VANDERBILT UNIVERSITY

INSTITUTE FOR SOFTWARE INTEGRATED SYSTEMS

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, 37235

FTSP Protocol Verification using SPIN

Branislav Kusy Sherif Abdelwahed

TECHNICAL REPORT

ISIS-06-704

FTSP Protocol Verification using SPIN

Branislav Kusy
kusy@isis.vanderbilt.edu

Sherif Abdelwahed
sherif@isis.vanderbilt.edu

Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN, 37203

Abstract

The FTSP protocol is used for synchronizing clocks across a set of sensor nodes which are connected to one another over a single- or multi-hop wireless communication channels. In this paper, we investigate the application of model checking technique to verify correctness properties for the FTSP wireless communication protocol. In particular, we consider the FTSP's error-resilient election algorithm and verify the mutual exclusion of the underlying control variables as well as the safety from buffer overflow. We discuss the problems encountered and comment on the overall experience with the application of model checking for FTSP verification.

1 Introduction

Networks built from small, relatively inexpensive nodes with embedded wireless communication, processing capabilities and different sensors and actuators are getting more and more attention in the research community. Complex networks built from thousands of such devices are expected to fill many aspects of our lives. The most significant difference between traditional distributed platforms and wireless sensor networks is the severe resource constraints of the sensor nodes. Also the topology of sensor network can be dynamically changing and the failures of single nodes are quite common. All this unique characteristics of sensor networks domain influence the design decisions and force the rejection of many traditional existing algorithms.

One of the basic middleware services of sensor networks is network-wide *time synchronization* (timesync). In applications like environmental monitoring [14], mobile target tracking [9, 17] or acoustic shooter localization [16], multiple sensor nodes acquire data from the environment and send the data to a base station node – a single node dedicated for bridging sensor network with a pc-class device. Data fusion performed at the base station depends heavily on the correct timing information related to the data. For example, an acoustic source localization algorithm may collect times of the arrivals of acoustic signal to the sensor nodes, compute the time differences of these arrivals between different nodes and search for the location of the source of the acoustic signal. A precise global timesync service available at each of the sensor nodes is critical for this application.

Timesync algorithms providing a mechanism to synchronize the local clocks of the nodes in the network have been extensively studied in literature. The leading timesync algorithms are the Reference Broadcast Synchronization (RBS) algorithm [5], the Timing-sync Protocol for Sensor Networks (TPSN) [8], and the Flooding Time-Synchronization protocol (FTSP) [15].

In the RBS protocol, receivers of a broadcasted reference message synchronize to each other by exchanging the time of arrival of the message. The TPSN algorithm performs node-to-node synchronization along edges of a spanning tree formed in the sensor network. In the FTSP protocol, a single node is elected to become the root of the network. The root then synchronizes all other nodes to its local clock.

Model checking [3] is a verification technique aimed at determining whether a system specification possesses a property expressed as a temporal logic formula. Model checking has enjoyed wide success in verifying, or finding design errors in real-life systems. An interesting account of a number of these success stories can be found in [4].

In this paper, we model the FTSP's error-resilient election algorithm and verify the safety and correctness properties of the protocol using model checking techniques. The verification process requires us to to introduce abstractions of the crucial hardware and software components that determine the execution of FTSP. The line between these abstractions being realistic, to simulate the execution well, and high-level enough to generate traceable state space during the verification process, is typically very delicate and the choice of modeling language for the abstractions is important. We picked Process Meta Language (PROMELA) [11] to build our abstractions because its capabilities were demonstrated in several successfully projects. Correctness specifications are then defined using linear time logic and the system is verified using the spin [11] model checking tool. Although we were able to create models with traceable state-space only for fairly simple sensor networks, we encountered and had to solve several interesting problems in the verification process that relate to the state-space explosion problem.

The most common approaches to verification of new protocols for sensor networks is simulation and live testing [12, 15, 2]. A TinyOS [1] probabilistic discrete event simulator TOSSIM [13] is commonly used in simulation and there exist different testbeds for live testing [7, 18]. However, formal verification is superior in discovering errors or design flaws in the protocol implementation as both simulation and live testing can not provide any guarantees about non-existence of such errors. Very little work has been done to formally verify operation of protocols in sensor networks.

