Using Integer Clocks to Verify the Timing-Sync Sensor Network Protocol

Xiaowan Huang, Anu Singh, Scott A. Smolka
Department of Computer Science, Stony Brook University

Abstract

We use the UPPAAL model checker for Timed Automata to verify the Timing-Sync time-synchronization protocol for sensor networks (TPSN). The TPSN protocol seeks to provide network-wide synchronization of the distributed clocks in a sensor network. Clock-synchronization algorithms for sensor networks such as TPSN must be able to perform arithmetic on clock values to calculate clock drift and network propagation delays. They must be able to read the value of a local clock and assign it to another local clock. Such operations are not directly supported by the theory of Timed Automata. To overcome this formal-modeling obstacle, we augment the UPPAAL specification language with the *integer clock* derived type. Integer clocks, which are essentially integer variables that are periodically incremented by a global pulse generator, greatly facilitate the encoding of the operations required to synchronize clocks as in the TPSN protocol. With this integer-clock-based model of TPSN in hand, we use UPPAAL to verify that the protocol achieves network-wide time synchronization and is devoid of deadlock. We also use the UPPAAL Tracer tool to illustrate how integer clocks can be used to capture clock drift and resynchronization during protocol execution.

1 Introduction

A *sensor network* is a collection of spatially distributed, autonomous sensing devices, used to perform a cooperative task. Sensor networks are widely deployed for unmanaged and decentralized operations such as military operations, surveillance, and health and environmental monitoring. Sensor networks differ from traditional distributed networks in their stringent resource constraints, network dynamics, node failure, and intermittent communication links. Consequently, protocols executing on sensor networks must be robust and reliable.

Many high-level sensor applications depend on an underlying *time-synchronization* mechanism, which provides network-wide time synchronization. Thus, protocols for time synchronization are critical for sensor applications, and ensuring their correctness is of the utmost importance.

In this paper, we use the UPPAAL model checker [7] for Timed Automata to verify the correctness of the *Timing-sync Protocol for Sensor Networks* (TPSN) [3]. TPSN is a time-synchronization protocol that performs pair-wise node synchronization along the edges of a network spanning tree. The local clock of a designated node, typically the *root* of the spanning tree, serves as the reference clock. Nodes may leave or join the network. TPSN enjoys several advantages over other sensor network time-synchronization protocols, including higher precision (less synchronization error) and the fact that its tree-based scheme makes it well-suited for multi-hop networks.

Clock-synchronization algorithms for sensor networks such as TPSN must be able to perform arithmetic on clock values in order to calculate clock drift and network propagation delays. They must be able to read the value of a local clock and assign it to another local clock. Such operations are not directly supported by the theory of Timed Automata. To overcome this formal-modeling obstacle, we augment UPPAAL's input language with the *integer clock* derived type. Integer clocks, which are essentially integer variables that are periodically incremented by a global pulse generator, greatly facilitate the encoding of the operations required to synchronize clocks as in the TPSN protocol.

With this integer-clock-based model of TPSN in hand, we use UPPAAL to verify that the protocol achieves network-wide time synchronization and is devoid of deadlock. Our model additionally takes

into account nodes entering and leaving the network. We also use the UPPAAL Tracer tool to illustrate how integer clocks can be used to capture clock drift and resynchronization during protocol execution.

This paper is organized as follows. Section 2 offers a description of the TPSN [3] protocol. Section 3 introduces the concept of integer clocks. Section 4 presents our UPPAAL model of TPSN; the corresponding verification and simulation results are given in Section 5. Section 6 considers related work, while Section 7 contains our concluding remarks. An online version of the paper, available from `www.cs.sunysb.edu/~sas/NFM10-full.pdf`, contains an Appendix in which a complete UPPAAL source-code listing of our TPSN model can be found.

2 The Timing-Sync Protocol for Sensor Networks

The Timing-sync Protocol for Sensor Networks (TPSN) aims to provide network-wide time synchronization in a sensor network [3]. The TPSN protocol first establishes a hierarchical structure in the network. It then achieves pair-wise synchronization along the edges of this structure such that all local clocks synchronize to a reference node. A more detailed discussion of the two phases of TPSN is now given.

