timing. Therefore we decided to try another state-of-the-art tool, specialized in distributed systems, that is, SPIN.

Moreover, SPIN is a mature tool that has been used successfully in numerous case studies. According to the SPIN site, examples include verification of the control algorithm for the flood control barrier near Rotterdam in the Netherlands (in the late nineties), logic verification of data and phone switch software of Lucent Technologies and even mission critical software: selected algorithms for a number of space missions were verified using SPIN.

We used SPIN version 5.1.6, dated May 9, 2008 and XSpin version 5.1.0 (April 24, 2008). Figure 5.1 depicts SPIN's GUI XSpin. The hardware used is a 13.3" MacBook (Intel Core2Duo processor on 2 GHz, 2 GB RAM) running Mac OS X 10.5 Leopard.



```
/* SPT Protocol model V7 for WSNs */

/* DEFINE CONSTANTS */

#define N               4       /* Number of nodes */
#define MAX_M           100     /* Maximum number of message rounds to be executed */
#define MAX_DIST        10000   /* Big number representing infinite distance */
#define ACCURACY        10      /* Multiplication factor for distance computation */
#define GATEWAY_COUNT   1       /* Number of gateways */

/* TYPEDEFS & DECLARATIONS */

typedef NodeData{               /* Node data:*/
        byte parent = 0;        /* - selected parent */
        short R[N] = 0;         /* - number of msgs this node received (per node) */
        short D[N] = MAX_DIST;  /* - distance-to-G of all nodes as perceived by this node */
}

NodeData nodes[N];              /* All node data */

short M;                        /* Global message round number */

typedef Tuple {byte u; byte v}
typedef NodeDimTuple{ Tuple to[N]; }
```

```
 Spin Version 5.1.6 -- 9 May 2008
Xspin Version 5.1.0 -- 24 April 2008
TclTk Version 8.4/8.4

<open /Users/wouter/Documents/Afstuderen/Tools/SPIN/Models/modelV7>
```

Figure 5.1: The main screen of the GUI XSpin 5.1.0 of the SPIN model checker

5.2 Model Construction

This section discusses our SPIN model of the SPT protocol. In contrast to the previous chapter, the final model is presented at once and we highlight the interesting aspects. First, we explain some terminology about modelling in PROMELA.

A model for SPIN is written in the specification language PROMELA and basically consists of data types, message channels and process types called **proctype**s (the textual equivalent of templates in UPPAAL). A **proctype** body consists of one or more data declarations and one or more statements. The notion of *executability* of statements is special and is explained below using our model. Processes are instantiated from process types which results in a model of concurrent processes. Each **proctype** instance has a unique predefined local variable called _pid. Processes communicate through the use of global variables or message channels that are either buffered (i.e. asynchronous communication) or unbuffered (i.e. synchronous communication, also known as handshake or rendezvous communication). The entire SPIN model is textually defined in PROMELA in a single file.

5.2.1 Protocol Model

We constructed a SPIN model according to the underlying ideas of the explicit non-deterministic links in UPPAAL model V3: u out of v messages are delivered, which ones is decided non-deterministically. To keep things manageable, the model file is split into three parts, presented in listings 5.1 – 5.3. In the sequel, we walk through these listings and point out the interesting aspects.

The first part of our PROMELA model, printed in listing 5.1, contains declarations of global constants and variables, as well as the necessary type definitions. The global constants[2] may look familiar since the same constants were used in the UPPAAL models. The type definition of type NodeData (lines 11–15) defines the data maintained by each node, which comprises a parent field for the currently selected parent, an array of received messages counters and an array for the dist-to-G of each node. Note the convenient way of initializing all entries of the distance array to the value of constant MAX_DIST (l. 14). Line 17 shows the declaration of a global array of this type NodeData of length N, which represents all node data. We explicitly decided to declare this data globally since local data (local to processes) cannot be accessed from verification properties. This also made us choose for a global message round variable M, to count the number of passed message rounds.

[2]In fact, constants are *pre-processor macros*: each occurrence in the code is replaced by its defined value.

```
 1  /* SPT Protocol model for WSNs − W.M. Everse */
 2
 3  /* DEFINE CONSTANTS */
 4  #define N              4              /* Number of nodes */
 5  #define MAX_M          100            /* Max number of msg rounds */
 6  #define MAX_DIST       10000          /* Represents 'infinite' distance */
 7  #define ACCURACY       10             /* Multiplication factor */
 8  #define GATEWAY_COUNT  1              /* Number of gateways */
 9
10  /* TYPEDEFS & DECLARATIONS */
11  typedef NodeData{                     /* Node data:*/
12      byte parent = 0;                  /* − selected parent */
13      short R[N] = 0;                   /* − counts received msgs (per node) */
14      short D[N] = MAX_DIST;            /* − dist−to−G (per node) */
15  }
16
17  NodeData nodes[N];                    /* All node data */
18
19  short M;                              /* Global msg round number */
20
21  typedef Tuple {byte u; byte v}
22  typedef NodeDimTuple{ Tuple to[N]; }
23  hidden NodeDimTuple C[N];             /* Connectivity Matrix, NxN */
24  hidden NodeDimTuple H[N];             /* History Matrix, NxN */
25
26  typedef NodeDimBool{ bool to[N]; }
27  hidden NodeDimBool Msgs[N];           /* Message Exchange Matrix, NxN */
28
29  byte ctrl = N;                        /* Used for transferring control */
```

Listing 5.1: PROMELA Model - Constants, Typedefs and Global Declarations

The model uses a connectivity matrix C and a history matrix H: both N×N matrices. There is however no data type in PROMELA to directly define a matrix and arrays of arrays are not supported. Lines 21–24 provide a work around: a 2-dimensional array (i.e. a matrix) is defined as an array of type NodeDimTuple, that on its turn is defined to be an array of tuples, using the **typedef** construct twice. The tuples (containing bytes u and v) are the entries of matrix C (and H) and correspond to the parameters u and v of the generalized link template of UPPAAL model V3: u *out of* v *messages come across*. History matrix H keeps track of the number of messages that came across so far. We return to the use of these matrices later on. The keyword **hidden** prevents SPIN from taking these data into account in the global system states, similar to UPPAAL's keyword **meta**.

Compared to UPPAAL, modelling message broadcast is less easy using SPIN, since it does not support broadcast channels. Ruys [50] (section 4.11) describes three approaches to modelling multicast or broadcast protocols for SPIN: a communication bus, a matrix of channels or a dedicated broadcast service. Nevertheless, we introduce a fourth approach, using yet another N×N matrix called Msgs (l. 27). Each message round, its entries indicate whether the corresponding nodes will receive a probe message. For example: at the beginning of a message round, an entry Msgs[1].to[2] of this matrix is set to 1 (i.e. true) if node 2 should receive a message from node 1 that message round. We will learn from listing 5.3 below which process is responsible for updating matrix Msgs.

```
/*INLINE DECLARATIONS */
inline isGateway(id){                      /* Check if id is a gateway */
    (id < GATEWAY_COUNT);
}

inline receive(id){                        /* Update counters if received a msg */
    atomic{
        do
        :: (k < N) && (Msgs[k].to[id]) ->
                nodes[id].R[k]++;
                if
                :: isGateway(k) -> nodes[id].D[k] = 0    /* dist-to-G of G is always 0 */
                :: else -> nodes[id].D[k]=nodes[k].D[k]
                fi;
                k++
        :: (k < N) && !(Msgs[k].to[id]) -> k++
        :: (k==N) -> k=0; break
        od
    }
}

inline getMinimum(id){                     /* Compute minimum distance */
    atomic{
        minval = MAX_DIST;
        do
        :: (k < N) ->
            if
            :: (nodes[id].R[k] > 0) && (k != id) && (nodes[id].D[k] < MAX_DIST) ->
                try = (ACCURACY * M / nodes[id].R[k]) + nodes[id].D[k];
                try = ( ((ACCURACY*M)%nodes[id].R[k])>=(nodes[id].R[k]/2)->(try+1):try );
                if
                :: (try <= minval) -> minval = try; nodes[id].parent = k
                :: else -> skip
                fi
            :: else -> skip
            fi;
            k++
        :: (k == N) -> k=0; try=0; nodes[id].D[id] = minval; break
        od;
    }
}

inline setC(x, y, p, q){                   /* Sets entries in connectivity matrix C */
    C[x].to[y].u = p;
    C[x].to[y].v = q;
    C[y].to[x].u = p;                      /* Due to symmetry */
    C[y].to[x].v = q;                      /* Due to symmetry */
}

inline canLoose(a,b){                      /* Check if msgs still may be lost */
    (C[a].to[b].u - H[a].to[b].u) < (C[a].to[b].v - H[a].to[b].v)
}

inline canSend(c,d){                       /* Check if msgs still may be sent */
    H[c].to[d].u < C[c].to[d].u
}

inline checkReset(e,f){                    /* Reset counters when needed */
    if                                     /* (modulo C[].to[].v) */
    :: (H[e].to[f].v == C[e].to[f].v) -> H[e].to[f].u = 0; H[e].to[f].v = 0;
    :: else -> skip
    fi
}
```

Listing 5.2: PROMELA Model - Inline Constructs

The next part of our model is printed in listing 5.2. It contains all so-called **inline** constructs. An **inline** construct defines a replacement text for a symbolic name. It is defined globally, i.e. on **proctype** level, and is called like a C procedure. An **inline** construct can have parameters but it cannot return a value to the caller and it does not define a new variable scope. Each invocation is replaced by the body of the corresponding **inline** (with actual parameters), much like the pre-processor macros used to define constants.

Listing 5.2 contains seven **inline** definitions. They are enumerated below together with the behaviour they specify. It can be observed that large part of this behaviour corresponds to behaviour that we also specified in the UPPAAL model. The use of these **inline** constructs will become clear as soon as we move to the discussion of the third part of the model: the processes. For now, we just summarize their behaviour below:

1. isGateway(id) – determines whether the given id is a gateway node id.
2. receive (id) – handles the receive action of node id: all probe messages of all other nodes in a message round are received at once (delivery in a certain message round is defined in matrix Msgs).
3. getMinimum(id) – computes the minimum dist-to-G of node id and sets the corresponding parent.
4. setC(x, y, p, q) – sets entries of the connectivity matrix C. The first two parameters denote the node ids and the last two indicate the link quality between these nodes: p out of q messages are delivered. Note that symmetry of C is ensured.
5. canLoose(a, b) – checks if messages still may be lost according to the connectivity matrix C and the history matrix H. This corresponds to the guard on the lower branch of the generalized link template of UPPAAL model V3 (page 52).
6. canSend(c, d) – checks if messages still may be sent according to the connectivity matrix C and the history matrix H. This corresponds to the guard on the upper branch of the generalized link template of UPPAAL model V3 (p. 52).
7. checkReset(e, f) – resets the values in the history matrix when needed. This corresponds to the local function checkReset() of the generalized link template of UPPAAL model V3 (p. 52).

The third and last part of our PROMELA model is printed in listing 5.3. It comprises the process type declarations and instantiations that specify the behaviour of the model. The process instances use the global variables and **inline** constructs that we showed before. There are two process types in our model: node() at lines 94–103 and globalSend() at lines 105–142, both without parameters. In fact, lines 94 and 105 not only define a **proctype**, but they also instantiate it immediately. This is indicated by the keyword **active**. Line 94 instantiates N processes of **proctype** node() with _pid =0,1,2...(N−1). Line 105 results in exactly one instance of the **proctype** globalSend(), with _pid=N. Before illustrating how these N+1 processes behave and interact, we first need to explain the concept of *executability*.

```
 93  /* PROCESS DECLARATIONS */
 94  active[N] proctype node(){                          /* Node process */
 95   endN:  do
 96      :: ctrl == _pid ->                              /* Wait for control */
 97          byte k;                                     /* Used in inline constructs */
 98          short try,minval;                           /* Used in inline construct */
 99          getMinimum(_pid);                           /* Compute minimum dist-to-G */
100          receive(_pid);                              /* 'receive' from all other nodes */
101          ctrl++                                      /* Transfer control to next process */
102      od
103  }
104
105  active proctype globalSend(){                       /* Simulates sending globally */
106      byte i,j;
107
108      d_step{                                         /* Fill symmetric connectivity matrix */
109                                                      /* Using inline setC(from, to, u, v) */
110          /*arbitrary topo, unique SPT [30/18] */
111          setC(0,1,1,7); setC(0,2,1,10); setC(0,3,9,10);
112          setC(1,3,3,10);
113          setC(2,3,4,5)
114      }
115
116      do                                              /* For each msg round */
117      :: (M < MAX_M) -> atomic{
118          do                                          /* Fill msgs matrix */
119          :: (i<N) -> do
120              :: (j < N && C[i].to[j].u == 0) -> j++  /* No link between i and j */
121              :: (j < N && C[i].to[j].u != 0) ->      /* There is a link between i and j */
122                  if
123                  :: canLoose(i,j) ->                 /* ND-choice: write 0 in msgs matrix */
124                      Msgs[i].to[j] = 0               /* 0 means msg gets lost */
125                  :: canSend(i,j) ->                  /* ND-choice: write 1 in msgs matrix */
126                      Msgs[i].to[j] = 1;              /* 1 means msg will come across */
127                      H[i].to[j].u++                  /* Counts nr of delivered msgs */
128                  fi;
129                  H[i].to[j].v++;                     /* Counts msg opportunities */
130                  checkReset(i,j);
131                  j++
132              :: (j==N) -> j=0; break
133              od;
134              i++
135          :: (i==N) -> i=0; break
136          od;
137          ctrl = GATEWAY_COUNT;}                      /* Transfer control to 1st node process */
138          ctrl == _pid;                               /* Wait for control */
139          M++
140      :: (M == MAX_M) -> break                        /* Max nr of msg rounds reached, stop */
141      od
142  }
```

Listing 5.3: PROMELA Model - Process Declarations

Executability

As said before, the notion of *executability* is special in PROMELA. It is the basic means in the language for modelling process synchronizations [22]. Any PROMELA statement is either *executable* or *blocked*, depending on the state of the composed system. Assignment and print statements are always *executable*. Expressions can be used as statements (for instance at line 138 in lst. 5.3) and are executable (i.e. passable) if and only if they evaluate to the boolean value *true* (which is equivalent to a non-zero integer value). A process that reaches a point in its body where it has no executable statements left, simply blocks.

At any point in time during execution of our model, there is always precisely one non-blocked (i.e. executable) process. This means that the order of process execution in the model is fixed. This is achieved using global variable ctrl which holds a _pid of the non-blocked process. It is initialized with the value of N, resulting in process globalSend() to be the sole *executable* process in the initial state of the composed model. Each node process in the system passes control to the next process by increasing this control variable at the end of its body (line 101).

An interesting PROMELA construct in the light of executability is the **atomic** construct. It is a so-called compound statement: a sequence of statements that is defined to be executable if and only if its *guard* (i.e. the first statement of the sequence) is executable. The semantics of the **atomic** sequence defines that once it starts executing (i.e. with the first statement of the sequence), *all* steps in the sequence will complete, before any other process is given the chance to execute (in the interleaving of process executions in the composed system). Non-determinism is allowed, but if any statement inside an atomic sequence happens to be unexecutable, the atomicity token is lost ('the atomic chain is broken') and another process can take over control [22].

proctype globalSend()

The single instance of process type globalSend() is considered the main process of our model: model execution starts with execution of this process. First it initializes the connectivity matrix C using **inline** setC() (l. 108–114). The construct **d_step** is very similar to the **atomic** construct and defines an indivisible sequence of actions. It is however executed in a *single* deterministic step and all of its statements (except for the guard) may explicitly *not* be unexecutable, otherwise an error will occur. This avoids the generation and exploration of the intermediate states. The topology defined by lines 108–114 is depicted in figure 5.2.

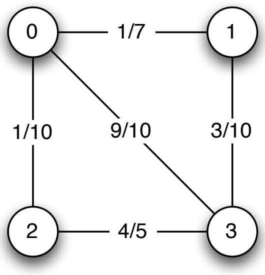


Figure 5.2: 4-node topology specified by lines 108 –114 of listing 5.3

The process continues by entering a loop that simulates the broadcast of a probe message for *all* nodes in the network *at once*, per message round. In other words: every message round (i.e. every loop), it fills the Msgs matrix with 0's and 1's (lines 122–128), transfers control to the first non-gateway node process (line 137), and waits until all non-gateway node processes have executed their body (line 138). An entry in the Msgs matrix Msgs[i].to[j]=1 (or 0) indicates a successful (or unsuccessful) transmission of a probe message from node i to node j. The choice between a 0 and a 1 is made non-deterministically, based on the connectivity and history matrices (similar to the approach in the non-deterministic link template of UPPAAL model V3 on page 52). Lines 122 – 128 show the PROMELA **if** construct[3] that realizes this choice, using **inline** definitions canLoose(i,j) and canSend(i,j) as guards of its options.

proctype Node()

The broadcast functionality specified in the active thread of the nodes (page 36) is implemented by globalSend(), described above. The remaining node functionality (dist-to-G computation and message reception) is implemented in **proctype** Node().

Initially, all N node processes are blocked due to the unexecutable guard expression ctrl==_pid. The processes that represent gateway nodes (for now only the process with _pid=0) will always remain blocked since, in our protocol, gateways do not have additional functionality. The first non-gateway node is enabled as soon as the globalSend() process transfers control to it in line 137. This node process then first 'invokes' **inline** getMinimum(id), then **inline** receive(id) and finally it increases ctrl to transfer control to the next node process. It then becomes blocked again.

Distance computation and parent selection is handled by getMinimum(id). This **inline** construct is very similar to the local node function getMinimum() in our UPPAAL models: based on the received probe messages of other nodes (containing perceived dist-to-Gs), it computes its shortest distance to the gateway and selects the corresponding parent. The rather awkward code (l. 51–70 in lst. 5.2) illustrates that PROMELA focusses on model description instead of computation.

What remains is how a node 'receives' probe messages, i.e. what are the contents of **inline** receive(id)? Lines 35–49 in lst. 5.2 provide the answer. The receive(id) construct walks over the *relevant* column of the Msgs matrix (which is filled before by globalSend()). The relevant column is the one that contains information about whether the current node _pid=id received messages this message round (the 'to'-column corresponding to _pid). For each received probe message (a '1' in its column), the node updates its 'received

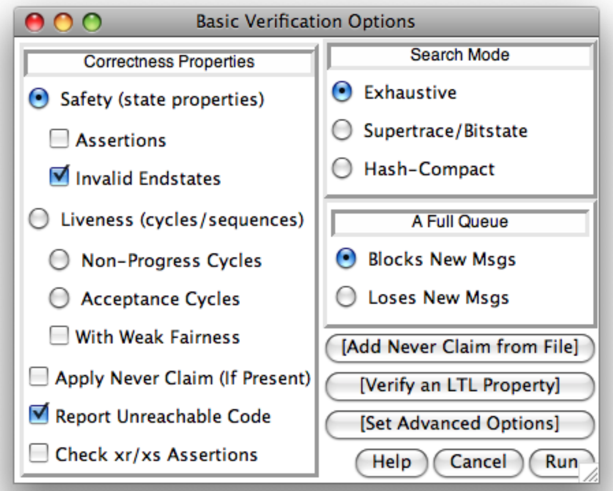
[3]A PROMELA **if** construct can have multiple options (with guards) and it is blocked if none of its options are executable. An option is executable if its guard is executable. If multiple options are executable, a choice is made non-deterministically by SPIN.

messages counter' for the corresponding sender (line 39) and it records the distance to the gateway of the sender (lines 40–43). In our SPIN model, message sending is thus simulated by just copying global data based on a non-deterministic choice of whether a message is received or not.

During model construction, we simulated our model to validate our ideas. In the appendix about SPIN, that is appendix C, we devoted a section to the simulation process using SPIN and XSpin. It also contains an example of the (satisfying) simulation results.

5.3 Verification

This section describes the verification process using SPIN. We verified the model described in the previous section using correctness properties that are very similar to the properties that we verified using UPPAAL. The detailed results are contained in appendix C. Here we provide a global overview of the results, point out noteworthy results and discuss them.



Basic Verification Options

Correctness Properties
- [✓ Safety (state properties)]
  - Assertions
  - [✓ Invalid Endstates]
- Liveness (cycles/sequences)
  - Non-Progress Cycles
  - Acceptance Cycles
  - With Weak Fairness
- Apply Never Claim (If Present)
- [✓ Report Unreachable Code]
- Check xr/xs Assertions

Search Mode
- [✓ Exhaustive]
- Supertrace/Bitstate
- Hash-Compact

A Full Queue
- [✓ Blocks New Msgs]
- Loses New Msgs

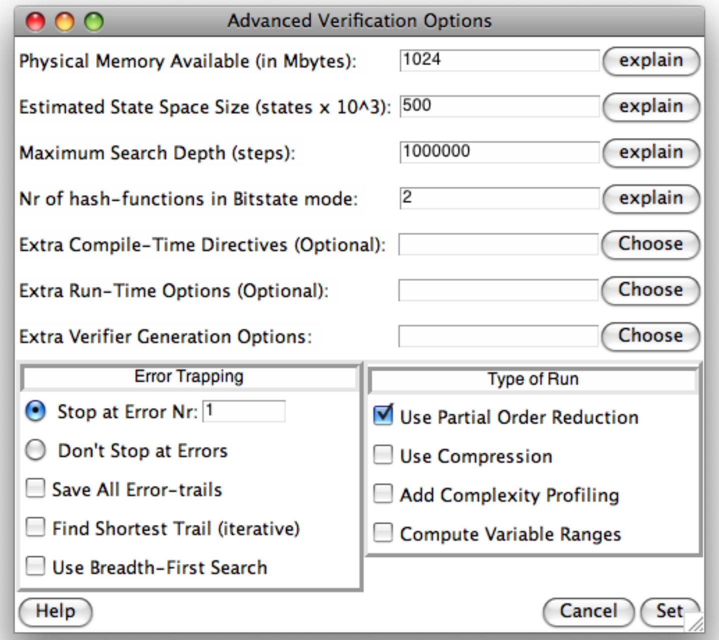
Figure 5.3: The Basic Verification Options Panel

Verification using SPIN and the XSpin GUI works as follows. The file containing the PROMELA source of the model to be verified is opened in the editor. XSpin creates a copy of the source in a temporary file called pan_in[4], in order to prevent overwriting the original. The verification options can

[4] *Protocol Analyzer (PAN)* is the name of SPIN's earliest predecessor. Now, it is probably better explained as an acronym for *Process Analyzer* [22].

be set by selecting 'Set Verification Options..' in the 'Run..' menu , which results in the panel depicted in figure 5.3.

SPIN can check safety properties and liveness properties. The former comprise state properties as assertions (i.e. PROMELA statements that are always executable and must evaluate to **true**) and invalid end state checks (i.e. deadlock checks, we return on these below). The latter comprises path properties. Clicking the 'Run' button on this panel results in a three-step process: SPIN generates an (ANSI C) model-specific verifier called pan.c, which is compiled to obtain an executable verifier. The resulting executable pan is subsequently executed to perform the actual verification, which is again timed using the UNIX utility `time` (as with UPPAAL, we reported the 'real' time component in our results). Both compilation and execution depend on the basic and advanced options specified. The advanced verification option can be accessed using the '[Set Advanced Options]' button, which results in the panel in figure 5.4. The values in this figure reflect the settings during our verification runs, unless stated otherwise.


Advanced Verification Options

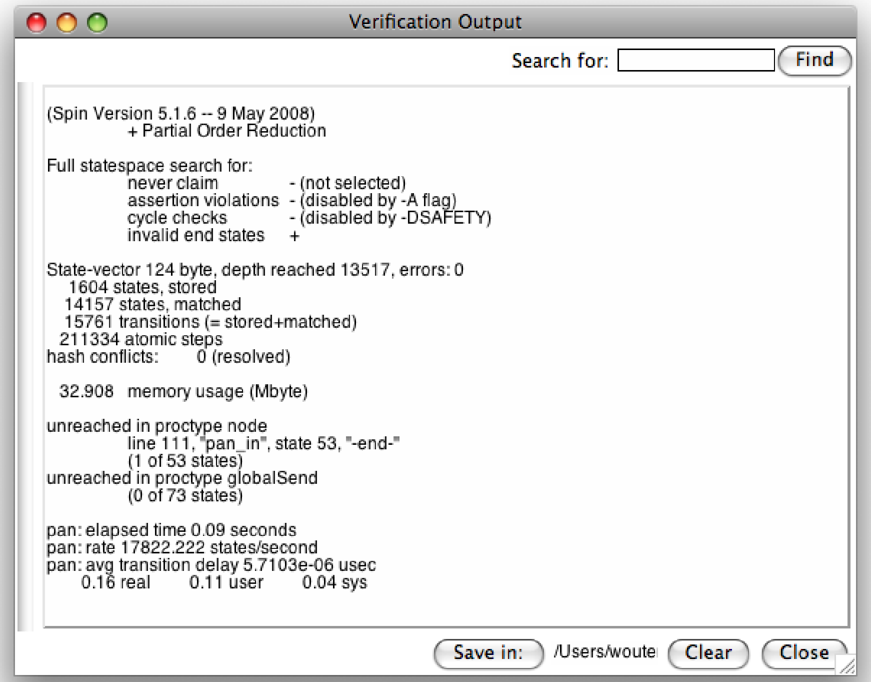
Physical Memory Available (in Mbytes):	1024
Estimated State Space Size (states x 10^3):	500
Maximum Search Depth (steps):	1000000
Nr of hash-functions in Bitstate mode:	2
Extra Compile-Time Directives (Optional):	
Extra Run-Time Options (Optional):	
Extra Verifier Generation Options:	

Error Trapping
- ● Stop at Error Nr: 1
- ○ Don't Stop at Errors
- ☐ Save All Error-trails
- ☐ Find Shortest Trail (iterative)
- ☐ Use Breadth-First Search

Type of Run
- ☑ Use Partial Order Reduction
- ☐ Use Compression
- ☐ Add Complexity Profiling
- ☐ Compute Variable Ranges

Figure 5.4: The Advanced Verification Options Panel

We used SPIN version 5.1.6, dated May 9, 2008 and XSpin version 5.1.0 (April 24, 2008). The hardware used is a 13.3" MacBook (Intel Core2Duo processor on 2 GHz, 2 GB RAM) running Mac OS X 10.5 Leopard. The output of a verification run as described above is orderly displayed by XSpin. The output for an example verification run (that checked for invalid end states) is shown in figure 5.5.



```
(Spin Version 5.1.6 -- 9 May 2008)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (not selected)
        assertion violations    - (disabled by -A flag)
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 124 byte, depth reached 13517, errors: 0
    1604 states, stored
   14157 states, matched
   15761 transitions (= stored+matched)
  211334 atomic steps
hash conflicts:         0 (resolved)

  32.908   memory usage (Mbyte)

unreached in proctype node
        line 111, "pan_in", state 53, "-end-"
        (1 of 53 states)
unreached in proctype globalSend
        (0 of 73 states)

pan: elapsed time 0.09 seconds
pan: rate 17822.222 states/second
pan: avg transition delay 5.7103e-06 usec
    0.16 real         0.11 user         0.04 sys
```

Figure 5.5: The Verification Output Window

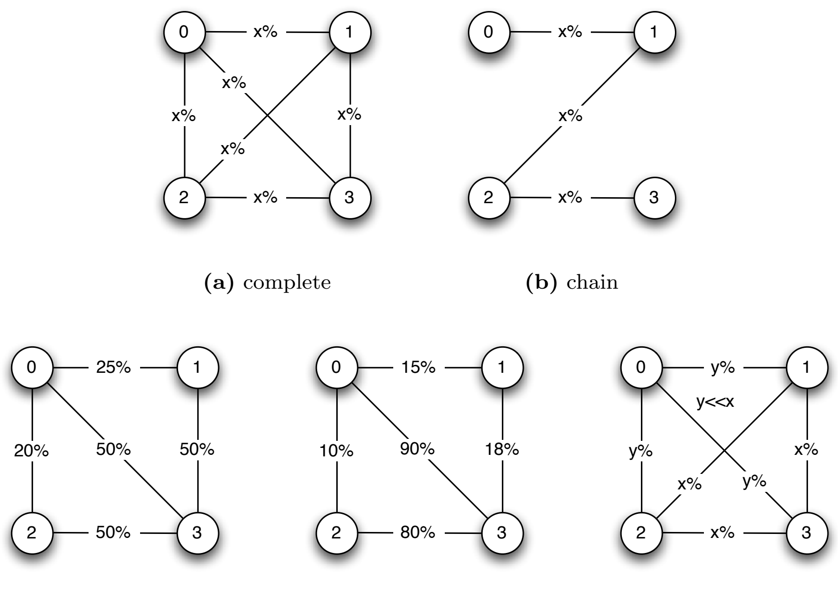
5.3.1 Verification Parameters

Recall from the previous chapter (UPPAAL, ch. 4) that the verification process depends on a number of internal and external parameters. This causes an instance explosion of potential models to verify, so we have to restrict ourselves by only verifying instances for some selected set of parameter values. The (applicable) parameters and their selected values are shortly repeated below.

1. *Properties* – we structurally verified the following properties: deadlock freedom, correct parent selection and correct distance computation.
2. *Number of nodes* – start with N=2 nodes and increase until infeasible.
3. *Topologies* – The kind of topology, figure 4.6 is reprinted for convenience in figure 5.6 below.

4. *Maximum number of message rounds* – We verified instances with MAX_M equal to 10, 20, 50 and 100.

5. *Precision* – constant ACCURACY equals 1 or 10.



(a) complete **(b)** chain

(c) multiple SPTs **(d)** multiple SPTs **(e)** bad gateway

Figure 5.6: Interesting topologies for verification (figure 4.6 reprinted)

In this chapter we do not consider topology 5.6(e). It is considered in chapter 8, which is about verification experiments and protocol variants.

5.3.2 Deadlock Freedom

Recall from the previous chapter that checking for deadlocks is useful because it might uncover errors in the model and errors in the system that is modelled (such that the user is forced to investigate the source of the deadlock). Moreover, all states of the model must be explored which results in an indication of the size of the generated state space.

As said before, SPIN can check for the occurrence of *invalid end states*. An invalid end state is SPIN's formalization of a system deadlock state. The notion of a deadlock in SPIN is somewhat more nuanced than in UPPAAL: a system deadlock is a state of the composed system without successor states, in which at least one process did not reach a valid local end state. A valid (local) end state of a process is either the state in which it reached the end of its body, or any state of the process that is explicitly marked to be a valid end state. An example of such an explicitly marked end state (using label endN:) in **proctype** Node is found on line 95 of listing 5.3 (page 76).

Deadlock freedom cannot be expressed in LTL and SPIN does not define a special formula for it like UPPAAL does. Instead, SPIN directly supports checking for deadlocks by providing the option to check for invalid end states, as shown in figure 5.3.

Verification Results

Tables C.1 – C.3 in appendix C display the results of the verification runs we performed on our PROMELA model using SPIN, considering deadlock freedom. The first result to notice is that SPIN did not find a deadlock state of the PROMELA model during any of these runs. The table entries show the number of states generated for the model and the time the verification took. Tables C.1 and C.2 show the results for the complete topology of figure 5.6(a) with all links 100% and 10% respectively. Table C.3 shows the results of the chain topology with all links 10%.

Note that there is no table with the results for a chain topology with all links 100%. This is because these results equal the results for the complete topology with all links 100% (C.1). But how can this be explained? After examining the models and the results we observed that in both cases there is no branching in the state space since there never is a non-deterministic choice (due to the 100% links canLoose() is always false). Therefore, the number of stored states is the same, due to the atomic statements (804 stored states for N=4, MAX_M=50). However, these statements do cause a subtle difference in the number of atomic steps (and thus in the total search depth): for N=4, MAX_M=50, the complete model results in 6385 atomic steps and the chain model in 5291 atomic steps.

If we take a closer look at table C.1, the following additional observations can be made:

- It only reports *stored* states, as opposed to the other tables that also contain *matched* states (denoted in italics). The latter are states of the model that were already explored (and stored by the verifier, thus so are their successors) and it is not needed to re-explore them. As said before, there is no branching in the state space due to the 100% links (no non-deterministic choice). Here, the absence of matched states thus indicates the absence of cycles in the state space.
- For 50 message rounds, verifying 4 nodes results in 804 states and verifying 8 nodes results in 1604 states. These runs took approximately 0,09 and 0,10 seconds respectively.

- The number of states grows proportionally in the number of nodes
- The number of states grows proportionally in the number of message rounds.
- Number of states $s = 4 + 4 \cdot N \cdot MAX\_M$. Apparently, each node generates 4 states every message round.
- Results of verification runs are independent of the value of ACCURACY.

Table C.2 shows what happens if the link quality is low (complete topology with all links 10%): the number of stored states is the same as in the 100% case ($4{+}4{\cdot}N{\cdot}MAX\_M$), but now there are also matched states. The number of matched states m grows proportionally in the number of message rounds, but exponentially in the number of nodes. Further examination results in the formula $m = \frac{9}{10}MAX\_M \cdot (2^{N(N-1)} - 1)$. This formula contains some recognizable elements:

- $N(N-1)$: the number of *directed* links between N nodes in this topo.
- $\frac{9}{10}$ equals $1-p$ where p is the fraction of messages that comes across, i.e. the link quality (each link has quality $p = \frac{1}{10}$ here). The formula for the number of matched states yields the exact result of SPIN whenever MAX_M is a multiple of the denominator of the link quality (which is the period over which the message loss is determined).

The number of matched states thus grows very fast in the number of nodes (MAX_M=50, N=8 yields $3.2 \cdot 10^{18}$ matched states)!

Finally, table C.3 displays the results for chain topologies with all links 10%. Again, the number of stored states equals the previous cases but the number of matched states is far less compared to the complete topology with 10% links. This is what we expected since fewer links results in fewer messages being sent and thus fewer lost messages. We were unable to infer a formula for the number of matched states in this case.

Stored and Matched States

We were able to state formulæ for the number of stored states s and the number of matched states m of our model reported by SPIN:

- $s = 4 + 4 \cdot N \cdot MAX\_M$
- $m = p \cdot MAX\_M \cdot (2^E - 1)$

where p is some probability that depends on the link quality and E is the number of *directed* edges in the topology under consideration. This indicates that the number of stored states is independent of the number of links in the topology. However, the number of matched states grows much faster and thus determines the feasibility of a verification. Therefore, the determinant of the feasibility of a verification run is the number of directed edges in the topology. Section C.3.3 in the appendices experimentally shows that the number of matched states is independent of the number of nodes. It also attempts to derive an expression for p, but this attempt was unsuccessful.

UPPAAL vs. SPIN

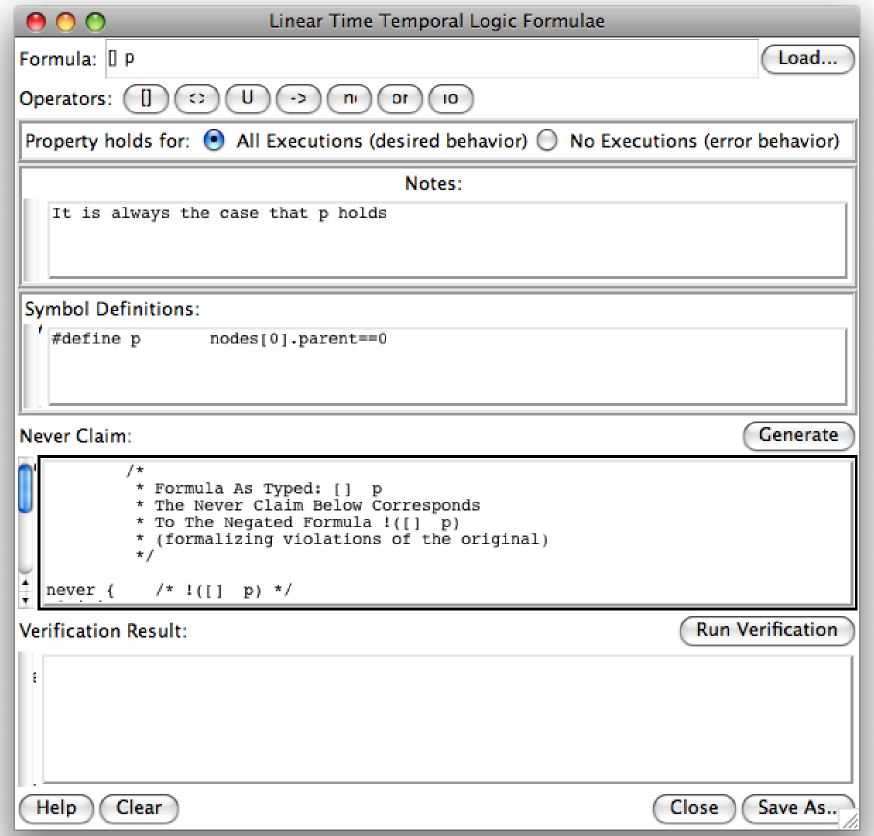
Our SPIN model is based on the ideas of UPPAAL model V3, with fixed execution order. If we compare the results of checking deadlock freedom in UPPAAL model V3 and the SPIN model, we must conclude that SPIN performs better: it is able to check a complete topology of 5 nodes with links of 10% within reasonable time (about 7 minutes), while UPPAAL was not able to check the same topology for more than 3 nodes.

5.3.3 Correct Parent and Distance

In the previous chapter we checked whether the protocol behaves correctly by checking whether all nodes found a correct parent w.r.t. the SPT or whether each node found its correct distance (i.e. Expected Transmission Count (ETX)) to the gateway. We were able to do this rather elegant by making use of UPPAAL's user-defined functions. Using SPIN, things are slightly less elegant but fortunately, not impossible.