This paper is organized as follows. Section 2 presents the flooding time synchronization protocol and describes the most important abstractions we used to model important components of the protocol. We describe the linear time logic (LTL) for linear time model checking in Section 3. Consequently, we define the correctness properties in Section 4. This section also provides verification results for the defined correctness criteria as well as execution statistics of the SPIN tool. Section 5 summarizes the problems we encountered, our solutions to these problems and our experience with the formal verification of FTSP. Finally, we offer our conclusions and future work directions in section 6.

2 FTSP - Flooding Time Synchronization Protocol

In the context of sensor networks, timesync commonly refers to the problem of synchronizing clocks across a set of sensor nodes which are connected to one another over a single- or multi-hop wireless communications channel. Out of many timesync algorithms described in the literature, we choose FTSP [15] for our verification abstraction. We describe FTSP in more details in this section, concentrating on the election part of the protocol that we later model in PROMELA. There are two types of events that FTSP reacts to: timer interrupts and radio message transmissions. Therefore, simulation and verification of FTSP require us to model the radio channel and the clock that communicate with possibly many sensor nodes running FTSP. We look at the properties of typical radio channels and clock crystals used in sensor networks and conclude this section by discussing the properties of PROMELA models of the radio channel, clock and sensor nodes that reasonably well approximate the real-world deployment scenarios.

2.1 The FTSP protocol

The basic problem, solved by FTSP, is to time-synchronize a node which is not yet aware of the global time, with the rest of the network. It can be assumed that there exists a node that is already synchronized and located within a radio range of the unsynchronized node. FTSP assumes that the transmission and the reception of a radio message happens at the same time (radio communication propagates with the speed of light). The following mechanism is used to propagate timing information in FTSP: the already-synchronized sender includes global time of the message transmission in the message and the unsynchronized receiver records the corresponding local time at the reception of that message. This way, the receiver calculates the offset of the global and local times from a single radio message, or in other words it becomes synchronized. Consequently, the global time information can further propagate to unsynchronized nodes located in the neighborhood of the receiver.

This basic scheme can be generalized to propagate the global time from a single node, the root of the network, to all nodes in the network. The problem that remains to be solved by FTSP, is to enforce that only a single node can become the root of the network. FTSP's election algorithm that decides based on unique node IDs, however, becomes more complex because it needs to handle link and node failures in the unreliable world of sensor networks.

The following details of the FTSP implementation are relevant to our later verification abstraction. FTSP does not assume any external global time source, the nodes synchronize their clocks to the clock of a selected node, called the *root*. A timesync hierarchy rooted at this node is created where the root is at level 0; nodes in the broadcast range of the root are at level 1 and so on. To maintain such a hierarchy, encountering hardware failures, dynamic position changes and partitioning of the network, all nodes periodically broadcast timesync messages. The clocks of the first level nodes are synchronized by receiving a broadcast message from the root. Similarly, the second level nodes get the global time estimates from the nodes at level 1 and consequently the global time information spreads to all nodes in the network. The broadcasted timesync message contains two data fields: `rootID` and `seqNum`. `rootID` field contains the ID of the root to which the transmitting node is synchronized. The `seqNum` field contains the newest sequence number that was heard from the root so far. The sequence number is increased by one by the root when it transmits the new timesync message and no other node can increase its value. Both the current `rootID` and the highest sequence number are stored as local variables `myRootID` and `mySeqNum` at each node.

There are basically two events the nodes react to:

1. timesync radio message reception events: a node reacts by updating its current root and current highest sequence number based on the information contained in the message, and
2. timer interrupts that are signaled periodically: a node reacts by broadcasting a timesync message.

However, the corresponding event handlers as published in [15] and shown in Figures 1, 2 are more complex because they need to resolve the following problems:

```
 1 event Radio.receive(TimeSyncMsg *msg)
 2 {
 3   if ( msg->rootID < myRootID )
 4       myRootID = msg->rootID;
 5   else if ( msg->rootID > myRootID || msg->seqNum <= mySeqNum )
 6       return;
 7
 8   mySeqNum = msg->seqNum;
 9   if ( myRootID < myID )
10       heartBeats = 0;
11
12   if ( numEntries >= NUMENTRIES_LIMIT  && getError(msg) > TIME_ERROR_LIMIT )
13       clearRegressionTable();
14   else
15       addEntryAndEstimateDrift(msg);
16 }
```

Figure 1: The handling of new timesync messages.