- **Level-discovery phase:** Level discovery occurs when the network is first deployed. In this phase, a hierarchical structure (spanning tree) is established for the network in which every node is assigned a level. A distinguished node called the **root node** is assigned level 0 and initiates this phase by broadcasting a *level_discovery* packet containing the identity and level of the sender. The immediate neighbors of the root node receive this packet and assign themselves a level one greater than the level they have received; i.e., level 1. They then broadcast a new *level_discovery* packet containing their own level. This process is continued and eventually every node in the network is assigned a level. After been assigned a level, a node neglects any future *level_discovery* packets, thereby ensuring that no flooding congestion takes place during this phase. Ultimately, a spanning tree is created with the root node at level 0.

- **Synchronization phase:** The root node initiates the synchronization phase, during which pair-wise synchronizations involving a level-i node and a level-$(i-1)$ node are carried out using handshake (two-way) message exchanges. Eventually, all nodes in the network synchronize their clocks to that of the root node. Consider the synchronization of level-i node A and level-$(i-1)$ node B; i.e., A is a child of B in the spanning tree and A is attempting to synchronize its clock with that of B. The following sequence of messages comprise the handshake: A at local time $T1$ sends a *synchronization_pulse* packet containing its level number and the value of $T1$ to B. B receives this packet at local time $T2$, where $T2 = T1 + \Delta + d$, Δ is the clock drift between the two nodes, and d is the propagation delay. B, at local time $T3$, sends back an *acknowledgment* packet to A containing its level number and the values of $T1$, $T2$, and $T3$. Finally, A receives the *acknowledgment* packet at local time $T4$. Assuming that the clock drift and the propagation delay do not change in this small span of time, A can calculate the clock drift and propagation delay as:

  $$\Delta = \frac{(T2-T1)-(T4-T3)}{2} \quad ; \quad d = \frac{(T2-T1)+(T4-T3)}{2}$$

  Knowing the drift, A can correct its clock accordingly, so that it synchronizes to B.

  The root node initiates the synchronization phase by broadcasting a *time_sync* packet. Upon receiving this packet, nodes belonging to level 1 wait for a random amount of time before synchronizing with the root node. Randomization is used to avoid contention at the MAC layer of the communications protocol stack. Upon receiving an acknowledgment, these nodes adjust their clock to the

root node. Given that every level-2 node has at least one level-1 node in its neighbor set, nodes belonging to level-2 overhear this message exchange, upon which they wait for some random amount of time before initiating the message exchange with level-1 nodes. This use of randomization is to ensure that level-1 nodes have been synchronized before level-2 nodes begin the synchronization phase. A node sends back an acknowledgment to a *synchronization_pulse* only after it has synchronized itself, thereby ensuring that multiple levels of synchronization do not occur in the network. This process is carried out throughout the network, and consequently every node is eventually synchronized to the root node.

- **Special Cases:** The TPSN algorithm employs heuristics to handle special cases such as a sensor node joining an already established network, and nodes dying randomly. **Node joining:** This case concerns a sensor node that joins an already-established network (in particular, after the level-discovery phase is over) or that does not receive a *level_discovery* packet owing to MAC-layer collisions. In either case, it will not be assigned a level in the spanning tree. Every node, however, needs to be a part of the spanning tree so that it can be synchronized with the root. When a new node is deployed, it waits for some time to be assigned a level, failing which it timeouts and broadcasts a *level_request* message. Neighboring nodes reply to this request by sending their own level. On receiving the levels from its neighbors, the new node assigns itself a level one greater than the smallest level it has received, thereby joining the hierarchy. This is a *local* level-discovery phase. **Node dying:** If a sensor node dies randomly, it may lead to a situation where a level-i node does not possess a level-$(i-1)$ neighbor. In this case, the level-i node would not get back an acknowledgment to its *synchronization_pulse* packet, preventing it from synchronizing to the root node. As part of collision handling, a node retransmits its *synchronization_pulse* after some random amount of time. After a fixed number of retransmissions of the *synchronization_pulse*, a node assumes that it has lost all its neighbors on the upper level, and therefore broadcasts a *level_request* message to discover its level. Assuming the network is connected, the node will have at least one neighboring node and will receive a reply, after which The TPSN protocol considers four retransmissions to be a heuristic for deciding non-availability of an upper-level neighbor.