As described in the first section of this chapter, correctness properties for SPIN models must be specified using a subset of the temporal logic LTL. To this end, XSpin provides a useful LTL property manager (figure 5.7).
The desired model behaviour is expressed as an LTL formula using symbols that are defined in the 'Symbol Definition' field using macros. Figure 5.7 provides an example: the formula `[] p`, pronounced 'always p', with `p` denoting the expression `nodes[0].parent==0`. In other words: it is always the case that the parent field of node 0 (which is the gateway) is equal to 0. Once the formula is specified, pushing the 'Generate' button results in the corresponding *never claim*[5]. This is a PROMELA process specifying the behaviour corresponding to the *negated* formula. According to Holzmann [22], "a never claim is normally used to specify either finite or infinite system behaviour that should *never* occur". As can be seen, it is also possible to specify error behaviour, in which case the never claim is generated from the formula as typed (i.e. not negated). The generated never claim can be included into the model to be verified to check if the specified behaviour can

[5]It is shown in the mid eighties that for every temporal logic formula there exists a Büchi automaton that accepts precisely those runs that satisfy the formula [22].



Formula: [] p

Property holds for: [✓ All Executions (desired behavior)] No Executions (error behavior)

Notes:

```
It is always the case that p holds
```

Symbol Definitions:

```
#define p       nodes[0].parent==0
```

Never Claim:

```
        /*
         * Formula As Typed: []  p
         * The Never Claim Below Corresponds
         * To The Negated Formula !([]  p)
         * (formalizing violations of the original)
         */

never {    /* !([]  p) */
```

Verification Result:

Figure 5.7: The LTL Property Manager

occur in the model. SPIN will flag it as an error if an accepting run can be found that matched the behaviour expressed [22].

We specified a number of LTL properties in order to check whether the model of the protocol is able to find correct parents and correct distances to the gateway, for all nodes. We will now present these properties and explain them. The first property we specified is the following:

```
[]<> (p && q)
```

In words: it is always the case that eventually p and q hold. We defined p as follows:

```
#define p    M==MAX_M
```

This means that in every state of the model, eventually a state should be reachable in which the global message round counter `M` equals the maximum number of message rounds (i.e. `MAX_M` message rounds have passed by). The state in which `MAX_M` is reached is a final state because verification terminates in it, as specified by the model. We can now use this (partial) property for parent checking but also for distance checking, depending on how `q` is defined:

```
#define q   nodes[1].parent==u &&
            nodes[2].parent==v &&
            nodes[3].parent==w
```

or

```
#define q   nodes[1].D[1]==u &&
            nodes[2].D[2]==v &&
            nodes[3].D[3]==w
```

The before-mentioned property together with these definitions of `p` and `q` now expresses that in all states of the model eventually a state can be reached in which all non-gateway nodes found the specified parent or the specified distance (i.e. the values specified for `u`, `v` and `w`) after `MAX_M` message rounds. Note that this property may not hold for small values of `MAX_M`. The corresponding UPPAAL properties are 3 and 4 in appendix B (section B.5.2).

As in the UPPAAL case, we also specified LTL properties for SPIN that allow for determining a *threshold* message round number M_T: it is always the case that if all nodes reached message round `x`, they all selected the specified parent and keep selecting the specified parent up to message round `MAX_M` (or computed the specified distance and keep computing the specified distance up to `MAX_M`):

```
[] (p -> q)
```

where `q` is defined as before (for resp. parent checking or distance checking). The symbol definition of `p` is printed below:

```
#define p    M >= x
```

with `x` a specified value that can be adjusted to find M_T. The corresponding UPPAAL properties are 7 and 8 in appendix B (section B.5.2). The LTL properties explained above are numbered and listed in appendix C, section C.3.2, together with detailed results of verifying them. We will discuss these results below.

Verification Results

We verified our PROMELA model of the protocol for three 4-node topologies (just like we did with UPPAAL). First, we verified a rather arbitrary topology that can be obtained by substituting 33% for 18% in topology 5.6(d) on page 82. Second, we verified topology 5.6(c) which contains two SPTs and simple link qualities. Third, we verified 5.6(d), which is also a topology with two SPTs, but with more complex link qualities resulting in fractions in distance computation. We refer to tables C.4 and C.5 in section C.3.2 for the detailed results. Here we will discuss the results more globally.

First of all, it is important to remark that our protocol model finds correct parents and distances for each node for *each* verification run we did. Therefore all runs are labelled OK and as a result, the OK/NOK labels were omitted in the result tables. This again increases our confidence in the correctness of the protocol. Moreover, the results also indicate that the protocol properly handles topologies in which multiple SPTs are present.

The SPIN results also mimic the UPPAAL results regarding the message round thresholds: finding correct parents requires less accurate measurements of link quality, as opposed to finding correct distances. Therefore, the thresholds found for parent checking are significantly lower compared to the distance checking case. More important is that we again may conclude that the protocol stabilizes. Take for instance topology 1: from 126 message rounds on and up to `MAX_M=500` message rounds, the model finds exact correct distances for all non-gateway nodes. This means that the SPT found (the routing tree) will not continuously change. However, for certain link qualities the threshold may become rather high due to fluctuation: in the case of topology 3 (page 82), the message round threshold for distance checking is 1001 (with `MAX_M`=2000). We learned by simulation that the dist-to-G of node 1 keeps fluctuating between 66 and 67. From round 1001 on, it finally stays 66 up to round 2000. This is probably a result of the rounding during distance computation (simulation also showed that it is not the difference between the two possible parents: different distances via the same parent also occur).

UPPAAL vs. SPIN

The trends in the SPIN results mimic the trends seen in the UPPAAL case, which increases the confidence in the results. SPIN is able to check some runs that are infeasible using UPPAAL. This is the case for topology 3 (figure 5.6(d)). SPIN also performs the verification runs much quicker, which is of course more convenient (all runs in SPIN were finished within at most 3 seconds while UPPAAL often needs far more then 10 seconds). On the other hand, UPPAAL supports user defined functions that we used to rather elegantly check for correct parents and distance. These functions also allowed for adding a tolerance bound for distance checks rather easily. In SPIN, things were less elegant, but not impossible. We did however not check tolerance bounds but this is justified since we actually do not need them: unlike UPPAAL, SPIN is able to check the exact distances within reasonable time.

5.4 Conclusion

In this section we summarize our experiences, problems and results encountered in this chapter. We will first enumerate our experiences with SPIN and its usability. Next, a number of conclusions about the SPIN model are enumerated together with the main results of its verification. We finalize this chapter with some concluding remarks about the protocol.

5.4.1 SPIN Experiences

SPIN is a powerful command line model checker. For many users, the first introduction to the tool is the default graphical interface XSpin, which greatly improves usability. It is however less easy to start with for a beginner, compared to UPPAAL. This is probably because of the less transparent character of the website, online documentation, underlying theory and tool implementation. Therefore, a rather indispensable and highly valuable resource is the SPIN 'primer and reference manual' by Holzmann [22]. Below we enumerate our findings about SPIN, PROMELA and XSpin for model construction, simulation and verification.

Model Construction

- A SPIN model is written in the specification language PROMELA and basically consists of data types, message channels, process types and (dynamic) process instantions.
- PROMELA focusses on process modelling rather than on computation.
- PROMELA's *executability* semantics is a basic means for modelling process synchronizations.
- There is a theoretical bound on the number of active processes and on the number of message channels (both max. 255). In practice this is however hardly a bound.
- PROMELA does not support floating point data types to encourage abstraction from computational aspects. There is however a possibility to embed C code.

- Modelling message broadcast requires a considerable amount of effort. Ruys [50] (section 4.11) describes some approaches and we introduced yet another approach using a global matrix.
- It is rather difficult to model probabilistic behaviour. But according to Holzmann [22]: "In a well-designed system, erroneous behaviour should be impossible, not just improbable".
- SPIN comes with a PROMELA syntax checker but the error messages are sometimes very vague.
- PROMELA models are saved in plain text (less structured, compared to eXtensible Mark up Language (XML), less portable, plain text needs a specific parser).
- XSpin's usability really needs revision. For example, some windows required a resize before displaying all buttons and labels properly, and it has very poor editor capabilities (scrolling, undo/redo, unsaved changes, syntax highlighting).

Simulation

- SPIN comes with a simulator to validate the model's behaviour.
- It supports random, guided and interactive simulation and the possibility to skip simulation steps.
- Generated simulation traces can be saved to and loaded from files.
- Error traces of the verifier can be loaded into the simulator for detailed examination.
- During interactive simulation, XSpin is capable of indicating current enabled statements using useful text selections.
- Running a simulation results in many output windows of which positions and sizes are not remembered.
- All simulation output is often only written to the output windows when simulation ends (data values, sequence chart).

Verification

- SPIN generates an optimized verifier of a PROMELA model, that can be compiled and run separately (realizing a significant performance gain).
- SPIN is able to check safety and liveness properties through the use of assertions, state labels and never claims.

- Liveness properties are specified in a subset of LTL.
- The specified LTL property is converted into a never claim which is a PROMELA process describing behaviour that should never occur.
- Local process data cannot be accessed in LTL properties.
- XSpin generates an orderly report of the verification statistics.

5.4.2 The Model

The list below contains observations and conclusions about our SPIN model:

- Our SPIN model is based on the idea of UPPAAL model V3 (non-deterministic links) with fixed execution order.
- Message broadcast is modelled using a global matrix that indicates for each node whether it will receive a message. It is updated every message round by a single process. Each message round, this process non-deterministically decides (based on given connectivity information and history) whether a message comes across or not. This implies fixed send order.
- Each node is modelled by a process that computes distance to the gateway and processes received messages. The nodes can only execute in fixed order (fixed receive order).

Results and conclusions about the verification of these models are collected below:

- For the verified topologies, the model is free of deadlocks (invalid end states).
- For topologies with solely 100% links, there are no states re-encountered by SPIN (i.e. matched states), since there is no branching in the (acyclic, finite) state space.
- The number of stored states $s = 4 + 4 \cdot N \cdot MAX\_M$.
- The number of matched states $m = p \cdot MAX\_M \cdot (2^E - 1)$. Here p is some probability that depends on the link quality and E is the number of *directed* edges in the topology under consideration. If all links have quality q then $p = 1 - q$.
- For the verified topologies, the model selects correct parents (w.r.t. the SPT rooted at the gateway).
- For the verified topologies, the model computes correct distances to the gateway (w.r.t. the SPT rooted at the gateway).

- Checking for correct parents is preferable above checking for correct distances since correct parents are found after fewer message rounds.
- The determinant of the feasibility of a verification run is the number of directed edges in the topology, rather than the number of nodes. A verification still ends within reasonable time (say less than 10 minutes) whenever the number of links is maximally 10 (corresponding to 20 directed edges). This corresponds to an order of magnitude of 10^7 transitions.
- A topology with 11 links (22 directed edges) already takes more than 20 minutes. This corresponds to an order of magnitude of 10^8 transitions. The number of message rounds, link qualities and the number of nodes have less influence.

5.4.3 The SPT Protocol

As was the case with UPPAAL, we did not found severe errors in the SPT protocol, which increases confidence in its correctness. It does however not prove its correctness since we can only verify the protocol for certain interesting topologies, not for all possible ones.

Our verification process showed that there may be fluctuation in distance computation of the protocol. This often does not form a problem since it fluctuates around the correct value. If the fluctuation is relatively high and all link qualities are near to each other, this may influence the parent selection process. In that case errors might occur.

CHAPTER 6

The Hidden Problem

"The only real mistake is the one from which we learn nothing."
John Powell

This chapter describes a characteristic subtlety of our SPIN model, which may appear as a serious flaw to an experienced SPIN/PROMELA expert. The problem is related to PROMELA's keyword **hidden** so we start with describing its use. Then we recall the use of this keyword in our model and we analyze the impact and consequences in this particular case. Based on this analysis we present some solution directions which try to exploit this subtlety. Finally, we draw conclusions about the learned lessons.

6.1 The Keyword **hidden**

As explained in chapter 5, we used the PROMELA keyword **hidden** in our model. It prevents SPIN from taking the (as hidden) declared data into account in the global system states. Typical use of hidden variables is for so-called 'scratch' variables. These are auxiliary variables only used for temporary storage at some point during model execution: their value is not read later on (their value does not matter during further model execution). Not hiding such data could needlessly result in a larger reachable state space [22]. Use of the **hidden** keyword may thus reduce the number of system states drastically, resulting in more feasible verifications.

To illustrate this, consider an arbitrary SPIN model that contains the global declaration: **hidden bool** b;. Suppose that this model defines a state space of x states (i.e. the model can be in x different states). Now, if the Boolean variable b would not be hidden, each new value stored in it would result in a new global state of the model! In this particular example, this would result in twice as much states in the worst case (i.e. $2x$), since a Boolean type has only *two* possible values. We said *worst-case* because this is under the assumption that each possible value of the variable b is actually stored at least once during execution of the model. Clearly, the impact of (unhiding) variables of bigger types is worse. Or the other way around: hiding variables can significantly reduce the size of the state space.

According to Holzmann [22], the **hidden** keyword should be used with caution. Hiding too much information from the state descriptor can easily result in the verifier performing an incomplete search with unreliable outcome. Moreover, the verifier takes the information for granted and does not check whether the use of the **hidden** keyword is unwarranted.

6.2 Hidden Matrices

Recall that our SPIN model works with 3 hidden N×N matrices, each matrix declared as an array of N arrays of N elements:

```
typedef Tuple {byte u; byte v}
typedef NodeDimTuple{ Tuple to[N]; }
hidden NodeDimTuple C[N];                  /* Connectivity Matrix, NxN */
hidden NodeDimTuple H[N];                  /* History Matrix, NxN */

typedef NodeDimBool{ bool to[N]; }
hidden NodeDimBool Msgs[N];                /* Message Exchange Matrix, NxN */
```

Listing 6.1: Declaration of 3 hidden matrices

1. A connectivity matrix C[N], defining (symmetric) link qualities of the links between each node pair;
2. A history matrix H[N] to keep track of the number of delivered probe messages (so far) between each node pair;
3. A message exchange matrix Msgs[N] which is updated each message round, indicating whether a probe message is successfully delivered between each pair of nodes, that round.

We decided to hide these matrices because they are part of the mechanism to simulate the probabilistic links, rather than part of the protocol we wanted to verify. After all, the matrices do not contribute to the state of the protocol, they just enforce an approximation of probabilistic behaviour of the wireless links between the nodes.

However, in order to be able to effectively verify the SPT protocol, any model of it should realize two important aspects: on the one hand it should of course mimic the protocol behaviour (since that should be verified), and on the other hand it should also simulate the probabilistic link behaviour, (since an important task of the protocol is estimating the link qualities, which obviously requires the link behaviour). Both aspects are essential for a model of the protocol and thus should both contribute to the state of the model. Our original motivation to hide the matrices did not account for the second aspect: the matrices should contribute to the model state but they are not since they were hidden.

Moreover, these matrices do not function as typical 'scratch' variables in the model (as explained in the previous section). Indeed, they do function as such for the protocol aspect of the model, but they do explicitly *not* for the link aspect: the values actually do matter since they are used later on during model execution. This often results in a problem during *backtracking*, explained below.

6.2.1 Backtracking the State Space

SPIN is an on-the-fly model checker: the model states are generated and checked at the same time, rather than first generating the complete reachable state space of the model and then performing the checks. Since there often is branching in the state space (a state can have several successor states) there must be an algorithm to explore the states systematically. By default, SPIN follows the depth-first search algorithm [22], meaning that all states are explored in a depth-first manner. This works as follows.

The initial state is checked and its successors are generated ('the initial state is *expanded*'). Then, its first successor is checked and expanded, and then the first successor resulting from this expansion is checked and expanded, etcetera, etcetera, until a state is reached that cannot be expanded (it has no outgoing transitions to successor states). At this point (i.e. the end of a trace), the algorithm 'walks back' over the generated trace of successor states, until it reaches a state that has still other unexpanded successors left. This is called *backtracking*. Now the algorithm continues with checking and expanding, in the same (depth-first) manner. The process terminates if all states are expanded and explored.

6.2.2 Backtracking and **hidden**

If there are hidden variables in the model, problems might occur during backtracking (if **hidden** is used wrongly). A state descriptor of a model state contains the values of all variables, except for the hidden variables (because that is why they were hidden). The values of the hidden variables can be thought of as saved outside of the state descriptor. After backtracking, the algorithm continues with all variable values as prescribed in the state descriptor of the state under consideration. Note that the values of the hidden variables might have been changed since the generation of the state under consideration (remember that we came there after backtracking). This is why the values of hidden variables cannot have meaning and should not be used 'later on': after backtracking, the hidden values are (very likely) changed, which results in an incomplete or unreliable state space exploration.

Figure 6.1 visualizes this: the hidden variable var is changed twice (var'') and after backtracking from state x to state b, the algorithm continues exploration with the unexpanded successor state y. Therefore, reading the value of the hidden variable later on would mean that the behaviour of the model depends on the exploration algorithm of the verifier.

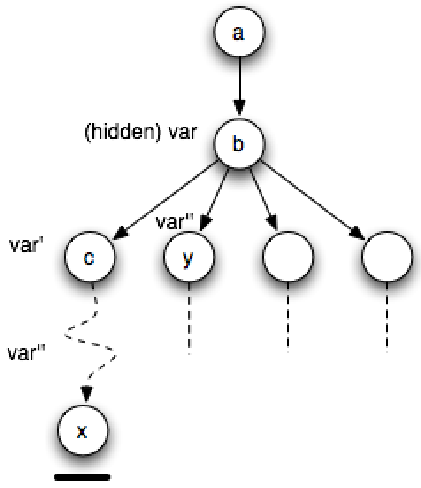


Figure 6.1: A visualization of the **hidden** problem

6.2.3 The Hidden Problem in our Model

Branching in the state space (a state having multiple successor states, like for instance state b in figure 6.1) has two possible sources: non-determinism in the execution order of the processes in the model or non-determinism explicitly present in one or more processes. In our model, we fixed the execution order of the processes (using global variable ctrl) so the only source of branching is the non-determinism in the globalSend process. The relevant fragment is printed in listing 6.2.

```
do                                              /* Fill msgs matrix */
:: (i<N) -> do
    :: (j < N && C[i].to[j].u == 0) -> j++      /* No link between i and j */
    :: (j < N && C[i].to[j].u != 0) ->          /* There is a link between i and j */
        if
        :: canLoose(i,j) ->                     /* ND-choice: write 0 in msgs matrix */
            Msgs[i].to[j] = 0                   /* 0 means msg gets lost */
        :: canSend(i,j) ->                      /* ND-choice: write 1 in msgs matrix */
            Msgs[i].to[j] = 1;                  /* 1 means msg will come across */
            H[i].to[j].u++                      /* Counts nr of delivered msgs */
        fi;
        H[i].to[j].v++;                         /* Counts msg opportunities */
        checkReset(i,j);
        j++
    :: (j==N) -> j=0; break
    od;
    i++
:: (i==N) -> i=0; break
od;
```

Listing 6.2: Non-determinism in the globalSend process

The code in listing 6.2 is executed every message round. It non-deterministically fills the message exchange matrix Msgs, based on the values in the history matrix H, in order to enforce the given link qualities in the connectivity matrix C. The number of different states resulting from this code fragment is determined by the number of possible message exchange matrices. Recall that this matrix contains Boolean entries indicating whether or not a probe message comes across between the corresponding node pairs. Of course, messages can only come across from one node to another if there is a link defined between them in the connectivity matrix. Therefore, this results worst-case (i.e. a completely connected topology) in $2^{N(N-1)}$ message exchange matrices (each of the $N(N-1)$ links between the N nodes can or cannot deliver a message).

The core of the problem

The model is in a certain state before executing this code fragment (which is in an **atomic** construct). It is important to note that, regarding the above, this state has 2^E successor states (with E the number of directed links between the nodes in the topology that is modelled). But, since these successors only differ in the hidden matrices Msgs and H, SPIN cannot distinguish between them: to SPIN, these states are all equal! In other words: *only one of the successors is expanded and explored because the others are matched to it!* We thus have to conclude:

There is *no* branching in the state space of our model.

Besides the insight stated above, this also explains the formula for the number of matched states (chapter 5) $m = p \cdot MAX\_M \cdot (2^E - 1)$: of all 2^E possibilities *every message round*, only one (this explains the minus 1) is explored, the rest is matched (E being the number of directed links).

6.2.4 Consequences

As said in the previous section, any model of our protocol should model two aspects: the protocol behaviour and the behaviour of the underlying probabilistic links. The latter aspect was already found to be hard to implement using a non-probabilistic modelling language. We approximated probabilistic behaviour using non-determinism, but by hiding the matrices Msgs and H, this approximation turns out to be very poor: only one of many possible probe message sequences is checked. This is explained in the following example.

For instance, every ten message rounds, a link of quality $\frac{1}{10}$ should deliver exactly one probe message. This message can be delivered in one of every ten rounds. Our model does not verify all ten possibilities but only one. Nevertheless, the quantitative aspect of the link quality is maintained (each possibility results in the desired quality of $\frac{1}{10}$) and that is the explanation why our verification results were correct w.r.t. the expected SPT tree.

Unfortunately, we could not reveal the details of the exploration order of SPIN, so we do not exactly know the resulting probe message sequence (i.e. is the message delivered in the first of every ten rounds? Or in the second? Or maybe just in the tenth?). We do know that the exploration order is deterministic which indicates that it is always the same periodic message sequence (i.e. the message is always delivered in for instance every 10th round).

It could be noted that we did not mention the connectivity matrix C as part of the problem. This is because hiding the connectivity matrix C does not have undesired consequences. Indeed, it is used throughout the model but it is constant (it never changes after the **d_step** construct in which it is initialized). Hiding it does result in a smaller state descriptor though.

6.3 Solution Directions

Obviously, the problem sketched in the previous section can be solved rather naive by just ‘unhiding’ the matrices Msgs and H. We elaborate on this solution in the first subsection below. We also present the results of some initial experiments that we did using this solution.

Besides the naive solution, it might be interesting to consider other, more sophisticated solution directions. The second and third subsections below discuss two slightly more sophisticated solutions, which may have some advantages compared to the naive approach. Again, we did some initial experiments and we present the results.

6.3.1 Naive Approach

The most straightforward solution to the problem sketched is to remove the keyword **hidden** from the declaration of the Msgs and H matrices. This will decrease the number of matched states drastically, at the cost of a huge increase of the state space (stored states). This is a result of the fact that now all possible values of these matrices result in new model states (there is branching in the state space).

The number of possible matrices is exponential in the number of nodes: for 2 nodes completely connected, there are $2^2 = 4$ different values for the Msgs matrix at the very first branching in the state space; a complete topology of 4 nodes already results in $2^{12} = 4096$ different values. Note that due to the link quality mechanism, there might be fewer different values possible at later branching points. Note also that the value of matrix H in message round M depends on its previous value (in round $M - 1$) and on the value of matrix Msgs in round M. Therefore, it does not introduce new states and the number of successor states at a branching is thus determined by the number of possible different values of matrix Msgs.

It is interesting to do some initial experiments in order to analyze the change in the number of states in this case. Moreover, an important question is whether verification remains feasible at all.

Naive Solution Experiments

The description of a set of cohesive experiments in this chapter comprises five elements: a hypothesis, a motivation thereof, the setup, the results and a conclusion about the stated hypothesis. These elements are enumerated below for the experiments concerning the naive solution.

1. Hypothesis: Not hiding matrices Msgs and H in our model will cause (meaningful) verification to become infeasible. We define 'meaningful verification' as verification of models of at least 3 nodes and M message rounds, where $M \geq 2v_{max}$ and v_{max} is the largest denominator in the connectivity matrix C. All link qualities should be less than 1.

2. Motivation:

    - The number of reachable states will explode, since the number of states at a branching is determined by the number of different values of the Msgs matrix. Moreover, branching occurs every message round!
    - The number of nodes for a meaningful verification is at least 3 in order to show that the protocol is capable to select a correct parent.

- The number of message rounds is at least two times the largest denominator in the connectivity matrix in order to allow the probabilism approximation mechanism of the model to simulate at least two periods for each link.
- It is trivial that link qualities should be less than 1 in order to activate the probability approximation.

3. SETUP: We perform SPIN's invalid end state check on a complete and chain topology of respectively N=3 and N=4 nodes with all links 10% (i.e. $\frac{1}{10}$). These topologies can be found in figure 5.6 on page 82. We compare the results to the original model. Hardware, version and verification settings of SPIN are as mentioned in chapter 5, page 80. We report the number of stored and matched states, the number of MAX_M and the time taken by the verification. The value of MAX_M is set to 20 or lower (depending on feasibility). An infeasible verification is one that runs out of memory (i.e. requiring more physical memory than the pre-set of 1024 MB).

4. RESULTS: The table below displays the verification results. Each topology column comprises three sub columns: time, stored/matched states and the maximum number of message rounds MAX_M. The results in italics are for the corresponding original model (i.e. the same model with matrices Msgs and H hidden).

Table 6.1: Naive Solution Experiments - results

N	Chain Topo 10%			Complete Topo 10%		
3	1.79s	433228/48205	20	38.5s	6849747/1602681	18
orig	*0.08s*	*244/270*	*20*	*0.09s*	*220/1071*	*18*
4	14.9s	3033672/271486	12	0.36s	69635/0	1
orig	*0.09s*	*196/693*	*12*	*0.10s*	*20/4095*	*1*

5. CONCLUSION: The results confirm our expectations: the naive solution causes a huge increase in the number of stored states and a (less) huge decrease of the number of matched states. Only one verification run is meaningful (as defined above): the 3-node chain topology. It is important to note that the corresponding value of MAX_M in this run is not an upper bound for feasibility (in contrast to the MAX_M values of the other verification runs). This result is rather disappointing since it is not very likely that such a small topology can uncover all potential problems of the protocol. Therefore the hypothesis is confirmed.

6.3.2 Partial Hiding

Instead of ‘unhiding’ both matrices, as was done in the preceding subsection, it may be interesting to investigate the effects of unhiding only one of the matrices. The philosophy behind it is that the number of possible values for one matrix is less than for two, resulting in a smaller state space, as there are less successor states at each branching.

The first option is to unhide the message exchange matrix Msgs while leaving matrix H hidden. This means that after backtracking to a next successor at a branching, the value of H is ‘corrupted’. However, this option is justified since H is used by the probability approximation mechanism to keep track of the delivered messages (per link) *within a periodic number of executed message rounds* (this is the denominator v of the corresponding entry in connectivity matrix C). Every time this periodic number of rounds is executed, the corresponding entry of H is *reset* (by the inline construct checkReset()):

```
inline checkReset(e,f){                     /* Reset counters when needed */
    if                                      /* (modulo C[].to[].v) */
    :: (H[e].to[f].v == C[e].to[f].v) -> H[e].to[f].u = 0; H[e].to[f].v = 0;
    :: else -> skip
    fi
}
```

Indeed, this means that after backtracking a corrupted matrix H is used, but only for a limited number of message rounds (that is, worst case, the maximum periodic number of message rounds v_{max}, i.e. the greatest denominator present in the connectivity matrix). Matrix H just records the established link qualities for a periodic number of message rounds for each link, but these values only function as starting point: the probability approximation mechanism takes decisions based on these values, in order to enforce the qualities given in the connectivity matrix C. A wrong starting point will only temporary result in poor quality approximation, which is rather quickly corrected by the mechanism. Nevertheless, we would like to call it a feature rather than a bug, as things do appear more unpredictable to the protocol. Moreover, compared to the naive solution, it will result in fewer stored states, therefore pushing the feasibility bound somewhat further.

The second option is to hide the message exchange matrix Msgs while unhiding the history matrix H. This is justified since the values in the message exchange matrix are regenerated every message round (by the globalSend() process). Like the first option, after backtracking, SPIN continues exploring the state space with a corrupt message exchange matrix. That is however only for the duration of one message round. As with the first option, this will result only temporary in poor quality approximation (w.r.t. the connectivity matrix), but also in a smaller state space.

Partial Hiding Experiments

1. Hypothesis: Hiding only one of the matrices H and Msgs in our model still results in useful model behaviour, but it yields a reduction of the state space of the model (w.r.t. the naive solution).

2. Motivation:

   - Two aspects are modelled: protocol behaviour and probabilistic link behaviour. Only the latter is affected by the change, but only temporary. The global resulting link behaviour that is exposed to the protocol part of the model remains as given in the connectivity matrix C.
   - The number of possible values for one matrix is smaller than for two. Since this number determines the branching degree in the state space (explained in the previous subsection), there are less successors states at each branching.

3. Setup: We verify the parents that are selected by the protocol on each node in three different models:

   - HM: the naive solution model (both H and Msgs not hidden);
   - HHM: matrix H hidden, matrix Msgs is not hidden;
   - HMH: matrix Msgs hidden, matrix H is not hidden.

   We verify the chain topology on three nodes connected by 10% links (see previous subsection). Constant MAX_M is set to 30. The verification uses the following LTL property:

   - `[]<>  (p && q)`
   - `#define p M==MAX_M`
   - `#define q nodes[1].parent==0 && nodes[2].parent==1`
   - "It is always the case that eventually the maximum number of message rounds is reached and that node 1 selected parent node 0 and node 2 selected parent node 1."

   We report the time taken by the verification, the number of stored and matched states, the size of the state vector and the total memory used.

4. Results: The results are shown in table 6.2. The property was verified successfully for all models. Of the two 'partial hiding' solutions, hiding matrix Msgs (model HMH) yields the largest reduction of the state space w.r.t. the naive solution (model HM): the gain in the number of stored states is 1679666 states. The gain in time is 23.59 seconds

Table 6.2: Partial Hiding Experiments - results

Model	Time(s)	Stored/Matched	State Vector	Memory
HM	65.44	4998243/6414060	116B	661MB
HHM	43.36	3440493/4576634	100B	411MB
HMH	41.85	3318577/4411724	108B	394MB

and the verification requires 267MB less of total memory. Hiding matrix H (model HHM) results in the smallest state descriptor (100B). The number of matched states as percentage of the number of stored states is 128% for the naive solution and 133% for both 'partial hiding' solutions.

5. Conclusion: The first part of our hypothesis states that the model behaviour is not significantly affected by partial hiding. This is confirmed by our experiments, as the protocol finds correct parents (indeed for this very simple, trivial topology but it is a promising start). The second part of the hypothesis (partial hiding will yield a state space reduction) is confirmed by the third column of table 6.2.

6.3.3 Controlled Branching

Our first solution direction was presented as a naive solution which caused maximal branching in the state space: all possible values of the matrices H and Msgs resulted in a maximal number of successor states at a branching. The second solution direction tried to restrict the branching degree by reducing the number of successors through hiding part of the possible values (either values from H or Msgs). This subsection describes a third solution direction that goes one step further: we try to control the branching degree.

As explained in this chapter, our original model hides matrices H and Msgs which results in a state space without branching. The idea of controlled branching is to take this original model as starting point and define an additional variable that depends on the value of these matrices. This variable should not be hidden from the state descriptor, such that the number of possible values of it determines the branching degree. The number of possible values stored in this variable is determined by the dependency upon the matrices H and Msgs: the type of dependency determines the branching degree. We can thus control the branching degree in the state space by defining how this new variable depends upon H and Msgs.

The dependency relation between the matrices and the new variable induces a partitioning of all possible matrix values. The maximal branching degree (i.e. the number of successors at a branching point) equals the number of partitions, rather than the number of all possible matrix values[1]. In this light, we can say that our original model only defines one partition that represents one possible instance of probabilistic behaviour over the message rounds. It would of course strongly increase our confidence in the protocol if we could verify a self-controlled number of instances more (for example, instances that are likely to occur in reality).

Controlling the branching degree allows for tuning the verification without breaking the link behaviour, nor the protocol behaviour. A low degree results in few simulated instances of the probabilistic link behaviour, but it pushes the feasibility bound (larger networks can be verified). Establishing a high branching degree results in better approximation of the link behaviour, at the cost of feasibility.

A simple example of a dependency relation in order to control the branching degree is to use the control variable to store the number of entries of Msgs containing a '1', each message round (i.e. the number of messages that will be delivered in a message round). This will result in $1 + E$ partitions that group all Msgs matrices that contain resp. 0, 1, 2, ..., E such entries, where E is the number of directed edges. So, for a complete topology on N nodes, this will result in a maximum branching degree of $1 + N(N - 1)$ instead of $2^{N(N-1)}$ (which is the number of possible Msgs matrices). Below, we experiment with this proof of concept of controlled branching.

Controlled Branching Experiments

1. Hypothesis: Extension of our original model with the Controlled Branching technique, as described above, results in a state space with (controllable) branching, while the protocol still finds correct parents for each possible branch.

2. Motivation:

    - In the original model, all successor states at a branching point are matched (considered equal) since they only differ in the matrices H and Msgs, which are hidden.
    - The branch control variable is not hidden and thus prevents the matching of all successors. However, successors in the same partition are still considered equal and are therefore still matched.
    - The protocol will find correct parents since each branch does establish the desired link qualities.

[1]Note that the branching degree is not constant: it is at its maximum at the start of each period of message rounds and decreases during this period, as a result of the probability approximation mechanism.

3. SETUP: We started our experiments with a model in which we count delivered probe messages (as described above) per message round. The model is our original model but with an additional global integer variable called branchControl. Every time an entry of Msgs is set to 1 (by process globalSend), it is increased (branchControl++, l. 128).

   It is important to note the dummy expression branchControl>=0 in line 119: it is always true. It reads the value of branchControl, and by doing so, prevents SPIN from recognizing the variable as write-only. It turned out that otherwise the model is optimized, since SPIN is capable to detect that the variable is never read. *Without this expression, the desired branching effect does not occur.*

```
116     do                                                  /* For each msg round */
117     :: (M < MAX_M) -> atomic{
118         do                                              /* Fill msgs matrix */
119         :: (i<N && branchControl>=0) -> do              /* Dummy use of branchControl*/
120             :: (j < N && C[i].to[j].u == 0) -> j++      /* No link between i and j */
121             :: (j < N && C[i].to[j].u != 0) ->          /* There is a link between i and j */
122                 if
123                 :: canLoose(i,j) ->                     /* ND-choice: write 0 in msgs matrix */
124                     Msgs[i].to[j] = 0                   /* 0 means msg gets lost */
125                 :: canSend(i,j) ->                      /* ND-choice: write 1 in msgs matrix */
126                     Msgs[i].to[j] = 1;                  /* 1 means msg will come across */
127                     H[i].to[j].u++;                     /* Counts nr of delivered msgs */
128                     branchControl++                     /* Counts nr of delivered msgs */
129                 fi;
130                 H[i].to[j].v++;                         /* Counts msg opportunities */
131                 checkReset(i,j);
132                 j++
133             :: (j==N) -> j=0; break
134             od;
135             i++
136         :: (i==N) -> i=0; break
137         od;
138         ctrl = GATEWAY_COUNT;}                          /* Transfer control to 1st node process */
139         ctrl == _pid;                                   /* Wait for control */
140         M++
141     :: (M == MAX_M) -> break                            /* Max nr of msg rounds reached, stop */
142     od
```

Listing 6.3: Adjusted portion of the globalSend process

   We also try to drastically reduce the branching degree: by inserting the line 'branchControl = branchControl%2;' between lines 137 and 138. This is a modulo 2 computation: now branchControl can only have two values, thus there are only two partitions left at each branching point: namely the two representing all Msgs matrices with an odd resp. and even number of '1's.

   We verified the complete 4-node topology with all links 10% with SPIN's invalid end state check. We also verified our arbitrary 4-node topology (which is depicted again in figure 6.2), in order to check whether the protocol finds correct parents in this model. The value of MAX_M is set to 20, the other constants were set to their defaults (i.e. as in the original model).

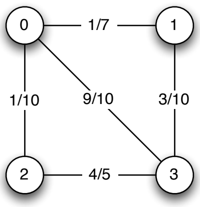


Figure 6.2: Arbitrary topo

The LTL property we used to check for correct parents:

- `[]<>  (p && q)`
- `#define p M==MAX_M`
- `#define q nodes[1].parent==3 && nodes[2].parent==3 && nodes[3].parent==0`
- "It is always the case that eventually the maximum number of message rounds is reached and that nodes 1 and 2 selected parent node 3 and node 3 selected parent node 0."

We report the time taken by the verification runs, the number of stored and matched states, total actual memory usage and the maximum search depth reached.

4. Results: The different models with Branch Control are indicated using the prefix BC. These are the details:

   - ORIG: the original model (without BC), complete topo, 10% links, check for invalid end states;
   - BCC20: complete topo, 10% links, check for invalid end states;
   - BCC20mod: complete topo, 10% links, branchControl modulo 2 as described above (odd/even partitions), check for invalid end states;
   - BCA20p: arbitrary topo from fig. 6.2, check for correct parents;
   - BCC20p: complete topo, 10% links, check for correct parents.

   The results are presented in table 6.3 below. The fourth experiment in the table resulted in an error: the protocol did not find correct parents for all nodes. In the topology under consideration (complete, all links 10%) all non-gateway nodes should select the gateway (node 0) as parent. According to SPIN however, node 2 has selected node 3 as its parent, after 20 message rounds. This number of rounds is probably too low, in combination with this low link quality.

Table 6.3: Controlled Branching Experiments - results

Model	Time(s)	Stored/Matched	Depth	Memory
ORIG	0.44	324/73710	2709	32MB
BCC20	21.40	2096070/986902	2760	320MB
BCC20mod	2.06	5544/288508	2684	34MB
BCA20p	107.77	3203561/11287990	3022	456MB
BCC20p	*error*	*parents incorrect!*		

5. Conclusion: Controlling the branching in the state space can be accomplished using a variable that depends on the values of the hidden matrices. Moreover, this results in a smaller state space, compared to the naive and the 'partial hiding' solutions (the verification of the same models is unfeasible w.r.t. memory, for these solutions). A dependency resulting in less possible values for the control variable (such as in model BCC20mod) results in fewer branches and thus fewer states. The protocol does find correct parents except for the BCC20p case. This is a consequence of the low link qualities and the small number of message rounds: the probability approximation mechanism can only deliver 2 probe messages per link, resulting in a poor quality estimation at the nodes. We consider the hypothesis as confirmed. The experiments indicate that Controlled Branching is a complex abstraction technique that needs extensive future research.