The root election problem: FTSP solves this problem by electing the node with the lowest ID as the root of the network. A simple election process is utilized (lines 3-6 in Figure 1): each node ignores timesync messages with `rootID` higher than `myRootID` variable. Otherwise, the message is further processed and if `rootID` is smaller than `myRootID`, the `myRootID` variable is updated. Consequently, a node is the root if and only if its `myRootID` variable equals to its ID.

Root node failure: FTSP needs to detect that the root node has failed and ensure that a new node becomes the root. The sequence numbers and periodic timesync message transmissions were introduced for this: if the root fails the `mySeqNum` variable at each node in the network will stop increasing, since the sequence number can be increased only by the root (lines 15-16 in Figure 2). Each node keeps track of the time when it received a timesync message with the higher sequence number by counting the timer interrupts in the `heartbeats` variable: `heartbeats` variable is increased by one in each timer interrupt and is zeroed when the `mySeqNum` variable is updated (line 3 in Figure 2, and lines 10-11 in Figure 1). If `heartbeats` variable at some node reaches the limit, the node declares itself to be the root of the hierarchy (lines 5-7 in Figure 2).

Redundancy of information: a node can get timesync messages from the lower levels which are located further from the root and have worse approximation of the global time. The `seqNum` field of the timesync message is introduced to deal with this problem. Each node accepts only if the `seqNum` of the message is higher than the highest sequence number received so far which is stored in `mySeqNum` (line 9 in Figure 1).

The only portions of FTSP that are not modeled in our verification abstraction are the radio message timestamping, the global time calculation and clock drift estimation corresponding to lines 13-17 in Figure 1.

```
event Timer.fired()
{
  ++heartBeats;

  if ( myRootID != myID  && heartBeats >= ROOT_TIMEOUT )
      myRootID = myID;

  if ( numEntries >= NUMENTRIES_LIMIT || myRootID == myID ){
      msg.rootID = myRootID;
      msg.seqNum = mySeqNum;
      Radio.send(msg);

      if ( myRootID == myID )
          ++mySeqNum;
  }
}
```

Figure 2: Periodic sending of timesync messages.

2.2 Verification Abstraction

We choose PROMELA to build our verification models. As mentioned in the previous text, we model only portion of FTSP, namely the one that deals with the maintenance of a single root and replacing the failed root by a different node. However, simulation and verification of this FTSP model require modelling of the radio channel and the clock as well. Properties of both the radio channel and clock are guided by the properties of the FTSP implementation hardware and software platform: UC Berkeley Mica2 [10] and TinyOS [1], respectively. We model radio channel, clock and sensor nodes as separate processes in PROMELA:

2.2.1 The Radio Channel Model

The radio channel process is capable of transmitting radio messages via broadcasting using the following mechanism: it accepts requests to transmit radio messages from a sensor node process on `radioSend` message channel. After receiving the transmission request, the radio channel process sends the message to all nodes in the network on `radioRcv` message channel. The radio channel is location aware, it sends the message only to the nodes that are within the radio range of the transmitter. This is accomplished by including the location of the transmitter as a parameter of the radio transmission request and having the locations of all nodes globally known. The radio range is uniform for all nodes and can be controlled by `RADIO_RANGE` constant. The channel also models non-deterministic failures of the radio links: a radio message will be dropped by the radio channel with some probability. The transmission of a single message is atomic, all requests for sending consequent radio messages from other nodes are blocked until the current transmission completes.

2.2.2 The Clock Model

The clock process is modeling the local free running clock at each of the nodes. The clock is periodically interrupting the processor which triggers the timesync message sending at each

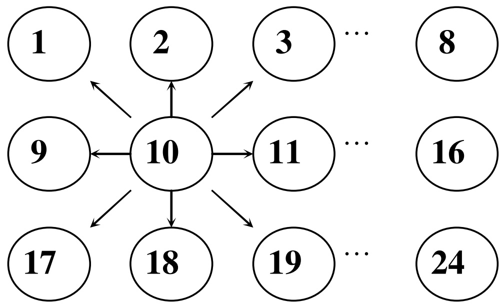

Figure 3: Example of the 2-dimensional grid for GRID_WIDTH=8, and NUM_NODES=24.

node. The clock interrupts have the same frequency at each node - it can not happen that node A gets two clock interrupts while node B did not get any interrupt.