  If the node that dies is the root node, level-1 nodes will not receive an acknowledgment and hence will timeout. Instead of broadcasting a *level_request* packet, they run a leader-election algorithm to determine a new root node. The newly elected root node re-initiates the level-discovery phase.

3 Integer Clocks

UPPAAL is an integrated tool environment for the specification, simulation and verification of real-time systems modeled as networks of Timed Automata extended with data types. Primitive data types in UPPAAL are: clock, bounded integer, boolean and channel. There are also two compound types: array and structure, as in the C language. Timed Automata in UPPAAL are described as *processes*. A process consists of process parameters, local declarations, states and transitions. Process parameters turn into process-private constants when a process is instantiated. Local declarations describe the set of private variables to which a running process has access. States correspond to the vertices of a Timed Automaton in graphical form. Transitions are the edges connecting these vertices. A transition specifies a source and destination state, a guard condition, a synchronization channel, and updates to private or global data. A *system* in UPPAAL is the parallel composition of previously declared processes.

Clocks in Timed Automata A Timed Automaton is equipped with a finite set of *clocks*: variables whose valuation is a mapping from variable name to a time domain. In the theory of Timed-Automata [1], an assignment to a clock x is restricted to $x \rightarrow R^+$, where R^+ is a non-negative real domain. That is, a

clock can only be assigned a constant value.

The UPPAAL model checker, which is based on the theory of Timed Automata, places similar restrictions on assignments to clocks. In UPPAAL, a clock is variable of type *clock* and can only be assigned the value of an integer expression. Moreover, UPPAAL transition guards are limited to conjunctions of simple *clock conditions* or *data conditions*, where a clock condition is a clock compared to an integer expression or another clock, and a data condition is a comparison of two integer expressions.

Integer Clocks Many real-time systems require more clock manipulations than UPPAAL can provide. For example, it is not possible in UPPAAL to read a clock value and send it as a message along a channel, as is required, for example, in time-sychronization protocols such as TPSN and FTSP [8]. The reason is that in UPPAAL, it is not possible to advance a clock by any arbitrary amount. Instead, when a transition occurs, the resulting clock values, with the exception of those that are reset to zero by the transition, are constrained only to lie within the region (convex polyhedron) formed by the conjunction of the invariant associated with the transition's source state and the transition guard. One therefore cannot in general determine the exact value of a clock when a particular transition occurs.

To address this problem, we introduce the notion of an *integer clock*, which is simply an integer variable whose value advances periodically. To enforce clock advance, we introduce a stand-alone *Universal Pulse Generator* (UPG) process which, at every time unit, broadcasts a signal (pulse) to all other UPPAAL processes in the system. All processes having integer clocks are required to catch this signal and increment their respective integer clocks. Using this mechanism, we mimic discrete-time clocks that possess an exact desired value when a Timed Automaton transition occurs. Because integer clocks are just integers whose valuations are under the control of the UPG, it is possible to directly perform arithmetic operations on them as needed.