6.4 Conclusion

The problem sketched in this chapter was found in one of the last stages of this research project. Because of the limited time left at that point of the project, we decided to analyse its source and the consequences. We proposed three different solution directions to the problem and did some initial experiments.

Our original model that was presented in the previous chapter consists of two important aspects: the protocol behaviour and the underlying link behaviour. The main result of the analysis is the insight that there is no branching in the state space of this model, due to the hidden matrices H and Msgs. Only one of many instances of the probabilistic link behaviour is verified (there is only one path in the state space). The quantity of this instance is correct (i.e. as given in the connectivity matrix). This explains the correct results of the verification performed (described in the previous chapter).

The traditional model checking approach aims to prove (the absence of) certain properties of a model, by checking all possible states the model can be in. It is part of the paradigm of mathematical reasoning. This often turns out to be a problem due to the combinatorial state space explosion. Moreover, model checking practitioners found that is hard to construct flawless models and properties for realistic systems. This is a consequence of the lack of methods to bridge the gap between informal system descriptions and formal models [9]. Brinksma [8] argues that model checking is also part of an experimental paradigm: the experimental atmosphere of increasing tool performance and determining how much of a problem can be verified. Moreover, in practice, verification models are constructed using insight, heuristics and experience; these models can only be validated using experimental methods.

Our approach cannot satisfy the traditional approach to model checking, because of the implementation of the probability approximation mechanism, combined with the hidden matrices: many states are wrongly considered to be equal. Our approach does however satisfy the experimental approach, which we call 'bug hunting': verification can never guarantee total correctness but it can increase or decrease confidence in the model (by showing presence of bugs).

We can conclude that hiding the matrices Msgs and H in our model has undesired effects. The functional behaviour is insufficient but the quantitative aspect of this behaviour is as desired: the links will have the quality given in the connectivity matrix. Therefore, our verification results are not useless: our confidence in the correctness of the protocol is increased by these results as the protocol finds the correct parents.

6.4.1 Solutions and Further Research

We proposed three solution directions: a naive approach that 'unhides' the hidden matrices, a 'partial hiding' approach in which only one of the matrices is 'unhidden' and a 'controlled branching' approach. The first results in maximal branching in the state space, the second in a slightly smaller branching degree and the third in a controllable (customizable) branching degree.

Using this 'controlled branching', well-connected topologies of at most four nodes can be verified for a limited number of message rounds (up to 20). However, the determinant of the feasibility of a verification run remains the number of directed edges: a topology on 5 nodes with less links might also be feasible.

The 'controlled branching' solution direction is the most promising and needs extensive further research. In order to come to a more feasible verification, we abstract away from all possible instances of approximated probabilistic link behaviour. These instances are grouped into partitions by a dependency relation. Further research should determine how to define a dependency resulting in useful partitions. Ideally, each partition should group equivalent behaviour that is representative and likely to occur in reality.

6.4.2 Learned Lessons

As quoted at the beginning of this chapter, one should learn from made mistakes. Our concrete lessons learned are the following:

- The keyword **hidden** can easily result in erroneous or unexpected behaviour. It should therefore be used with much caution!
- SPIN optimizes a model before it is verified: a variable that is never read is recognized as a write-only variable, which is omitted from the state descriptor. Therefore, we had to add the dummy expression branchControl>=0 in the third solution direction, to let the model read the value of branchControl and prevent SPIN from discarding it from the state.
- Discovering a mistake is usually demotivating but it turns out to be a gateway to interesting research: without our ‘hidden problem’, we would not have come to the ‘controlled branching’ solution direction, which allows for an extra dimension of customization in our verification process.
- Moreover, analyzing the problem led to thorough insight in the problem, the model and the verification.

CHAPTER 7

Verification using PRISM

This chapter describes our modelling and verification activities using the symbolic model checker PRISM. The structure of this chapter is similar to that of chapters 4 and 5: we will first introduce the reader to the tool and its underlying theory and motivate why it has been chosen. Then we elaborate on the process of model construction with PRISM and subsequently we report on the verification experiences and results. Finally we summarize our findings in the concluding section.

The aforementioned chapters extensively introduced many of the issues encountered when modelling and verifying the Shortest Path Tree (SPT) protocol (e.g. modelling broadcast, execution order, link quality). We refer to these if necessary and focus here on modelling and verification with PRISM.

7.1 Tool Introduction

The Probabilistic Symbolic Model Checker (PRISM) is a tool for formal modelling and analysis of systems which exhibit random or probabilistic behaviour. PRISM is developed at the university of Birmingham and allows for analysing quantitative properties of these so-called stochastic systems [21]. The first public release of PRISM was version 1.2, dated September 2001. Since then, it has been in constant development. The current version is 3.2, dated June 15, 2008 in which PRISM's Graphical User Interface (GUI) is improved. It is a rather mature tool now which has been successfully deployed in a wide application domain: from real-time communication protocols to biological signalling pathways [21]. PRISM is free and open source software, available at `http://www.prismmodelchecker.org`.

7.1.1 Underlying Theory

PRISM supports three types of probabilistic models:

- A Discrete-Time Markov Chain (DTMC) is probably the simplest probabilistic model and it is defined as a tuple $(S, s_0, \mathcal{P})$ with a set S of states, an initial state $s_0 \in S$ and a transition probability matrix $\mathcal{P} : S \times S \rightarrow [0, 1]$, where $\mathcal{P}(s, s')$ is the probability of making a transition from one state s to another state s'. The sum of the probabilities from state s must equal 1, i.e. $\sum_{s' \in S} \mathcal{P}(s, s') = 1$ for all $s \in S$ [31, 42]. In DTMCs, time is modelled as discrete time steps and probabilities are also discrete [21].

- A Markov Decision Process (MDP) extends a DTMC by allowing both probabilistic and non-deterministic behaviour: in any state there is a non-deterministic choice between a number of discrete probability distributions over states [31]. MDPs allow for modelling of asynchronous parallel systems.

- A Continuous-Time Markov Chain (CTMC) does not support non-determinism, but time is modelled in a continuous fashion through the use of the negative exponential distribution [21]. It is defined as a tuple $(S, s_0, \mathcal{R})$ with a set of states S, an initial state $s_0 \in S$ and a transition rate matrix $\mathcal{R} : S \times S \rightarrow \mathbb{R}_{\geq 0}$ where $\mathcal{R}(s, s')$ is the rate of making a transition from state s to s'. The probability of moving from s to s' within t time units is $1 - e^{-\mathcal{R}(s,s') \cdot t}$ [31, 42].

PRISM's model specification language augments these models with *costs* and *rewards*, which are real values that can be assigned to states and transitions. This allows for reasoning about quantitative measures like for instance 'energy consumption' or 'number of messages lost' [21]. The modelling language is a simple, state-based language based on the Reactive Modules formalism [1]. The property specification language for PRISM is based on the temporal logics Probabilistic Computation Tree Logic (PCTL) and Continuous Stochastic Logic (CSL). The former is a probabilistic extension of CTL used in the context of DTMCs and MDPs, whereas the latter is used in the context of CTMCs. We refer to the literature for details about these logics (e.g. [31, 42]).

Explicit vs. Symbolic Model Checking

In contrast to *explicit* model checkers such as for instance SPIN, PRISM is a *symbolic* model checker. Tools in the first category explicitly generate and explore the states of the model under consideration. However, this often results in a combinatorial explosion (i.e. the state space explosion problem, described in chapter 2). Symbolic methods try to avoid this problem by using a symbolic representation for the state space. All possible system states are encoded using a set of boolean variables and a specific set of states is identified by a propositional formula over these variables. A common data structure used to represent these sets of states is (some form of) a Binary Decision Diagram (BDD), since each formula has its unique BDD representation. The performance of symbolic model checking methods depends critically on the variable ordering that is chosen for the BDDs, but since this is an NP-complete problem, there are only heuristics to find a good ordering. Originally, explicit techniques were used mostly in software verification settings, whilst symbolic techniques were found mainly in the hardware domain. We refer to the literature for more information about explicit vs. symbolic model checking and for the details of symbolic techniques [3, 22, 31, 44].

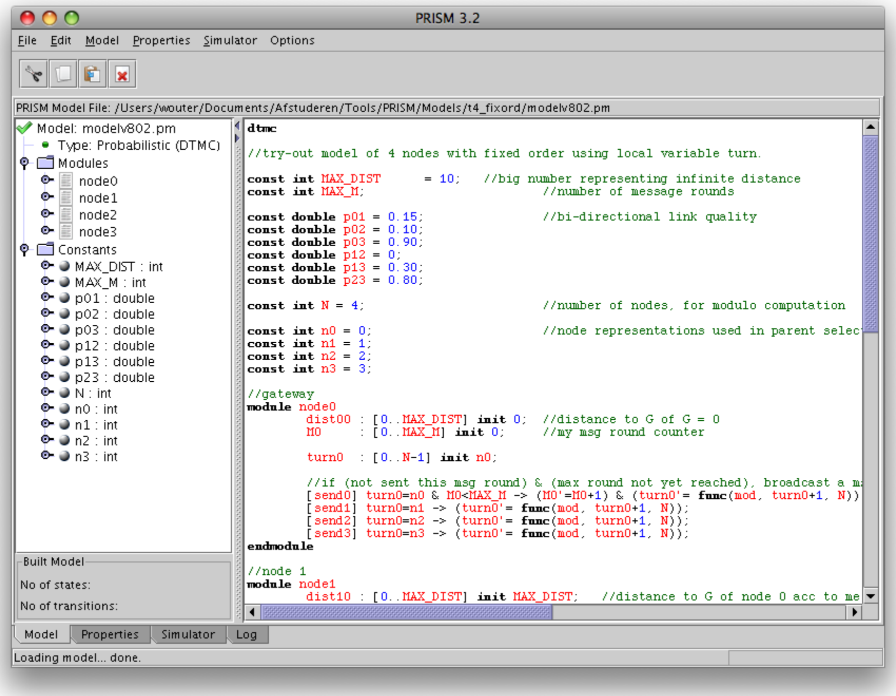
7.1.2 Tool Motivation

Using both UPPAAL and SPIN, we found that modelling link qualities (i.e. probabilities) is a complex rather than a straightforward job. We used (semi[1]) non-deterministic behaviour to accomplish an approximation of the desired stochastic behaviour. However, these tools are in principle not really suitable to model probabilism. In contrast, a tool that actually is very suitable for modelling probabilities is PRISM. It is designed to be a *probabilistic* model checker, allowing the verification of systems which exhibit stochastic behaviour.

Like UPPAAL and SPIN, PRISM is a mature tool which has been used successfully to analyse a wide range of case studies in many different application domains. According to the PRISM website, examples include randomized distributed algorithms such as the randomized consensus protocol, communication protocols (e.g. Bluetooth device discovery, IEEE 1394 FireWire root contention, Gossip protocol), security, biological process modelling and more.

We used PRISM version 3.2, dated June 15, 2008, compiled from source code on Mac OS. Figure 7.1 is a screen shot of PRISM's GUI, to get a first impression. The hardware used is a 13.3" MacBook (Intel Core2Duo processor on 2 GHz, 2 GB RAM) running Mac OS X 10.5 Leopard.

[1]Our implementations so far were *semi* non-deterministic since there is a recurring period of time (message rounds) over which link qualities are maintained, based on history.



```
dtmc

//try-out model of 4 nodes with fixed order using local variable turn.

const int MAX_DIST      = 10;   //big number representing infinite distance
const int MAX_M;                        //number of message rounds

const double p01 = 0.15;                //bi-directional link quality
const double p02 = 0.10;
const double p03 = 0.90;
const double p12 = 0;
const double p13 = 0.30;
const double p23 = 0.80;

const int N = 4;                        //number of nodes, for modulo computation

const int n0 = 0;                       //node representations used in parent selec
const int n1 = 1;
const int n2 = 2;
const int n3 = 3;

//gateway
module node0
        dist00 : [0..MAX_DIST] init 0;  //distance to G of G = 0
        M0     : [0..MAX_M] init 0;     //my msg round counter

        turn0  : [0..N-1] init n0;

        //if (not sent this msg round) & (max round not yet reached), broadcast a m
        [send0] turn0=n0 & M0<MAX_M -> (M0'=M0+1) & (turn0'= func(mod, turn0+1, N))
        [send1] turn0=n1 -> (turn0'= func(mod, turn0+1, N));
        [send2] turn0=n2 -> (turn0'= func(mod, turn0+1, N));
        [send3] turn0=n3 -> (turn0'= func(mod, turn0+1, N));
endmodule

//node 1
module node1
        dist10 : [0..MAX_DIST] init MAX_DIST;   //distance to G of node 0 acc to me
```

Figure 7.1: The GUI of PRISM 3.2 in edit mode

7.2 Model Construction

This section reports on our modelling activities using PRISM. We describe the process we went through and point out the difficulties encountered.

A PRISM model is specified textually using a simple, state-based language, based on the Reactive Modules formalism of Alur and Henzinger [1]. A system is modelled as the parallel composition of a set of *modules* that interact with each other. A module consists of a set of local, finite-range *variables*, together with a set of probabilistic guarded *commands* [21]. The values of the local variables of a module determine the *local state* of the module. The *global state* of the entire model comprises the local state of all modules. The behaviour of a module is described by specifying *probabilistic guarded commands*, which take the form:

```
[] guard -> prob1 : update1 + ... + probN : updateN;
```

A *guard* is a predicate over *all* variables in the model (a module can read other module's variables). An *update* is a change of the module's variable values (i.e. a state transition), associated with a certain probability. A module can make a transition (execute an update) if the corresponding guard is true (and based on the associated probabilities). The type of a model can be DTMC, MDP or CTMC (denoted resp. `dtmc`, `mdp` and `ctmc`) and is specified typically at the start of the file. For the former two types, probabilities on the right hand side of a command must sum up to one.

Each state transition of a module can be synchronized with state transitions of other modules using *synchronization labels*. Such a label is placed left to the guard of a command, inside the square brackets. This allows for forcing two or more modules to make transitions simultaneously.

7.2.1 Protocol Model

Using UPPAAL and SPIN, we were able to build rather generic models: the topology and the number of nodes are parameters of the model. We were able to do so, since both modelling languages support advanced concepts such as process types, process instantiation and process parameterization. Unfortunately, PRISM's modelling language is rather restricted: for instance basic arrays are not supported, nor is module instantiation and parameterization. However, PRISM provides a mechanism called *module renaming*. This allows for module duplication by renaming all variables and labels specified in a module. Module renaming is however less powerful than module parameterization. Therefore, the PRISM model of our protocol is far less generic compared to our UPPAAL or SPIN models: the topology cannot be adjusted in a flexible and convenient way.

```
//nondeterminism in execution order: mdp
mdp

//number of message rounds
const int MAX_M;

//big number representing infinite distance
const int MAX_DIST  = 10000;

//bidirectional link qualities
const double p01 = 0.15;
const double p02 = 0.1;
const double p03 = 0.9;
const double p12 = 0;
const double p13 = 0.33;
const double p23 = 0.8;

//node representations used for parent selection
const int n0 = 0;
const int n1 = 1;
const int n2 = 2;
const int n3 = 3;

//distance to G of G is always 0
const int dist00 = 0;
```

Listing 7.1: PRISM Model V1 - Global Constants

Listing 7.1 shows the first part of our PRISM model. It consists merely of the definition of global constants for the maximum number of message rounds, maximum distance, link probabilities, auxiliary node representations and the distance to the gateway of the gateway itself (which obviously equals 0). Additionally, it defines the model type to be an MDP since, as we will see soon, this is a non-deterministic discrete-time model.

```
//gateway node
module node0
    M0     : [0..MAX_M] init 0;             //my msg round counter

    //if (not sent this msg round) & (max round not yet reached),
    //broadcast a msg (by sync) and increase msg round nr
    [send0] (M0<=M1 & M0<=M2 & M0<=M3) & (M0<MAX_M)  -> (M0'=M0+1);
endmodule
```

Listing 7.2: PRISM Model V1 - Gateway Module

Listing 7.2 shows the PRISM module that models a gateway node. The state of this module is determined by the local variable `M0`, which represents a local message round counter. Its value range is declared to be `[0..MAX_M]` and its initial value is 0. The gateway module has just one command that synchronizes with other modules on the label `send0`. Message broadcast is modelled by synchronization, the corresponding label indicates the sending node. Before explaining this in more detail, let us first turn to the node module.

```
//node 1
module node1
    dist10 : [0..MAX_DIST] init MAX_DIST;   //distance to G of node 0 acc to me
    dist11 : [0..MAX_DIST] init MAX_DIST;   //my own distance to G
    dist12 : [0..MAX_DIST] init MAX_DIST;   //distance to G of node 2 acc to me
    dist13 : [0..MAX_DIST] init MAX_DIST;   //distance to G of node 3 acc to me

    recv10 : [1..1+MAX_M*10] init 1;        //#msgs I received from node 0
    recv12 : [1..1+MAX_M*10] init 1;        //#msgs I received from node 2
    recv13 : [1..1+MAX_M*10] init 1;        //#msgs I received from node 3

    parent1: [0..3];                        //the parent node I selected

    M1     : [0..MAX_M] init 0;             //my msg round counter

    //if (not sent this msg round) & (max round not yet reached),
    //broadcast a msg (by sync), update M1, dist-to-G and parent
    [send1] (M1<=M0 & M1<=M2 & M1<=M3) & (M1<MAX_M) ->
            (M1'=M1+1) & (dist11'=computedist1) & (parent1'=setparent1);

    //receive (sync) msg from node 0 with probability p01
    [send0] true -> p01: (dist10'=dist00) & (recv10'=recv10+10) + (1-p01): true;

    //receive (sync) msg from node 2 with probability p12
    [send2] true -> p12: (dist12'=dist22) & (recv12'=recv12+10) + (1-p12): true;

    //receive (sync) msg from node 3 with probability p13
    [send3] true -> p13: (dist13'=dist33) & (recv13'=recv13+10) + (1-p13): true;
endmodule
```

Listing 7.3: PRISM Model V1 - Node Module

The model we present here is a model of a specific 4-node topology with one gateway node. The node module printed in listing 7.3 represents one of the three remaining 'ordinary' nodes. Its local state comprises four distance-to-G variables (one for each node) and three receive counters (to count probe messages from every possible other node). Note that a node module in a model of a 5-node topology would have five distance variables and four receive counters: this illustrates our point that our PRISM models are less generic than our UPPAAL or SPIN models due to the absence of basic arrays. The state of the node module further comprises a variable for the selected parent and, like the gateway module, a local message round counter.

Since this is a model of four nodes, each (non-gateway) node contains four commands labelled with four synchronization labels (`send0` – `send3`). The first command models the *broadcast* of a probe message by the node itself. The other commands are for *receiving* a probe message of other nodes. We explain these two types of commands below.

Broadcast – The event of `node1` broadcasting a message is modelled by the synchronization of all nodes on the label `send1`. Of course, `node1` can only broadcast if it did not already do so this message round and if it did not yet reach `MAX_M` (the guard on line 51, lst. 7.3). If the guard is satisfied, `node1` updates its message round counter, its distance-to-G and its selected parent (line 52, lst. 7.3). These updates happen with a probability of one, since no probability is specified.

Receive – Theoretically, `node1` can receive messages from all other nodes (including the gateway). Therefore, the event of receiving a probe message by `node1` is modelled by the module synchronization on labels `send0`, `send2` and `send3`. After all, these labels indicate the broadcast of a probe message by the corresponding nodes. Receiving a probe message boils down to storing the message content (distance-to-G of the sender) and updating a receive counter. This is where link quality comes into play: the relevant updates are performed with the corresponding probability. The only condition to be satisfied is the possibility for synchronization, so the guard is always `true`.

Note that in principle we model a complete topology, since all possible node interactions (sending and receiving commands) are present. This turned out to be required for module renaming to be applicable: all node modules must have the same number of commands and variables. We can still model links that are not present by simply setting the corresponding probability to 0. Note also that the gateway module in listing 7.2 is in fact a heavily stripped node module: it cannot receive probe messages (i.e. besides label `send0`, no synchronization with other nodes).

```
63 //define node 2 by module renaming
64 module node2 = node1 [  //send transition:
65             send1=send2, M1=M2, M2=M1, dist11=dist22, parent1=parent2,
66             //recv from node 0 (common neighbour):
67             p01=p02, dist10=dist20, recv10=recv20,
68             //recv from node 3 (common neighbour):
69             p13=p23, dist13=dist23, recv13=recv23,
70             //recv from node 1 (reversed):
71             send2=send1, dist12=dist21, dist22=dist11,recv12=recv21,
72             //parent selection:
73             n2=n1
74 ]endmodule
75
76 //define node 3 by module renaming
77 module node3 = node1 [  //send transition:
78             send1=send3, M1=M3, M3=M1, dist11=dist33, parent1=parent3,
79             //recv from node 0 (common neighbour):
80             p01=p03, dist10=dist30, recv10=recv30,
81             //recv from node 2 (common neighbour):
82             p12=p23, dist12=dist32, recv12=recv32,
83             //recv from node 1 (reversed):
84             send3=send1, dist13=dist31, dist33=dist11,recv13=recv31,
85             //parent selection
86             n3=n1
87 ]endmodule
```

Listing 7.4: PRISM Model V1 - Module Renaming

Listing 7.4 shows the aforementioned module renaming technique. It allows us to define two more modules by renaming all variables and labels in module `node1`. This must be done carefully and in a structured manner since renaming easily results in an incorrect module. Fortunately, PRISM provides a convenient option to view the full parsed model, in which module renaming is already applied. This allows for checking whether the renaming is specified correctly.

```
88 //formula for distance computation
89 formula computedist1 = min(ceil(10*M1/recv10 + dist10), ceil(10*M1/recv12 + dist12),
90                  ceil(10*M1/recv13 + dist13), MAX_DIST);
91 //formula for parent selection
92 formula setparent1 = (computedist1=ceil(10*M1/recv10 + dist10)) ? n0 :
93                  (computedist1=ceil(10*M1/recv12 + dist12)) ? n2 : n3 ;
```

Listing 7.5: PRISM Model V1 - Formulæ

The final part of this model is printed in listing 7.5. It contains a *formula* for distance computation and one for parent selection. Formulæ are used to avoid code duplication: an expression is linked to an identifier, which then can be used throughout the model as shorthand for the expression. According to the PRISM manual, the expansion of formulæ is done prior to the module renaming process such that the formula contents are renamed, not the formula itself.

Distance computation – Lines 89 and 90 of listing 7.5 contain the formula `computedist1` for the calculation of the distance to the gateway of node 1. Note that due to module renaming, this formula also holds for node 2 and node 3. The dist-to-G of a node is a minimum of three alternatives (in this 4-node topology) and a fourth alternative (`MAX_DIST`) to keep the result in the range of the module's distance variable (e.g. `dist11` for node 1). Note the multiplication factor of 10 in the distance computation, which is a workaround to avoid zero dividing. We explain this workaround in detail in section D.2 of appendix D (it also explains the range of the receive counters).

Parent selection – The formula for parent selection on lines 92 and 93 of listing 7.5 makes use of a common shorthand expression for conditional expressions (`test ? iftrue : iffalse`). It determines the parent node for which the least distance was computed by using the formula for distance computation. The node representations such as `n0` and `n2` are required for proper renaming of this formula for the other modules.

During model construction, we simulated our models to validate the behaviour. In appendix D, which is about PRISM, we devoted a section to the simulation process (section D.2). During simulation of an early version of our model, we found a flaw in it and we explain how this is solved (the aforementioned workaround). The first section of appendix D contains two PRISM models (models V2 and V3) that are *based* on the model (V1) that was presented here. This is explained in the next section, which is about verification of our model.

7.3 Verification

This section contains our verification activities using PRISM. Unfortunately, we encountered some problems with our model that prevented us from verifying the model in a useful way. We discuss these problems and their causes below. First, we explain how verification with PRISM works.

Once the model to be verified is specified correctly in PRISM's modelling language, and its global behaviour is validated and debugged using the simulator, the next step towards verification is *building* it. During the building process, PRISM builds a probabilistic model from the model specification, i.e. it computes the set of states (reachable from the initial state(s)) and the transition matrix. An error is reported if any deadlocked states are encountered during the build (i.e. reachable states without outgoing transitions). If this is the case, it can be resolved by having PRISM add self-loops to these states. Once the build process is completed successfully, the model can be verified.

As already mentioned in the first section of this chapter, the properties to be checked for the probabilistic model are specified in variants of probabilistic temporal logics: either PCTL (DTMCs and MDPs) or CSL (CTMCs). PRISM allows for checking *qualitative* properties as well as *quantitative* properties. The former constitutes properties based on reachability analysis (typically resulting in `true` or `false`), while the latter category involves numerical computation (resulting in a computed value, e.g. a probability) [31]. Examples of respectively a qualitative and a quantitative property are:

- `P>=1 [ F terminate ]` – the system eventually terminates with probability 1.
- `P=? [ !mod2ready U mod1ready ]` – the probability that module 1 is ready before module 2 is ready.

The labels above (such as `terminate,mod1ready,mod2ready`) must of course be defined to determine the corresponding set of states. Furthermore, PRISM supports so-called *experiments*, a powerful means for quantitative properties to compute their value for a range of parameters and plot the results.

7.3.1 Build Problems

The protocol model V1 we came up with for PRISM (as presented in this chapter) took a considerable amount of effort and required many intermediate versions. On the one hand, this is a result of the restricted modelling language, on the other hand also from some issues related to verification: building the probabilistic model from our description.

Build Error

At some point we found out that PRISM was unable to construct the specified model. Instead, during the building process an error was reported:

> Error: Probabilities in command 2 of module "node1" sum to less than one (e.g. 0.85) for some states. Perhaps some of the updates give out-of-range values. One possible solution is to strengthen the guard.

It turned out that the guards of the receiving commands (the second, third and fourth command of the node module) are too weak. Indeed, they are always `true`. In theory however, the corresponding commands cannot be executed in the composed system, since they must be executed *synchronously* with a (sending) command (of another module) of which the guard is *not* true. In other words: the synchronization mechanism should prevent that these commands can be executed (and thus preventing the variables to be updated with an out-of-range value). Moreover, this expected theoretical behaviour is exactly what is shown by the simulator, so the given error appeared rather unexpectedly.

The reason that PRISM reports this error is an artefact of the implementation of the build process: before actually constructing the probabilistic model, PRISM first performs a number of checks on each module in isolation (among which the check for variables going out of range). The effects of the parallel composition of the modules are simply not considered. According to PRISM developer Dr. Gethin Norman, this greatly simplifies the checks resulting in a much shorter model construction time. PRISM developer Dr. David Parker even plans to fix this issue in the future[2].

The solution to this problem thus is to 'strengthen the guards', as suggested by the error message. Specifically, the variables that do not pass the out-of-range test are the receive counters that count the received probe messages (`recvxy`): the corresponding updates only result in an increase which causes PRISM to report the out-of-range error, although this never can occur in the composed system (due to synchronization). Originally, (without the workaround for zero dividing, see appendix section D.2), it would be necessary to replace the guards of the receiving commands by the guards `recvxy < MAX_M` (with `x,y` node numbers). However, because of the workaround, this changes in `recvxy < (10*MAX_M-8)`. The updated model is model V2 in appendix D.

Build Time

Once we updated the guards, PRISM was able to build the probabilistic model from our specification. However, for a very basic test specification containing two ordinary nodes (just broadcasting and receiving), the time taken by PRISM to construct the model was around 11 minutes! This is excessively long for such a small and basic model (only 4 states and 5 transitions).

We learned from the PRISM discussion group that this is also related to the way that PRISM (and symbolic model checkers in general) builds models. To cite Dr. Parker: "...PRISM builds the full 'potential' model (including all possible states). It then computes the set of states that are reachable. Finally, it restricts the model to these states. In many cases, this turns out to be a very efficient way to do model construction. In some cases (especially like this, when there is a small number of reachable states in a potentially large model), it is less efficient."

We thus can solve the problem of the very long build time by reducing the 'potential' state space of the model. To achieve that, we adjusted the value of the `MAX_DIST` constant to 10 instead of 10000. This drastically reduces the full potential state space, since this constant is used to define the variable ranges of all distance-to-G variables. Furthermore, we should also keep the value of the `MAX_M` constant as low as possible (since it determines the range

[2]There has been contact with these PRISM developers via the PRISM discussion group at http://groups.google.com/group/prismmodelchecker

of the receive counters). As said, model V2 in the appendix is the updated version of model V1, which was presented in this chapter. It can be build (for small values of `MAX_M`) in reasonable time.

Deterministic Execution Order

In order to reduce the state space of our model, we fixed the execution order of the modules, just like we did in our UPPAAL and SPIN models. In model V1 and V2, the execution order of the modules is non-deterministic (therefore its type is MDP). Model V3 however, which is listed in appendix D, is model V2 but with deterministic (fixed) order of execution of the modules. Therefore, model V3 describes a DTMC.

The fixed order is achieved by adding a local variable `turnx` (where `x` is a node number) to each module. It is used to keep track of the node whose turn it is to broadcast a probe message. All module transitions are synchronized and each transition of each module updates this local `turn` variable. Without the use of a formula to do this, it would result in many duplicate code. Therefore, we added two new formulæ `nextG` and `next` for use in respectively the gateway module and the node modules to update the `turn` variable:

```
//formulae to update turn variable modulo N
formula nextG = func(mod, turn0+1, N);
formula next = func(mod, turn1+1, N);
```

7.3.2 Verification Attempts

The previous section described some issues of the model building process, that led to modifications of our original PRISM model (resulting in model V2 and V3, app. D). The question arises whether these adjusted models are still valid for our protocol and suitable for verification. In this subsection we try to answer this question.

Table 7.1 below provides the statistics resulting from building the models V2 and V3 for various values of the maximum number of message rounds `MAX_M` (i.e. 1, 2, .., 5), using the command line. The models were built from the specification in appendix D: with the topology being the 4-node topology (with unique SPT) that we already saw earlier (fig. 5.6(d) on page 82, with 33% substituted for 18%). The table shows the time taken by the build process, the number of states and transitions in the resulting model and the number of encountered deadlock states. In order to be able to actually verify these models, we need to instruct PRISM to automatically add a self-loop to each of these deadlock states (using the command line switch `-fixdl`).

Table 7.1: Build statistics for models V2 and V3 with `MAX_DIST=10`

Model	`MAX_M`	t(s)	#states	#trans	#deadlock
V2	1	0.61	941	1864	432
V3	1	0.50	185	184	128
V2	2	5.48	73057	245944	21024
V3	2	4.31	11673	15544	6624
V2	3	20.8	2995509	10353868	872317
V3	3	11.6	321563	472952	171676
V2	4	210.7	70530565	279743196	15516072
V3	4	32.1	4833212	9692214	1627488
V2	5	(out-of-memory)			
V3	5	261.2	45887086	97879544	12695345
V3	6	(out-of-memory)			

The table clearly illustrates that building a model (from model specifications V2 and V3) for values of `MAX_M` that are higher than 5 is not feasible on our machine. This means that we are unable to verify the behaviour of the protocol for 5 or more message rounds. Moreover, the `MAX_DIST` constant is set to 10, while it should be a much higher number, now all nodes will advertise a distance-to-G of maximal 10 (even if it is more). Increasing `MAX_DIST` is not an option since that will result in an increasing state space, which leads to a decreasing bound for the number of message rounds that can be verified.

On top of all other problems we encountered during modelling and verification in PRISM, and together with the rather large amount of effort we already invested in it, this result forced us to draw the hardly honourable conclusion that serious verification of the protocol using these PRISM models is not possible.

The third section of appendix D contains two example properties (a check for deadlock freedom and one for correct parent selection), for illustrative purposes. The first property could be successfully verified for model V2 with `MAX_M=4` (built in approx. 210 seconds, table 7.1) in about 3.2 seconds. The verification of the second property did not finish within 15 minutes (for the same model).

7.4 Conclusion

In this section we summarize our experiences, problems and results encountered in this chapter. We will first enumerate our experiences with PRISM and its usability. Next, a number of conclusions about the PRISM models are enumerated together with the main results of our verification activities. We finalize this chapter with some concluding remarks about the protocol.

7.4.1 PRISM Experiences

Coming from UPPAAL and SPIN, modelling for PRISM is different. PRISM's modelling language is simple but far less expressive. Moreover, PRISM is a symbolic model checker and this has consequences for the model building process. Once the model is specified, the entire model must be built which may take long (the complete potential state space is generated, i.e. all possible variable valuations). PRISM's GUI on the other hand, is well-designed and intuitive.

Model Construction

- A PRISM model is written in PRISM's modelling language: a simple, state-based language based on the Reactive Modules principle of Alur and Henzinger [1].
- A PRISM model is either a DTMC, an MDP or a CTMC and it comprises constants, variables and modules that may operate synchronized.
- The PRISM language strongly focusses on modelling rather than on computation (even more than SPIN's PROMELA).
- PRISM's modelling language is rather primitive and restricted: basic arrays are not supported, nor is module instantiation, nor is module parameterization. This simplicity is inconvenient while modelling.
- The module renaming technique provided by PRISM can be used to avoid duplicate code, but it is rather primitive (based on a simple text replace). Renaming turns out a labouring and rather error-prone process for the modeller, especially if a module comprises many variables and labels. In these situations, the 'parsed model viewer' provided by PRISM can thus be of great help.
- As said, PRISM's GUI is well-designed, intuitive and it feels robust. The GUI's model editor immediately provides feedback on the constructed model by code colouring and instant parsing.

Simulation

- PRISM provides a highly usable simulator for debugging the model's behaviour.
- It supports interactive (manual) and automatic simulation (both single step and multiple steps), as well as backtracking the simulation trace.
- It is not necessary to build the model to simulate its behaviour.
- A generated simulation path can be exported to a text file.
- Prior to simulation, PRISM asks for the value of constants that are left undefined in the model.

Verification

- A PRISM model can be verified once it has been built successfully from its specification.
- During the building process, PRISM computes the reachable set of states from the initial state(s) of the probabilistic model.
- Prior to building, PRISM performs some checks on the model. However, these checks are performed on each module in isolation (module synchronization in the composed system is not taken into account).
- During the building process, PRISM first computes the full potential state space (i.e. all possible variable valuations), after which it computes the reachable states. This is often very efficient, but in some cases it is less efficient.
- PRISM allows for checking both quantitative and qualitative properties.
- Prior to verification, PRISM asks for the value of constants that are left undefined in the model.
- PRISM supports so-called experiments that allow for computing and plotting values for quantitative properties, for a range of model parameters (i.e. undefined constants).

7.4.2 The Model

Unfortunately and despite our effort, we were unable to construct a useful protocol model for comparable verification with UPPAAL and SPIN. The process does however provide valuable insight into modelling and simulation using PRISM. Below we enumerate observations and conclusions about our PRISM models.

- Because of the restricted modelling language, our PRISM models are far less generic, compared to the UPPAAL and SPIN models: nor the number of nodes, nor the topology that is modelled in a particular model specification, can be adjusted in a flexible way.
- It is rather hard to express distance computation and parent selection efficiently. It required us a work-around (described in appendix D) to prevent zero-dividing.
- Link qualities were easily modelled using probabilities (as expected).
- Model V1 is a 4-node model with one gateway. It is an MDP since the execution order of the node modules is non-deterministic. This model could not be built because of incorrect guards and variable ranges that were too large.
- Model V2 is an updated version of model V1 that can be built (stronger guards and smaller variable ranges).
- Model V3 is an updated version of model V2: we predefined the execution order of the modules, thereby turning it into a DTMC.

The models could be built for at most 4 nodes, `MAX_DIST=10` and `MAX_M=5`. Verification will thus yield useless results since at most five message rounds are examined. We learned from previous chapters that this is not enough for a node to compute a reliable distance to the gateway. A simple qualitative property could be successfully verified but the verification of a quantitative property did not end within 15 minutes (appendix D), even for such a relatively small model.

7.4.3 The SPT Protocol

Compared to model construction for SPIN or UPPAAL, constructing a model of the SPT protocol for PRISM was significantly harder and more time-consuming. Modelling and verification of our protocol using PRISM required most effort. Based on the results described in this chapter, we cannot conclude anything about our protocol.

CHAPTER 8

Variants and Experiments

As stated in the first chapter, the objective of this research project is to take the first essential steps that are required in the process of developing a platform or methodology for formal verification experiments with Wireless Sensor Network (WSN) protocols. This chapter presents some examples of concrete experiments, using our SPT protocol.

In the first section below, we explain why there is a need to do formal verification experiments with WSN protocols. In the subsequent sections we elaborate on the variants of our protocol we experimented with. We finalize this chapter, as usual, with a summarizing section with conclusions.

8.1 Verification Experiments

As stated in the problem statement in chapter 1, WSN protocols are rather complex because of the underlying characteristics of these networks (e.g. unreliable links, scarce resources like energy, memory, computational power). Therefore, it is often the case that a 'mathematical provable' protocol (proven to be correct) cannot be implemented directly in practice. This is the result of all kinds of assumptions (i.e. abstractions from implementation details), which were required in order to come to a feasible proof.

This is also the foundation of the need for formal verification experiments with WSN protocols: the idea is to model the strongly idealized ('provable') protocol and use that as starting point for adding more realistic details. The behaviour of the more realistic variants obtained in that way can be verified using model checking experiments. This may eliminate the need for a mathematical proof of correctness of these variants.

Another useful application of verification experiments would be the ability to easily and rather quickly validate changes in the protocol. This would for example enable a protocol designer with a revolutionary idea for improving a protocol to rather quickly check his or her brilliance (and, for example, to rule out some remaining uncertainties). Model checking experiments thus can be a useful tool to support the design of WSN protocols.

Sometimes, the underlying characteristics of a protocol may give rise to certain questions that may be answered using experiments. In our case for example, the SPT protocol is a routing protocol that has to deal with the underlying link probabilities of the network. Questions that may arise are: does the parent selection process of the nodes ever result in cycles? Is it possible for the gateway to become disconnected from the rest of the network? This would of course be highly undesirable.