2.2.3 The Sensor Node Model

Each node of the system is modeled as a separate process. A unique identifier `ID` is defined at the initialization phase of the process. Three local variables `myRootID`, `mySeqNum`, and `heartbeats` are defined to store ID of the root to which a node is currently synchronizing, the highest sequence number received from the current root and number of clock interrupts since the node received a new message from the root, respectively. Further, `myLoc` variable represents 2-dimensional location of the node that aids the location aware message transmission. We define a 2-dimensional equally spaced grid of the nodes with two constants: `GRID_WIDTH` defining the number of nodes along x-axis and the `NUM_NODES` defining the number of all nodes in the grid. See Figure 3 for an example. The sensor node process then basically reacts to message receiving events and the timer events assuming no particular order of these events.

3 Linear Time Model Checking

Formal logic can be used to describe precisely many practical forms of specification, including, safety, liveness, and properties. The most widely studied are temporal logics which were introduced for specification and verification purposes in computer science. Temporal logics support the formulation of properties of system behavior over time. Different interpretations of temporal logics exist depending on how one considers the system to change with time. Linear temporal logic allows the statement of properties of execution sequences of a system. Branching temporal logic allow the user to write formulas which include some sensitivity to the choices available to a system during its executions. It allows the statement of properties of about possible execution sequences that start in a state.

In this paper we consider only specification expressed using the LTL linear time logic [3, 6]. This logic is formally based on finite state models where at each moment there may be several different possible futures. Technically speaking linear time boils down to the fact that from the initial state the system can have a set of different possible execution sequences. The underlying notion of the semantics of a linear time temporal logic is thus a set of infinite sequences. Each sequence is intended to represent a single possible computation. The set of sequences itself thus represents all possible computations.

The syntax of linear time logic (LTL) is defined as follows. The most elementary expressions in LTL are atomic propositions. The set of atomic propositions is denoted by AP with typical elements p, q and r. We define the syntax of LTL in Backus-Naur Form. For $p \in AP$ the set of LTL formulas is defined by

$$\phi ::= \top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{X}\phi \mid \phi\mathsf{U}\phi$$

where X and U are linear temporal operators that express a property over a single path. Other frequently used derived operators include G (always) and F future. In literature the symbols $\Box$ (`[]`) and $\Diamond$ (`<>`) are sometimes used to denote G and F, respectively.

In general, LTL formulas are interpreted over a finite state machine model $\mathcal{M} = (S, \delta, L)$ where S is a set of states endowed with a transition relation $\delta \subseteq S \times S$ and $L : S \to \mathcal{P}(\texttt{Atoms})$ is a labeling function. The semantics of LTL is defined by a satisfaction relation, denoted by $\models$, between a model $\mathcal{M}$, one of its states s, and a formula ϕ. We write $\mathcal{M}, s \models \phi$ to indicate that ϕ is true in the state s in the model $\mathcal{M}$. The model $\mathcal{M}$ may be omitted since we always speak about the same model. The satisfaction relation is defined as follows. Consider the path $\pi = s_1\ s_2\ \ldots$ we write π^i for the suffix starting at starting at s_i, that is, $\pi^i = s_i\ s_{i+1}\ \ldots$. Let $\pi = s_1 s_2 \ldots$ be a path. Then:

$\pi \models \top$
$\pi \models p$ iff $p \in L(s_1)$
$\pi \models \neg\phi$ iff $\pi \not\models \phi$
$\pi \models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$
$\pi \models \mathsf{X}\phi$ iff $\pi^2 \models \phi$
$\pi \models \phi_1\ \mathsf{U}\ \phi_2$ iff $\exists j$ such that $\pi^j \models \phi_2$ and $\forall i < j\ \pi^i \models \phi_1$

The interpretation of the operators G and F can be derived using the above definition based on the identities $\mathsf{F}\phi = \top\mathsf{U}\phi$ and $\mathsf{G}\phi = \neg\mathsf{F}\neg\phi$. A more detailed introduction to LTL model checking can be found in [6].