To improve code readability, we introduce the new derived type *intclock* as follows: `typedef int intclock;` and the UPG process is given by:

```
broadcast chan time_pulse;
process universal_pulse_generator()
{   clock t;
    state S { t <= 1 };
    init S;
    trans
        S -> S { guard t == 1; sync time_pulse!; assign t = 0; }; }
```

Processes deploying integer clocks must, at every state, specify transitions (one per integer clock) that respond to `time_pulse` events. For example, the following transition specifies a perfect (zero-drift) integer clock `x`: `S -> S { sync time_pulse?; assign x = x+1;}`

In the following example, process `A` sends out its current clock value to another process `B` after waiting a certain amount of time after initialization. Without using integer clocks, `A`'s clock cannot be read.

```
chan AtoB;
meta int msg;
process A()
{
   const int wait = 3;
   meta intclock x;
   state INIT, SENT;
   init INIT;
   trans
       INIT -> INIT { sync time_pulse?; assign x = x+1; },/* time advance */
       SENT -> SENT { sync time_pulse?; assign x = x+1; }, /* time advance */
       INIT -> SENT { guard x >= wait; sync AtoB!; assign msg = x; };
}
```

The drawback of using integer clocks is an increase (albeit modest) in model complexity, through the addition of one process (the UPG) and the corresponding time-advance transitions, one for every state of every process. The additional code needed to implement integer clocks is straightforward, and can be seen as a small price to pay for the benefits integer clocks bring to the specification process.

Time Drift Time-synchronization protocols such as TPSN are needed in sensor networks due to the phenomenon of *time drift*: local clocks advancing at slightly different rates. Time drift can be modeled using integer clocks by allowing a process to either ignore or double-count a UPG pulse event after a certain number of pulses determined by the *time-drift rate* (TDR). Integer clocks with time drift are used in Section 4 to model a sensor network containing nodes with a positive TDR (pulse events are double-counted periodically), a negative TDR (pulse events are ignored periodically), and a TDR of zero (no time drift) for the root node. As we show in Section 5, the TPSN protocol achieves periodic network-wide resynchronization with the root node.

For example, an integer clock having a TDR of 5 means that its value is incremented by 2 every 5 time units (and by 1 every other time unit). An integer clock having a TDR of -10 means that its value remains unchanged every 10 time units. A TDR of 0 means that the integer clock strictly follows the UPPAAL unit time-advance rate. The UPPAAL code for implementing integer clocks with TDR-based time drift appears in function `local_time_advance()` of the TPSN specification; see Appendix A in the online version of the paper.

4 Uppaal Model of TPSN

In this section, we use integer clocks to model TPSN in UPPAAL. In particular, we use integer clocks for each node's local time. We still, however, rely fundamentally on UPPAAL's built-in clock type for other modeling purposes, including, for example, the modeling of message channel delays. We consider a sensor network whose initial topology is a ring of N nodes indexed from 0 to $N-1$. Each node therefore has exactly two neighbors, and nodes are connected by communication channels. Node 0 is considered to be the root node. Note that the choice of a ring for the initial topology is not essential. The TPSN protocol constructs a spanning tree from the network topology, and the synchronization phase operates over this spanning tree.

Nodes and channels are modeled as UPPAAL processes. A node process maintains information about its neighbors, level in the spanning tree, parent, and local clock. Nodes send and receive packets through the channels processes. We cannot simply use UPPAAL's built-in `chan` type to model channels because we need to model the delay in the network, which is required in the protocol. Upon sending or receiving packets, a node switches its state and alter its data accordingly.

A node process takes three parameters, its id and the id of its two neighbors, and is declared in UPPAAL as follows: `process node(int[0,N-1] id, int[0,N-1] neighbor1, int[0,N-1] neighbor2)`

Channels also have ids. Channel k represents the communication media surrounding node k, and therefore is only accessible to node k and its neighbors.