8.1.1 Experimentation

The upcoming sections describe some experiments that we did, in order to show the use of formal experimentation. The structure is as follows: we start explaining the context and the motivation of some issue(s) concerning our SPT protocol. This gives rise to a cohesive set of experiments which we describe by enumerating five elements:

1. Hypothesis – a formulation of our hypothesis about the issue(s), giving rise to the experiments;
2. Motivation – a summary of the motivation for the hypothesis;
3. Setup – a description of the setup of the verification, since we require our experiments to be transparent and repeatable;
4. Results – the tabulated results of the experiments;
5. Conclusion – a conclusion based on the results, about the stated hypothesis.

Section 8.2 describes experiments with the original SPIN model of the SPT protocol, in order to investigate whether a disconnected gateway can occur. Next, subsections 8.3.1 and 8.3.2 each report about a variant of the protocol, for which we had to adjust the model. Both variants resolve an impractical aspect of the protocol as presented in chapter 3. We searched for interesting situations during verification of these variants, in particular (temporary) cycle occurrence and instability or fluctuation.

8.2 Disconnected Gateway and Parent Cycles

The SPT protocol is a routing protocol: the nodes strive to build a shortest path tree rooted at the gateway, based on received information from periodically broadcasted probe messages. The main problem here is the unreliable stochastic nature of the underlying wireless links. This unreliability gives rise to the question whether the entire network is always connected. A disconnected gateway node for example, is highly undesirable since the sensor data that should be forwarded to it (for later processing) will just cycle through the network without getting anywhere.

A general example topology on four nodes that paves the way for a disconnected gateway scenario is the one given in figure 8.1.

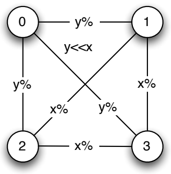


Figure 8.1: General topology for illustrating the disconnected gateway scenario

This topology is completely connected but the important thing is that $y << x$: the links between the gateway (node 0) and the rest of the nodes have quality y, which is significantly less than the quality x of the rest of the links. An occasional scenario that may occur is the following: suppose that the probe message broadcasted by the gateway in the very first message round luckily comes across (and gets at one ore more of the other nodes). As the links between the gateway and nodes 1, 2 and 3 have a very low quality, it is very likely that these nodes do not receive anything from the gateway in the next message rounds. It is however very likely that they do hear from each other, since they are mutually well-connected. Moreover, they will advertise to have an attractive distance to the gateway. Therefore, nodes 1, 2 and 3 may select each other as parent and, by doing so, create a parent loop. The gateway may thus become disconnected: there is no node that selected the gateway as parent.

8.2.1 Disconnected Gateway Experiments

1. Hypothesis – Situations in which the gateway is disconnected (i.e. in which there is no node that selected the gateway as parent) do occur with the SPT protocol.
2. Motivation – These situations are a direct consequence of the probabilistic nature of the wireless links in combination with the protocol

characteristics, as explained in the scenario above.

3. Setup – The following Linear Temporal Logic (LTL) property is used to check whether a disconnected gateway scenario occurs:

   - `[] p`
   - `#define p nodes[1].parent==0 || nodes[2].parent==0 || nodes[3].parent==0`
   - “It is always the case that at least one of the non-gateway nodes selected the gateway to be its parent”.

   We would also like to check whether the gateway becomes connected again. In other words: verify whether the disconnected gateway scenario is temporary or not. The following LTL property enables us to do so:

   - `[]<> p`
   - “It is always the case that *eventually* `p` holds”, with `p` as defined above.

   We verify the topology on four nodes, depicted in figure 8.1, with link quality parameters $x = \frac{9}{10}$ and $y = \frac{1}{10}$.

   We check the properties for our original SPIN model (chapter 5) and for the model with ‘controlled branching’ (by counting the number of delivered probe messages), that is model BCC20 from chapter 6, adjusted to the topo from figure 8.1. Model constant MAX_M=20, other constants are set to their defaults.

   Hardware, version and verification settings of SPIN are as mentioned in chapter 5, page 80. We report the given counterexample if properties are violated, or just `false`. Otherwise we report `true`.

4. Results – Table 8.1 displays the results. The counterexamples are abbreviated: `n1.p` denotes the parent selected by node 1. Both counterexamples are equal and occur in message round 14 (`M=13`): nodes 1 and 2 selected parent node 3 and node 3 selected parent node 2.

Table 8.1: Disconnected Gateway Experiments - results

Model:	`[] p`	`[]<> p`
ORIG	`M=13: n1.p=n2.p=3, n3.p=2`	`true`
BCC20	`M=13: n1.p=n2.p=3, n3.p=2`	`false`

5. CONCLUSION – the hypothesis stated that a disconnected gateway situation is possible, and this is confirmed by our experiments. Moreover, this situation is only temporary in the original model. This in contrast to the model with branch control: SPIN finds a path in the state space on which the gateway does not become connected again! Both models simulated 20 message rounds, so now the impact of the branch controlled model becomes clear: in the original model, only one instance of probabilistic link behaviour is exposed to the protocol part, in which (luckily) the gateway becomes connected again. In the branch controlled model, more instances of probabilistic behaviour are exposed to the protocol part, resulting in a path that violates the property. This violation is however a result of the relatively small number of simulated message rounds (i.e. 20).

8.2.2 Notes on Parent Cycles

In a disconnected gateway situation, there is no node that selected the gateway as its parent. All nodes must have selected some parent so there must be a cycle: for example node x_1 has parent x_2, which has parent x_i, $\ldots$, which has parent x_1 again. The presence of such cycles is of course highly undesirable as the purpose of the protocol is to enable the nodes to route sensor data messages to the gateway. However, such situations might be acceptable if we would know that the parent cycles are only temporary.

Of course, parent cycles of ordinary nodes might also occur when the gateway is connected. In theory, such cycles can comprise an arbitrary number of nodes, up to the number of non-gateway nodes in the network. Since there are many possible instances of these cycles, it is hard to check for their absence in a general way. It is however perfectly possible to check for instance the absence of all possible parent cycles of two (non-gateway) nodes in a topology of N nodes and a single gateway (there are at most $\frac{1}{2}(N-1)(N-2)$ such cycles).

As said, the temporary occurrence of parent cycles might be acceptable, but if cycles occur too frequently or persist long enough, network performance will degrade drastically. Adding extra nodes (ordinary nodes or even extra gateway nodes) to the network might be a solution to this problem. Model checking experiments might be helpful to determine parts of the network that are likely to induce parent cycles, such that extra nodes can be added there, in order to decrease the probability of parent cycles occurrence. This topic might be interesting for future research.

8.3 Infinite Memory Assumption

As often is the case with WSN protocols, the SPT protocol presented in chapter 3 is idealized in the sense that it is based on some important assumptions. These assumptions were made to be able to mathematically prove its correctness. For example, it assumes symmetric link qualities. Another important assumption, that we are going to concentrate on in this section, is the infinite memory assumption: our SPT protocol disregards the restricted amount of memory available to a sensor node. This is indicated by the unbounded number of executed message rounds and by the unbounded number of neighbours about which information is stored. Both aspects are discussed in the subsequent subsections.

8.3.1 Finite Sliding Window Variants

The SPT protocol assumes an infinite window of message rounds over which it computes (estimates) link quality. In reality however, it is necessary that the window is finite due to memory constraints of the nodes. Moreover, the protocol should adapt to potential changes in the topology quickly and this is bothered by endlessly probing the link quality. An approach to solve this is by making the window finite and thus introducing a *sliding window.*

The SPT protocol as described in chapter 3 assumes an infinite window: the distance to the gateway is computed based on all data perceived *ever*, i.e. from the very moment that the network became operational (fig. 3.7, page 36). Therefore the nodes need to store at least the (unbounded) message round number and the (unbounded) number of received messages per neighbour. This is of course not feasible in a real sensor network, since there will be a time that the nodes run out of memory (especially sensor network nodes, because of their resource constraints).

In our SPIN model of the protocol, we had to restrict this 'infinity' by defining constant MAX_M for the maximum number of message rounds to be executed. This is required to be sure that the verification terminates. By setting this constant to a large number, we could simulate an infinite window of message rounds over which distance is computed. In fact, the nodes compute a cumulative moving average of the link quality (over an ever increasing window).

We would like to modify the protocol such that this source of infiniteness is eliminated, since that would be one step closer to a practical implementation. The idea is too fix the window size over which link quality is computed: nodes will compute a simple moving average of the link quality over a *sliding* window of a fixed number of message rounds. This means that old enough data is simply discarded. If we modify our original mode as described, we can easily perform all kinds of verification experiments: does the protocol still function correct? Do nodes find correct parents? Do nodes find correct distances-to-G? What is a good size for the sliding window? What happens if the window is too small or too big? Can we think of and implement other strategies to estimate link quality?

This last question might also be interesting w.r.t. topology changes: how fast does the protocol adapt to these changes? Instead of the simple moving average of link quality over a sliding window, we can of course also try to use a Weighted Moving Average (WMA) or even an Exponential Weighted Moving Average (EWMA). Below we concisely discuss the basic changes to our SPIN model in order to implement a finite sliding window with a simple Moving Average (MA). The full Process Meta Language (PROMELA) sources of that model, as well as our models with WMA and EWMA, can be found in appendix E.

Model Changes

The changes to our SPIN model start with renaming constant MAX_M to WIN_SIZE, the number of message rounds in a window. We also add a new constant MAX_WIN to hold the maximum number of executed windows (similar to the function of MAX_M: for termination of verification). Further, we added a global variable to count the number of passed windows[1] (l.13). The changes can be found in listing 8.1:

```
 4  /* DEFINE CONSTANTS */
 5  #define N              4          /* Number of nodes */
 6  #define WIN_SIZE       10         /* Window size in msg rounds */
 7  #define MAX_WIN        10         /* Max number of windows */
 8  #define MAX_DIST       10000      /* Represents 'infinite' distance */
 9  #define ACCURACY       10         /* Multiplication factor */
10  #define GATEWAY_COUNT  1          /* Number of gateways */
11
12  /* TYPEDEFS & DECLARATIONS */
13  local byte WIN_CNT = 0;           /* Counts the nr of passed windows */
14
15  typedef NodeData{                 /* Node data:*/
16      byte parent = 0;              /* - selected parent */
17      byte R[N] ;                   /* - counts received msgs (per node) */
18      chan W[N] = [WIN_SIZE] of {bool};  /* - most recent window with recv info (per node) */
19      short D[N] = MAX_DIST;        /* - dist-to-G (per node) */
20  }
21
22  NodeData nodes[N];                /* All node data */
```

Listing 8.1: Changed constants and declarations

To realize a finite sliding window of message rounds, each node has to maintain information about which messages it received in the most recent window (and from which nodes). In other words: instead of the information

[1]The PROMELA keyword local indicates that this global variable is accessed by only one process: SPIN's Partial Order Reduction (POR) algorithm can take advantage thereof.

whether or not it received a message in the current message round (Boolean matrix Msgs), a node now needs this information for the WIN_SIZE most recent message rounds. In fact, the Msgs matrix needs a third dimension of size WIN_SIZE. Since we found it more natural to store this information at the nodes, we added an array of *buffered message channels* (or *queues*) to the NodeData type definition (l.18). These channels are of capacity WIN_SIZE and function here like first-in-first-out queues of Booleans (i.e. the Booleans from the matrix Msgs).

Listing 8.2 shows the changed inline construct receive(id). It updates the received messages counter (l.53) and appends the 'new info' to the queue (l.55). 'New info' corresponds to whether or not this node (id) received a message from sender k this message round (the corresponding Boolean entry from the Msgs matrix). Note that the received messages counter just contains the number of 1's in the queue, i.e. the number of received messages in the last MAX_WIN message rounds.

```
41  inline receive(id){                          /* Update counters if received a msg */
42      atomic{
43          do
44          :: (k < N) ->
45              r = msgs[k].to[id];              /* Msg recvd this msg round? */
46  
47              if
48              :: nfull(nodes[id].W[k]) ->      /* If queue is not full yet */
49                  x=0
50              :: full(nodes[id].W[k]) ->       /* If queue is full */
51                  nodes[id].W[k]?x;            /* Get oldest msg round info */
52              fi;
53              nodes[id].R[k] = r-x+nodes[id].R[k];   /* Update counter */
54  
55              nodes[id].W[k]!r;                /* Append most recent info to queue */
56  
57              if
58              :: r -> if                       /* Update perceived distance if recvd */
59                  :: isGateway(k)-> nodes[id].D[k] = 0    /* dist-to-G of G is always 0 */
60                  :: else -> nodes[id].D[k]=nodes[k].D[k]
61                  fi
62              :: else -> skip
63              fi;
64  
65              k++
66          :: (k==N) -> k=0; r=0; break
67          od
68      }
69  }
```

Listing 8.2: Changed inline construct receive()

Of course, the distance computation is slightly changed too: instead of the message round number M, we should now use the window size. Moreover, if the number of rounds is less than WIN_SIZE (at the start of the protocol), we should use that number instead of M. We use the length of the corresponding queue, since it satisfies these needs (l.78 in listing 8.3).

```
            l = len(nodes[id].W[k]);
            try = (ACCURACY * l / nodes[id].R[k]) + nodes[id].D[k];
            try = ( ((ACCURACY*l)%nodes[id].R[k])>=(nodes[id].R[k]/2) ->(try+1):try );
```

Listing 8.3: Changed distance computation

Finally, the changed part of the globalSend process is rather straightforward. It is printed in listing 8.4. If WIN_SIZE message rounds are executed, M is reset and WIN_CNT is increased (line 175). If MAX_WIN windows passed by, the process terminates (line 176).

```
149   do                                              /* For each msg round */
150   :: (M<WIN_SIZE) &&
151      (WIN_CNT<MAX_WIN) -> atomic{
152      do                                           /* Fill msgs matrix */
153      :: (i<N) -> do
154         :: (j < N && C[i].to[j].u == 0) -> j++   /* No link between i and j */
155         :: (j < N && C[i].to[j].u != 0) ->       /* There is a link between i and j */
156            if
157            :: canLoose(i,j) ->                   /* ND-choice: write 0 in msgs matrix */
158               msgs[i].to[j] = 0                  /* 0 means msg gets lost */
159            :: canSend(i,j) ->                    /* ND-choice: write 1 in msgs matrix */
160               msgs[i].to[j] = 1;                 /* 1 means msg will come across */
161               H[i].to[j].u++                     /* Counts nr of delivered msgs */
162            fi;
163            H[i].to[j].v++;                       /* Counts msg opportunities */
164            checkReset(i,j);
165            j++
166         :: (j==N) -> j=0; break
167         od;
168         i++
169      :: (i==N) -> i=0; break
170      od;
171      ctrl = GATEWAY_COUNT;}                       /* Transfer control to 1st node process */
172      ctrl == _pid;                                /* Wait for control */
173      M++
174   :: (M==WIN_SIZE) && (WIN_CNT<MAX_WIN) ->        /* A window passed by, reset */
175      M=0; WIN_CNT++
176   :: (WIN_CNT==MAX_WIN) -> break                  /* Max nr of windows reached, stop */
177   od
```

Listing 8.4: Changed part of the globalSend process

We also constructed a finite window model in which the nodes maintain a WMA of the link quality, rather than the described simple MA. In the latter case, all data is equally important: MA smoothens fluctuations and emphasizes the direction of a trend. In the WMA case, older data is less important compared to more recent data (with linear decreasing weights). Translated to our protocol, this means that results (i.e. whether a message is received or not) from older message rounds weigh less than more recent results. Finally, we also constructed a finite window model that uses some form of a EWMA. As with WMA, older data is less important but the weighting factor (or smoothing factor) decreases exponentially. EWMA is also known as exponential smoothing or discounting. More information about Time Series Analysis can be found in the e-handbook of NIST/SEMATECH [41]. The PROMELA source of these models can be found in appendix E.

Verification Experiments

During modelling, we validated our ideas about the three finite sliding window models using SPIN's simulator mode. We also performed invalid end state checks (deadlock freedom): invalid end states were found in neither of these models (for different values of parameters such as topology, maximum number of rounds, window size, etc.). Moreover, we sporadically verified whether the nodes selected correct parents. They actually did in most cases that we verified, with a sufficiently large window size and number of maximum windows. Because of limited time, we did not structure these experiments and we therefore do not further discuss the results here. Nevertheless, these results were promising and increased our confidence in these models. Instead, we now move to another set of experiments that may answer an interesting question.

The question is: does the finite window variant of the protocol still result in a stable situation? The original protocol converges to a stable situation (i.e. one in which the parent relationships between the nodes form a Shortest Path Tree (SPT) rooted at the gateway). This is mathematically provable by using the so-called Law of Large Numbers (LLN), a well-known theorem from probability theory: given a random variable (link quality) that is periodically sampled (reception of probe messages each message round), it describes the convergency of the sample mean (estimated link quality) to the expected value of the variable, as the number of samples (message rounds) increases.

The problem that is introduced in the finite window variant is that the number of samples stops increasing when it reached the window size. This may result in an estimated link quality that is not close enough to the expected value. Therefore, the estimated link quality may fluctuate, which will cause the nodes to keep changing their selected parent forever: no stable situation is reached anymore.

1. Hypothesis – The finite sliding window may prevent the protocol to reach a stable situation (w.r.t. selected parents), resulting in nodes that keep changing their parent 'forever'.

2. Motivation – In the original protocol, reaching a stable situation w.r.t. selected parents is based on an ever increasing number of probes (samples) of the link quality, such that the sample mean converges to the actual link quality. A finite window variant of the protocol may prevent this convergency, since the number of probes (i.e. the window size) is limited. If the window size is to small, this may result in a fluctuating link quality approximation which causes the nodes to continuously switch parents (parent instability).

3. Setup

- The model under consideration is the finite sliding window model with a simple MA link quality estimator, as described above (source code in first section of appendix E).
- The topology under consideration is our arbitrary topology, depicted again in figure 8.2.

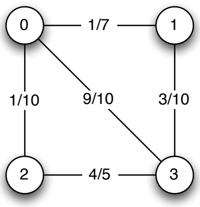


Figure 8.2: Arbitrary topo

- In order to find unstable situations, check the following LTL properties: `[]<>p` (p1) and `[](q->p)` (p2) with:
  - `#define p nodes[1].parent==3 && nodes[2].parent==3`
    `nodes[3].parent==0`
  - `#define q  r > x`
  - `#define r  M + WIN_SIZE*WIN_CNT`
  - 'It is always the case that eventually correct parents are selected' and 'it is always the case that correct parents are selected if the number of passed message rounds `r` is greater than a given value `x`'.

  The first property will tell us whether the protocol is able to select correct parents for different values of model parameters WIN_SIZE and MAX_WIN; the second property reveals the influence of the number of executed message rounds (in fact the number of samples) on parent selection. It is a way to check for convergency of the sample mean.
- Hardware, version and verification settings of SPIN are as mentioned in chapter 5, page 80.
- We report `true` (`t`) or `false` (`f`) for the checked properties, for different values of parameters WIN_SIZE and MAX_WIN.

4. Results – The tabulated results of the experiments can be found in tables 8.2 and 8.3 below.

5. Conclusion – The observations and conclusions following from the results of these experiments are enumerated below:

Table 8.2: Finite Sliding Window Experiments - results for p1

Property 1: $[] <> p$								
MAX_WIN:	3	4	5	6	7	8	30	31
WIN_SIZE=5	f	t	f	t	t	t	t	f
WIN_SIZE=10	t	t	t	t	t	t	t	t

Table 8.3: Finite Sliding Window Experiments - results for p2

Property 2: $[](q- > p)$				
x:	5	10	50	100
WIN_SIZE=5	f	f	f	f
WIN_SIZE=10	f	t	t	t
WIN_SIZE=15	f	f	f	f
WIN_SIZE=20	f	t	t	t

(a) Table 8.2 shows us that in a model with a window size of five message rounds, the result of the checked property (always eventually correct parents selected) depends on the value of constant MAX_WIN. This indicates instability of the parent tree, since the correct stable parent tree was found and is lost later on, while the protocol progresses. This is probably a result of the small window size.

(b) Indeed, table 8.3 shows that in a model with WIN_SIZE=5, the protocol is unable to select correct parents, even after 100 rounds. This indicates that there is no proper convergency towards the expected value possible, when considering only the last five message rounds.

(c) Table 8.2 also shows us what happens in a model with a window size of ten rounds: here the protocol finds correct parents for all values of MAX_WIN in the table. This situation seems stable. Moreover, the results in 8.3 confirm this suspicion, since it is always the case that the protocol finds correct parents if the number of passed message rounds is greater than 10.

(d) The lower two rows of table 8.3 support the idea that stable situations do occur if the window size is a multiple of the largest divider in the connectivity matrix (10 in this case). If this is the case, this would probably be a consequence of the periodicity in our implementation of the probability approximation mechanism of the model.

(e) The results of these experiments make it very plausible that unstable situations, in which nodes keep switching parents, do occur so we consider the hypothesis to be confirmed.

8.3.2 Neighbourhood Management

As said before, the SPT protocol as presented assumes infinite memory. One source of excessive memory use is discussed in the previous subsection: the infinite window of message rounds. There is another, inefficient memory consuming aspect of the protocol: each node maintains information of *all* other nodes. In large, high-density networks this is not desirable or even not feasible. Although this aspect is theoretically speaking finite, we can significantly reduce the memory consumption by restricting the number of neighbouring nodes of which each node maintains information. Moreover, this will also result in a reduction in energy consumption.

Indeed, there is a need for neighbourhood management: nodes should only maintain information about a (carefully) selected set of neighbours, stored in a (fixed-size) neighbour table. Neighbourhood management as such has aspects in common with cache management. The central question is, as stated in [56], "how does a node determine, over time, in which nodes it should invest its limited neighbour table resources to maintain link statistics?" This table is the only place where a node maintains link statistics, so how does a potential neighbour, that is not in the table yet, conquer a place in the table?

Three essential components of neighbourhood management are insertion, reinforcement and eviction [56]:

- *Insertion* – Once a probe message is received from an 'unknown' neighbour (i.e. one that is not in the table yet), it must be determined whether to insert it. This may be done based on for instance geographical information, signal strength associated with the message, a simple statistical method, preference, etc. The insertion policy should avoid a high insertion rate.

- *Reinforcement* – on reception of a probe message from a known neighbour, its corresponding entry in the neighbour table must be reinforced. The specific action that is performed on a table entry that is reinforced depends on the eviction policy.

- *Eviction* – if a new candidate neighbour node is found, which is not in the (fully populated) neighbour table yet, an old entry with neighbour information must be evicted (discarded). There are several policies to do so. One is based on the FREQUENCY algorithm, that keeps a frequency count of all entries [56]: reinforcement increases this count, and every new candidate for insertion in a full table causes the frequency count of all entries to decrease. If the frequency count of an entry becomes 0, the entry is dropped, and the next candidate for insertion can be inserted.

Adding neighbourhood management to our model turned out to be rather challenging. We implemented a simple form of neighbourhood management in our SPIN model of chapter 5. The full source of this variant is contained in appendix E. Here we will concisely discuss the interesting aspects of modelling neighbourhood management.

Minor changes required for neighbourhood management include adding a constant NEIGHBOURS for the number of neighbours in the neighbour table, defining a type Neighbour for containing neighbour data (that is maintained by each node for each neighbour) and modifying the NodeData type definition such that it contains a field for the dist-to-G of a node and a so-called *neighbour table*: an array NBTable of NEIGHBOURS entries of type Neighbour. Also, instead of ranging over all possible nodes, the inline construct getMinimum() (that computes the minimum dist-to-G) now should range over the entries in the neighbour table of the node under consideration.

```
inline manageNeighbour(id, nb){              /* Manage neighbour nb */
    byte c;                                  /* Range over neighbour entries in NBTable */
    byte index;                              /* NBTable index of entry to be updated */

    c=0; index=-1;
    do                                       /* Find an entry to update */
    :: (c<NEIGHBOURS) ->
        if                                   /* If nb in NBTable: set index to reinforce, */
                                             /* or if index not set, to 1st entry with F==0 */
        :: (nodes[id].NBTable[c].ID == nb) || (index==-1 && nodes[id].NBTable[c].F == 0) ->
            index = c
        :: else -> skip
        fi;
        c++
    :: (c==NEIGHBOURS) -> c=0; break
    od;

    if
    :: (index < 0) ->                            /* If table full and no entry with F==0, */
        do                                       /* decrease all frequency counters */
        :: (c < NEIGHBOURS) -> nodes[id].NBTable[c].F--; c++
        :: (c == NEIGHBOURS) -> c=0; break
        od
    :: else ->                                   /* Reinforce or replace */
        nodes[id].NBTable[index].R =
        ((nodes[id].NBTable[index].ID == nb) -> (nodes[id].NBTable[index].R+1) : 1);
        nodes[id].NBTable[index].F =
        ((nodes[id].NBTable[index].ID == nb) -> (nodes[id].NBTable[index].F+1) : 1);
        nodes[id].NBTable[index].ID = nb;        /* Update ID */
        if                                       /* Update dist-to-G */
        :: isGateway(nb) -> nodes[id].NBTable[index].D = 0    /* dist-to-G of G is always 0 */
        :: else -> nodes[id].NBTable[index].D = nodes[nb].D
        fi
    fi;
}
```

Listing 8.5: Neighbourhood Management - inline construct ManageNeighbour()

In order to implement the logic for neighbourhood management, we added an inline construct manageNeighbour() (figure 8.5). This *poor man's procedure* manages the neighbour table by maintaining the policies for insertion, reinforcement and eviction. Every time a node (indicated by parameter id) receives a probe message (from a sender indicated by nb), this construct is executed. Senders are stored in the table until it is full (naive insertion). If a sender is already in the table, its frequency count F is increased (reinforcement). If a sender is not in the table (i.e. its is a potential neighbour) and the table is full, all frequency counters are decreased. An entry with a frequency count of zero is overwritten with the potential neighbour; if there are no such entries, it is discarded. Entries with a frequency count of zero thus will sooner or later be overwritten (eviction).

Yet, this variant should be seen as an initial proof of concept that requires extensive further research. Therefore, and because of limited time, we did not perform structured experiments with this variant. We do know that SPIN does not find any invalid end states for this model. Simulation did however show that correct parent selection depends heavily on the number of neighbours in the table, as well as on the topology under consideration. Parent selection seems correct if the size of the neighbour table equals (not surprisingly) `N−1` and, more promising, `N−2`, with `N` the number of nodes. Moreover, in this case of simple neighbourhood management, it is very likely that the protocol does not converge to a stable situation since the neighbour table contents may change continuously.

8.4 Conclusion

Formal verification experiments with WSN protocols would have several benefits. First, they can answer questions about the protocol or its underlying characteristics. Second, they provide a light-weight method to rather quickly validate changes or additions to the protocol. Third, they may eliminate mathematical proof of less idealized variants of the protocol.

This chapter illustrated the use of verification experiments. We presented a basic template for a cohesive set of experiments, containing a hypothesis and its motivation, the (repeatable) setup for the experiments and their tabulated results, and a conclusion that either confirms or rejects the stated hypothesis.

We experimented with disconnected gateway scenarios, with finite sliding window variants of the protocol and with (modelling) neighbourhood management. The basic template for the experiments turned out to be very useful as it provides a method for scoping and structuring the experiments. Experimentation resulted in interesting conclusions:

- Disconnected gateway situations do occur, but these are only temporary situations.
- Parent cycles do occur.
- Estimating mean link quality over a finite sliding window of message rounds can be done in different ways. We implemented a simple cumulative MA, a WMA and an EWMA.

- Experiments have shown that it is very plausible that unstable situations (in which nodes keep switching parents) do occur in a finite sliding window variant of the protocol.
- Adding neighbourhood management to our model turned out to be rather challenging, mainly due to the implementation of the policies for insertion, reinforcement and eviction.
- We were able to implement a simple form of neighbourhood management, as a proof of concept. Simulation of this variant made it clear that further research is required.

8.4.1 Future Work

Most of the time, performing formal verification experiments (and probably experiments in general) turned out to result in additional interesting questions and more experiments. We summarized some interesting issues below, which we count in for future work due to time limitations.

- Experiments with parent cycles and with potential solutions to the problem (e.g. adding more gateway nodes, adding cycle detection);
- Comparison of finite sliding window variants;
- Experiments with finite sliding window models to find optimal configuration of parameters;
- Experiments with (less naive) neighbourhood management and improving the implementation, to find characteristics of this variant such as for instance stabilization and topology dependency;
- Experiments with infinite models: we always verified a bounded (finite) number of message rounds (or windows), but SPIN may be able to check some properties on unbounded models;
- Experiments with a combination of the finite sliding window variant and the neighbourhood management variant;
- Experiments with a dynamically changing topology (w.r.t convergency to a stable situation);
- Experiments with random node failure, since node failure is a common event in real WSNs;
- Experiments with asymmetric link quality, as the protocol is based on symmetric links.

CHAPTER 9

Conclusion and Future Work

Is formal verification (specifically: Model Checking) suitable for supporting the design of protocols for WSNs?

This chapter contains the conclusions of our research, as well as directions for further research. We first summarize all important results of our research project. Next, we use the results to formulate answers to the sub questions, which were stated in the introduction (chapter 1). The answers to the sub questions together suggest an answer to our main research question, on which we will elaborate subsequently. Finally, we present directions for future research.

9.1 Summary of Results

We took the first steps towards a platform for formal verification experiments for WSN protocols by investigating the feasibility of formal verification of a particular protocol. This section provides an itemized overview of the important results of our research:

- *Modelling Insights* – We gathered valuable insights during model construction:
  - Insight into problematic aspects of modelling the SPT protocol, such as link quality (probabilism), message broadcast and distance computation;
  - Insight into the complexity of the model of the SPT protocol and in complexity reduction techniques, such as fixed execution order of the nodes (ch. 4) and controlled branching (ch. 6);

  - Insight into model construction for the selected tools, such as underlying theory, flexibility of modelling languages and best practices.

- *Verification Insights* – We gathered valuable insights during model verification:
  - Checking for correct parent selection is preferable above checking for correct distance computation since correct parents are found after fewer message rounds;
  - The feasibility of a verification run depends heavily on the topology under consideration (number of nodes and links and link quality);
  - The verification process has many input parameters and this results in an instance explosion which makes it infeasible to verify all possibilities.

- *Verification Boundaries* – Both UPPAAL (model V2) and SPIN are capable to verify a completely connected 4-node topology with all links 10%, for a limited number of message rounds. However, probability approximation in the SPIN model is better. Feasibility can be tuned by changing the number of links between the nodes or by changing the quality of the links, such that verification of certain topologies of 5 nodes might be feasible as well. Meaningful verification of a PRISM model turned out to be infeasible.

- *Confidence in the SPT Protocol* – No errors were found during verification of our models.

- *Protocol Description* – We provided a clear description of the SPT protocol by specifying both an informal and a formal description (ch. 3).

- *Concrete Models and Properties* – The construction of models of the SPT protocol required non-trivial effort, which resulted in a number of concrete models for UPPAAL, SPIN and PRISM, contained in the appendices of this thesis (together with the associated correctness properties).

- *Tool Experience* – The concluding sections of chapters 4, 5 and 7 contain experiences with the capabilities and properties of the corresponding tool w.r.t. modelling, simulation and verification.

9.2 Conclusions

In this section we draw conclusions by answering the research questions that were stated in the introduction. We start with the sub questions after which we answer the main research question.

9.2.1 Sub Questions

1. What are the boundaries, problems and experiences of modelling the protocol?

The *boundaries* of modelling the SPT protocol are determined by the expressiveness of the modelling language. The UPPAAL model required least effort, followed by the SPIN model. The PRISM model required most effort. This correlates with the degree of expressiveness of the respective languages. Modelling languages are optimized for specification, rather than for computation. Therefore, the same issues recur in different models.

Problems encountered during modelling were related to modelling message broadcast (mainly with SPIN and PRISM), modelling probabilistic link behaviour (mainly with UPPAAL and SPIN) and modelling distance computation (mainly with PRISM).

We *experienced* convenient usability of UPPAAL and PRISM, both come with a robust and intuitive model editor, in contrast to SPIN. We also *experienced* that model construction for PRISM took significantly more effort than it took for UPPAAL and SPIN. This is a consequence of the rather restricted modelling language.

2. What are the boundaries, problems and experiences of verifying the protocol?

The *boundaries* of verification of the SPT protocol are determined by the combinatorial state space explosion, as a result of the parallel node processes and non-determinism. We tried to restrict this explosion by adding abstractions as fixed execution order (all models) and controlled branching (only in SPIN model). Our UPPAAL (model V1) and SPIN models can be checked up to about 4 nodes (for worst-case topologies: completely connected with low-quality links). We did not manage to construct a useful PRISM model of that order.

Problems encountered were related to the instance explosion of the verification process, caused by many input parameters such as number of message rounds to be executed, topology (links and quality), number of nodes etc. A *problem* encountered with PRISM, is that models must be built from their specification, prior to any verification. The build process is rather inefficient for our type of models (many relatively large variable ranges of a number of parallel modules). Therefore we could not construct a useful model for verification.

We *experienced* that checking for correct parent selection is to be preferred above checking for correct distance computation, since it requires less resources: the former does not require exactly computed distances. Another *experience* is that feasibility of verification depends on many parameters, of which topology and number of nodes has greatest influence.

3. What aspects are best modelled/verified using which verification tool?

UPPAAL: message broadcast is best modelled using the powerful concept of *broadcast synchronization channels*. UPPAAL's modelling language is quite expressive: it provides convenient features like parameterizable process templates, partial instantiation and user-defined functions. Rational numbers are not supported. UPPAAL's property specification language allows for convenient access to all model variables. UPPAAL is a real-time model checker, optimized for verification of timing aspects. We hardly exploited its timing features in our models since we did not focus on timing within the SPT protocol.

SPIN: implementing message broadcast and distance computation requires more effort than in the UPPAAL case: native PROMELA (i.e. PROMELA without embedded C-code) is less expressive than UPPAAL's modelling language. There is however an option to embed C-code into the models. In contrast to PRISM, it does support parameterizable process types, which turned out to be a requirement for constructing more general models.

PRISM: in contrast to the other tools, PRISM is a probabilistic model checker: it allows for easily modelling link quality (i.e. probabilities). The implementation of message broadcast and distance computation did however require most effort (of the three tools). This is a result of the very restricted expressiveness of PRISM's modelling language.

4. How does the topology of a network influence the results?

The network topology defines the way in which constituent parts are interrelated or arranged. In our models this boils down to the total number of links within the network and their quality. Obviously, we can conclude that more links results in more states. It turned out that the number of directed links (rather than the number of nodes) is the determinant of the feasibility of a verification run. This is a result of modelling the links as non-deterministic processes (and of the fact that the execution order of the nodes is fixed).

Our verification results further show that links of low quality result in more states, compared to links of high quality.

5. How does the number of nodes influence the results?

As stated in the previous answer, the number of nodes is not the determinant of the feasibility of a verification run. This is the result of the fixed execution order of the nodes. Without fixing the order, it would be the determinant of feasibility, due to the combinatorial explosion of different possibilities (orders of execution). In that case, the number of model states would be exponential in the number of nodes. Fixing the order causes this relation to become linear.

6. Is the protocol under consideration correct?

As said before "model checking is only as good as the model". Since verification models are abstractions of reality and constructed based on insight, heuristics and experience, we cannot prove the correspondence of our models to the real protocol design. We can only make this plausible by describing and motivating our model design decisions. Therefore, we cannot state that 'the protocol is correct'. We can however state that it is very likely that the protocol is correct since we did not find any counterexamples: no errors were found during formal specification, simulation and verification. Moreover, confidence in this protocol grows further due to the facts that it has been proven correct mathematically and implemented and simulated in MATLAB.

7. What recommendations about a platform for formal verification experiments for WSN protocols can be given w.r.t. the gained experience?

Based on the verification results, we recommend SPIN for performing these experiments. Although model construction may require more effort compared to UPPAAL, it performs better on verification (4 nodes completely connected with 10% links vs. 3 nodes for UPPAAL, model V2). If the focus is on timing properties of the protocol, UPPAAL must be used (SPIN and PRISM do not support time).

We also recommend to investigate suitable abstraction techniques in order to push the feasibility bound further. Since feasibility depends heavily on topology, it might be interesting to abstract away from specific topologies, following for example the idea of Câmara et al. [10]. This may be an approach to feasible verification using PRISM as well. Further research on this topic is needed. Other abstractions that might be applicable focus on restricting the number of possible instances of the link behaviour (such as Controlled Branching, ch. 6), which result in a more quantitative verification. Further research is needed on this topic as well. We return on this quantitative verification below.

9.2.2 Main Research Question

Now that we answered the sub questions, we move to our main research question and attempt to answer it.

> *Is formal verification (specifically: Model Checking) suitable for supporting the design of protocols for WSNs?*

This question is much more complex than one should think at first sight. Based on our research, it cannot be answered with a simple yes or no. We first have to define the terms 'suitable' and 'supporting the design'. We start with the latter.

Designers of WSN protocols have only few tools to check the validity of their design. A possibility is to mathematically prove that the design is correct, but this usually requires many assumptions and simplifications. Another possibility is simulation and testing, for instance using MATLAB, as in our case. This may however not uncover all undesirable aspects of a protocol. The main research question should be interpreted in this light: *supporting the design of protocols* can thus be translated to *checking the validity of the design of protocols.*

Whether or not formal verification is *suitable* for supporting protocol designers in their work, depends on *how* and *to what extent* it can be used. The first term describes the method and its complexity: is the method straightforward, usable and easy to understand or does it require expert knowledge and a lot of experience? The second term indicates the use of formal verification in protocol design: does it yield considerable added value in designing protocols?

