

# Model-checking the Flooding Time Synchronization Protocol

Allan I. McInnes

Electrical and Computer Engineering  
University of Canterbury, Christchurch, New Zealand  
e-mail: allan.mcinnnes@canterbury.ac.nz

**Abstract**—Large-scale wireless sensor networks must be reliable, since they are intended to be operated without human intervention. Using well-understood building-blocks is one method of increasing confidence in the reliability of a sensor network design. In this paper, we use model-checking to analyze and characterize the Flooding Time Synchronization Protocol, a synchronization protocol that is distributed along with the TinyOS sensor network operating system. We apply a number of abstraction techniques to keep the model state-space small, and as a result are able to verify several properties of FTSP networks that have not previously been checked. Our results provide greater confidence in FTSP, and also establish some limitations on the size of FTSP networks. Our FTSP model provides a basis for further model-checking of FTSP.

**Index Terms**—Wireless sensor networks, Time synchronization, Model checking, Process algebra.

## I. INTRODUCTION

The Flooding Time Synchronization Protocol (FTSP) [1] is a synchronization protocol developed for use in low-power sensor networks. An implementation of FTSP is distributed as an experimental component of the TinyOS sensor network operating system [2]. Since FTSP is a building block for sensor network applications that rely on synchronization between nodes, it is important that the protocol be robust.

FTSP blends leader election and time synchronization into a single protocol. The protocol is conceptually simple, but unanticipated node interactions may produce unexpected network behavior. Although FTSP has been extensively tested, at least one deployed sensor network using FTSP has encountered such unexpected behavior [3]. It is therefore worthwhile to consider additional methods of uncovering errors in the FTSP design besides testing.

Model-checking is a method of discovering design errors by systematically exploring the state-space of an abstract model of a design. Model-checking was previously applied to FTSP by Kusy and Abdelwahed [4]. However, they were only able to fully model-check 2-node networks. These single-hop networks are unable to exhibit behaviors caused by lack of direct contact between nodes.

Our contributions to model-checking FTSP are:

- We define a model of FTSP (section III) using the process algebra Communicating Sequential Processes (CSP) [5]. This model appears to be more efficient than Kusy and Abdelwahed's model, in the sense that it has far fewer states and can thus be used to check substantially larger networks. Our choice of CSP was motivated by a desire to include the FTSP model within

a larger effort to apply CSP to problems in sensor network design [6].

- We specify several properties of FTSP networks using CSP refinement assertions (section IV) suitable for checking using the FDR2 model-checker [7].
- We report the results of using FDR2 to check FTSP networks of up to seven nodes (section IV-B). Our results are largely positive: we found that all of the network configurations we checked always converge to a single root node and global time. However, we found there is no guarantee that the global time will be the root local time. We also found that there is a bound on the size of FTSP networks related to the size of the sequence number representation. FTSP networks with a radius exceeding this bound will not function correctly.

We place our work in context, and briefly compare our results to Kusy and Abdelwahed's, in section V.

## II. BACKGROUND

Before describing our model of FTSP, we first review the design of FTSP and the fundamentals of CSP.

### A. The Flooding Time Synchronization Protocol

Maróti *et al.* [1] designed FTSP to provide network-wide time synchronization with errors in the micro-second range, and to be scalable to networks containing hundreds of nodes. Individual FTSP nodes essentially perform two operations: they receive and process messages from other nodes to update their estimate of the global time, and they periodically broadcast synchronization messages (beacons) that carry their current estimate of the global time. Listings 1 and 2 show simplified pseudocode for FTSP, derived from the current TinyOS FTSP implementation.

The node responsible for maintaining the global time of the FTSP network is called the root. FTSP uses a dynamic leader election scheme to decide the root node. If the root fails, the other nodes will eventually time out and initiate a new election (lines 12–15 of Listing 2). Once a node has become the root it begins transmitting synchronization messages time-stamped with the current global time.