Using Integer Clocks The TPSN protocol involves clock-value reading (a parent node sends its local time to a child node during the synchronization phase) and clock-value arithmetic (calculating Δ and d). We use integer clocks to model these operations. In a node process, we declare an integer clock as: `meta intclock local_time;`

A node process has seven states. There are therefore seven time-advance transitions in its specification; e.g. `initial -> initial { sync time_pulse?; assign local_time = local_time+1; }`

Level-Discovery Phase Three UPPAAL states are used to model the level-discovery phase: `initial`, `discovered`, `discovered-neighbors`. When the system starts, all nodes except the root enter the

`initial` state. The root goes directly to the `discovered` state, which indicates that a node has obtained its level in the hierarchy (zero for the root). After a certain delay, the root initiates the level-discovery phase by broadcasting a `level_discovery` message to its neighbors and then enters the `discovered-neighbors` state, means it has done the task to discover its neighbors' level. A node in the `initial` state enters the `discovered` state upon receiving a `level_discovery` message, or simply ignores this message if it is already in `discovered` or `discovered-neighbors` state. A node in the `discovered` state waits for some delay and then emits a `level_discovery`, changing its state to `discovered-neighbors`. As described in Section 2, this procedure will occur all over the network until eventually all nodes are in the `discovered-neighbors` state, indicating the entire network is level-discovered.

In our model, there are only two level-discovery transitions: from `initial` to `discovered` and from discovered to `discovered-neighbors`. Consider, for example, the former, which occurs upon receipt of a `level_discovery` message:

```
initial -> discovered {
    select i:int[0,N-1];
    guard level == UNDEF && neighbors[i];
    sync lev_recv[i]?;
    assign level = msg1[i]+1, parent = i;
},
```

If a node is not yet discovered (`guard level == UNDEF`) and the `level_discovery` message (through channel `lev_recv[i]`) is sent by its neighbor (`guard neighbor[i]`), then it sets its level to one greater than its neighbor's level (`msg1[i]+1`); i.e. this neighbor becomes its parent.

Synchronization Phase We consider state `discovered-neighbors` to be the initial state of the synchronization phase, and introduce four additional states: `sync-ready`, `sync-sent`, `sync-received` and `synchronized`. The root, which is the absolute standard of time, broadcasts a `time_sync` message to its neighbors, and then goes directly from `discovered-neighbors` to `synchronized`. The `time_sync` message notifies the root's children that they can now synchronize with their parent. The root's children then transit from `discovered-neighbors` to `sync-ready` (see Section 2). A node in the `sync-ready` state will send a `synchronization_pulse` to its parent and then enter `sync-sent`. The parent, which must be already synchronized, goes from `synchronized` to `sync-received`, where it will remain for a certain amount of delay. When the delay is over, it returns to the `synchronized` state and sends an `acknowledgment` message to the child. The child then adjusts its local clock using the information carried in these messages and goes to the `synchronized` state. All of these messages are actually broadcasted, which allows a child's child to overhear the `synchronization_pulse` message and turn itself into `sync-ready`. The state transition diagram for this phase is depicted in Figure 1.

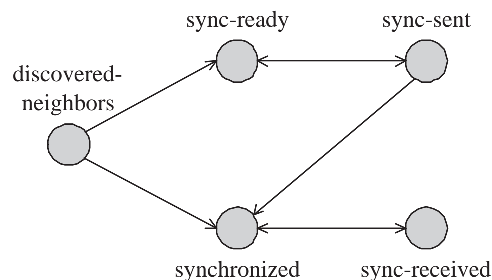

Figure 1: State Transition Diagram for the Synchronization Phase

For example, consider the transition a node makes from `sync-sent` to `synchronized` upon receipt of an `acknowledgment` message:

```
sync_sent -> synchronized {
    select i:int[0,N-1];
    guard id != ROOT_ID && i == parent;
    sync ack_recv[i]?;
    assign adjust_local_time();
}
```

This transition occurs if the node is not the root (`guard id != ROOT_ID`) and the incoming `acknowledgment` message is from its parent as determined during level-discovery (`guard i==parent`). The transition's affect is to adjust the node's local time (an integer clock) by calling function `adjust_local_time()`, which simply calculates the time drift Δ and adds Δ to `local_time`.