Based on our research and on the current state of the art, we tend to answer negatively to the main question. First of all, we showed that configurations of at most 4 or 5 nodes can be checked, and it is doubtful whether this limitation still yields enough added value. Furthermore, using formal methods (i.e. model checking in this case) to validate protocol designs does require a considerable amount of expert knowledge and experience in model construction and property specification. However, once a model of a protocol is constructed, and desired properties to be verified are specified, the added value is experimentation: given the model and properties, it is relatively easy to experiment with modifications (just push the button). Unfortunately, applying the desired modifications correctly may again require expert knowledge and experience in order to understand the model. Moreover, this is also the case for specifying additional properties and interpreting the results. We therefore fully agree with the following citation of Theo Ruys (taken from Ruys [50], page 68):

> "Model checkers are often put forward as 'press-the-button' tools: given a model and a property, pressing the 'verify' button of the model checker is sufficient for the tool to prove or disprove the property. If both the model and the property are readily available, this claim might be true. However, the formalisation of both the model and the properties is usually not a trivial task. Furthermore, due to the infamous state space explosion problem, both the model and the property to be verified should be coded as efficiently as possible for the model checker that is being used."

On the other hand, experimentation using formal verification turned out to be a rather powerful tool that has potential to support the design of protocols very well. If expert knowledge is available, formal verification may actually be very suitable, considering the experimental approach. Moreover, some limitations we found may disappear using another modelling paradigm: we modelled all nodes and their interactions explicitly, but it may be profitable to attempt a so-called *macro-programming* approach: abstracting away from separate nodes and messages (similar to the MATLAB model in the appendix).

Based on our research, we must conclude that, using state of the art formal verification techniques, only small network configurations can be checked. However, the 'Hidden Problem' (ch. 6) provided us with deep insights into the quantitative character of the verification: the probabilistic links induce many behaviours of the model and only a representative subset of these can be checked, instead of all (as shown with the Controlled Branching technique). Using model checking in this way results in a technique that fits somewhere between simulation and the traditional (qualitative) view on model checking. More research is required to find suitable abstraction techniques, that account for this quantitative character of the verification.

9.3 Main Contributions

In this section we concisely summarize the two main contributions of our research project.

- Current state of the art formal verification techniques can handle only very restricted WSN configurations of maximally four or five nodes. This induces the need for suitable abstraction techniques. These abstraction techniques should account for the *quantity* of the probabilistic link behaviour. The qualitative character of verification of WSN protocols changes and becomes more quantitative. More research is required to find suitable abstraction techniques that account for this quantitative aspect.

- We introduced the idea of a methodology or platform that supports the design of WSN protocols. This would be of great help for protocol designers. We propose a platform containing three stages:
    1. Mathematical proof of a (strongly) simplified protocol design;
    2. Matlab simulation to validate (possibly less simplified) protocol behaviour, in particular large scenarios;
    3. Formal verification experiments using model checking, to experiment with (more realistic, less simplified) variants of the protocol.

9.4 Future Research

This thesis gives rise to a number of interesting topics for future research. The main topics are listed below:

- Modelling explicit topologies comes with an instance explosion of possible topologies. Furthermore, feasibility of the verification depends heavily on the topology under consideration. Therefore further research should be done to abstract away from specific topologies;
- A promising solution direction for the 'Hidden Problem' (described in chapter 6) is the 'Controlled Branching' approach. This abstraction technique is based on a dependency relation that induces a partitioning of the probabilistic link behaviour. Future research should determine how to define this relation such that it results in representative partitions;
- More general: further research can be performed in order to find suitable abstraction techniques, that exploit the quantitative character of the verification of WSN protocols;
- The previous chapter about protocol variants and experiments (ch. 8) resulted in interesting future work, such as:
    - Research on the effect of multiple gateways in a single network (w.r.t. the disconnected gateway problem);
    - Research on neighbourhood management;
    - Research on dynamically changing topologies.

APPENDIX A

Matlab Implementation of the SPT Protocol

This appendix contains the MATLAB source of an abstract implementation of the SPT protocol. It is developed by Leon Evers of the Pervasive Systems (PS) group of the University of Twente. This section also contains some of the plots that are generated during a simulation of a network of 200 nodes.

A.1 Source

The MATLAB model consists of four source files:

- `spanningtree.m` – the main file of the implementation, contains tuning parameters and uses the other source files.
- `recvperc.m` – defines a function that calculates an s-curve of distance vs. receive probability.
- `optimtree.m` – describes a function to plot the optimal tree, based on the generated positions and the aforementioned s-curve.
- `constructtree.m` – constructs the tree chosen by the protocol and plots it.

These files are listed below. As can be seen in lines 4–7 of listing A.1, the number of nodes (constant `num`) is set to 200 and the total number of message rounds that will be simulated (constant `runtime`) equals 800. Lines 64–76 show that after `runtime/4` rounds, the node positions are shuffled randomly.

```
clear;

%configuration constants
num = 200; % nr of nodes
runtime = 800; % nr. message rounds (WE: was 800)
roundmsgs = 1; % msgs per round
initmsgs = 0; % initial nr of messages before protocol starts

% random seed initialisation
% the line below chooses a fixed random seed
s = 1.947383660000000e+009;
% uncomment next line to use different random seed
%s = rand('seed')
rand('seed', s);    % WE: initializes the random generator to method 'seed' and value s

txpow = rand(1) * num/2; % transmission power

% all node positions
pos = rand(2,num);
%%
% ds is distance between all nodes
ds = zeros(num,num);
for i=1:num;ds (i,1:num) = sqrt((pos(1,i) − pos(1,:)).^2 + (pos(2,i) − pos(2,:)).^2);end;

% pds contains reception probability from/to all nodes
pds = zeros(size(ds));
for i=1:numel(ds); pds(i) = recvperc(ds(i), txpow); end;

%% calculate root link quality and hop count
[m, root] = min(sum(pos));
[qual hops parent] = optimtree(pos, pds, root, false);
notroot = 1:length(qual) ≠ root;

%% plot results
figure(2);plot(sortrows([qual; hops]'));
title (['Quality and nr. hops. avg retransmissions = ' num2str(mean(qual(notroot) ./
     hops(notroot)))]);
% retransmissions is measure of network density

%%
msgs = false(num, num);
msgsum = zeros(num, num);
%same parameters based on perceived reception prob
nodehops = nan(1,num);
nodequal = inf(1,num);
nodeparent = uint16(zeros(1,num));
rcvprob = zeros(1, num);
neighqual = inf(num, num);
neighhops = nan(num, num);

parentcorrectperc = nan(1, runtime);
optimdist = nan(1, runtime);
nodedist = nan(1, runtime);

nodehops(root) = 0;
nodequal(root) = 0;
nodeparent(root) = root;

rcvprob = zeros(num,num,3);
for j = 1:runtime
    oldqual = nodequal;
    [msgsum rcvprob neighqual neighhops nodeparent nodehops nodequal] = constructtree(pos, pds,
         root, msgsum, rcvprob, neighqual, neighhops, nodeparent, nodehops, nodequal, true);
    if nodequal == oldqual; break; end

    if j == round(runtime / 4)      %reset positions!
        pos = rand(2,num);
        ds = zeros(num,num);
        for i=1:num;ds (i,1:num) = sqrt((pos(1,i) − pos(1,:)).^2 + (pos(2,i) − pos(2,:)).^2);end;
        pds = zeros(size(ds));
        for i=1:numel(ds); pds(i) = recvperc(ds(i), txpow); end;
        [m, root] = min(sum(pos));
        [qual hops parent] = optimtree(pos, pds, root, true);
        notroot = 1:length(qual) ≠ root;
        nodehops(root) = 0;
        nodequal(root) = 0;
        nodeparent(root) = root;
    end
```

Listing A.1: MATLAB source - spanningtree.m (part 1/2)

```

%     figure(4);plot(sortrows([nodequal; nodehops]'));
%     title (['Quality and nr. hops after ' num2str(j) ' msg rounds']);

    realqual = nan(1,num);
    parentpds = zeros(1,num);
    for i = find(nodeparent > 0)
        parentpds(i) = pds(nodeparent(i),i);
    end
    realqual(root) = 0;
    for i = 1:num
        donodes = nodeparent > 0 & isnan(realqual);
        if sum(donodes) == 0; break; end
        realqual(donodes) = realqual(nodeparent(donodes)) + 1./parentpds(donodes);
    end
    if i == num; disp('Tree not connected.');end
    donodes = ¬isnan(realqual) & notroot;
    optimdist(j) = mean(qual(donodes) ./ realqual(donodes));
    nodedist(j) = mean(nodequal(donodes) ./ realqual(donodes));
    parentcorrectperc(j) = mean(nodeparent(donodes) == parent(donodes));
end

figure(5);plot([optimdist' nodedist' parentcorrectperc']);
ylim([0.6 1.01])
title('Average quality of chosen tree');
xlabel('message rounds');
legend('optimal / chosen', 'perceived / chosen', '% correct parents' ,'Location','SouthEast');
```

Listing A.2: MATLAB source - spanningtree.m (part 2/2)

```
function p = recvperc(ds, txpow)
%
% calculates s−curve of distance vs receive probability.
%
% to test it, evaluate the next line (select and press F9)
% plot(recvperc([1:200]/200, 10))

p = −ds ./ (ds + exp(4−txpow .∗ ds))+1;
```

Listing A.3: MATLAB source - recvperc.m

```
function [qual hops parent] = optimtree(pos, pds, root, draw)

num = size(pos,2);

hops = nan(1,num); % nr. of hops
qual = nan(1,num); % total link quality, lower is better
parent = uint16(zeros(1,num)); % parent

% initialize gateway
hops(root) = 0;
qual(root) = 0;
parent(root) = root;

for i = 1:num
    %try to improve link quality when adding an extra hop
    [newqual, newparent] = min(repmat(qual',1,num) + (1./ pds));
    % nodes that improve quality
    goodqual = ¬(newqual ≥ qual);
    parent(goodqual) = newparent(goodqual);
    hops(goodqual) = i;
    oldqual = qual;
    qual(goodqual) = newqual(goodqual);
    if qual == oldqual; draw = true; end
    if draw
        figure(1); plot(pos(1,:),pos(2,:), '.');
        title('Optimal tree');
        axis equal;
        axis tight;
        maxcolor = max(hops) ∗ 1.2;
        for i = 1:num
            line([pos(1,i) pos(1,parent(i))], [pos(2,i)
                 pos(2,parent(i))],'Color',repmat((hops(i)/maxcolor),3,1));
        end
    end
    if qual == oldqual; break; end
end
```

Listing A.4: MATLAB source - optimtree.m

```
function [msgsum rcvprob neighqual neighhops nodeparent nodehops nodequal] = constructtree(pos,
      pds, root, msgsum, rcvprob, neighqual, neighhops, nodeparent, nodehops, nodequal, draw)

alpha = 0.02;

num = size(pos,2);
notroot = 1:num ≠ root;

% messages that arrive this round
msgs = rand(size(pds)) ≤ pds;
%update local state based on message contents
q = repmat(nodequal',1,num);
h = repmat(nodehops',1,num);
neighqual(msgs) = q(msgs);
neighhops(msgs) = h(msgs);

msgsum = msgsum + msgs;
j = msgsum(1,1);

% perceived reception probability
%rcvprob = msgsum ./ j;

% WE: Single Exponential Smoothing:
% rcvprob(:,:,1) = rcvprob(:,:,1) .* (1—alpha) + msgs .* alpha;

% WE: Some variant of Exponential Smoothing:
rcvprob(:,:,3) = rcvprob(:,:,3) .* (1—alpha) + msgs .* alpha;
rcvprob(:,:,2) = rcvprob(:,:,2) .* (1—alpha) + rcvprob(:,:,2) .* alpha;
rcvprob(:,:,1) = 2*rcvprob(:,:,3) — rcvprob(:,:,2);

% build tree based on rcvprob
[newqual, newparent] = min(neighqual + 1./rcvprob(:,:,1));
goodqual = (newqual < Inf) & notroot;
nodeparent(goodqual) = newparent(goodqual);
nodehops(goodqual) = neighhops(newparent(goodqual) + (find(goodqual)—1).*num) + 1;
nodequal(goodqual) = newqual(goodqual);

if draw
    figure(3); plot(pos(1,:),pos(2,:), 'r.');
    axis equal;
    axis tight;
    title(['Chosen tree after ' num2str(j) ' msg rounds']);
    %     maxcolor = max(nodehops) * 1.2;
    maxcolor = max(nodequal(nodequal < Inf)) * 1.2;
    for i = find(nodeparent>0)
        line([pos(1,i) pos(1,nodeparent(i))], [pos(2,i) pos(2,nodeparent(i))], 'Color',
              repmat([nodequal(i)/maxcolor],3,1));
    end

end
```

Listing A.5: MATLAB source - constructtree.m

A.2 Plots

This sections contains some plots generated by an execution run of the MATLAB implementation, as listed in the previous section. 800 message rounds are simulated.

The first plot shows the optimal tree that is generated based on the randomly generated node positions and the s-curve of distance vs. receive probability. The second plot shows the tree chosen by the protocol, after 200 message rounds, which is actually a very good approximation of the optimal tree.

The next two plots show the optimal tree and the chosen tree after randomly shuffling node positions (after 200 message rounds). Observe the chaotic situation to which the protocol is exposed (node positions are shuffled but the nodes still have the parents based on the old situation).

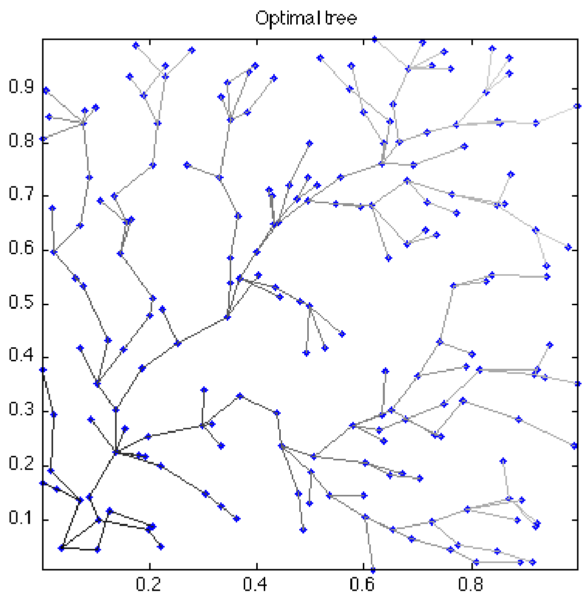


Figure A.1: The optimal tree w.r.t. the generated node positions

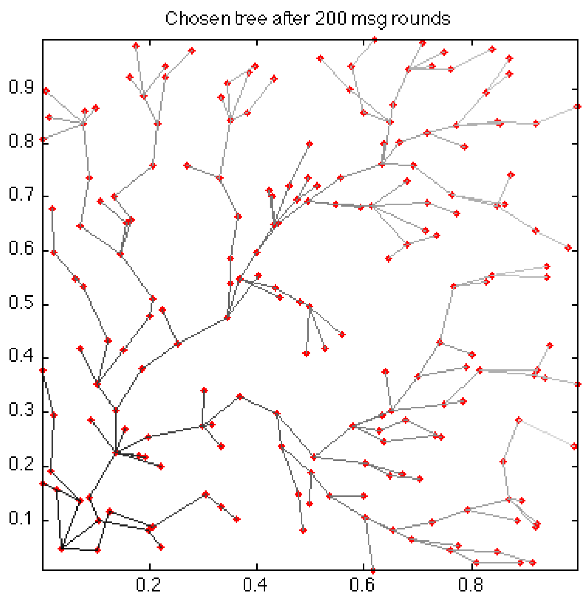


Figure A.2: The chosen tree after simulation of 200 message rounds

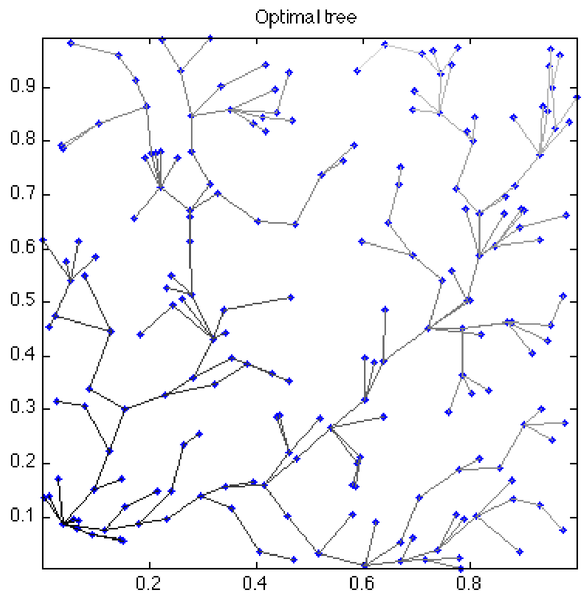


Figure A.3: The optimal tree w.r.t. the generated node positions (after shuffling)

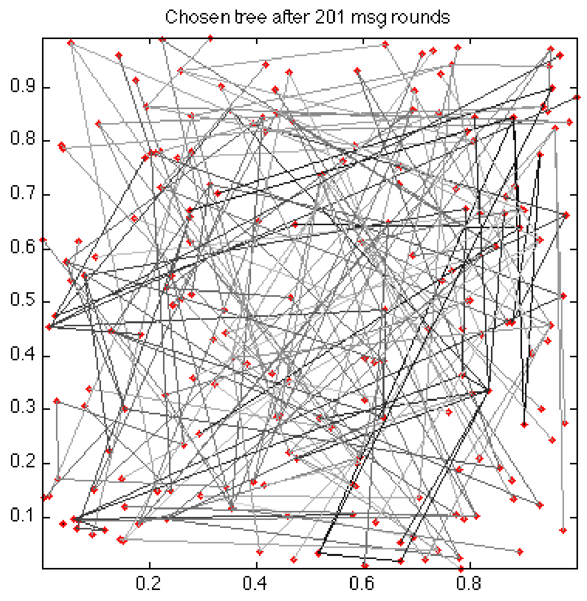


Figure A.4: The chosen tree after simulation of 201 message rounds (after shuffling)

The next plot shows the chosen tree after 800 simulated rounds and finally the last plot shows some statistics (also after 800 rounds).

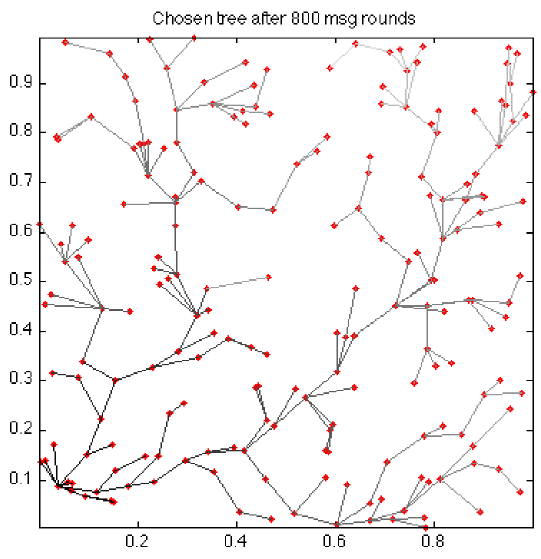


Figure A.5: The chosen tree after simulation of 800 message rounds

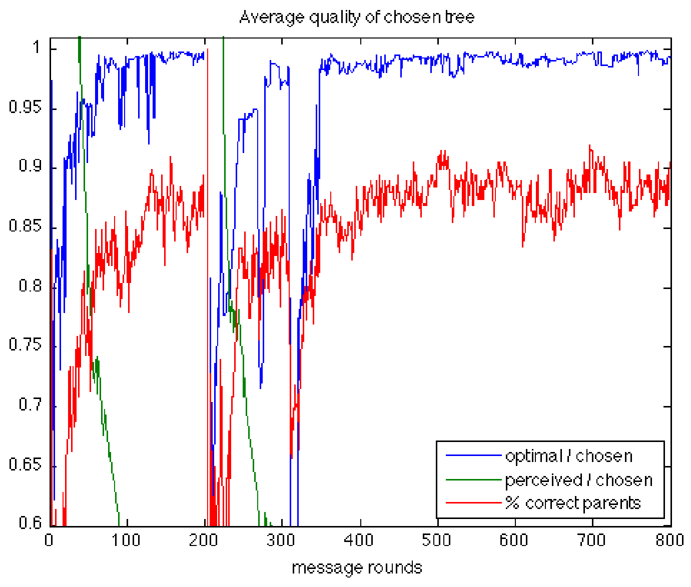


Figure A.6: Statistics

APPENDIX B

UPPAAL Models, Simulation and Verification Results

This appendix contains the complete source of our UPPAAL models, a section about the simulation of the behaviour of these models using UPPAAL's built-in simulator and a section with detailed verification results.

B.1 Protocol Model V1

```
/*****
Shortest Path Tree protocol for Wireless Sensor Networks
Author: W.M. Everse

Simple model
     All nodes can hear each other with link quality of 100%
     Nodes operate synchronously
*****/

// Number of nodes, including gateway(s) G
const int N = 4;

// Maximum number of message rounds
const int MAX_M = 10;

// Big number to represent 'infinite' distance
const int MAX_DIST = 10000;

// Define type Node_id, parameter of Node template
typedef int[0,N-1] Node_id;

// Synchronization channel to model message sending/receiving
broadcast chan send;

// Model a message as a struct
// msg.s_id = sender id and msg.dist = distance
meta struct{
     Node_id s_id;
     int dist;
} msg;

// Determine whether the given node is a gateway node
bool isGateway(Node_id node) {
     return node == 0;
}
```

Listing B.1: Global declarations of model V1

```
/* <parameter>const Node_id i</parameter> */    //explained in text

// Node clock
clock x;

// Local round number
int M;

// Dist-to-G per node (D[y] = dist-to-G from y, perceived by this node)
int D[Node_id] = {MAX_DIST, MAX_DIST, MAX_DIST, MAX_DIST};

// Msg counters per node (R[y] = #messages received from node y)
int R[Node_id];

// The selected parent
meta Node_id parent;

// Function to determine the minimum distance
int getMinimum(){
    meta int minval = MAX_DIST;       // To hold the min. found so far
    meta int try;                     // To hold the next value
    if( isGateway(i) ) return 0;      // Minimum dist-to-G of G is 0
    if( M == 0 ) return MAX_DIST;     // First round return MAX_DIST
    for(j : Node_id){
        if( R[j] > 0 && j != i && D[j] < MAX_DIST ){
            try = M/R[j] + D[j];
            if( (M % R[j]) >= (R[j] / 2) ) try++; // Round to nearest int
            if( try <= minval){
                minval = try;
                parent = j;
            }
        }
    }
    return minval;                    // Return the min. found
}

// Function executed on message reception
void receive(){
    R[msg.s_id]++;                    // Increase msg counter of sender
    D[msg.s_id] = msg.dist;           // Update Dist-to-G of sender
}
```

Listing B.2: Local declarations of node template of model V1

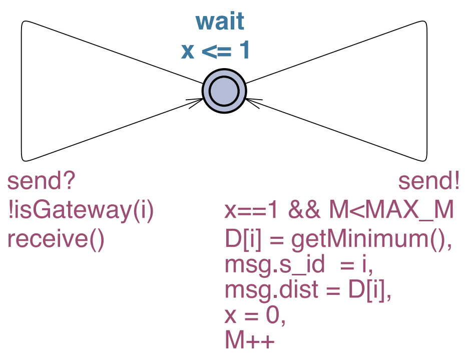


Figure B.1: Node Template of model V1

```
// This instantiates template Node(const Node_id i)
// for all values in Node_id.
system Node;
```

Listing B.3: System declaration of model V1

B.2 Protocol Model V2

```
/*****
Shortest Path Tree protocol for Wireless Sensor Networks
Author: W.M. Everse
Connectivity Matrix model
    Link qualities between nodes are defined in percentages
        in connectivity matrix C[][] below
    Nodes operate synchronously
    Nodes with i<GATEWAY_COUNT are gateways
*****/

// Number of nodes, including gateway(s) G
const int N = 4;

// Number of gateway nodes
const int GATEWAY_COUNT = 1;

// Maximum number of message rounds
const int MAX_M = 50;

// Big number to represent 'infinite' distance
const int MAX_DIST = 10000;

// Define type Node_id, parameter of Node template
typedef int[0,N-1] Node_id;

meta int turn=3;
const bool NOTURN = true;

// Symmetric connectivity matrix with percentages
// if C[x][y]==100 then x can hear y with quality of 100%
const int[0,100] C[Node_id][Node_id] = {
    { 0, 15, 10, 90},
    {15,  0,  0, 33},
    {10,  0,  0,  0},
    {90, 33,  0,  0}
};

// Synchronization channel to model message sending/receiving
broadcast chan send;

// Model a message as a struct
// msg.s_id = sender id and msg.dist = distance
meta struct{
    Node_id s_id;
    int dist;
} msg;

// Determine whether the given node is a gateway node
bool isGateway(Node_id node) {
    return node < GATEWAY_COUNT;
}

// Constant for the accuracy of the distance computed
// 1, 10, 100 corresponds resp. to 0, 1 or 2 positions behind comma
const int ACCURACY = 10;

/** The functions below are for reference in verification properties **/

// Set the correct distance to G per node in array d2G[]
// You can also set an upper and lower tolerance bound (epsUpper and epsLower)
bool isCorrectDistanceToG(Node_id node, int dist){
    const int d2G[Node_id] = {0, 67, 24, 11};
    const int epsUpper = 0;     //0 = no upper tolerance bound
    const int epsLower = 0;     //0 = no lower tolerance bound

    return (dist >= d2G[node]-epsLower) && (dist <= d2G[node]+epsUpper);
}

// Set the correct parent(s) per node
bool isCorrectParent(Node_id node, Node_id parent){
    if(isGateway(node)) return parent==0;
    if(node==1) return parent==3 || parent==0;
    if(node==2) return parent==3;
    if(node==3) return parent==0;
    return false;
}
```

Listing B.4: Global declarations of model V2

```
/* <parameter>const Node_id i</parameter> */

// Node clock
clock x;

// Local round number
int M;

// Dist-to-G per node (D[y] = dist-to-G from y, perceived by this node)
int D[Node_id] = {MAX_DIST, MAX_DIST, MAX_DIST, MAX_DIST};

// Msg counters per node (R[y] = #messages received from node y)
int R[Node_id];

// The selected parent
meta Node_id parent;

// Function to determine the minimum distance
int getMinimum(){
    meta int minval = MAX_DIST;      // To hold the min. found so far
    meta int try;                    // To hold the next value
    if( isGateway(i) ) return 0;     // Minimum dist-to-G of G is 0
    if( M == 0 ) return MAX_DIST;    // First round return MAX_DIST
    for(j : Node_id){
        if( R[j] > 0 && j != i && D[j] < MAX_DIST ){
            try = ACCURACY * M/R[j] + D[j];
            if( (ACCURACY * M % R[j]) >= (R[j] / 2) ) try++; // Round to nearest integer
            if( try <= minval){
                minval = try;
                parent = j;
            }
        }
    }
    return minval;                   // Return the min. found
}

// This counter is used to realize link qualities, see receive() below
meta int lost_from[Node_id];

// Function executed on message reception
// It also enforces the given link qualities (from connection matrix)
void receive(){
    // Compute the current enforced lost ratio
    int lost_ratio = 100 * lost_from[msg.s_id] / (M+1);
    if(lost_ratio < (100 - C[i][msg.s_id]) )
        lost_from[msg.s_id]++;       // The message got lost
    else
    {   // Message received
        R[msg.s_id]++;               // Increase msg counter of sender
        D[msg.s_id] = msg.dist;      // Update Dist-to-G of sender
    }
}
```

Listing B.5: Local declarations of node template of model V2

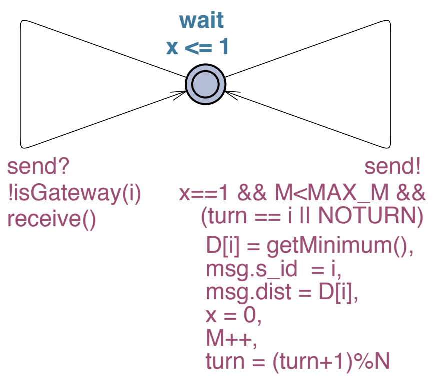


Figure B.2: Node Template of model V2

```
// This instantiates template Node(const Node_id i)
// for all values in Node_id.
system Node;
```

Listing B.6: System declaration of model V2

B.3 Protocol Model V3

```
/*****
Shortest Path Tree protocol for Wireless Sensor Networks
Author: W.M. Everse
Non-Deterministic Link model
    Links are modelled as explicit non-deterministic processes
    Nodes operate synchronously
    Nodes with i<GATEWAY_COUNT are gateways
*****/

// Number of nodes, including gateway(s) G
const int N = 4;

// Number of gateway nodes
const int GATEWAY_COUNT = 1;

// Maximum number of message rounds
const int MAX_M = 50;

// Big number to represent 'infinite' distance
const int MAX_DIST = 10000;

// Define type Node_id, parameter of Node template
typedef int[0,N-1] Node_id;

meta Node_id turn;
const bool NOTURN = true;

// Synchronization channels to model message sending/receiving
broadcast chan send[Node_id];
broadcast chan recv[Node_id];

// Model a message as a struct
// msg.s_id = sender id and msg.dist = distance
meta struct{
    Node_id s_id;
    int dist;
} msg;

// Determine whether the given node is a gateway node
bool isGateway(Node_id node) {
    return node < GATEWAY_COUNT;
}

// Constant for the accuracy of the distance computed
// 1, 10, 100 corresponds resp. to 0, 1 or 2 positions behind comma
const int ACCURACY = 10;

/** The functions below are for reference in verification properties **/

// Set the correct distance to G per node in array d2G[]
// You can also set an upper and lower tolerance bound (epsUpper and epsLower)
bool isCorrectDistanceToG(Node_id node, int dist){
    const int d2G[Node_id] = {0, 41, 23, 11};
    const int epsUpper = 0;     //0 = no upper tolerance bound
    const int epsLower = 0;     //0 = no lower tolerance bound

    return (dist >= d2G[node]-epsLower) && (dist <= d2G[node]+epsUpper);
}

// Set the correct parent(s) per node
bool isCorrectParent(Node_id node, Node_id parent){
    if(isGateway(node)) return parent==0;
    if(node==1) return parent==3 || parent==0;
    if(node==2) return parent==3;
    if(node==3) return parent==0;
    return false;
}
```

Listing B.7: Global declarations of model V3

```
/* <parameter>const Node_id i</parameter> */

// Node clock
clock x;

// Local round number
int M;

// Dist-to-G per node (D[y] = dist-to-G from y, perceived by this node)
int D[Node_id] = {MAX_DIST, MAX_DIST, MAX_DIST, MAX_DIST};

// Msg counters per node (R[y] = #messages received from node y)
int R[Node_id];

// The selected parent
meta Node_id parent;

// Function to determine the minimum distance
int getMinimum(){
    meta int minval = MAX_DIST;      // To hold the min. found so far
    meta int try;                    // To hold the next value
    if( isGateway(i) ) return 0;     // Minimum dist-to-G of G is 0
    if( M == 0 ) return MAX_DIST;    // First round return MAX_DIST
    for(j : Node_id){
        if( R[j] > 0 && j != i && D[j] < MAX_DIST ){
            try = ACCURACY * M/R[j] + D[j];
            if( (ACCURACY * M % R[j]) >= (R[j] / 2) ) try++; // Round to nearest integer
            if( try <= minval){
                minval = try;
                parent = j;
            }
        }
    }
    return minval;                   // Return the min. found
}

// Function executed on message reception
void receive(){
    R[msg.s_id]++;                   // Increase msg counter of sender
    D[msg.s_id] = msg.dist;          // Update Dist-to-G of sender
}
```

Listing B.8: Local declarations of node template of model V3

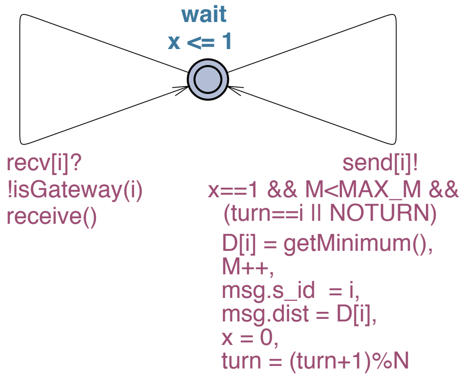


Figure B.3: Node Template of model V3

```
/* <parameter>const Node_id src, const Node_id tgt, const int u, const int v</parameter> */
/*****
LinkAccross

Nondeterministic process that represents an unidirectional link
    between two nodes (parameters 'src' and 'tgt').

The quality of this link is determined by parameters u and v:
    u on v messages get across. Which ones is decided
    nondeterministically.
*****/

// counts number of messages that came across
int[0,u] s = 0;

// counts period
int[0,v] t = 0;

// resets counters if v msgs passed
void checkreset(){
    if(t==v){
        s=0;
        t=0;
    }
}
```

Listing B.9: Local declarations of link template of model V3

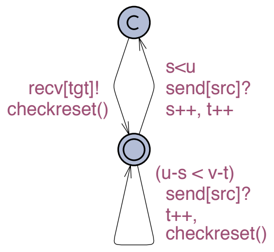


Figure B.4: Link Template of model V3

```
// Declare link processes
link01 = LinkAcross(0,1,1,7);       //Link Quality = 14,3%
link02 = LinkAcross(0,2,1,10);      //Link Quality = 10%
link03 = LinkAcross(0,3,9,10);      //Link Quality = 90%

link13 = LinkAcross(1,3,1,3);       //Link Quality = 33%
link31 = LinkAcross(3,1,1,3);

link23 = LinkAcross(2,3,8,10);      //Link Quality = 80%
link32 = LinkAcross(3,2,8,10);

// Instantiate template Node(const Node_id i)
// for all values in Node_id and the link processes.
system Node,
    link01, link02, link03,
    link13, link31,
    link23, link32;
```

Listing B.10: System declaration of model V3

B.4 Validation by Simulation

We validated the behaviour of our models using the built-in simulator of UPPAAL. In this section we will provide some insights in the simulation process and in the results.

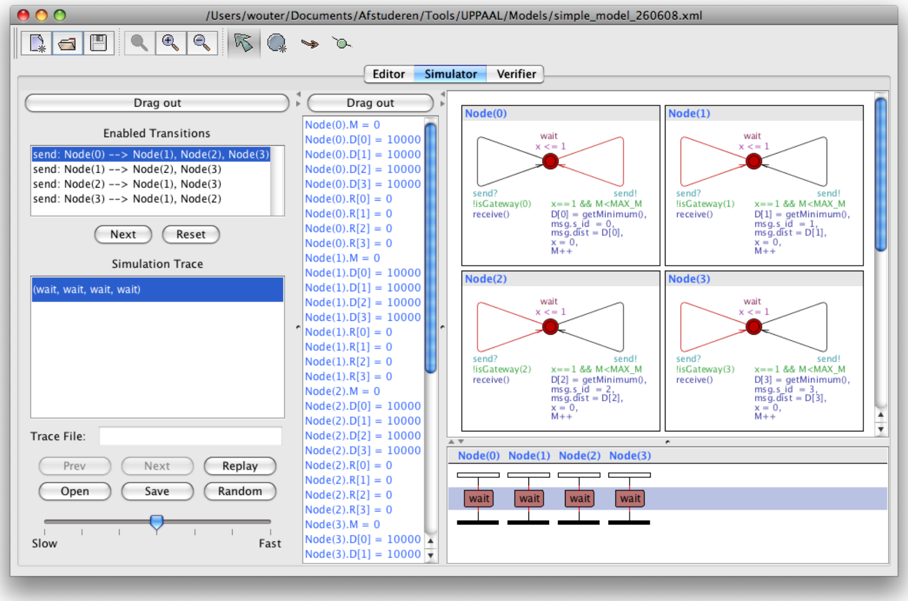
Figure B.5 shows a screen shot of the simulator. It consists of four panels, the most left one being the *simulation control* panel, that shows the simulation trace so far and allows for either selecting an enabled transition manually, or running a random simulation. Next of it, we find the *variables* panel that shows us all variable values in the current selected state of the simulation trace. Right of this panel, we see the *processes* panel that depicts a graphical representation of the processes that are composed in the system under simulation, showing the current state of each process. The lower rightmost panel is the *Message Sequence Chart (MSC)* panel which displays a MSC view of the generated trace.

B.4.1 Protocol Model V1

In the first version of our model, all nodes hear all other nodes with a link quality of 100%. Figure B.5 shows the initial state of the composed system. All four nodes can start broadcasting a probe message, therefore there are four transitions enabled. If we execute the selected transition we can see the changes in the composed system state, indicating that node 0 did a broadcast of a probe message and nodes 1, 2 and 3 received this message (i.e. synchronized) and updated their information according to it (see figure B.6). These nodes all have their received messages counter for messages from node 0 (`Node(i).R[0]` with `i=1,2,3`) set to 1, and their recorded dist-to-G of node 0 (`Node(i).D[0]`) set to 0[1].

Note that node 0's local message round counter now equals 1 and that the 3 remaining enabled transitions are the broadcasts of nodes 1, 2 and 3. When they all have broadcasted a probe message in this message round (i.e. message round 0) all nodes will be ready to broadcast in message round 1 again. This continues until all local message round counters reached the value of `MAX_M`. In that case, there is no enabled transition left and the composed system is *deadlocked*: our simulation run ends.

[1]As node 0 is considered to be the gateway, it follows from its local declarations (line 22 of lst. B.2 on page 158) that it will always broadcast a dist-to-G of 0.