4 Verification of the Properties

SPIN allows us to verify different properties of our models via assertion checking, cycle detection and satisfaction of LTL formulas.

4.1 The Assertion Checking

We specified multiple simple safety properties in the code expressed with an assert statement. The SPIN verification tool then automatically proves that these safety property valid. We verified the mutual exclusivity of the `control` variable, and to verify the safety of the buffer dedicated to each of the node processes:

4.2 The LTL Formulas

We used LTL formulas to specify and verify the following correctness properties of the FTSP protocol:

4.2.1 Synchronization to the common root

One of the most important requirements of timesync algorithms is that eventually, all nodes synchronize to the same unique root, if the network is connected. All nodes in the network need to synchronize the same global time which is maintained by a single node, the root of the network.

```
TimeSyncCommonRoot:
  #define p (myRootID[0] !=255)
  #define q (myRootID[0] == myRootID[1] &&
              myRootID[0] == myRootID[2] &&
              ...
              myRootID[0] == myRootID[n]   )

Formula As Typed: [] ((p) -> (<> (q)))
```

Figure 4: All nodes synchronize to the same root.

The formula can be seen in Figure 4. It is written for a n-node system, with assigned unique ids from 0 to n forming a connected network. The timesync algorithm should eventually choose the node with the smallest id (i.e. id=0) to be the root. The LTL formula states that if node 0 becomes initialized, all other nodes in the network will eventually become synchronized to its global clock.

4.2.2 Each node gets eventually synchronized

It can not happen that a node would stay in uninitialized state. This pathological case would allow for the previous property to be true even though the system would not behave according to our needs.

```
EventSynced:
  #define p (myRootID[0] == 255)

Formula As Typed: [] ( <> ( !p))
```

Figure 5: All nodes eventually get synchronized.

4.2.3 Ideal channel and ideal network

Ideal channel is not losing messages and ideal network contains no nodes that would die. If the messages are not lost in the radio channel and the nodes do not die, all the nodes in the network always get new activity from the current root, therefore they will not change the root in the future, after the root was chosen.

4.3 The safety and LTL verification results

Before showing the results for the system described in this section, we would like to clarify that we implemented also a version of the protocol where the clock and the radio were not so tightly

```
EventSyncedForever:
  #define p   (myRootID[0] != 255)
  #define q   (myRootID[0] == myRootID[1] &&
               myRootID[0] == myRootID[2] &&
               ...
               myRootID[0] == myRootID[n])

Formula As Typed: [] ((p) -> (<> ([](q))))
```

Figure 6: In an ideal network, all nodes synchronize to the same root forever.

connected (i.e. strictly changing their steps). However, the resulting system had much bigger state space which prohibited the verification of LTL formulas.

4.3.1 Safety verification

System proved to be complicated, even though we used many optimization techniques. Therefore the systems for which we were able to verify LTL properties were somewhat small. The safety properties were verified with the following results:

```
NUM_NODES = 2, GRID_WIDTH = 2, NODEBUF_SIZE = 1, IDEAL CHANNEL
Depth=  682165 States=   9e+06 Transitions= 1.69413e+07

NUM_NODES = 4, GRID_WIDTH = 2, NODEBUF_SIZE = 1, IDEAL CHANNEL
Depth= 13498582 States= 3.8e+07 Transitions= 1.00776e+08

NUM_NODES = 2, GRID_WIDTH = 2, NODEBUF_SIZE = 2, IDEAL CHANNEL
Depth= 2633906 States= 1.5e+07 Transitions= 2.93885e+07
```

4.3.2 LTL verification

The verification was done for the following systems:

- `NUM_NODES = 2, GRID_WIDTH = 2,`
  `NODEBUF_SIZE = 1, IDEAL CHANNEL`

  TimeSyncCommonRoot:
  ```
  Depth= 1281491 States= 1e+07
  Transitions= 1.91818e+07
  ```
  EventSynced:
  ```
  Depth= 1281491 States= 1e+07
  Transitions= 1.91818e+07
  ```
  EventSyncedForever:
  ```
  Depth= 1339047 States= 1.7e+07
  Transitions= 3.23263e+07
  ```