Time Drift TPSN addresses time drift in a sensor network by periodically performing resynchronization to ensure that each node's local clock does not diverge too much from the root. Since we have modeled local clocks using integer clocks, we cannot use arbitrary rational numbers for the TDR. Rather, we let the root advance its local clock at the pace of the UPG-generated pulse events and non-root nodes either skip or double-count a pulse after every certain number of pulses determined by an integer TDR. Since the TDR is fixed for each node, time drift appears linear in the long run. Figure 2 illustrates integer-clock-based time drift in a 5-node sensor network having respective TDRs $[0,-8,10,16,-10]$.

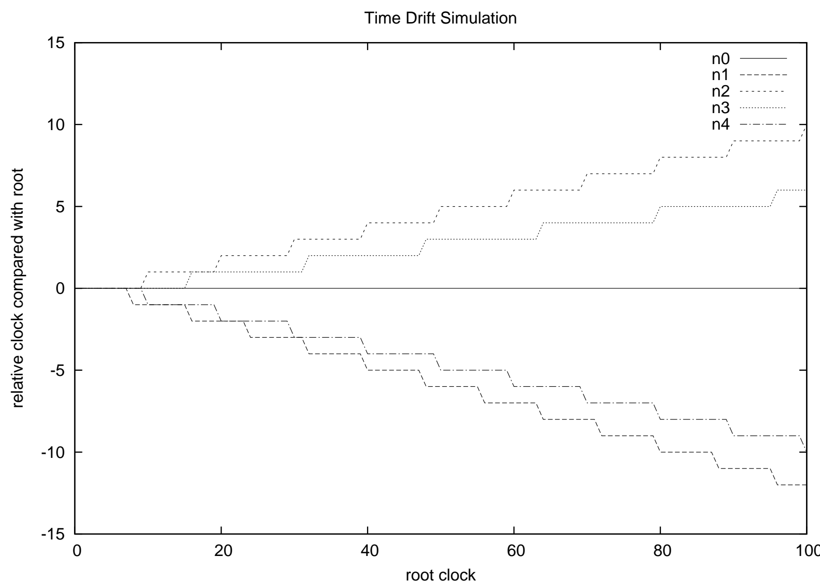

Figure 2: Integer-clock-based Time Drift

Resynchronization The resynchronization phase is almost identical to the synchronization phase except that the root will periodically jump from the `synchronized` state to the *discovered-neighbor* state. As in the synchronization phase, the root will soon re-enter the *synchronized* state and re-broadcast `time_sync`. We also let nodes in the `synchronized` state return to the `sync-ready` state on receiving `time_sync` or `synchronization_pulse`. Thus, another `time_sync` message will trigger another round of the synchronization phase, which is precisely the intent of resynchronization. The primary issue here is to set the resynchronization interval long enough to ensure synchronization has first completed.

System The system-level specification consists of N nodes, N channels, and the UPG processes:

```
n0 = node(0,1,2); n1 = node(1,0,3); n2 = node(2,0,4); n3 = node(3,1,4); n4 = node(4,2,3);
c0 = channel(0);  c1 = channel(1);  c2 = channel(2);  c3 = channel(3);  c4 = channel(4);
system n0, n1, n2, n3, n4, c0, c1, c2, c3, c4, universal_pulse_generator;
```

The choice of node-process parameters reflects the initial ring topology we are considering for the system. The system definition can be easily scaled using the process-parameter mechanism, as long as one carefully sets the neighbors of each node.

5 Simulation and Verification Results

We verified our model of the TPSN protocol given in Section 4, including the level-discovery, synchronization and periodic-resynchronization phases, using the UPPAAL model checker. We regard the root node as the node with the ideal clock (zero time drift) and all other nodes have (distinct and fixed) positive and negative TDRs. Since nodes are initially unsynchronized, each node is given a different initial clock value. Our model is also parameterized by the minimum and maximum node-response delay (20 and 40, respectively), channel delay (4), and minimum and maximum resynchronization interval (10 and 20, respectively), for which we have provided appropriate values.

Simulation Results We used the UPPAAL TRACER tool, and a resynchronization interval of 200 time units, to obtain simulation results for our model with periodic resynchronizations of the local clocks of all nodes. Figure 3 is obtained by extracting the local clock values of all nodes from a random simulation trace. This was accomplished by first saving the history of states to a trace file and then using the external program UPPAAL TRACER to convert the data in the trace file into to a readable form. Figure 2 of Section 4 was obtained similarly.