/Users/wouter/Documents/Afstuderen/Tools/UPPAAL/Models/simple_model_260608.xml

Enabled Transitions:

```
send: Node(0) --> Node(1), Node(2), Node(3)
send: Node(1) --> Node(2), Node(3)
send: Node(2) --> Node(1), Node(3)
send: Node(3) --> Node(1), Node(2)
```

Simulation Trace:

```
(wait, wait, wait, wait)
```

```
Node(0).M = 0
Node(0).D[0] = 10000
Node(0).D[1] = 10000
Node(0).D[2] = 10000
Node(0).D[3] = 10000
Node(0).R[0] = 0
Node(0).R[1] = 0
Node(0).R[2] = 0
Node(0).R[3] = 0
Node(1).M = 0
Node(1).D[0] = 10000
Node(1).D[1] = 10000
Node(1).D[2] = 10000
Node(1).D[3] = 10000
Node(1).R[0] = 0
Node(1).R[1] = 0
Node(1).R[2] = 0
Node(1).R[3] = 0
Node(2).M = 0
Node(2).D[0] = 10000
Node(2).D[1] = 10000
Node(2).D[2] = 10000
Node(2).D[3] = 10000
Node(2).R[0] = 0
Node(2).R[1] = 0
Node(2).R[2] = 0
Node(2).R[3] = 0
Node(3).M = 0
Node(3).D[0] = 10000
Node(3).D[1] = 10000
```

Node(0)

```
wait
x <= 1
send?
!isGateway(0)
receive()
send!
x==1 && M<MAX_M
D[0] = getMinimum(),
msg.s_id  = 0,
msg.dist = D[0],
x = 0,
M++
```

Node(1)

```
wait
x <= 1
send?
!isGateway(1)
receive()
send!
x==1 && M<MAX_M
D[1] = getMinimum(),
msg.s_id  = 1,
msg.dist = D[1],
x = 0,
M++
```

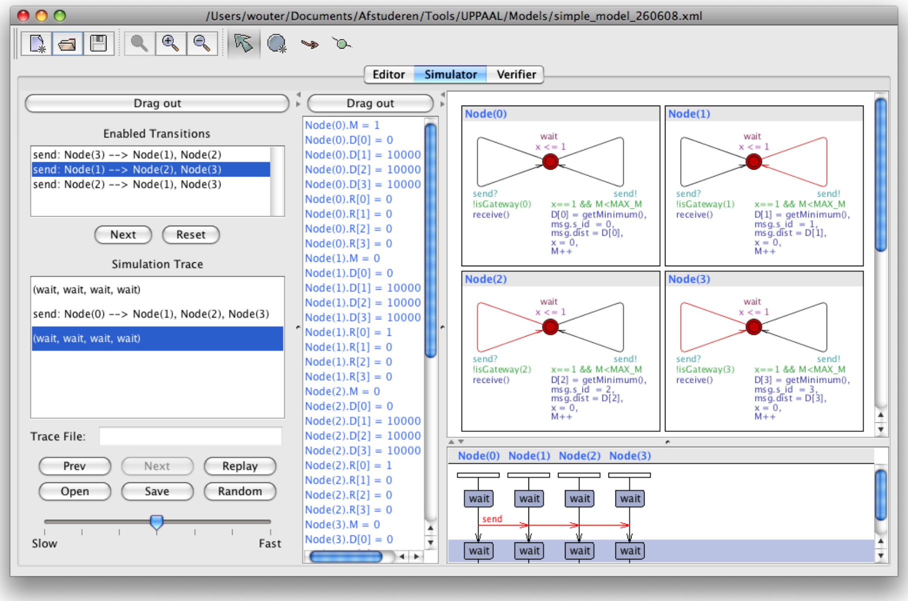
Node(2)

```
wait
x <= 1
send?
!isGateway(2)
receive()
send!
x==1 && M<MAX_M
D[2] = getMinimum(),
msg.s_id  = 2,
msg.dist = D[2],
x = 0,
M++
```

Node(3)

```
wait
x <= 1
send?
!isGateway(3)
receive()
send!
x==1 && M<MAX_M
D[3] = getMinimum(),
msg.s_id  = 3,
msg.dist = D[3],
x = 0,
M++
```

Figure B.5: UPPAAL's Simulator (initial state of model V1)



/Users/wouter/Documents/Afstuderen/Tools/UPPAAL/Models/simple_model_260608.xml

Enabled Transitions:

```
send: Node(3) --> Node(1), Node(2)
send: Node(1) --> Node(2), Node(3)
send: Node(2) --> Node(1), Node(3)
```

Simulation Trace:

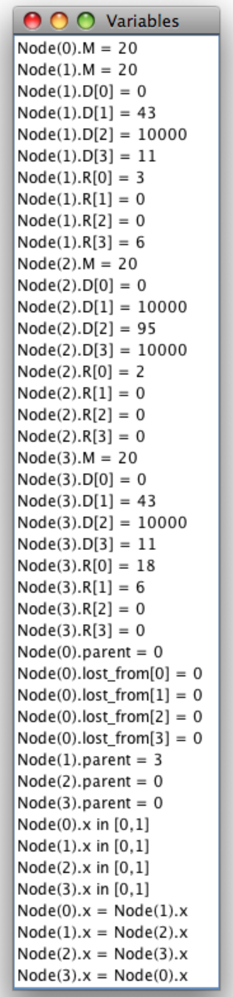
```
(wait, wait, wait, wait)
send: Node(0) --> Node(1), Node(2), Node(3)
(wait, wait, wait, wait)
```

```
Node(0).M = 1
Node(0).D[0] = 0
Node(0).D[1] = 10000
Node(0).D[2] = 10000
Node(0).D[3] = 10000
Node(0).R[0] = 0
Node(0).R[1] = 0
Node(0).R[2] = 0
Node(0).R[3] = 0
Node(1).M = 0
Node(1).D[0] = 0
Node(1).D[1] = 10000
Node(1).D[2] = 10000
Node(1).D[3] = 10000
Node(1).R[0] = 1
Node(1).R[1] = 0
Node(1).R[2] = 0
Node(1).R[3] = 0
Node(2).M = 0
Node(2).D[0] = 0
Node(2).D[1] = 10000
Node(2).D[2] = 10000
Node(2).D[3] = 10000
Node(2).R[0] = 1
Node(2).R[1] = 0
Node(2).R[2] = 0
Node(2).R[3] = 0
Node(3).M = 0
Node(3).D[0] = 0
```

Figure B.6: State of model V1 after simulating a broadcast of node 0

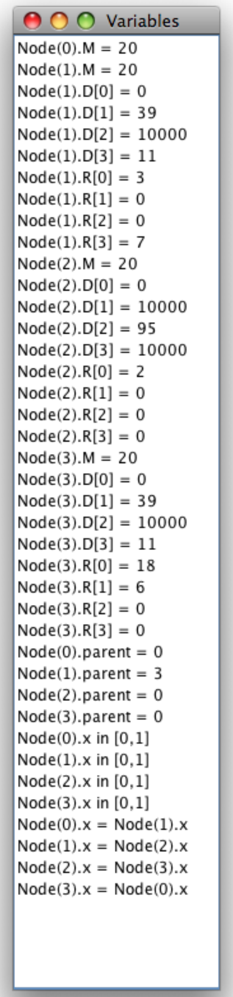
B.4.2 Protocol Model V2 & V3

In order to examine the behaviour of our models V2 and V3 we also did a lot of simulation. Among other things, we did random simulation runs with `MAX_M` set to 20 and with the connectivity matrix (of V2) specifying the 4-node topology from figure 4.3 (page 48). We also executed random simulation runs on model V3, configured with the same topology. The contents of the variable simulation panel (with some less relevant variables omitted, for example the arrays from node 0 since they are not altered during simulation) at the end of a random simulation run are depicted in figure B.7.



```
Variables
Node(0).M = 20
Node(1).M = 20
Node(1).D[0] = 0
Node(1).D[1] = 43
Node(1).D[2] = 10000
Node(1).D[3] = 11
Node(1).R[0] = 3
Node(1).R[1] = 0
Node(1).R[2] = 0
Node(1).R[3] = 6
Node(2).M = 20
Node(2).D[0] = 0
Node(2).D[1] = 10000
Node(2).D[2] = 95
Node(2).D[3] = 10000
Node(2).R[0] = 2
Node(2).R[1] = 0
Node(2).R[2] = 0
Node(2).R[3] = 0
Node(3).M = 20
Node(3).D[0] = 0
Node(3).D[1] = 43
Node(3).D[2] = 10000
Node(3).D[3] = 11
Node(3).R[0] = 18
Node(3).R[1] = 6
Node(3).R[2] = 0
Node(3).R[3] = 0
Node(0).parent = 0
Node(0).lost_from[0] = 0
Node(0).lost_from[1] = 0
Node(0).lost_from[2] = 0
Node(0).lost_from[3] = 0
Node(1).parent = 3
Node(2).parent = 0
Node(3).parent = 0
Node(0).x in [0,1]
Node(1).x in [0,1]
Node(2).x in [0,1]
Node(3).x in [0,1]
Node(0).x = Node(1).x
Node(1).x = Node(2).x
Node(2).x = Node(3).x
Node(3).x = Node(0).x
```

(a) Protocol model V2



```
Variables
Node(0).M = 20
Node(1).M = 20
Node(1).D[0] = 0
Node(1).D[1] = 39
Node(1).D[2] = 10000
Node(1).D[3] = 11
Node(1).R[0] = 3
Node(1).R[1] = 0
Node(1).R[2] = 0
Node(1).R[3] = 7
Node(2).M = 20
Node(2).D[0] = 0
Node(2).D[1] = 10000
Node(2).D[2] = 95
Node(2).D[3] = 10000
Node(2).R[0] = 2
Node(2).R[1] = 0
Node(2).R[2] = 0
Node(2).R[3] = 0
Node(3).M = 20
Node(3).D[0] = 0
Node(3).D[1] = 39
Node(3).D[2] = 10000
Node(3).D[3] = 11
Node(3).R[0] = 18
Node(3).R[1] = 6
Node(3).R[2] = 0
Node(3).R[3] = 0
Node(0).parent = 0
Node(1).parent = 3
Node(2).parent = 0
Node(3).parent = 0
Node(0).x in [0,1]
Node(1).x in [0,1]
Node(2).x in [0,1]
Node(3).x in [0,1]
Node(0).x = Node(1).x
Node(1).x = Node(2).x
Node(2).x = Node(3).x
Node(3).x = Node(0).x
```

(b) Protocol model V3

Figure B.7: Values of a selected set of variables after simulation of 20 message rounds

Although there are slight differences, both models seem to behave properly: node 1 selected node 3 as parent on the shortest path to node 0. The other nodes properly selected node 0 as parent in the routing tree. The distance to the gateway[2] of each node (`Node(i).D[i]`) sufficiently approximates the real value. For example, node 3 has a 90% link with node 0. This corresponds to an ETX value of $\frac{100}{90} = 1\frac{1}{9} \approx 1,11$. Node 3 finds `Node(3).D[3]=11` because we multiply all results with a global constant `ACCURACY` with default value 10. Doing so allows for a more precise result (in this case one digit behind the decimal point), as UPPAAL does not support floating point numbers.

B.5 Verification Results

Notation used in this section:

- The number of nodes is denoted by `N`
- The number of message rounds verified is denoted by `MAX_M`
- The global constant `ACCURACY` is set to 10 by default. Verification runs with `ACCURACY` set to 1 are annotated with the superscript: ^{A1}
- Verification runs with ordered process execution are annotated with the superscript: ^{O} and are executed starting with process id 0.
- The time reported is the `real time` measured by the UNIX utility `time`
- Table entries with a sole dash (-) indicate that the corresponding verification run was terminated abnormally (aborted since it did not end within reasonable time).
- Empty table entries correspond to less interesting verification runs that we did not execute.
- Occasionally, the big M is used for millions, to abbreviate large numbers of states (e.g. 26520000 is abbreviated to 26,52M).

B.5.1 Deadlock Freedom

On all paths, it is always the case that if a deadlock state occurs, it is the state in which for all nodes hold that their local message round counter `M` equals `MAX_M`:

```
A[] deadlock imply ( forall(i:Node_id) Node(i).M == MAX_M )
```

[2]Recall from chapter 3 that this actually is an Expected Transmission Count (ETX)-value: the expected number of transmissions needed to get a message at the gateway.

Protocol Model V2

Table B.1: UPPAAL V2 Results for topo 4.6(a), p. 57: complete, all links 100%

MAX_M:	10		20		50		100	
N	#s	t(s)	#s	t(s)	#s	t(s)	#s	t(s)
2	47	.018	77	.020	167	.022	317	.025
4	677	.063	827	.071	1277	.105	2027	.152
6	9437	1.09	10067	1.17	11957	1.40	15107	1.75
8	135917	24.2	138467	25.2	146117	26.7	158867	28.5
2^{A1}	33	.017	63	.018	153	.021	303	.026
4^{A1}	201	.036	351	.044	801	.073	1551	.126
6^{A1}	1233	.168	1863	.240	3753	.453	6903	.807
4^{O}	41	.021	81	.027	201	.029	401	.035
8^{O}	81	.039	161	.041	401	.060	801	.096
12^{O}	121	.056	241	.081	601	.117	1201	.193
$\downarrow$								
100^{O}							10001	33.7

Table B.2: UPPAAL V2 Results for topo 4.6(a), p. 57: complete, all links 10%

MAX_M:	10		20		50		100	
N	#s	t(s)	#s	t(s)	#s	t(s)	#s	t(s)
2	32	.017	91	.020	268	.022	563	.031
3	79	.020	2663	.127	33239	1.46	89459	3.90
4	200	.029	1091653	64	*oom*	*1086*	-	-
4^{O}	41	.021	81	.027	201	.029	401	.035

Table B.3: UPPAAL V2 Results for topo 4.6(b), p. 57: chain, all links 100%

MAX_M:	10		20		50		100	
N	#s	t(s)	#s	t(s)	#s	t(s)	#s	t(s)
3	226	.029	299	.032	509	.042	859	.057
4	1224	.095	1412	.108	1862	.137	2612	.181
5	7634	.620	8254	.677	9184	.749	10734	.871
4^{A1}	266	.035	416	.047	866	.073	1616	.118
4^{O}	41	.022	81	.026	201	.028	401	.038

Table B.4: UPPAAL V2 Results for topo 4.6(b), p. 57: chain, all links 10%

MAX_M:	10		20		50		100	
N	#s	t(s)	#s	t(s)	#s	t(s)	#s	t(s)
3	76	.021	1150	.061	62371	2.57	185553	7.69
4	169	.030	18113	.974	-	>720	-	-
4^{A1}					4,33M	253	5,60M	334
4^{O}	41	.022	81	.024	201	.028	401	.034
4					*26,52M*	*894*	*oom*	*1050*
$\mathit{4}^{A1}$					*4,33M*	*144*	*5,60M*	*187*

Remarks:

- "*oom*" in tables B.2 and B.4 means "out-of-memory". This entry is verified on verification server BIG1 (32-bit UPPAAL induces a $2^{32} = 4$ GB memory limit). For more information, please refer to section 4.3.4.
- The two bottom rows (italicized) of table B.4 contain results of verification runs that were also verified on server BIG1. Again refer to section 4.3.4 for more information.

Protocol Model V3

Table B.5: UPPAAL V3 Results for topo 4.6(a), p. 57: complete, all links 100%

MAX_M:	10		20		50		100	
N	#s	t(s)	#s	t(s)	#s	t(s)	#s	t(s)
2	75	.020	125	.023	275	.026	525	.031
4	5964	.219	7394	.267	11684	.403	18834	.631
5	52845	2.39	60035	2.70	81605	3.70	117555	5.67
6	459510	28.4	494060	30.6	597710	37.3	770460	48.3
2^{A1}	54	.020	104	.021	254	.026	504	.030
4^{A1}	1911	.089	3341	.134	7631	.278	14781	.500
5^{A1}	11454	.541	18644	.868	40214	1.82	76164	3.44
6^{A1}	68469	4.31	103019	6.49	206669	13.9	379419	23.8
2^{O}	31	.021	61	.022	151	.022	301	.025
4^{O}	201	.035	401	.041	1001	.057	2001	.092
5^{O}	481	.053	961	.072	2401	.132	4801	.243
6^{O}	1121	.102	2241	.164	5601	.351	11201	.662
8^{O}	5761	.621	11521	1.20	28801	2.89	57601	5.73

Table B.6: UPPAAL V3 Results for topo 4.6(a), p. 57: complete, all links 10%

MAX_M:	10		20		50		100	
N	#s	t(s)	#s	t(s)	#s	t(s)	#s	t(s)
2	95	.021	194	.024	491	.031	986	.042
3	14411	.548	433488	16.0	2,66M	101.1	6,21M	238.8
4	-	-						
2^{A1}	95	.022	194	.025	491	.030	986	.043
3^{A1}	14411	.550	401736	14.8	930619	34.5	1,09M	41.3
4^{A1}	-	-						
2^{O}	57	.022	114	.025	285	.025	570	.033
3^{O}	4386	.153	112852	3.76	716401	24.9	1,68M	58.5
4^{O}	>6M	>400						

Table B.7: UPPAAL V3 Results for topo 4.6(b), p. 57: chain, all links 100%

`MAX_M`:	10		20		50		100	
`N`	#s	t(s)	#s	t(s)	#s	t(s)	#s	t(s)
4	5237	.185	6069	.209	7959	.289	11109	.363
5	43582	1.61	47430	1.89	52680	1.94	61430	2.26
6	392809	17.1	415378	18.0	428788	18.7	451138	19.9
4^{A1}	1129	.057	1759	.078	3649	.135	6799	.229
5^{A1}	4995	.206	6745	.272	11995	.458	20745	.771
6^{A1}	22373	.974	26843	1.17	40253	1.73	62603	2.68
4^{O}	101	.030	201	.035	501	.043	1001	.053
5^{O}	141	.030	281	.034	701	.057	1401	.071
6^{O}	181	.034	361	.043	901	.064	1801	.098
8^{O}	261	.046	521	.061	1301	.098	2601	.161

Table B.8: UPPAAL V3 Results for topo 4.6(b), p. 57: chain, all links 10%

`MAX_M`:	10		20		50		100	
`N`	#s	t(s)	#s	t(s)	#s	t(s)	#s	t(s)
3	2928	.128	111718	4.07	2,12M	84.1	5,64M	225.8
4	122643	5.79	-	>900	-	-	-	-
3^{A1}	2928	.132	101717	3.70	447296	17.0	551774	21.2
4^{A1}	122643	5.92	-	>900	-	-	-	-
3^{O}	1049	.049	29498	.940	450982	14.9	1,16M	40.6
4^{O}	26530	.870	-	>900	-	-	-	-

B.5.2 Correct Parent and Distance

1. Along all paths eventually a state is reached in which for each node holds that its parent is a correct parent:

```
A<> forall(i:Node_id)
          isCorrectParent(i, Node(i).parent)
```

2. Along all paths eventually a state is reached in which for each node holds that its distance is correct (regarding a certain tolerance ϵ_l and ϵ_u, specified in the function):

```
A<> forall(i:Node_id)
          isCorrectDistanceToG(i, Node(i).D[i])
```

3. Along all paths eventually a state is reached in which each node selected a correct parent after `MAX_M` message rounds:

```
A<> forall(i:Node_id)
          (Node(i).M == MAX_M) &&
          isCorrectParent(i, Node(i).parent)
```

4. Along all paths eventually a state is reached in which each node computed the correct distance (regarding a certain tolerance) after `MAX_M` message rounds:

   ```
   A<> forall(i:Node_id)
             (Node(i).M == MAX_M) &&
             isCorrectDistanceToG(i, Node(i).D[i])
   ```

5. Along all paths it is always the case that whenever all nodes reached `MAX_M`, they also all selected a correct parent:

   ```
   A[] (forall(i:Node_id) Node(i).M == MAX_M) imply
       (forall(j:Node_id)
              isCorrectParent(j, Node(j).parent))
   ```

6. Along all paths it is always the case that whenever all nodes reached `MAX_M`, they also all computed the correct distance:

   ```
   A[] (forall(i:Node_id) Node(i).M == MAX_M) imply
       (forall(j:Node_id)
              isCorrectDistanceToG(j, Node(j).D[j])
   ```

7. Along all paths it is always the case that if all nodes reached message round `x`, they all selected a correct parent and keep selecting a correct parent up to message round `MAX_M`:

   ```
   A[] (forall(i:Node_id) Node(i).M >= x) imply
       (forall(j:Node_id)
              isCorrectParent(j, Node(j).parent))
   ```

8. Along all paths it is always the case that if all nodes reached message round `x`, they all computed the correct distance and keep computing the correct distance up to message round `MAX_M`:

   ```
   A[] (forall(i:Node_id) Node(i).M >= x) imply
       (forall(j:Node_id)
              isCorrectDistanceToG(j, Node(j).D[j])
   ```

Notes for the verification runs with these properties:

- We checked three 4-node topologies, in the tables below called T1, T2 and T3. T1 is a unique SPT topology obtained by substituting 33 for 18 in topology 4.6(d), T2 is topology 4.6(c) (two SPTs, simple link quality values) and T3 is topology 4.6(d) (two SPTs, more complex link qualities).
- OK means property is satisfied
- NOK means property fails
- In case of parent checking, the function `isCorrectParent(i,j)` returned `true` if nodes 0 and 3 selected parent node 0, and if nodes 1 and 2 selected parent node 3. For T2 and T3 it returned also `true` if node 1 selected parent 0.
- In case of distance checking, the function `isCorrectDistanceToG(i,d)` assumed the following distances (of resp. node 0, 1, 2 and 3):
  - Topology T1: 0, 41, 24, 11
  - Topology T2: 0, 40, 40, 20
  - Topology T3: 0, 67, 24, 11

  The values of ϵ_u and ϵ_l are reported in case of distance checks.
- In some cases the time (in seconds) taken by a verification run is also reported.
- Unless stated otherwise in the tables below, default settings are:
  - `ACCURACY=10`
  - `MAX_M=50`
  - `NOTURN=true`
- Superscript ^{t0} at the model version indicates that the global constant `NOTURN=false` and `turn=0` for the entire table row.
- Superscript ^{800} at the model version indicates that the global constant `MAX_M=800`.

Table B.9: UPPAAL Model V2 & V3 results for property 1 (correct parents)

P1	T1: 4.6(d)[33/18]	T2: 4.6(c)	T3: 4.6(d)
V2	.065s, OK	.105s, OK	.025s, OK
V3	88.3s, OK	34.3s, OK	276.0s, OK

Table B.10: UPPAAL Model V2 & V3 results for property 2 (correct distance)

P2	T1: 4.6(d)[33/18]	T2: 4.6(c)	T3: 4.6(d)
V2	$\epsilon_u = \epsilon_l = 0$ NOK $\epsilon_u = 2,\ \epsilon_l = 1$ OK $\epsilon_u = \epsilon_l = 0$ (MAX_M=400) OK	$\epsilon_u = \epsilon_l = 0$ NOK $\epsilon_u = 1,\ \epsilon_l = 0$ OK $\epsilon_u = \epsilon_l = 0$ (MAX_M=400) OK	$\epsilon_u = \epsilon_l = 0$ NOK $\epsilon_u = 5,\ \epsilon_l = 1$ OK $\epsilon_u = \epsilon_l = 0$ (MAX_M=400) OK
V3	(>300s) - $turn_0$,$\epsilon_u = \epsilon_l = 0$ OK $turn_2$ NOK	(>300s) - $turn_0$ NOK $turn_0$, MAX_M=100,$\epsilon_u = \epsilon_l = 0$ OK	(>300s) - $turn_0$ NOK $turn_0$, MAX_M=100 NOK

Table B.11: UPPAAL Model V2 & V3 results for property 3 (correct parents)

P3	T1: 4.6(d)[33/18]	T2: 4.6(c)	T3: 4.6(d)
V2	2.38s OK	6.39s OK	5.37s OK
$V3^{t0}$	23.1s OK	25.9s OK	>404s -

Table B.12: UPPAAL Model V2 & V3 results for property 4 (correct distance)

P4	T1: 4.6(d)[33/18]	T2: 4.6(c)	T3: 4.6(d)
V2	$\epsilon_u = \epsilon_l = 0$ NOK $\epsilon_u = 3,\ \epsilon_l = 1$ OK	$\epsilon_u = \epsilon_l = 0$ NOK $\epsilon_u = 1,\ \epsilon_l = 0$ (MAX_M=100) OK	$\epsilon_u = \epsilon_l = 0$ NOK $\epsilon_u = 5,\ \epsilon_l = 1$ OK
$V3^{t0}$	$\epsilon_u = \epsilon_l = 0$ NOK $\epsilon_u = 1,\ \epsilon_l = 1$ OK	$\epsilon_u = \epsilon_l = 0$ NOK $\epsilon_u = 0,\ \epsilon_l = 2$ OK	$\epsilon_u = \epsilon_l = 0$ NOK $\epsilon_u = 3,\ \epsilon_l = 6$ NOK $\epsilon_u = 3,\ \epsilon_l = 7$ -

Table B.13: UPPAAL Model V2 & V3 results for property 7 (correct parents)

P7	T1: 4.6(d)[33/18]	T2: 4.6(c)	T3: 4.6(d)
V2	1.31s x=10	.615s x=12	2.77s x=4
$V3^{t0}$	13.9s x=10	13.9s x=22	-

Table B.14: UPPAAL Model V2 & V3 results for property 8 (correct distance)

P8	T1: 4.6(d)[33/18]	T2: 4.6(c)	T3: 4.6(d)
$V2^{800}$	$\epsilon_u = 3,\ \epsilon_l = 1$ x=48 $\epsilon_u = 2,\ \epsilon_l = 1$ x=67 $\epsilon_u = 1,\ \epsilon_l = 1$ x=128	$\epsilon_u = 1,\ \epsilon_l = 0$ x=86 $\epsilon_u = \epsilon_l = 0$ x=88	$\epsilon_u = 2,\ \epsilon_l = 1$ x=184 $\epsilon_u = \epsilon_l = 1$ x=284 $\epsilon_u = 0,\ \epsilon_l = 1$ x=584
$V3^{t0}$	(MAX_M=100) x=43	(MAX_M=100) x=58	-

APPENDIX C

SPIN Model, Simulation and Verification Results

This appendix contains the complete source of our SPIN model, a section about the simulation of the behaviour of the model using SPIN's simulator mode, and a section with detailed verification results.

C.1 Protocol Model

```
/* SPT Protocol model for WSNs - W.M. Everse */

/* DEFINE CONSTANTS */
#define N              4                    /* Number of nodes */
#define MAX_M          100                  /* Max number of msg rounds */
#define MAX_DIST       10000                /* Represents 'infinite' distance */
#define ACCURACY       10                   /* Multiplication factor */
#define GATEWAY_COUNT  1                    /* Number of gateways */

/* TYPEDEFS & DECLARATIONS */
typedef NodeData{                           /* Node data:*/
    byte parent = 0;                        /* - selected parent */
    short R[N] = 0;                         /* - counts received msgs (per node) */
    short D[N] = MAX_DIST;                  /* - dist-to-G (per node) */
}

NodeData nodes[N];                          /* All node data */

short M;                                    /* Global msg round number */

typedef Tuple {byte u; byte v}
typedef NodeDimTuple{ Tuple to[N]; }
hidden NodeDimTuple C[N];                   /* Connectivity Matrix, NxN */
hidden NodeDimTuple H[N];                   /* History Matrix, NxN */

typedef NodeDimBool{ bool to[N]; }
hidden NodeDimBool Msgs[N];                 /* Message Exchange Matrix, NxN */

byte ctrl = N;                              /* Used for transferring control */
```

Listing C.1: PROMELA Model - Constants, Typedefs and Global Declarations

```
/*INLINE DECLARATIONS */
inline isGateway(id){                       /* Check if id is a gateway */
    (id < GATEWAY_COUNT);
}

inline receive(id){                         /* Update counters if received a msg */
    atomic{
        do
        :: (k < N) && (Msgs[k].to[id]) ->
                nodes[id].R[k]++;
                if
                :: isGateway(k) -> nodes[id].D[k] = 0   /* dist-to-G of G is always 0 */
                :: else -> nodes[id].D[k]=nodes[k].D[k]
                fi;
                k++
        :: (k < N) && !(Msgs[k].to[id]) -> k++
        :: (k==N) -> k=0; break
        od
    }
}

inline getMinimum(id){                      /* Compute minimum distance */
    atomic{
        minval = MAX_DIST;
        do
        :: (k < N) ->
            if
            :: (nodes[id].R[k] > 0) && (k != id) && (nodes[id].D[k] < MAX_DIST) ->
                try = (ACCURACY * M / nodes[id].R[k]) + nodes[id].D[k];
                try = ( ((ACCURACY*M)%nodes[id].R[k])>=(nodes[id].R[k]/2)->(try+1):try );
                if
                :: (try <= minval) -> minval = try; nodes[id].parent = k
                :: else -> skip
                fi
            :: else -> skip
            fi;
            k++
        :: (k == N) -> k=0; try=0; nodes[id].D[id] = minval; break
        od;
    }
}

inline setC(x, y, p, q){                    /* Sets entries in connectivity matrix C */
    C[x].to[y].u = p;
    C[x].to[y].v = q;
    C[y].to[x].u = p;                       /* Due to symmetry */
    C[y].to[x].v = q;                       /* Due to symmetry */
}

inline canLoose(a,b){                       /* Check if msgs still may be lost */
    (C[a].to[b].u - H[a].to[b].u) < (C[a].to[b].v - H[a].to[b].v)
}

inline canSend(c,d){                        /* Check if msgs still may be sent */
    H[c].to[d].u < C[c].to[d].u
}

inline checkReset(e,f){                     /* Reset counters when needed */
    if                                      /* (modulo C[].to[].v) */
    :: (H[e].to[f].v == C[e].to[f].v) -> H[e].to[f].u = 0; H[e].to[f].v = 0;
    :: else -> skip
    fi
}
```

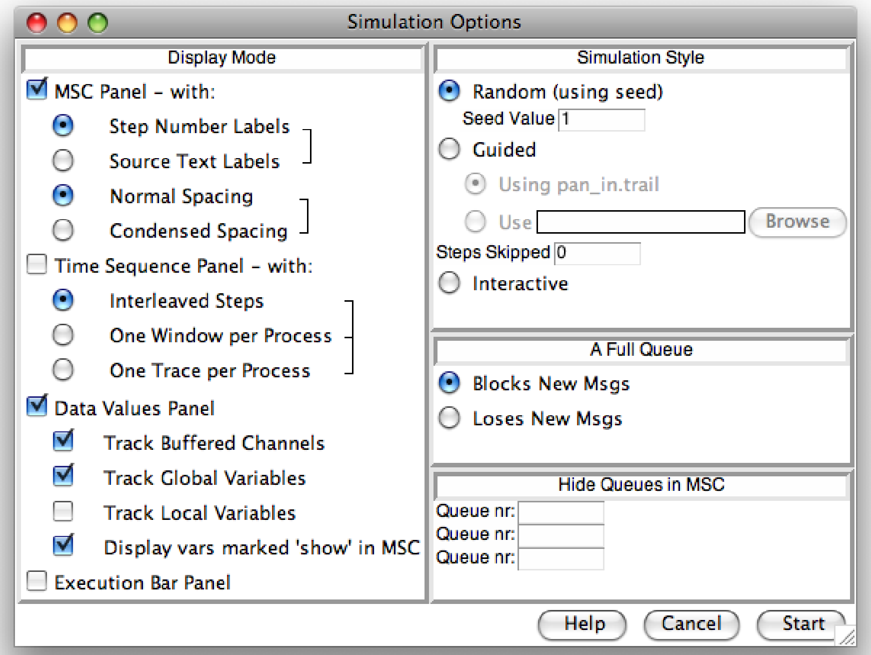
Listing C.2: PROMELA Model - Inline Constructs

```
/* PROCESS DECLARATIONS */
active[N] proctype node(){                            /* Node process */
 endN:  do
     :: ctrl == _pid ->                               /* Wait for control */
          byte k;                                     /* Used in inline constructs */
          short try,minval;                           /* Used in inline construct */
          getMinimum(_pid);                           /* Compute minimum dist-to-G */
          receive(_pid);                              /* 'receive' from all other nodes */
          ctrl++                                      /* Transfer control to next process */
     od
}

active proctype globalSend(){                         /* Simulates sending globally */
     byte i,j;

     d_step{                                          /* Fill symmetric connectivity matrix */
                                                      /* Using inline setC(from, to, u, v) */
          /*arbitrary topo, unique SPT [30/18] */
          setC(0,1,1,7); setC(0,2,1,10); setC(0,3,9,10);
          setC(1,3,3,10);
          setC(2,3,4,5)
     }

     do                                               /* For each msg round */
     :: (M < MAX_M) -> atomic{
          do                                          /* Fill msgs matrix */
          :: (i<N) -> do
               :: (j < N && C[i].to[j].u == 0) -> j++  /* No link between i and j */
               :: (j < N && C[i].to[j].u != 0) ->      /* There is a link between i and j */
                    if
                    :: canLoose(i,j) ->               /* ND-choice: write 0 in msgs matrix */
                         Msgs[i].to[j] = 0            /* 0 means msg gets lost */
                    :: canSend(i,j) ->                /* ND-choice: write 1 in msgs matrix */
                         Msgs[i].to[j] = 1;           /* 1 means msg will come across */
                         H[i].to[j].u++               /* Counts nr of delivered msgs */
                    fi;
                    H[i].to[j].v++;                   /* Counts msg opportunities */
                    checkReset(i,j);
                    j++
               :: (j==N) -> j=0; break
               od;
               i++
          :: (i==N) -> i=0; break
          od;
          ctrl = GATEWAY_COUNT;}                      /* Transfer control to 1st node process */
          ctrl == _pid;                               /* Wait for control */
          M++
     :: (M == MAX_M) -> break                         /* Max nr of msg rounds reached, stop */
     od
}
```

Listing C.3: PROMELA Model - Process Declarations

C.2 Validation by Simulation

During model construction, we validated the behaviour of our PROMELA model of the protocol using SPIN in simulation mode. The GUI XSpin displays the simulation options orderly in the 'Simulation Options' panel (figure C.1), accessible via the 'Set Simulation Parameters..' action of the 'Run..' menu.


Simulation Options

Display Mode
- [✓] MSC Panel - with:
  - [✓ Step Number Labels] Source Text Labels
  - [✓ Normal Spacing] Condensed Spacing
- [ ] Time Sequence Panel - with:
  - [✓ Interleaved Steps] One Window per Process / One Trace per Process
- [✓] Data Values Panel
  - [✓] Track Buffered Channels
  - [✓] Track Global Variables
  - [ ] Track Local Variables
  - [✓] Display vars marked 'show' in MSC
- [ ] Execution Bar Panel

Simulation Style
- [✓ Random (using seed)] Guided / Interactive
  - Seed Value: 1
  - Guided: [✓ Using pan_in.trail] Use ____ (Browse)
  - Steps Skipped: 0

A Full Queue
- [✓ Blocks New Msgs] Loses New Msgs

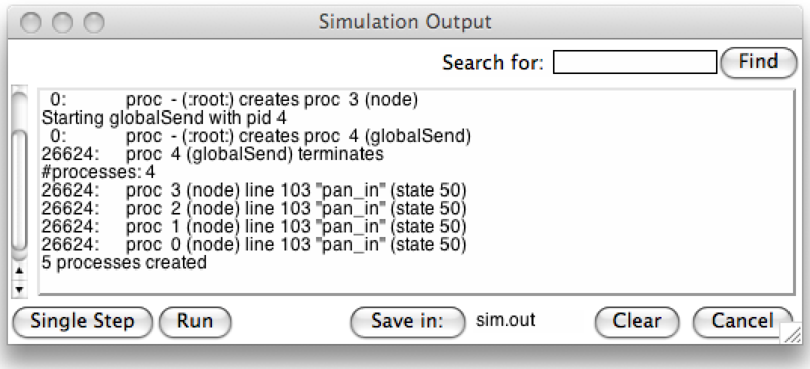
Hide Queues in MSC
- Queue nr:
- Queue nr:
- Queue nr:

Figure C.1: The Simulation Options Panel

As can be seen in the options panel, there are three styles of simulation: random, guided and interactive. A random simulation allows for running a simulation run of the model by letting SPIN decide on which transitions to take (and resolve non-determinism). A guided simulation needs a pan_in.trail [1] file, normally created during a verification run that resulted in a counterexample of a correctness property. The number of steps that should be skipped before displaying a random or guided simulation can be specified. An interactive simulation allows for a custom simulation run, leaving the choice of which transition to take to the user.

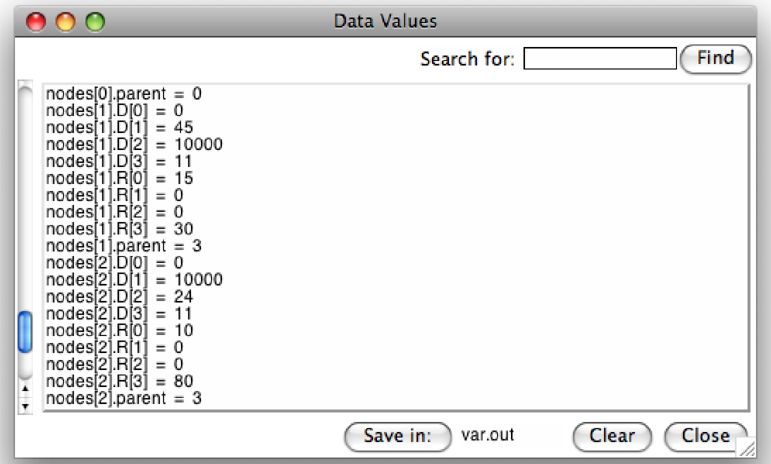
[1] A model that is opened in XSpin is copied into a file called pan_in, in order to avoid overwriting the original. The file to which a possible counterexample is written during verification is also called pan_in, but with the .trail extension.