- `NUM_NODES = 2, GRID_WIDTH = 2,`
  `NODE_BUF_SIZE = 2, IDEAL CHANNEL`

TimeSyncCommonRoot:

```
Depth= 4846851 States= 1.6e+07
Transitions= 3.09278e+07
```

EventSynced:

```
Depth= 4847493 States= 1.6e+07
Transitions= 3.09217e+07
```

EventSyncedForever:

```
Depth= 4816863 States= 1.7e+07
Transitions= 3.27155e+07
```

- `NUM_NODES = 4, GRID_WIDTH = 2,`
  `NODE_BUF_SIZE = 1, IDEAL CHANNEL`

TimeSyncCommonRoot:

```
Depth= 1792 States= 2.6e+07
Transitions= 6.11405e+07
```

EventSynced and EventSyncedForever:

In the first 2 minutes, bitstate search used more than 700MB memory, the virtual disk memory pages had to be used which caused long time delays, we eventually gave up after 30 minutes when almost no progress was visible (depth was reaching limit=$15 * 10^6$). We also tried the breadth first search to find some counter-examples but unsuccessfully. We tried decreasing the maximum depth search limit which saved some memory but then the search reached the maximum depth limit (about $1 * 10^7$) without completion. Next, we tried the exhaustive search with compression which used less memory than the bitstate search, however after 15 min it consumed the whole available memory (1GB) and the depth of the search became $1.4 * 10^7$ which was hopeless. We tried the hash-compact search with 1GB memory and $1.6 * 10^7$ limit for the maximum depth of the search. After 25 minutes, this method showed much slower progress than the previous methods and used most of its memory so we decided to cancel it.

5 Verification Experience

We describe the experience when working with PROMELA language and SPIN/XSPIN tool in this section.

We used cygwin version of spin (http://spinroot.com) together with Xspin407 graphical interface that was executed with TCL 8.4.4.0. The computer we used is Intel Pentium IV, 2.6GH, 512MB RAM, Windows XP Professional. We set the parameters for the verification in SPIN the following way:

- physical memory available = 1GB,,

- estimated state space = $500 * 10^3$ states,
- maximum search depth = $15 * 10^6$ steps,
- used partial order reduction,
- search mode: supertrace/bitstate, and
- new msgs were blocked on the full queue.

All verification results mentioned in the previous section took less than 30 minutes to verify. Our computer could usually do $3 * 10^6$ to $5 * 10^6$ states in 1 minute therefore the most of the simple systems (i.e. with 2 nodes) were done in less than 2 minutes. The larger examples took over 10 minutes, still a reasonable time. Much bigger problem was the memory consumption. In almost every more complicated example either the depth of the search went off limits (over $15 * 10^6$) or all 1GB of memory was taken even by search in a relatively small depth. Since we have specified more available memory than was the limit of the physical memory, the virtual memory had to be used which made the search too slow.

To decrease search state space, we had to change our former model of the system where the clock model and the radio model interleaved in arbitrary fashion. Event though we managed to implement safe system that seemed to work under simulation, the state space of the system was so large, that even 2-nodes system verification was at the edge of computability on our computer (taking 1 GB of memory and 15 minutes to verify). Therefore we decided to constrain the system's choices and introduced strict changing of radio and clock processes. The size of the exploration space when compared to the arbitrary interleaving decreased significantly, i.e. the depth of the search for the new algorithm reached about 10^5 for a 2-node system as compared to the old algorithm's depth of more than 10^7.

We used atomic sections in the code as much as possible, as we noticed that the depth of the search and the size of the explored state space can be dramatically reduced this way. Another approach was to decrease the sizes of the variables as much as possible: our current code mostly uses byte variables.

We encountered several problems in this verification case study. The first problem we discovered was that we had a loop in the clock process that allowed the clock to take the control infinitely often - starving all the other components. The verification then just took a long time without finding any problem (the truncated search). However, the problem was easily discovered when verifying LTL properties.

The second problem, quite similar to the previous one was with the self-loop which existed in one of our components. We implemented `else->skip;` commands in a do-loop which makes sense in the simulation, where the non-determinism works. However, the non-determinism is non-existent in the verification process (this caused problems especially after working with SPIN, where fairness can be explicitly defined).

We could not try out the non-deterministic message dropping in the radio channel as we originally planed to because of the state space explosion problems.