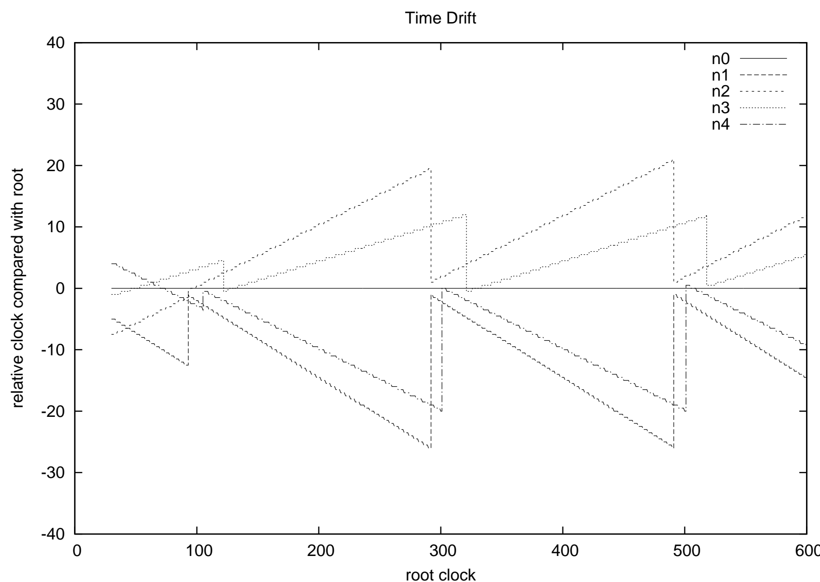

Figure 3: Integer Time Drift with a resynchronization interval of 200 time units

In Figure 3, plotting starts at time 30 since the root's starting clock value is set to 30. Between time 30 and 100, the network executes the level-discovery phase. The synchronization phase takes place

approximately between time 100 and 120. From there, all local clocks advance at their own rate, which, as can be seen, consists of four relative clocks each having a distinct TDR. At time 300, and then again at time 500, the system performs a resynchronization, which brings non-root local-clock values very close to 0. Notice that different nodes get synchronized at different times. Nodes 1 and 2 are synchronized at the same time because they are adjacent to the root. Nodes 3 and 4 are synchronized after them because of node response time and packet delay.

Verification Results We used UPPAAL to verify the following properties of our model. (Correctness properties can be specified in UPPAAL using a subset of the CTL temporal logic.)

No Deadlock: $A\,[\,]\, not \;\; deadlock$
Absence of deadlock.

Synchronized: $A\,[\,]\,(<>n_i.state == synchronized)$
All nodes enter a *synchronized* state at least once. This implies that the level-discovery and synchronization phases are correctly implemented.

Relative Time Bounded: $A\,[\,]\,abs(n_i.local_clock - n_0.local_clock) < X$
A node's local time relative to the root is bounded at all times ($X = 25$), which implies correct repeated resynchronization.

Relative Time Close: $A\,[\,]\,(<>abs(n_i.local_clock - n_0.local_clock) < Y)$
A node's relative time will always eventually get very close to 0 ($Y = 5$), another implication of correct repeated resynchronization.

We used UPPAAL to perform model checking of these properties on networks having $N = 3-5$ nodes. The corresponding verification results (CPU time and memory usage) are given in Table 1. In all cases, UPPAAL reported that the four properties are true. (The second and fourth properties, which are "infintiely often" properties, are verified only up to the time bound determined by the maximum UPPAAL integer value of 32,767.) All results are obtained on a Linux PC with a 2GHz Intel Pentium CPU and 2GB memory. UPPAAL's default settings were used.