Besides the XSpin main window, starting the simulation in XSpin results in at least three additional windows: a Simulation Output window that shows the transitions taken, a Data Values window that shows the global data values, and optionally also the local data values, and a Sequence Chart window that displays a MSC of the simulation. Unfortunately, the sequence chart and the data is often only showed at the end of a simulation run, rather then 'real-time' during simulation. Examples of these three windows and their contents at the end of a random simulation of the model (as presented above) are shown in figures C.2 – C.4.



```
  0:       proc  - (:root:) creates proc  3 (node)
Starting globalSend with pid 4
  0:       proc  - (:root:) creates proc  4 (globalSend)
26624:     proc  4 (globalSend) terminates
#processes: 4
26624:     proc  3 (node) line 103 "pan_in" (state 50)
26624:     proc  2 (node) line 103 "pan_in" (state 50)
26624:     proc  1 (node) line 103 "pan_in" (state 50)
26624:     proc  0 (node) line 103 "pan_in" (state 50)
5 processes created
```

Figure C.2: The Simulation Output Window



```
nodes[0].parent = 0
nodes[1].D[0] = 0
nodes[1].D[1] = 45
nodes[1].D[2] = 10000
nodes[1].D[3] = 11
nodes[1].R[0] = 15
nodes[1].R[1] = 0
nodes[1].R[2] = 0
nodes[1].R[3] = 30
nodes[1].parent = 3
nodes[2].D[0] = 0
nodes[2].D[1] = 10000
nodes[2].D[2] = 24
nodes[2].D[3] = 11
nodes[2].R[0] = 10
nodes[2].R[1] = 0
nodes[2].R[2] = 0
nodes[2].R[3] = 80
nodes[2].parent = 3
```

Figure C.3: The Data Values Window

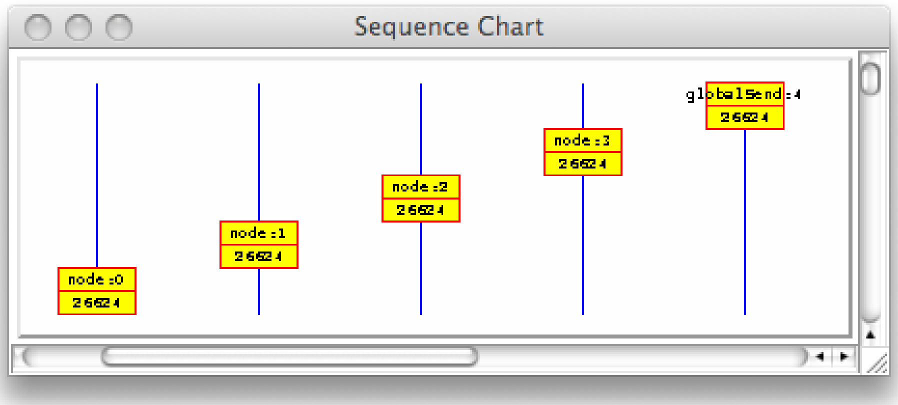


Figure C.4: The Sequence Chart Window

The figures show the end state of the random simulation run of our model as we presented it in this thesis. We can check the correctness of this simulation run by checking the node data values displayed in C.3. The corresponding topology is depicted below.

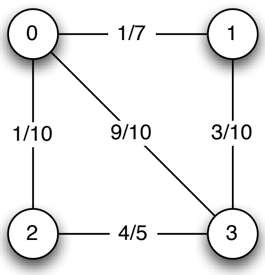
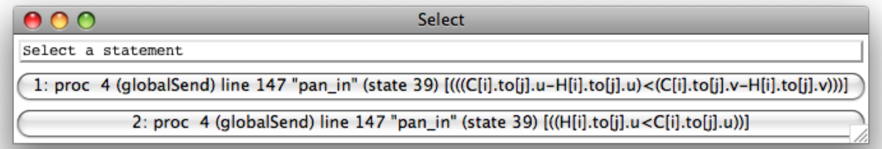


Figure C.5: 4-node topology specified by lines 108 –114 of listing C.3

It can be seen (partly) in figure C.3 that this simulation run resulted in correct values. For instance, both node 1 and node 2 found a correct parent (i.e. node 3) and they both found the correct distance to the gateway (i.e. node 1 found 45 and node 2 found 24). Note that the following parameter settings were used for this simulation: ACCURACY=10, MAX_M=100.

An interactive simulation results in one more window: the Select window, that enables the user to select one of the enabled transitions, whenever there are several enabled transitions. This select window is shown in figure C.6.



```
Select a statement
1: proc  4 (globalSend) line 147 "pan_in" (state 39) [((((C[i].to[j].u-H[i].to[j].u)<(C[i].to[j].v-H[i].to[j].v)))]
2: proc  4 (globalSend) line 147 "pan_in" (state 39) [((H[i].to[j].u<C[i].to[j].u))]
```

Figure C.6: The Select Window

In figure C.6, there are two enabled transitions. These are the transitions corresponding to the (statement) expressions in the **inline** constructs canLoose(a,b) and canSend(c,d) (see listing C.2). Our experience on our MacBook is that this window often needs a small manual resize before it shows all buttons (enabled transitions) properly.

C.3 Verification Results

Notation used in this section:

- The number of nodes is denoted by N
- The number of message rounds verified is denoted by MAX_M
- The global constant ACCURACY is set to 10 by default. Verification runs with ACCURACY set to 1 are annotated with the superscript: ^{A1}
- The time reported is the `real time` measured by the UNIX utility `time`, as reported by SPIN's verification output.
- The number of states reported are the number of *stored states* as indicated by SPIN. The number of *matched states* (if any) is reported in italics (on a new line).
- A down arrow (↓) in the number-of-nodes column indicates similar results for a higher numbers of nodes
- Table entries with a sole dash (-) indicate that the corresponding verification run was terminated abnormally (aborted since it did not end within reasonable time).
- Empty table entries correspond to less interesting verification runs that we did not execute.

C.3.1 Deadlock Freedom

Absence of deadlock is checked in SPIN by checking for invalid end states.

Table C.1: SPIN–results for topo 5.6(a), p. 82: complete, all links 100%

MAX_M:	10		20		50		100	
N	#s	t(s)	#s	t(s)	#s	t(s)	#s	t(s)
2	84	.08	164	.08	404	.08	804	.08
4	164	.08	324	.08	804	.09	1604	.10
8	324	.09	644	.09	1604	.10	3204	.13
↓								
4^{A1}	164	.08	324	.08	804	.09	1604	.10

Table C.2: SPIN–results for topo 5.6(a), p. 82: complete, all links 10%

MAX_M:	10		20		50		100	
N	#s	t(s)	#s	t(s)	#s	t(s)	#s	t(s)
2	84	.08	164	.08	404	.08	804	.08
	27		*54*		*135*		*270*	
4	164	.30	324	.46	804	.99	1604	1.74
	36855		*73710*		*184275*		*368550*	
5	204	44.6	404	89.3	1004	223	2004	439
	9,44M		*18,87M*		*47,19M*		*94,37M*	
4^{A1}	164	.30	324	.46	804	.99	1604	1.74
	36855		*73710*		*184275*		*368550*	

Table C.3: SPIN–results for topo 5.6(b), p. 82: chain, all links 10%

MAX_M:	10		20		50		100	
N	#s	t(s)	#s	t(s)	#s	t(s)	#s	t(s)
3	124	.08	244	.08	604	.09	1204	.09
	135		*270*		*675*		*1350*	
4	164	.09	324	.09	804	.12	1604	.12
	567		*1134*		*2835*		*5670*	
8	324	1.12	644	2.12	1604	5.12	3204	9.77
	147447		*294894*		*737235*		*1,47M*	
↓								

C.3.2 Correct Parent and Distance

1. It is always the case that eventually a state can be reached in which all non-gateway nodes found the specified parent (i.e. the values specified for parameters `u`, `v` and `w`) after `MAX_M` message rounds:

```
[]<> (p && q)
#define p    M==MAX_M
#define q    nodes[1].parent==u &&
             nodes[2].parent==v &&
             nodes[3].parent==w
```

2. It is always the case that eventually a state can be reached in which all non-gateway nodes found the specified distance (i.e. the values specified for parameters u, v and w) after MAX_M message rounds:

   ```
   []<> (p && q)
   #define p    M==MAX_M
   #define q    nodes[1].D[1]==u &&
                nodes[2].D[2]==v &&
                nodes[3].D[3]==w
   ```

3. It is always the case that if all nodes reached message round x, they all selected the specified parent (i.e. the values specified for parameters u, v and w) and keep selecting the specified parent up to message round MAX_M:

   ```
   [] (p -> q)
   #define p    M >= x
   #define q    nodes[1].parent==u &&
                nodes[2].parent==v &&
                nodes[3].parent==w
   ```

4. It is always the case that if all nodes reached message round x, they all computed the specified distance (i.e. the values specified for parameters u, v and w) and keep computing the specified distance up to message round MAX_M:

   ```
   [] (p -> q)
   #define p    M >= x
   #define q    nodes[1].D[1]==u &&
                nodes[2].D[2]==v &&
                nodes[3].D[3]==w
   ```

Notes for the verification runs with these properties:

- We checked three 4-node topologies, called T1, T2 and T3:
  - T1 is a unique SPT topology obtained by substituting 33 for 18 in topology 5.6(d).
  - T2 is topology 5.6(c) (two SPTs, simple link quality values).
  - T3 is topology 5.6(d) (two SPTs, more complex link qualities).
- In case of parent checking (properties 1 and 3), property parameters u, v and w are specified respectively as follows:
  - for T1: 3, 3, 0

  - for T2 and T3: (3 or 0), 3, 0

- In case of distance checking (properties 2 and 4), property parameters `u`, `v` and `w` are specified respectively as follows:
  - for T1: 41, 24, 11
  - for T2: 40, 40, 20
  - for T3: 66, 24, 11
- Unless stated otherwise in the tables, the default settings for the verification runs are MAX_M=50 and ACCURACY=10.
- Table entries contain:
  - number of states in format: stored/matched(visited)
  - number of seconds the run took
  - for properties 3 and 4: the value found for parameter `x`
  - value of MAX_M if other than 50
- IEC = Invalid End state Check, for comparison purposes.

Table C.4: SPIN results for properties 1 & 3 (correct parents)

	T1: 5.6(d)[33/18]	T2: 5.6(c)	T3: 5.6(d)
P1	1605/27544(2406) 0.29s	1605/89296(2406) 0.66s	1605/27208(2406) 0.29s
P3	804/6487 0.13s x=7	804/21925 0.24s x=5	804/6403 0.13s x=5
IEC	*804/6486* *0.12s*	*804/21924* *0.18s*	*804/6402* *0.12s*

Table C.5: SPIN results for properties 2 & 4 (correct distance)

	T1: 5.6(d)[33/18]	T2: 5.6(c)	T3: 5.6(d)
P2	3205/49568(4806) 0.48s MAX_M=100	1605/89296(2406) 0.66s	3205/56048(4806) 0.52s MAX_M=100
IEC	*1604/11592* MAX_M=100 *0.15*s	*804/21924* *0.18s*	*1604/13212* MAX_M=100 *0.16s*

P4	8004/58747
	MAX_M=500
	0.65s
	x=126
		8004/219226
		MAX_M=500
		1.68s
		x=45
			32004/266857
			MAX_M=2000
			2.56s
			x=1001

C.3.3 Stored and Matched States

This section experimentally validates the results of the found formulæ for the number of stored states s and the number of matched states m of our model against the numbers reported by SPIN. It also experimentally shows that the number of matched states m of our model reported by SPIN is independent of the number of nodes. Finally, we also attempted to experimentally find an expression for p in the formula for matched states.

The formulæ:

- $s = 4 + 4 \cdot N \cdot MAX\_M$
- $m = p \cdot MAX\_M \cdot (2^E - 1)$

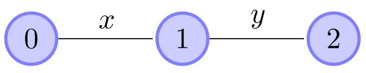
where N and MAX_M are model parameters denoting respectively the number of nodes and the maximum number of message rounds to verify. p is some probability that depends on the link quality and E is the number of *directed* edges in the topology under consideration.

We listed SPIN's output for an invalid end state check of our model with ten 50% links ($E = 20$ directed links of quality $q = 0.5$) below. Further, model parameter MAX_M=10 and $p = 1 - 0.5 = 0.5$ (we already saw that $p = 1 - q$ if all links have the same quality q). The topology under consideration is a ring topology (i.e. a chain topology with an additional link between the first and the last node of the chain). The remaining links connect arbitrary pairs of non-neighbouring nodes in the ring. We start with N=5 nodes (thus completely connected), and increase the number of nodes up to N=10, which is just a ring of nodes. Finally, we also checked N=11, a ring topo in which a connection misses: thus just a chain on 11 nodes.

- N=5, formulæ outcome OK: 204/5242875, 25.42s.
- N=6, formulæ outcome OK: 244/5242875, 27.68s.
- N=7, formulæ outcome OK: 284/5242875, 31.48s.
- N=8, formulæ outcome OK: 324/5242875, 32.70s.
- N=9, formulæ outcome OK: 364/5242875, 38.48s.
- N=10, formulæ outcome OK: 404/5242875, 41.23s.
- N=11, formulæ outcome OK: 444/5242875, 45.43s.

The formulæ outcome is equal to the results reported by SPIN, the number of matched states is constant and it is much bigger than the number of stored states. The determinant of the feasibility of a verification run is the number of directed edges in the topology.

What about p in the formula for matched states? We know that for topologies in which all links have the same quality, say q, p equals $1-q$. But what about topologies in which different link qualities appear? As an attempt to find an expression for p, we take a very simple topology:



$$m = p \cdot MAX\_M \cdot (2^4 - 1) = 1500p$$

x	y	m	$p = \frac{m}{1500}$	$p(\%)$
$\frac{3}{4}$	$\frac{4}{4}$	75	$\frac{1}{20}$	5.0
	$\frac{3}{4}$	375	$\frac{1}{4}$	25.0
	$\frac{2}{4}$	450	$\frac{3}{10}$	30.0
	$\frac{1}{4}$	525	$\frac{7}{20}$	35.0
	$\frac{0}{4}$	75	$\frac{1}{20}$	5.0
	$\frac{9}{10}$	150	$\frac{1}{10}$	10.0
	$\frac{1}{10}$	570	$\frac{19}{50}$	38.0
	$\frac{1}{50}$	594	$\frac{99}{250}$	39.6
	$\frac{1}{100}$	597	$\frac{199}{500}$	39.8
$\frac{9}{10}$		417	$\frac{139}{500}$	27.8
$\frac{99}{100}$		309	$\frac{103}{500}$	20.6
$\frac{9}{10}$	$\frac{1}{10}$	390	$\frac{13}{50}$	26.0

The formula for the number of matched states for the case with this topology and $MAX\_M = 100$ is on the right of the figure. Below we listed the experiments we did using this simple scenario. It can be observed that the first part of the table approaches 40%. We were however unable to derive an expression for p based on link qualities.

APPENDIX D

PRISM Model, Simulation and Verification Results

This appendix contains the complete source of our PRISM models and sections about PRISM's simulator and verifier.

D.1 Protocol Model

The V1 model presented in chapter 7 could not be built by PRISM. Model V2 is an updated version of V1, which can be built (more info, see chap. 7).

D.1.1 Model V2

```
//nondeterminism in execution order: mdp
mdp

//number of message rounds
const int MAX_M;

//big number representing infinite distance
const int MAX_DIST  = 10;

//bidirectional link qualities
const double p01 = 0.15;
const double p02 = 0.1;
const double p03 = 0.9;
const double p12 = 0;
const double p13 = 0.33;
const double p23 = 0.8;

//node representations used for parent selection
const int n0 = 0;
const int n1 = 1;
const int n2 = 2;
const int n3 = 3;

//distance to G of G is always 0
const int dist00 = 0;
```

Listing D.1: PRISM Model V2 - Part 1

```
//gateway node
module node0
    M0      : [0..MAX_M] init 0;          //my msg round counter

    //if (not sent this msg round) & (max round not yet reached),
    //broadcast a msg (by sync) and increase msg round nr
    [send0] (M0<=M1 & M0<=M2 & M0<=M3) & (M0<MAX_M)  -> (M0'=M0+1);
endmodule

//node 1
module node1
    dist10 : [0..MAX_DIST] init MAX_DIST;   //distance to G of node 0 acc to me
    dist11 : [0..MAX_DIST] init MAX_DIST;   //my own distance to G
    dist12 : [0..MAX_DIST] init MAX_DIST;   //distance to G of node 2 acc to me
    dist13 : [0..MAX_DIST] init MAX_DIST;   //distance to G of node 3 acc to me

    recv10 : [1..1+MAX_M*10] init 1;        //#msgs I received from node 0
    recv12 : [1..1+MAX_M*10] init 1;        //#msgs I received from node 2
    recv13 : [1..1+MAX_M*10] init 1;        //#msgs I received from node 3

    parent1: [0..3];                        //the parent node I selected

    M1      : [0..MAX_M] init 0;              //my msg round counter

    //if (not sent this msg round) & (max round not yet reached),
    //broadcast a msg (by sync), update M1, dist-to-G and parent
    [send1] (M1<=M0 & M1<=M2 & M1<=M3) & (M1<MAX_M) ->
            (M1'=M1+1) & (dist11'=computedist1) & (parent1'=setparent1);

    //receive (sync) msg from node 0 with probability p01
    [send0] recv10<(10*MAX_M-8) -> p01: (dist10'=dist00) & (recv10'=recv10+10) + (1-p01):
         true;

    //receive (sync) msg from node 2 with probability p12
    [send2] recv10<(10*MAX_M-8) -> p12: (dist12'=dist22) & (recv12'=recv12+10) + (1-p12):
         true;

    //receive (sync) msg from node 3 with probability p13
    [send3] recv10<(10*MAX_M-8) -> p13: (dist13'=dist33) & (recv13'=recv13+10) + (1-p13):
         true;
endmodule


//define node 2 by module renaming
module node2 = node1 [  //send transition:
            send1=send2, M1=M2, M2=M1, dist11=dist22, parent1=parent2,
            //recv from node 0 (common neighbour):
            p01=p02, dist10=dist20, recv10=recv20,
            //recv from node 3 (common neighbour):
            p13=p23, dist13=dist23, recv13=recv23,
            //recv from node 1 (reversed):
            send2=send1, dist12=dist21, dist22=dist11,recv12=recv21,
            //parent selection:
            n2=n1
]endmodule

//define node 3 by module renaming
module node3 = node1 [  //send transition:
            send1=send3, M1=M3, M3=M1, dist11=dist33, parent1=parent3,
            //recv from node 0 (common neighbour):
            p01=p03, dist10=dist30, recv10=recv30,
            //recv from node 2 (common neighbour):
            p12=p23, dist12=dist32, recv12=recv32,
            //recv from node 1 (reversed):
            send3=send1, dist13=dist31, dist33=dist11,recv13=recv31,
            //parent selection
            n3=n1
]endmodule

//formula for distance computation
formula computedist1 = min(ceil(10*M1/recv10 + dist10), ceil(10*M1/recv12 + dist12),
                ceil(10*M1/recv13 + dist13), MAX_DIST);
//formula for parent selection
formula setparent1 = (computedist1=ceil(10*M1/recv10 + dist10)) ? n0 :
                (computedist1=ceil(10*M1/recv12 + dist12)) ? n2 : n3 ;
```

Listing D.2: PRISM Model V2 - Part 2

D.1.2 Model V3 (Fixed Order)

Model V3 is based on model V2, but the execution order of the node modules is fixed. This is achieved by adding a local variable `turnx` (where `x` is a node number) to each module. It is used to keep track of the node whose turn it is to broadcast a probe message. Since this eliminates the non-determinism in the model, the model type is a `dtmc` instead of a `mdp`.

```
//fixed execution order, no nondeterminism: dtmc
dtmc

//number of message rounds
const int MAX_M;

//big number representing infinite distance
const int MAX_DIST  = 10;

//bidirectional link qualities
const double p01 = 0.15;
const double p02 = 0.1;
const double p03 = 0.9;
const double p12 = 0;
const double p13 = 0.33;
const double p23 = 0.8;

//number of nodes for modulo computation
const int N = 4;

//node representations used for parent selection
const int n0 = 0;
const int n1 = 1;
const int n2 = 2;
const int n3 = 3;

//distance to G of G is always 0
const int dist00 = 0;

//gateway node
module node0
    M0      : [0..MAX_M] init 0;              //my msg round counter
    turn0   : [0..N-1] init n0;               //indicates whose turn it is

    //if (it is my turn) & (max round not yet reached),
    //broadcast a msg (by sync), increase msg round & update turn
    [send0] (turn0=n0) & (M0<MAX_M) -> (M0'=M0+1) & (turn0' = nextG);

    //update turn variable mod N
    [send1] (turn0=n1) -> (turn0'= nextG);
    [send2] (turn0=n2) -> (turn0'= nextG);
    [send3] (turn0=n3) -> (turn0'= nextG);
endmodule
```

Listing D.3: PRISM Model V3 - Part 1

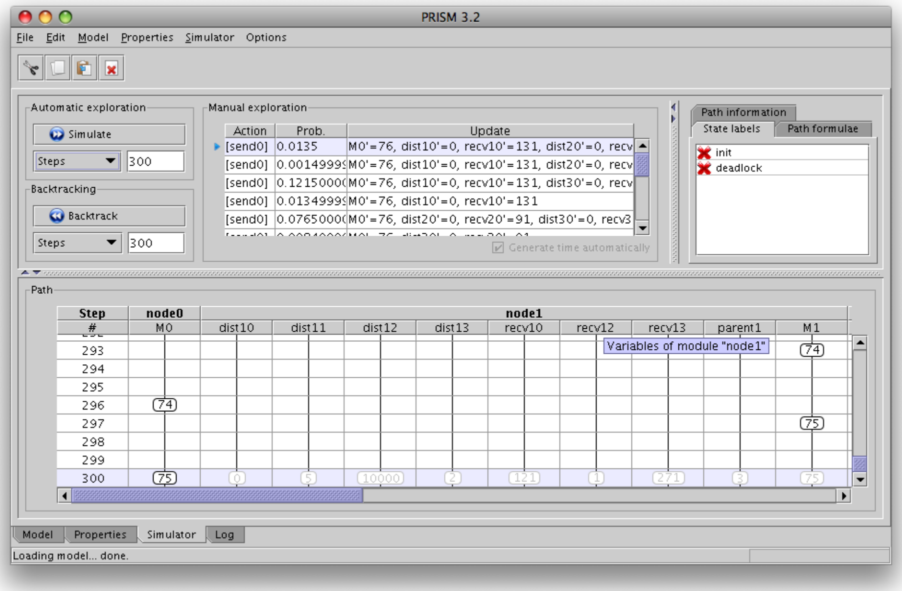
Each transition updates the value of the `turn` variable. Without the use of a formula to do this, it would result in many duplicate code. Therefore, we added two new formulæ `nextG` and `next` for use in respectively the gateway module and the node modules to update the `turn` variable (lines 110-112 in listing D.4 below).

```

//node 1
module node1
    dist10 : [0..MAX_DIST] init MAX_DIST;   //distance to G of node 0 acc to me
    dist11 : [0..MAX_DIST] init MAX_DIST;   //my own distance to G
    dist12 : [0..MAX_DIST] init MAX_DIST;   //distance to G of node 2 acc to me
    dist13 : [0..MAX_DIST] init MAX_DIST;   //distance to G of node 3 acc to me

    recv10 : [1..1+MAX_M*10] init 1;        //#msgs I received from node 0
    recv12 : [1..1+MAX_M*10] init 1;        //#msgs I received from node 2
    recv13 : [1..1+MAX_M*10] init 1;        //#msgs I received from node 3

    parent1: [0..N-1];                      //the parent node I selected

    M1      : [0..MAX_M] init 0;            //my msg round counter

    turn1   : [0..N-1] init n0;             //indicates whose turn it is

    //if (it is my turn) & (max round not yet reached),
    //broadcast a msg (by sync), update M1, dist-to-G, parent & turn
    [send1] (turn1=n1) & (M1<MAX_M) -> (M1'=M1+1) & (dist11'=computedist1) &
            (parent1'=setparent1) & (turn1'= next);

    //receive (sync) msg from node 0 with probability p01
    [send0] (turn1=n0) & (recv10<(10*MAX_M-8)) ->
            p01: (dist10'=dist00) & (recv10'=recv10+10) & (turn1'= next) +
            (1-p01): (turn1'= next);

    //receive (sync) msg from node 2 with probability p12
    [send2] (turn1=n2) & (recv12<(10*MAX_M-8)) ->
            p12: (dist12'=dist22) & (recv12'=recv12+10) & (turn1'= next) +
            (1-p12): (turn1'= next);

    //receive (sync) msg from node 3 with probability p13
    [send3] (turn1=n3) & (recv13<(10*MAX_M-8)) ->
            p13: (dist13'=dist33) & (recv13'=recv13+10) & (turn1'= next) +
            (1-p13): (turn1'= next);
endmodule

//define node 2 by module renaming
module node2 = node1 [  //send transition:
    send1=send2, M1=M2, M2=M1, dist11=dist22, parent1=parent2,
    //recv from node 0 (common neighbour):
    p01=p02, dist10=dist20, recv10=recv20,
    //recv from node 3 (common neighbour):
    p13=p23, dist13=dist23, recv13=recv23,
    //recv from node 1 (reversed):
    send2=send1, dist12=dist21, dist22=dist11,recv12=recv21,
    //parent selection and turn:
    n2=n1, n1=n2, turn1=turn2
]endmodule

//define node 3 by module renaming
module node3 = node1 [  //send transition:
    send1=send3, M1=M3, M3=M1, dist11=dist33, parent1=parent3,
    //recv from node 0 (common neighbour):
    p01=p03, dist10=dist30, recv10=recv30,
    //recv from node 2 (common neighbour):
    p12=p23, dist12=dist32, recv12=recv32,
    //recv from node 1 (reversed):
    send3=send1, dist13=dist31, dist33=dist11,recv13=recv31,
    //parent selection and turn:
    n3=n1, n1=n3, turn1=turn3
]endmodule

//formulae to update turn variable modulo N
formula nextG = func(mod, turn0+1, N);
formula next = func(mod, turn1+1, N);

//formula for distance computation
formula computedist1 = min(ceil(10*M1/recv10 + dist10), ceil(10*M1/recv12 + dist12),
            ceil(10*M1/recv13 + dist13), MAX_DIST);
//formula for parent selection
formula setparent1 = (computedist1=ceil(10*M1/recv10 + dist10)) ? n0 :
            (computedist1=ceil(10*M1/recv12 + dist12)) ? n2 : n3 ;
```

Listing D.4: PRISM Model V3 - Part 2

D.2 Validation by Simulation

PRISM's GUI provides a powerful simulator to debug (or validate) the behaviour of a model. It is depicted in figure D.1 below. Enabled transitions together with their associated probabilities are shown and can be selected manually. Automatic simulation of a number of steps is also possible, and so is backtracking (a number of steps) on a simulation path. Either the full state of all modules or a user-selected set of module variables is shown for each simulated step, thereby visualizing the simulated path. The initial state is often specified in the model itself but the simulator also allows for changing it at the start of a new simulation path. It also asks the user for a value for all undefined constants (such as `MAX_M` in our model).



Action	Prob.	Update
[send0]	0.0135	M0'=76, dist10'=0, recv10'=131, dist20'=0, recv
[send0]	0.00149999	M0'=76, dist10'=0, recv10'=131, dist20'=0, recv
[send0]	0.12150000	M0'=76, dist10'=0, recv10'=131, dist30'=0, recv
[send0]	0.01349999	M0'=76, dist10'=0, recv10'=131
[send0]	0.07650000	M0'=76, dist20'=0, recv20'=91, dist30'=0, recv3

Step	node0	node1								
#	M0	dist10	dist11	dist12	dist13	recv10	recv12	recv13	parent1	M1
293										74
294										
295										
296	74									
297										75
298										
299										
300	75	0	5	10000	2	121	1	271	3	75

Figure D.1: The simulator tab of the PRISM GUI

We used the simulator to validate our ideas about the constructed PRISM model. Our model V1 for example (as presented in chapter 7) can perfectly be simulated. As can be seen in the model source, the topology simulated is our (in the mean time well-known) topology on 4 nodes, figure 5.6(d) on page 82 with 33% substituted for 18%. Automatic simulation often results in for instance node 1 selecting node 3 as parent node, but not always: sometimes it selects node 0 as its parent. This is due to the 'real' probabilities computed by PRISM.

D.2.1 Workaround Zero Dividing

This subsection explains the workaround that we applied to avoid zero dividing in distance computation. In an early version of our model of a 4-node topology, the formula for distance computation was as follows:

```
//node 1 computes its distance
formula computedist1 = min(ceil(M1/recv10 + dist10), ceil(M1/recv12 + dist12),
                        ceil(M1/recv13 + dist13), MAX_DIST);
```

Since the receive counters `recvxy` (where `x,y` are node numbers) were initialized to 0, this resulted in zero dividing, which is undesired for obvious reasons. In PRISM simulation mode, it resulted for instance in arbitrary large, negative integer values (rather than an error, though). To avoid this, we modified the formula:

```
//node 1 computes its distance
formula computedist1 = (recv10>0 & recv12>0 & recv13>0) ? min(ceil(M1/recv10 + dist10),
                         ceil(M1/recv12 + dist12), ceil(M1/recv13 + dist13), MAX_DIST)
                       : MAX_DIST;
```

However, this is of course incorrect: now distance is only computed when a node received at least one message from *all* possible neighbours. But this may never be the case, resulting in a node that never computes its distance-to-G. A solution could be to check for all possibilities (i.e. did/did not receive at least one message for each possible neighbour), but the number of possibilities is exponential in the number of nodes and that would heavily reduce the extendibility of the model (which was already not very good).

A more convenient solution is a workaround: we initialize the receive counters `recvxy` to 1 instead of 0, thus eliminating zero dividing. This however results in incorrect distance computation: especially in the first few message rounds, all nodes think that they are very well-connected. To reduce the impact of this error, we increase the receive counters with 10 (instead of 1) every time a probe message is received and we multiply the message round counters with 10 in the formula for distance computation. Now the error introduced is 10 times smaller and it is simply ignored.

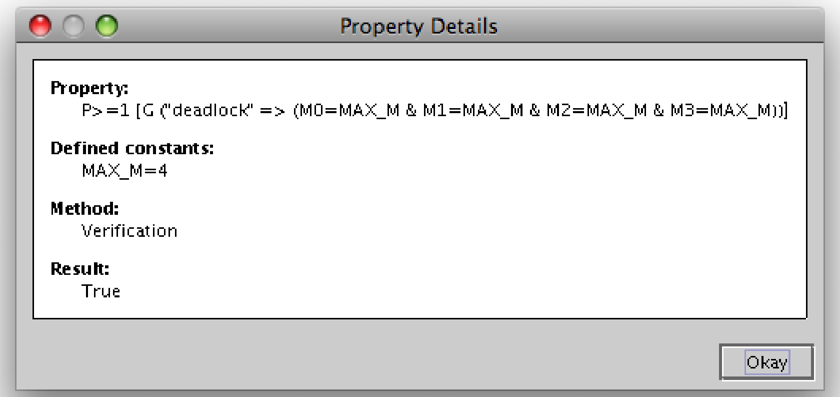
Note that we also had to adjust the declared range of the receive counters. In order to be able to use this workaround. The range should be `[1..1+MAX_M*10]` instead of `[0..MAX_M]`. Note also that the guards in model V2 and V3 take this range into account (`recvxy < (10*MAX_M-8)`).

D.3 Verification Examples

As said in chapter 7, PRISM can add self-loops to deadlock states that are encountered during model construction (building). These states can be referenced to from properties, using the state label `"deadlock"`. In our models, deadlock states should only occur at the 'end' of the verification run (when all nodes reached `MAX_M`), as it was the case with UPPAAL. Therefore, the following (qualitative) property can be used to check whether there are any other (unwanted) deadlocks:

```
//Property for deadlock freedom
P>=1 [G ("deadlock" => (M0=MAX_M & M1=MAX_M & M2=MAX_M & M3=MAX_M))]
```

In words: there is a probability of 1 that, whenever a deadlock state is found, all nodes reached the maximum message round number in that state (i.e. it is a 'valid' deadlock state). Verification of this property for model V2 with `MAX_M=4` in PRISM's GUI took 3.16 s, resulting in the following window:



Property:
P>=1 [G ("deadlock" => (M0=MAX_M & M1=MAX_M & M2=MAX_M & M3=MAX_M))]

Defined constants:
MAX_M=4

Method:
Verification

Result:
True

Figure D.2: Property details in PRISM's GUI

Another interesting (quantitative) property to check is the following:

```
//Property for parent selection
Pmin=? [F parent1=n3]
```

In words: what is the minimum probability that node 1 eventually selects node 3 as its parent? Unfortunately, PRISM did not manage to check it within 15 minutes (for the same model).

APPENDIX E

SPIN Models of Protocol Variations

This appendix contains the source of our SPIN models of variations of the SPT protocol.