Further, we encountered problems with verifying LTL properties even for trivial systems. Even though a system could be relatively easily verified for the safety, the merging of the two state spaces in LTL verification (one for the system and one for the verification) introduces more complicated system in general. In particular, the safety verification of a 4-node system was at the edge of computability on our pc, and adding more complexity to it with LTL verification made it incomputable.

The main issue of the SPIN toolset is obviously related to scalability; the verification engine could not scale well with the more complicated systems (more components). Apparently, trying to incorporate the clock into the system (even though with a byte-bounded time) significantly increased the complexity of the system model. Consequently, we could verify the LTL properties for only small set of components. On the other hand, the simulation tool in XSpin proved to be very helpful in understanding the flow of the protocol algorithm, even for more complicated examples. We were running the simulation for about 10 motes and observed that the algorithm worked correctly for the several critical traces.

6 Conclusions and Future Work

This paper is about the application of PROMELA language and the SPIN tool to verify a practical implementation of the FTSP protocol. In this paper, we described the setting of the model and correctness properties. Even though we did not manage to verify the LTL properties for more complicated examples, we proved some useful properties of the FTSP protocol which was previously implemented in TinyOS operating system. We came across interesting problems regarding the reducing the complexity of the system and were forced to solve these problems with non-trivial effort.

The possible future work includes looking into clock extensions to the SPIN verifier. This may allow for reducing the complexity of the system and may help to verify the protocol for more complicated systems.

We would also like to look into the behavior of the system under different scenarios. Particularly, when the current root of the network fails. The system that models the loosy radio channel and the nodes which die non-deterministically would be closer to the real world scenarios.

References

[1] Tinyos. `http://www.tinyos.net/media.html`.

[2] G. W. Allen, G. Tewari, A. Patel, M. Welsh, and R. Nagpal. Firefly-inspired sensor network synchronicity with realistic radio effects. In *in Proc. of ACM 3rd international conference on Embedded networked sensor systems*, pages 142–153, November 2005.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[4] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4), December 1996.

[5] J. E. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *The Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–163, December 2002.

[6] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier, 1990.

[7] E. Ertin, A. Arora, R. Ramnath, and M. Nesterenko. Kansei: A testbed for sensing at scale. In *in Proc. of 5th Intl Conf. Information Processing in Sensor Networks (IPSN SPOTS 2006)*, April 2006.

[8] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync protocol for sensor networks. In *The First ACM Conference on Embedded Networked Sensor System (SenSys)*, pages 138–149, November 2003.

[9] T. He, Ch. Huang, B. M. Blum, J. Stankovic, and T. Abdelzaher. Range-free localization schemes for large scale sensor networks. In *in Proc. of ACM 9th annual international conference on Mobile computing and networking*, pages 81–95, September 2003.

[10] J. Hill and D. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, November 2002.

[11] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[12] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *in Proc. of ACM 2nd international conference on Embedded networked sensor systems (SenSys)*, pages 81–94, November 2004.

[13] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *In Proc. of the ACM 1st international conference on Embedded networked sensor systems*, pages 126–137, 2003.

[14] A. Mainwaring, D. Culler, J. Polastre, et al. Wireless sensor networks for habitat monitoring. In *in Proc. of ACM 1st international workshop on Wireless sensor networks and applications*, pages 88–97, September 2002.

[15] M. Maróti, B. Kusý, Gy. Simon, and Á. Lédeczi. The flooding time synchronization protocol. In *in Proc. of ACM 2nd Conference on Embedded Networked Sensor Systems (SenSys)*, pages 39–49, November 2004.

[16] Gy. Simon, M. Maróti, Á. Lédeczi, et al. Sensor network-based countersniper system. In *in Proc. of ACM 2nd Conference on Embedded Networked Sensor Systems (SenSys)*, pages 1–12, November 2003.

[17] H. Wang, J. Elson, L. Girod, D. Estrin, and K. Yao. Target classification and localization in a habitat monitoring application. In *In Proc. of the IEEE ICASSP*, April 2003.

[18] G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: A wireless sensor network testbed. In *Proc. 4th Intl Conf. Information Processing in Sensor Networks (IPSN 05)*, pages 483–488, 2005.