N	No Deadlock	Synchronized	Relative Time Bounded	Relative Time Close
3	$0.61sec/21MB$	$2.06sec/24MB$	$0.62sec/21MB$	$2.11sec/24MB$
4	$6.5sec/22MB$	$68.0sec/31MB$	$6.7sec/22MB$	$70.2sec/31MB$
5	$6.1min/126MB$	$214.9min/181MB$	$6.3min/126MB$	$236.4min/181MB$

Table 1: Verification Results: Time and Memory Usage

6 Related Work

In [6], the FTSP flooding-time synchronization protocol) for sensor networks has been verified using the SPIN model checker. Properties considered include *synchronization of all nodes to a common root node*, and *eventually, all nodes get synchronized*, for networks consisting of 2-4 nodes. In [4], UPPAAL has been used to verify a time-synchronization protocol for a time-division-multiple-access (TDMA) communication model. The aim of this protocol is to ensure that the clocks of nearby (neighboring) nodes are synchronized. UPPAAL has also been used to demonstrate the correctness of the LMAC medium access control protocol for sensor networks [11], and the the minimum-cost forwarding (MCF) routing protocol [12] for sensor networks [5].

Time-triggered systems are distributed systems in which the nodes are independently-clocked but maintain synchrony with one another. In [9], real-time verification of time-triggered protocols has been performed using a combination of mechanical theorem proving, bounded model-checking and SMT (satisfiability modulo theories) solving. Calendar automata, which use sparse-time constraints, are used in [2] for the modeling and verification of the fault-tolerant, real-time startup protocol used in the Timed Triggered Architecture [10].

7 Conclusions

We used the UPPAAL model checker for Timed Automata to obtain a number of critical verification results for the TPSN time-synchronization protocol for sensor networks. Clock-synchronization algorithms for sensor networks such as TPSN must be able to perform arithmetic on clock values to calculate clock drift and network propagation delays. They must be able to read the value of a local clock and assign it to another local clock. The introduction of the *integer clock* (with time drift) derived UPPAAL type greatly facilitated the encoding of these operations, as they are not directly supported by the theory of Timed Automata. We also used the UPPAAL TRACER tool to illustrate how integer clocks can be used to capture clock drift and resynchronization during protocol execution.

References

[1] R. Alur and D. L. Dill. The theory of timed automata. *TCS*, 126(2), 1994.

[2] B. Dutertre and M. Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *In FORMATS/FTRTFT*, pages 199–214, 2004.

[3] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync protocol for sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys)*, pages 138–149. ACM, 2003.

[4] F. Heidarian, J. Schmaltz, and F. W. Vaandrager. Analysis of a clock synchronization protocol for wireless sensor networks. In *FM*, pages 516–531, 2009.

[5] W. Henderson and S. Tron. Verification of the minimum cost forwarding protocol for wireless sensor networks. In *IEEE Conference on Emerging Technologies and Factory Automation*, pages 194–201. IEEE Computer Society, 2006.

[6] B. Kusy and S. Abdelwahed. FTSP protocol verification using SPIN. Technical Report ISIS-06-704, Institute for Software Integrated Systems, May 2006.

[7] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1-2):134–152, 1997.

[8] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi. The flooding time synchronization protocol. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 39–49. ACM, 2004.

[9] L. Pike. Modeling time-triggered protocols and verifying their real-time schedules. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD'07)*, pages 231–238. IEEE, 2007.

[10] W. Steiner and M. Paulitsch. The transition from asynchronous to synchronous system operation: An approach for distributed fault-tolerant systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 329. IEEE Computer Society, 2002.

[11] L. van Hoesel and P. Havinga. A lightweight medium access protocol (LMAC) for wireless sensor networks: Reducing preamble transmissions and transceiver state switches. In *1st International Workshop on Networked Sensing Systems (INSS)*, pages 205–208, 2004.

[12] F. Ye, A. Chen, S. Lu, L. Zhang, and F. Y. A. Chen. A scalable solution to minimum cost forwarding in large sensor networks. In *IEEE 10th International Conference on Computer Communications and Networks*, pages 304–309, 2001.