E.1 Finite Sliding Window Variant MA

This section contains the PROMELA source of the finite sliding window variant of our original model, with a simple MA of the link quality.

```
/* SPT Protocol model for WSNs - W.M. Everse */
/* Experiments with finite sliding window, SMA */

/* DEFINE CONSTANTS */
#define N               4               /* Number of nodes */
#define WIN_SIZE        10              /* Window size in msg rounds */
#define MAX_WIN         10              /* Max number of windows */
#define MAX_DIST        10000           /* Represents 'infinite' distance */
#define ACCURACY        10              /* Multiplication factor */
#define GATEWAY_COUNT   1               /* Number of gateways */

/* TYPEDEFS & DECLARATIONS */
local byte WIN_CNT = 0;                 /* Counts the nr of passed windows */

typedef NodeData{                       /* Node data:*/
    byte parent = 0;                    /* - selected parent */
    byte R[N] ;                         /* - counts received msgs (per node) */
    chan W[N] = [WIN_SIZE] of {bool};   /* - most recent window with recv info (per node) */
    short D[N] = MAX_DIST;              /* - dist-to-G (per node) */
}

NodeData nodes[N];                      /* All node data */

local byte M;                           /* Global msg round number */

typedef Tuple {byte u; byte v}
typedef NodeDimTuple{ Tuple to[N] }
hidden NodeDimTuple C[N];               /* Connectivity Matrix, NxN */
hidden NodeDimTuple H[N];               /* History Matrix, NxN */

typedef NodeDimBool{ bool to[N]; }
```

Listing E.1: Finite Sliding Window MA - part 1/3

```
hidden NodeDimBool msgs[N];                          /* Message Exchange Matrix, NxN */

byte ctrl = N;                                       /* Used for transferring control */

/*INLINE DECLARATIONS */
inline isGateway(id){                                /* Check if id is a gateway */
    (id < GATEWAY_COUNT);
}

inline receive(id){                                  /* Update counters if received a msg */
    atomic{
        do
        :: (k < N) ->
            r = msgs[k].to[id];                      /* Msg recvd this msg round? */

            if
            :: nfull(nodes[id].W[k]) ->              /* If queue is not full yet */
                x=0
            :: full(nodes[id].W[k]) ->               /* If queue is full */
                nodes[id].W[k]?x;                    /* Get oldest msg round info */
            fi;
            nodes[id].R[k] = r-x+nodes[id].R[k];     /* Update counter */

            nodes[id].W[k]!r;                        /* Append most recent info to queue */

            if
            :: r -> if                               /* Update perceived distance if recvd */
                :: isGateway(k)-> nodes[id].D[k] = 0    /* dist-to-G of G is always 0 */
                :: else -> nodes[id].D[k]=nodes[k].D[k]
                fi
            :: else -> skip
            fi;

            k++
        :: (k==N) -> k=0; r=0; break
        od
    }
}

inline getMinimum(id){                               /* Compute minimum distance */
    atomic{
        minval = MAX_DIST;
        do
        :: (k < N) ->
            if
            :: (nodes[id].R[k] > 0) && (k != id) && (nodes[id].D[k] < MAX_DIST) ->
                l = len(nodes[id].W[k]);
                try = (ACCURACY * l / nodes[id].R[k]) + nodes[id].D[k];
                try = ( ((ACCURACY*l)%nodes[id].R[k])>=(nodes[id].R[k]/2) ->(try+1):try );
                if
                :: (try <= minval) -> minval = try; nodes[id].parent = k
                :: else -> skip
                fi
            :: else -> skip
            fi;
            k++
        :: (k == N) -> k=0; l=0; try=0; nodes[id].D[id] = minval; break
        od;
    }
}

inline setC(x, y, p, q){                             /* Sets entries in matrix C */
    C[x].to[y].u = p;
    C[x].to[y].v = q;
    C[y].to[x].u = p;                                /* Due to symmetry */
    C[y].to[x].v = q;                                /* Due to symmetry */
}

inline canLoose(a,b){                                /* Check if msgs still may be lost */
    (C[a].to[b].u - H[a].to[b].u) < (C[a].to[b].v - H[a].to[b].v)
}

inline canSend(c,d){                                 /* Check if msgs still may be sent */
    H[c].to[d].u < C[c].to[d].u
}
```

Listing E.2: Finite Sliding Window MA - part 2/3

```

inline checkReset(e,f){                               /* Reset counters when needed */
    if                                                /* (modulo C[].to[].v) */
    :: (H[e].to[f].v == C[e].to[f].v) -> H[e].to[f].u = 0; H[e].to[f].v = 0;
    :: else -> skip
    fi
}

/* PROCESS DECLARATIONS */
active[N] proctype node(){                            /* Node process */
 endN:  do
    :: ctrl == _pid ->                                /* Wait for control */
        byte k, l;                                    /* Used in inline constructs */
        bool r,x;                                     /* Used in inline construct */
        short try,minval;                             /* Used in inline construct */
        getMinimum(_pid);                             /* Compute minimum dist-to-G */
        receive(_pid);                                /* 'receive' from all other nodes */
        ctrl++                                        /* Transfer control to next process */
    od
}

active proctype globalSend(){                         /* Simulates sending globally */
    byte i,j;

    d_step{                                           /* Fill symmetric connectivity matrix */
                                                      /* Using inline setC(from, to, u, v) */
        /*arbitrary topo, unique SPT [30/18]
        setC(0,1,1,7); setC(0,2,1,10); setC(0,3,9,10);
        setC(1,3,3,10);
        setC(2,3,4,5)
        */
        /*arbitrary topo fig. 4.1c, mult SPT
        setC(0,1,1,4); setC(0,2,1,5); setC(0,3,1,2);
        setC(1,3,1,2);
        setC(2,3,1,2)
        */
        /*arbitrary topo fig. 4.1e, disconnected gateway*/
        setC(0,1,1,10); setC(0,2,1,10); setC(0,3,1,10);
        setC(1,2,1,1); setC(1,3,1,1);
        setC(2,3,1,1)
    }

    do                                                /* For each msg round */
    :: (M<WIN_SIZE) &&
        (WIN_CNT<MAX_WIN) -> atomic{
        do                                            /* Fill msgs matrix */
        :: (i<N) -> do
            :: (j < N && C[i].to[j].u == 0) -> j++    /* No link between i and j */
            :: (j < N && C[i].to[j].u != 0) ->        /* There is a link between i and j */
                if
                :: canLoose(i,j) ->                   /* ND-choice: write 0 in msgs matrix */
                    msgs[i].to[j] = 0                 /* 0 means msg gets lost */
                :: canSend(i,j) ->                    /* ND-choice: write 1 in msgs matrix */
                    msgs[i].to[j] = 1;                /* 1 means msg will come across */
                    H[i].to[j].u++                    /* Counts nr of delivered msgs */
                fi;
                H[i].to[j].v++;                       /* Counts msg opportunities */
                checkReset(i,j);
                j++
            :: (j==N) -> j=0; break
            od;
            i++
        :: (i==N) -> i=0; break
        od;
        ctrl = GATEWAY_COUNT;}                        /* Transfer control to 1st node process */
        ctrl == _pid;                                 /* Wait for control */
        M++
    :: (M==WIN_SIZE) && (WIN_CNT<MAX_WIN) ->          /* A window passed by, reset */
        M=0; WIN_CNT++
    :: (WIN_CNT==MAX_WIN) -> break                    /* Max nr of windows reached, stop */
    od
}
```

Listing E.3: Finite Sliding Window MA - part 3/3

E.2 Finite Sliding Window Variant WMA

This section contains the PROMELA source of the finite sliding window variant of our original model, with a Weighted Moving Average (WMA) of the link quality: Older data is less important compared to more recent data (linear decreasing weights). The position of the 0 or 1 in the window (in the queue) is its weight.

```
/* SPT Protocol model for WSNs - W.M. Everse */
/* Experiments with finite sliding window, WMA */

/* DEFINE CONSTANTS */
#define N               4               /* Number of nodes */
#define WIN_SIZE        10              /* Window size in msg rounds */
#define MAX_WIN         20              /* Max number of windows */
#define MAX_DIST        10000           /* Represents 'infinite' distance */
#define ACCURACY        10              /* Multiplication factor */
#define GATEWAY_COUNT   1               /* Number of gateways */

/* TYPEDEFS & DECLARATIONS */
local byte WIN_CNT = 0;                 /* Counts the nr of passed windows */

typedef NodeData{                       /* Node data:*/
    byte parent = 0;                    /* - selected parent */
    byte ETX[N] = 0;                    /* - expected transmission count (per node) */
    chan W[N] = [WIN_SIZE] of {bool};   /* - most recent window with recv info (per node) */
    short D[N] = MAX_DIST;              /* - dist-to-G (per node) */
}

NodeData nodes[N];                      /* All node data */

local byte M;                           /* Global msg round number */

typedef Tuple {byte u; byte v}
typedef NodeDimTuple{ Tuple to[N] }
hidden NodeDimTuple C[N];               /* Connectivity Matrix, NxN */
hidden NodeDimTuple H[N];               /* History Matrix, NxN */

typedef NodeDimBool{ bool to[N]; }
hidden NodeDimBool msgs[N];             /* Message Exchange Matrix, NxN */

byte ctrl = N;                          /* Used for transferring control */

/*INLINE DECLARATIONS */
inline isGateway(id){                   /* Check if id is a gateway */
    (id < GATEWAY_COUNT);
}
```

Listing E.4: Finite Sliding Window WMA - part 1/4

```
inline receive(id){                       /* Update counters if received a msg */
    atomic{
        do
        :: (k < N) ->
            r = msgs[k].to[id];           /* Msg recvd this msg round? */

            if
            :: full(nodes[id].W[k]) ->    /* If queue is full */
                nodes[id].W[k]?_          /* Discard oldest msg round info */
            :: nfull(nodes[id].W[k]) ->
                skip
            fi;

            nodes[id].W[k]!r;             /* Append most recent info to queue*/

            computeETX(id, k);

            if                            /* Update perceived distance if recvd */
            :: r ->
                if
                :: isGateway(k)->
                    nodes[id].D[k] = 0    /* dist-to-G of G is always 0 */
                :: else ->
                    nodes[id].D[k]=nodes[k].D[k]
                fi
            :: else -> skip
            fi;

            k++
        :: (k==N) -> k=0; r=0; break
        od
    }
}

inline computeETX(id, k){                               /* Compute WMA of ETX */
    do
    :: (weight < WIN_SIZE) && nempty(nodes[id].W[k]) ->
        nodes[id].W[k]?x;                               /* Get oldest value from queue*/
        result = result + weight*x;                     /* Multiply by weight & add to result */
        nodes[id].W[k]!x;                               /* Put value back in queue */
        weight++                                        /* Next */
    :: (weight == WIN_SIZE) || (empty(nodes[id].W[k])) -> break
    od;

    if
    :: (result > 0) ->
        result = weight*(weight-1)/(2*result)           /* Compute WMA of ETX */
    :: else -> result = 0                               /* Nothing received, set to 0 */
    fi;

    nodes[id].ETX[k] = result;                          /* Save result */

    x=0; weight=0; result=0;                            /* Reset variables */
}

inline getMinimum(id){                                  /* Compute minimum distance */
    /* vars 'minval', 'try' and 'k' are declared in node process */
    atomic{
        minval = MAX_DIST;
        do
        :: (k < N) ->
            if
            :: (nodes[id].ETX[k] > 0) && (k != id) && (nodes[id].D[k] < MAX_DIST) ->
                try = nodes[id].ETX[k] + nodes[id].D[k];
                if
                :: (try <= minval) -> minval = try; nodes[id].parent = k
                :: else -> skip
                fi
            :: else -> skip
            fi;
            k++
        :: (k == N) -> k=0; try=0; nodes[id].D[id] = minval; break
        od;
    }
}
```

Listing E.5: Finite Sliding Window WMA - part 2/4

```
inline setC(x, y, p, q){                          /* Sets entries in matrix C */
    C[x].to[y].u = p;
    C[x].to[y].v = q;
    C[y].to[x].u = p;                             /* Due to symmetry */
    C[y].to[x].v = q;                             /* Due to symmetry */
}

inline canLoose(a,b){                             /* Check if msgs still may be lost */
    (C[a].to[b].u - H[a].to[b].u) < (C[a].to[b].v - H[a].to[b].v)
}

inline canSend(c,d){                              /* Check if msgs still may be sent */
    H[c].to[d].u < C[c].to[d].u
}

inline checkReset(e,f){                           /* Reset counters when needed */
    if                                            /* (modulo C[].to[].v) */
    :: (H[e].to[f].v == C[e].to[f].v) -> H[e].to[f].u = 0; H[e].to[f].v = 0;
    :: else -> skip
    fi
}

/* PROCESS DECLARATIONS */
active[N] proctype node(){                        /* Node process */
 endN: do
    :: ctrl == _pid ->                            /* Wait for control */
        bool x; byte weight; short result;        /* Used in inline constructs */
        byte k;                                   /* Used in inline constructs */
        bool r;                                   /* Used in inline construct */
        short try,minval;                         /* Used in inline construct */
        getMinimum(_pid);                         /* Compute minimum dist-to-G */
        receive(_pid);                            /* 'receive' from all other nodes */
        ctrl++                                    /* Transfer control to next process */
    od
}

active proctype globalSend(){                     /* Simulates sending globally */
    byte i,j;

    d_step{                                       /* Fill symmetric connectivity matrix */
                                                  /* Using inline setC(from, to, u, v) */
        /*arbitrary topo, unique SPT [30/18] */
        setC(0,1,1,7); setC(0,2,1,10); setC(0,3,9,10);
        setC(1,3,1,3);  /* ATTENTION: 1/3 i.s.o. 3/10 */
        setC(2,3,4,5)

        /*test topo, N=2
        setC(0,1,5,10)
        */
    }

    do                                            /* For each msg round */
    :: (M<WIN_SIZE) &&
        (WIN_CNT<MAX_WIN) -> atomic{
        do                                        /* Fill msgs matrix */
        :: (i<N) -> do
            :: (j < N && C[i].to[j].u == 0) -> j++  /* No link between i and j */
            :: (j < N && C[i].to[j].u != 0) ->      /* There is a link between i and j */
                if
                :: canLoose(i,j) ->               /* ND-choice: write 0 in msgs matrix */
                    msgs[i].to[j] = 0             /* 0 means msg gets lost */
                :: canSend(i,j) ->                /* ND-choice: write 1 in msgs matrix */
                    msgs[i].to[j] = 1;            /* 1 means msg will come across */
                    H[i].to[j].u++                /* Counts nr of delivered msgs */
                fi;
                H[i].to[j].v++;                   /* Counts msg opportunities */
                checkReset(i,j);
                j++
            :: (j==N) -> j=0; break
            od;
            i++
```

Listing E.6: Finite Sliding Window WMA - part 3/4

```
         :: (i==N) -> i=0; break
         od;
         ctrl = GATEWAY_COUNT;}                    /* Transfer control to 1st node process */
         ctrl == _pid;                             /* Wait for control */
         M++
    :: (M==WIN_SIZE) && (WIN_CNT<MAX_WIN) ->       /* A window passed by, reset*/
         M=0; WINDOW_CNT++
    :: (WIN_CNT==MAX_WIN) -> break                 /* Max nr of windows reached, stop */
    od
}
```

Listing E.7: Finite Sliding Window WMA - part 4/4

E.3 Finite Sliding Window Variant EWMA

This section contains the PROMELA source of the finite sliding window variant of our original model, with an exponentially weighted moving average of the link quality: older data is less important compared to more recent data (exponentially decreasing weights).

```
/* SPT Protocol model for WSNs - W.M. Everse */
/* Experiments with EWMA discounting */

/* No ETX/distance, the metric is now just a link quality indicator
i.e (the smoothened average), its inverse may be used as distance indicator */

/* DEFINE CONSTANTS */
#define N               4               /* Number of nodes */
#define MAX_M           100             /* Max number of msg rounds */
#define MAX_DIST        10000           /* Represents 'infinite' distance */
#define ACCURACY        100             /* Multiplication factor */
#define GATEWAY_COUNT   1               /* Number of gateways */
#define ALPHA           10              /* Smoothing factor (per ACCURACY) */

/* TYPEDEFS & DECLARATIONS */
typedef NodeData{                       /* Node data:*/
    byte parent = 0;                    /* - selected parent */
    short S[N] = 0;                     /* - quality measure, exp smoothened avg (per node) */
    short D[N] = MAX_DIST;              /* - dist-to-G (per node) */
}

NodeData nodes[N];                      /* All node data */

short M;                                /* Global msg round number */

typedef Tuple {byte u; byte v}
typedef NodeDimTuple{ Tuple to[N]; }
hidden NodeDimTuple C[N];               /* Connectivity Matrix, NxN */
hidden NodeDimTuple H[N];               /* History Matrix, NxN */

typedef NodeDimBool{ bool to[N]; }
hidden NodeDimBool msgs[N];             /* Message Exchange Matrix, NxN */

byte ctrl = N;                          /* Used for transferring control */

/*INLINE DECLARATIONS */
inline isGateway(id){                   /* Check if id is a gateway */
    (id < GATEWAY_COUNT);
}

inline receive(id){                     /* Update counters if received a msg */
    atomic{
        do
        :: (k < N) && (msgs[k].to[id]) ->
            nodes[id].S[k] = ALPHA + ((ACCURACY-ALPHA)*nodes[id].S[k])/ACCURACY;
            if
            :: isGateway(k) -> nodes[id].D[k] = 0    /* dist-to-G of G is always 0 */
            :: else -> nodes[id].D[k]=nodes[k].D[k]
            fi;
            k++
        :: (k < N) && !(msgs[k].to[id]) ->
            nodes[id].S[k] = ((ACCURACY-ALPHA)*nodes[id].S[k])/ACCURACY; k++
        :: (k==N) -> k=0; break
        od
    }
}
```

Listing E.8: Finite Sliding Window EWMA - part 1/3

```

inline getMinimum(id){                              /* Compute minimum distance */
    /* vars 'minval', 'try' and 'k' are declared in node process*/
    atomic{
        minval = MAX_DIST;
        do
        :: (k < N) ->
            if
            :: (nodes[id].S[k] > 0) && (k != id) && (nodes[id].D[k] < MAX_DIST) ->
                try = (ACCURACY / nodes[id].S[k]) + nodes[id].D[k];
                try = ( (ACCURACY%nodes[id].S[k])>=(nodes[id].S[k]/2)->(try+1):try );
                if
                :: (try <= minval) -> minval = try; nodes[id].parent = k
                :: else -> skip
                fi
            :: else -> skip
            fi;
            k++
        :: (k == N) -> k=0; try=0; nodes[id].D[id] = minval; break
        od;
    }
}

inline setC(x, y, p, q){                            /* Sets entries in matrix C */
    C[x].to[y].u = p;
    C[x].to[y].v = q;
    C[y].to[x].u = p;                               /* Due to symmetry */
    C[y].to[x].v = q;                               /* Due to symmetry */
}

inline canLoose(a,b){                               /* Check if msgs still may be lost */
    (C[a].to[b].u - H[a].to[b].u) < (C[a].to[b].v - H[a].to[b].v)
}

inline canSend(c,d){                                /* Check if msgs still may be sent */
    H[c].to[d].u < C[c].to[d].u
}

inline checkReset(e,f){                             /* Reset counters when needed */
    if                                              /* (modulo C[].to[].v) */
    :: (H[e].to[f].v == C[e].to[f].v) -> H[e].to[f].u = 0; H[e].to[f].v = 0;
    :: else -> skip
    fi
}

/* PROCESS DECLARATIONS */
active[N] proctype node(){                          /* Node process */
 endN:  do
    :: ctrl == _pid ->                              /* Wait for control */
        byte k;                                     /* Used in inline constructs */
        short try,minval;                           /* Used in inline construct */
        getMinimum(_pid);                           /* Compute minimum dist-to-G */
        receive(_pid);                              /* 'receive' from all other nodes */
        ctrl++                                      /* Transfer control to next process */
    od
}

active proctype globalSend(){                       /* Simulates sending globally */
    byte i,j;

    d_step{                                         /* Fill symmetric connectivity matrix */
                                                    /* Using inline setC(from, to, u, v) */
        /*arbitrary topo fig. 8.2d, unique SPT [30/18] */
        setC(0,1,1,7); setC(0,2,1,10); setC(0,3,9,10);
        setC(1,3,3,10);
        setC(2,3,4,5)
    }
```

Listing E.9: Finite Sliding Window EWMA - part 2/3

```

    do                                                  /* For each message round */
    :: (M < MAX_M) -> atomic{
        do                                              /* Fill msgs matrix */
        :: (i<N) -> do
            :: (j < N && C[i].to[j].u == 0) -> j++      /* No link between i and j */
            :: (j < N && C[i].to[j].u != 0) ->          /* There is a link between i and j */
                if
                :: canLoose(i,j) ->                     /* ND-choice: write 0 in msgs matrix */
                    msgs[i].to[j] = 0                   /* 0 means msg gets lost */
                :: canSend(i,j) ->                      /* ND-choice: write 1 in msgs matrix */
                    msgs[i].to[j] = 1;                  /* 1 means msg will come across */
                    H[i].to[j].u++                      /* Counts nr of delivered msgs  */
                fi;
                H[i].to[j].v++;                         /* Counts msg opportunities */
                checkReset(i,j);
                j++
            :: (j==N) -> j=0; break
            od;
            i++
        :: (i==N) -> i=0; break
        od;
        ctrl = GATEWAY_COUNT;}                          /* Transfer control to 1st node process */
        ctrl == _pid;                                   /* Wait for control */
        M++
    :: (M == MAX_M) -> break                            /* Max nr of msg rounds reached, stop */
    od
}
```

Listing E.10: Finite Sliding Window EWMA - part 3/3

E.4 Neighbourhood Management Variant

This section contains the PROMELA source of the protocol variant with (simple) neighbourhood management implemented.

```
/* SPT Protocol model for WSNs - W.M. Everse */
/* Experiments with simple neighbourhood management */

/* DEFINE CONSTANTS */
#define N               4                       /* Number of nodes */
#define MAX_M           100                     /* Max number of msg rounds */
#define MAX_DIST        10000                   /* Represents 'infinite' distance */
#define ACCURACY        10                      /* Multiplication factor */
#define GATEWAY_COUNT   1                       /* Number of gateways */
#define NEIGHBOURS      2                       /* Number of neighbours in neighbour table */

/* TYPEDEFS & DECLARATIONS */
typedef Neighbour{                              /* Neighbour data: */
    byte  ID=255;                               /* - node id */
    short R = 0;                                /* - number of recvd msgs from neighbour */
    short D = MAX_DIST;                         /* - dist-to-G of this neighbour */
    short F = 0;                                /* - frequency counter */
}
```

Listing E.11: Neighbourhood Management Variant - part 1/4

```

typedef NodeData{                                      /* Node data: */
    byte parent = 0;                                   /* - selected parent */
    short D = MAX_DIST;                                /* - dist-to-G of this node*/
    Neighbour NBTable[NEIGHBOURS];                     /* - neighbour table */
}

NodeData nodes[N];                                     /* All node data */

short M;                                               /* Global msg round number */

typedef Tuple {byte u; byte v}
typedef NodeDimTuple{ Tuple to[N]; }
hidden NodeDimTuple C[N];                              /* Connectivity Matrix, NxN */
hidden NodeDimTuple H[N];                              /* History Matrix, NxN */

typedef NodeDimBool{ bool to[N]; }
hidden NodeDimBool msgs[N];                            /* Message Exchange Matrix, NxN */

byte ctrl = N;                                         /* Used for transferring control */

/*INLINE DECLARATIONS */
inline isGateway(id){                                  /* Check if id is a gateway */
    (id < GATEWAY_COUNT);
}

inline receive(id){                                    /* Manage neighbour if received a msg */
    atomic{
        do
        :: (k < N) && (msgs[k].to[id]) ->
                manageNeighbour(id,k);
                k++
        :: (k < N) && !(msgs[k].to[id]) -> k++
        :: (k==N) -> k=0; break
        od
    }
}

inline manageNeighbour(id, nb){                    /* Manage neighbour nb */
    byte c;                                        /* Range over neighbour entries in NBTable */
    byte index;                                    /* NBTable index of entry to be updated */

    c=0; index=-1;
    do                                             /* Find an entry to update */
    :: (c<NEIGHBOURS) ->
        if                                         /* If nb in NBTable: set index to reinforce, */
                                                   /* or if index not set, to 1st entry with F==0 */
        :: (nodes[id].NBTable[c].ID == nb) || (index==-1 && nodes[id].NBTable[c].F == 0) ->
            index = c
        :: else -> skip
        fi;
        c++
    :: (c==NEIGHBOURS) -> c=0; break
    od;

    if
    :: (index < 0) ->                              /* If table full and no entry with F==0, */
        do                                         /* decrease all frequency counters */
        :: (c < NEIGHBOURS) -> nodes[id].NBTable[c].F--; c++
        :: (c == NEIGHBOURS) -> c=0; break
        od
    :: else  ->                                    /* Reinforce or replace */
        nodes[id].NBTable[index].R =
        ((nodes[id].NBTable[index].ID == nb) -> (nodes[id].NBTable[index].R+1) : 1);
        nodes[id].NBTable[index].F =
        ((nodes[id].NBTable[index].ID == nb) -> (nodes[id].NBTable[index].F+1) : 1);
        nodes[id].NBTable[index].ID = nb;          /* Update ID */
        if                                         /* Update dist-to-G */
        :: isGateway(nb) -> nodes[id].NBTable[index].D = 0    /* dist-to-G of G is always 0 */
        :: else -> nodes[id].NBTable[index].D = nodes[nb].D
        fi
    fi;
}
```

Listing E.12: Neighbourhood Management Variant - part 2/4

```

inline getMinimum(id){                          /* Compute minimum distance */
    byte k, IDk;
    short Rk, Dk, try, minval;
    atomic{
        minval = MAX_DIST;
        do                                      /* Range over all neighbours in NBTable */
        :: (k < NEIGHBOURS) ->
            IDk = nodes[id].NBTable[k].ID;      /* For readability */
            Rk = nodes[id].NBTable[k].R;
            Dk = nodes[id].NBTable[k].D;

            if
            :: (Rk > 0) && (Dk < MAX_DIST) ->
                try = (ACCURACY * M / Rk) + Dk; /* Compute distance via this neighbour */
                try = ( (M%Rk) >= (Rk/2) -> (try+1) : try );
                if                              /* If minimum found, set parent */
                :: (try <=minval) -> minval=try; nodes[id].parent = IDk;
                :: else -> skip
                fi
            :: else -> skip
            fi;
            k++
        :: (k == NEIGHBOURS) -> k=0; try=0; nodes[id].D=minval; break
        od;
    }
}

inline setC(x, y, p, q){                        /* Sets entries  in connectivity matrix C */
    C[x].to[y].u = p;
    C[x].to[y].v = q;
    C[y].to[x].u = p;                           /* Due to symmetry */
    C[y].to[x].v = q;                           /* Due to symmetry */
}

inline canLoose(a,b){                           /* Check if msgs still may be lost */
    (C[a].to[b].u - H[a].to[b].u) < (C[a].to[b].v - H[a].to[b].v)
}

inline canSend(c,d){                            /* Check if msgs still may be sent */
    H[c].to[d].u < C[c].to[d].u
}

inline checkReset(e,f){                         /* Reset counters when needed */
    if                                          /* (modulo C[].to[].v) */
    :: (H[e].to[f].v == C[e].to[f].v) -> H[e].to[f].u = 0; H[e].to[f].v = 0;
    :: else -> skip
    fi
}

/* PROCESS DECLARATIONS */
active[N] proctype node(){                      /* Node process */
 endN:  do
    :: ctrl == _pid ->                          /* Wait for control */
        getMinimum(_pid);                       /* Compute minimum dist-to-G */
        receive(_pid);                          /* 'receive' from all other nodes */
        ctrl++                                  /* Transfer control to next process */
    od
}

active proctype globalSend(){                   /* Simulates sending globally */
    byte i,j;

    d_step{                                     /* Fill symmetric connectivity matrix */
                                                /* Using inline setC(from, to, u, v) */
        /*arbitrary topo, unique SPT [30/18]
        setC(0,1,1,7); setC(0,2,1,10); setC(0,3,9,10);
        setC(1,3,3,10);
        setC(2,3,4,5)
        */
        /*test topo for neighbourhood management, complete 3 nodes 50%
        setC(0,1,1,2); setC(0,2,1,2); setC(1,2,1,2)
        */
        /*test topo for neighbourhood management, square 4 nodes 50%
        setC(0,1,1,2); setC(0,2,1,2); setC(1,3,1,2); setC(2,3,1,2)
        */
        /*test topo for neighbourhood management, complete 4 nodes 50%
        setC(0,1,1,2); setC(0,2,1,2); setC(0,3,1,2); setC(1,2,1,2); setC(1,3,1,2);
        setC(2,3,1,2)
        */
```

Listing E.13: Neighbourhood Management Variant - part 3/4

```
        /*test topo for neighbourhood management, star topo, center is node 1 */
        setC(1,0,1,2);
        setC(1,2,1,2);
        setC(1,3,1,2)
        /*setC(1,4,1,6);
        setC(1,5,1,6);
        setC(1,6,1,6);
        setC(1,7,1,6)*/
    }

    do                                              /* For each msg round */
    :: (M < MAX_M) -> atomic{
        do                                          /* Fill msgs matrix */
        :: (i<N) -> do
            :: (j < N && C[i].to[j].u == 0) -> j++  /* No link between i and j */
            :: (j < N && C[i].to[j].u != 0) ->      /* There is a link between i and j */
                if
                :: canLoose(i,j) ->                 /* ND-choice: write 0 in msgs matrix */
                    msgs[i].to[j] = 0               /* 0 means msg gets lost */
                :: canSend(i,j) ->                  /* ND-choice: write 1 in msgs matrix */
                    msgs[i].to[j] = 1;              /* 1 means msg will come across */
                    H[i].to[j].u++                  /* Counts nr of delivered msgs */
                fi;
                H[i].to[j].v++;                     /* Counts msg opportunities */
                checkReset(i,j);
                j++
            :: (j==N) -> j=0; break
            od;
            i++
        :: (i==N) -> i=0; break
        od;
        ctrl = GATEWAY_COUNT;}                      /* Transfer control to 1st node process */
        ctrl == _pid;                               /* Wait for control */
       M++
    :: (M == MAX_M) -> break                        /* Max number of msg rounds reached */
    od
}
```

Listing E.14: Neighbourhood Management Variant - part 4/4

List of Acronyms

ACM	Association for Computing Machinery 69
ADC	Analog-to-Digital Converter 8
ANSI	American National Standards Institute 69
AODV	Ad hoc On demand Distance Vector 27
BDD	Binary Decision Diagram 112
CS	Computer Science 1
CSL	Continuous Stochastic Logic 111
CTL	Computation Tree Logic 39
CTMC	Continuous-Time Markov Chain 111
DD	Direct Diffusion 28
DTMC	Discrete-Time Markov Chain 111
DVD	Digital Versatile Disc 17
EEMCS	Electrical Engineering, Mathematics and Computer Science 64
ETX	Expected Transmission Count 167
EWMA	Exponential Weighted Moving Average 132
FMT	Formal Methods and Tools 1
FTSP	Flooding Time Synchronization Protocol 23
GEAR	Geographic and Energy Aware Routing 28
GPS	Global Positioning System 28
GUI	Graphical User Interface 110
ICT	Information and Communication Technology 17
IP	Internet Protocol 11
IUT	Implementation Under Test 18
LEACH	Low Energy Adaptive Clustering Hierarchy 28

LLN	Law of Large Numbers	135
LMAC	Lightweight Medium Access Control	24
LTL	Linear Temporal Logic	129
MA	Moving Average	132
MANET	Mobile Ad-hoc Network	26
MDP	Markov Decision Process	111
MSC	Message Sequence Chart	164
MST	Minimum Spanning Tree	16
OSI	Open Systems Interconnection	11
OSPF	Open Shortest Path First	27
PAN	Protocol Analyzer	79
PCTL	Probabilistic Computation Tree Logic	111
POR	Partial Order Reduction	132
PRISM	Probabilistic Symbolic Model Checker	110
PROMELA	Process Meta Language	132
PS	Pervasive Systems	150
RIP	Routing Information Protocol	27
SPIN	Simple Promela Interpreter	69
SPT	Shortest Path Tree	110
TCP	Transmission Control Protocol	11
UPPAAL	Uppsala-Aalborg Model Checker	38
WMA	Weighted Moving Average	132
WRP	Wireless Routing Protocol	27
WSN	Wireless Sensor Network	126
XML	eXtensible Mark up Language	90

List of Figures

List of Tables

Listings

Bibliography

[1] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999. ISSN 0925-9856. doi: http://dx.doi.org/10.1023/A:1008739929481.

[2] Sara Baase and Allen Van Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0201612445. Third Edition.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN 026202649X, 9780262026499.

[4] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[5] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, pages 232–243, 1995. URL http://www.uppaal.com.

[6] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981. ISBN 0138221227.

[7] P J Boltryk, C J Harris, and N M White. Intelligent sensors – a generic software approach. *Journal of Physics: Conference Series*, 15:155–160, 2005. URL http://stacks.iop.org/1742-6596/15/155.

[8] H. Brinksma. Verification is experimentation! *Software Tools for Technology Transfer*, 3(2):107–217, 2001. ISSN 1433-2779.

[9] H. Brinksma and A. H. Mader. On verification modelling of embedded systems. Technical Report TR-CTIT-04-03, University of Twente, Enschede, January 2004.

[10] Daniel Câmara, Antonio Alfredo F Loureiro, and Fethi Filali. Methodology for formal verification of routing protocols for ad hoc wireless networks. In *GLOBECOM 2007, 50th IEEE Global Communications Conference, November 26-30, 2007, Washington, USA*, Nov 2007. doi: 10.1109/GLOCOM.2007.137.

[11] Qing Cao, Lei Fang, Tarek Abdelzaher, John Stankovic, and Sang Son. Efficiency centric communication model for wireless sensor networks. In *in Proc. IEEE INFOCOM*, pages 1–12, 2006.

[12] Yuen-Hui Chee, Mike Koplow, Michael Mark, Nathan Pletcher, Mike Seeman, Fred Burghardt, Dan Steingart, Jan Rabaey, Paul Wright, and Seth Sanders. Picocube: a 1 cm3 sensor node powered by harvested energy. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 114–119, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-115-6. doi: http://doi.acm.org/10.1145/1391469.1391499.

[13] Douglas S. J. De Couto. *High-throughput routing for multi-hop wireless networks.* PhD thesis, Massachusetts Institute of Technology, 2004. Supervisor-Robert T. Morris.

[14] L. Evers, M. J. J. Bijl, M. Marin-Perianu, R. S. Marin-Perianu, and P. J. M. Havinga. Wireless sensor networks and beyond: A case study on transport and logistics. Technical Report TR-CTIT-05-26, University of Twente, Enschede, June 2005.

[15] A. Fehnker, M. Fruth, and A. McIver. Graphical modelling for simulation and formal analysis of wireless network protocols. In *Proc. Workshop on Methods, Models and Tools for Fault-Tolerance (MeMoT'07) at the 7th International Conference on Integrated Formal Methods (IFM'07)*, July 2007.

[16] A. Fehnker, L. F. W. van Hoesel, and A. H. Mader. Modelling and verification of the lmac protocol for wireless sensor networks. Technical Report TR-CTIT-07-09, Centre for Telematics and Information Technology, Enschede, February 2007.

[17] Ansgar Fehnker and Peng Gao. Formal verification and simulation for performance analysis for probabilistic broadcast protocols. In *ADHOC-NOW*, pages 128–141, 2006.

[18] Ralph P. Grimaldi. *Discrete and Combinatorial Mathematics: An Applied Introduction (fifth edition).* Pearson Education, Inc. / Addison-

Wesley, Boston, MA, USA, 2004. ISBN 0-321-21103-0. International Edition.

[19] M. Hendriks, G. Behrmann, K. G. Larsen, P. Niebert, and F. W. Vaandrager. Adding symmetry reduction to UPPAAL. In K. G. Larsen and P. Niebert, editors, *Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, number 2791 in LNCS, pages 46–59. Springer–Verlag, 2004.

[20] Jason Lester Hill. *System architecture for wireless sensor networks.* PhD thesis, University of California, Berkeley, 2003. Adviser-David E. Culler.

[21] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.

[22] Gerard J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual.* Addison-Wesley Professional, September 2003. ISBN 0321228626.

[23] Gerard J. Holzmann. *Design and validation of computer protocols.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991. ISBN 0-13-539925-4.

[24] Weirong Jiang, Shuping Liu, Yun Zhu, and Zhiming Zhang. Optimizing routing metrics for large-scale multi-radio mesh networks. *Wireless Communications, Networking and Mobile Computing, 2007. WiCom 2007. International Conference on*, pages 1550–1553, September 2007. doi: 10.1109/WICOM.2007.390.

[25] J.P. Katoen. Concepts, algorithms and tools for model checking. Lecture notes, Institut für Mathematische Maschinen und Datenverarbeitung (Informatik), Friedrich-Alexander-Universität, Erlangen, Nürnberg, Germany, March 1999.

[26] Maleq Khan, Gopal Pandurangan, and V.S. Anil Kumar. Energy-efficient distributed constructions of minimum spanning tree for wireless ad-hoc networks. Technical report, Department of Computer Science, Purdue University, West Lafayette, Indiana, October 2006.

[27] Taehyun Kim, Jaeho Kim, Sangshin Lee, Ilyeup Ahn, Minan Song, and Kwangho Won. An automatic protocol verification framework for the development of wireless sensor networks. In *TridentCom '08: Proceedings of the 4th International Conference on Testbeds and research*

infrastructures for the development of networks & communities, pages 1–5, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN 978-963-9799-24-0.

[28] Lakshman Krishnamurthy, Robert Adler, Phil Buonadonna, Jasmeet Chhabra, Mick Flanigan, Nandakishore Kushalnagar, Lama Nachman, and Mark Yarvis. Design and deployment of industrial sensor networks: experiences from a semiconductor plant and the north sea. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 64–75, New York, NY, USA, 2005. ACM. ISBN 1-59593-054-X. doi: http://doi.acm.org/10.1145/1098918.1098926.

[29] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach (4th Edition)*. Addison Wesley, March 2007. ISBN 0321497708.

[30] B. Kusy and S. Abdelwahed. Ftsp protocol verification using spin. Technical report, ISIS, Vanderbilt University, Nashville, Tennessee, May 2006.

[31] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):128–142, 2004.

[32] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct 1997.

[33] Jure Leskovec, Purnamrita Sarkar, and Carlos Guestrin. Modeling link qualities in a sensor network. *Informatica (Slovenia)*, 29 (4):445–452, 2005. URL http://dblp.uni-trier.de/db/journals/informaticaSI/informaticaSI29.html#LeskovecSG05.

[34] Konrad Lorincz and Matt Welsh. Motetrack: A robust, decentralized approach to rf-based location tracking. In Thomas Strang and Claudia L. Popien, editors, *LoCA*, volume 3479 of *Lecture Notes in Computer Science*, pages 63–82. Springer, 2005.

[35] Dimitrios Lymberopoulos and Andreas Savvides. Xyz: a motion-enabled, power aware sensor node platform for distributed sensor network applications. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 63, Piscataway, NJ, USA, 2005. IEEE Press. ISBN 0-7803-9202-7.

[36] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM. ISBN 1-58113-589-0. doi: http://doi.acm.org/10.1145/570738.570751.

[37] The MathWorks™. *Getting started with* MATLAB® *7*. The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098, September 2007. URL http://www.mathworks.com.

[38] A.K. McIver and A. Fehnker. Formal techniques for the analysis of wireless networks. In *In Proc. 2nd Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA06*, 2006.

[39] Geoff V. Merrett, Alex S. Weddell, Nick R. Harris, Bashir M. Al-Hashimi, and Neil M. White. A structured hardware/software architecture for embedded sensor nodes. In *17th International Conference on Computer Communications and Networks*, 2008.

[40] Sanket Nesargi and Ravi Prakash. Manetconf: Configuration of hosts in a mobile ad hoc network. In *INFOCOM*, 2002.

[41] NIST/SEMATECH. e-handbook of statistical methods. http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc4.htm, January 2003. National Institute of Standards and Technology. Handbook Paragraph 6.4. Introduction to Time Series Analysis. Last visited: June 1, 2009.

[42] H.A. Oldenkamp. Probabilistic model checking - a comparison of tools. Master's thesis, University of Twente, May 2007.

[43] Heemin Park, Jeff Burke, and Mani B. Srivastava. Design and implementation of a wireless sensor network for intelligent light control. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 370–379, New York, NY, USA, 2007. ACM. ISBN 978159593638X. doi: http://dx.doi.org/10.1145/1236360.1236407. URL http://dx.doi.org/10.1145/1236360.1236407.

[44] Dave Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, August 2002.

[45] Joseph Polastre, Jonathan Hui, Philip Levis, Jerry Zhao, David Culler, Scott Shenker, and Ion Stoica. A unifying link abstraction for wireless sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 76–89, New

York, NY, USA, 2005. ACM. ISBN 1-59593-054-X. doi: http://doi.acm.org/10.1145/1098918.1098928.

[46] C. S. Raghavendra, Krishna M. Sivalingam, Ramesh Govindan, and Parmesh Ramanathan, editors. *Proceedings of the Second ACM International Conference on Wireless Sensor Networks and Applications, WSNA 2003, San Diego, CA, USA, September 19, 2003*, 2003. ACM. ISBN 1-58113-764-8.

[47] Matthias Ringwald and Kay Römer. Deployment of sensor networks: Problems and passive inspection. In *Proceedings of the 5th Workshop on Intelligent Solutions in Embedded Systems (WISES '07)*, pages 180–193, Madrid, Spain, jun 2007.

[48] Kay Römer. Tracking real-world phenomena with smart dust. In *1st European Workshop on Wireless Sensor Networks (EWSN)*, pages 28–43, Berlin, Germany, January 2004. Springer-Verlag. ISBN 3-540-20825-9.

[49] Kay Römer and Matthias Ringwald. The deployment of sensor networks. *National Center of Competence in Research for Mobile Information and Communication Systems (NCCR-MICS) Newsletter*, pages 2–4, feb 2008.

[50] Theo C. Ruys. *Towards Effective Model Checking.* PhD thesis, University of Twente, Enschede, The Netherlands, March 2001. Promotor: prof. dr. H. Brinksma, Assistent Promotor: dr. ir. R. Langerak.

[51] D.P.L. Simons and M.I.A. Stoelinga. Mechanical verification of the IEEE 1394a root contention protocol using Uppaal2k. *Springer International Journal of Software Tools for Technology Transfer*, pages 469–485, 2001.

[52] A.S. Tanenbaum. *Computer Networks.* Prentice Hall, Upper Saddle River, NJ, fourth edition, 2003.

[53] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. pages 332–344. IEEE Computer Society Press, 1986.

[54] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 381–396, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1.

[55] Oskar Wibling, Joachim Parrow, and Arnold Neville Pears. Automatized verification of ad hoc routing protocols. In *FORTE*, pages 343–358, 2004.

[56] Alec Woo, Terence Tong, and David Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 14–27, New York, NY, USA, 2003. ACM Press.

[57] J. Wu. *Reliable Routing Protocols for Dynamic Wireless Ad Hoc and Sensor Networks.* PhD thesis, University of Twente, Enschede, the Netherlands, December 2007.