Receiving nodes store the local time-of-arrival and the global time-stamp value of the last eight received messages (lines 17–26 of Listing 1), and perform linear regression through these data-points to compute the offset and skew of their local time relative to the global time. When nodes receiving messages from the root have accumulated enough

Listing 1. Pseudocode for FTSP message reception

```

1 event Receive.receive(message_t* msg)
2 {
3     // processMsg task
4     if( msg.rootID < myRootID &&
5         ~(heartBeats < IGNORE_ROOT_MSG
6           && myRootID == myID) ) {
7         myRootID = msg->rootID;
8         mySeqNum = msg->seqNum;
9     } else if( myRootID == msg->rootID
10              && (int8_t)(msg->seqNum
11                  - mySeqNum) > 0 ) {
12         mySeqNum = msg->seqNum;
13     } else return;
14
15     if( myRootID < myID ) heartBeats = 0;
16
17     // addNewEntry function
18     if( (numEntries >= ENTRY_VALID_LIMIT
19         || myRootID == myID) &&
20         (timeError > THROWOUT_LIMIT
21         || timeError < -THROWOUT_LIMIT) ) {
22         if (++numErrors > 3) clearTable();
23         return;
24     }
25     // Add entry and update myGlobalTime
26 }

```

entries in their regression table to produce accurate global time estimates they are considered synchronized, and begin transmitting their own beacon messages to propagate the global time through the network (lines 18–25 of Listing 2).

In addition to the global time, the root also generates a sequence number (line 27 of Listing 2) that is added to each outgoing message and is propagated through the network. The sequence number allows nodes to determine whether a received message provides new information (lines 10–11 of Listing 1). Messages with sequence numbers older than the most recent received sequence number are ignored. Only the root node is allowed to increment sequence numbers.

### B. CSP and FDR2

FDR2 [7] is an industrial-strength tool that automatically checks models for conformance to specifications. The language used to build models for FDR2 is CSP [5]. CSP processes define sequences of events, and interact with other processes by synchronizing on shared events. Related events can be grouped into channels, so that each event corresponds to sending or receiving data through the channel.

As an example of CSP, consider a modeling an online bookstore. The model might include events that represent ordering a book, confirming the order, providing payment, and shipping the book. These events can be arranged as a sequential process using the prefix operator ( $\rightarrow$ ):

```

BookOrder =
  order?book -> confirm!book ->
  request_payment!cost(book) ->
  receive_payment?p -> ship!book -> SKIP

```

where  $?$  and  $!$  respectively indicate channel input and output, and **SKIP** is a primitive representing successful termination.

Listing 2. Pseudocode for FTSP message transmission

```

1 event Timer.fired()
2 {
3     // timeSyncMsgSend function
4     if( myRootID == 0xFFFF
5         && ++heartBeats >= ROOT_TIMEOUT ) {
6         mySeqNum = 0;
7         myRootID = myID;
8     }
9     if( myRootID != 0xFFFF ) {
10        // sendMsg task
11        if( heartBeats >= ROOT_TIMEOUT ) {
12            heartBeats = 0;
13            myRootID = myID;
14            ++mySeqNum;
15        }
16        if( numEntries < ENTRY_SEND_LIMIT
17            && myRootID != myID ) {
18            ++heartBeats;
19        } else {
20            msg.rootID = myRootID;
21            msg.seqNum = mySeqNum;
22            msg.globalTime = myGlobalTime
23            Radio.send(msg);
24            ++heartBeats;
25            if( myRootID == myID ) ++mySeqNum;
26        }
27    }
28 }

```

Although the sequential ordering process allows books to be ordered, it has the unfortunate habit of shipping a book no matter the received payment. More complex processes can be defined using other CSP operators, such as:

- Conditional execution:  
`if p == cost(book) then ... else ...`
- Alternative behaviors (external choice):  
`DisplayCart [] DisplayCatalog`
- Independent parallel execution:  
`Servers = Server(1) ||| Server(2)`
- Parallelism with synchronization on shared events:  
`Customer [|OrderEvents|] Servers`

In addition to a notation for describing concurrent systems, CSP provides a rich theory of process refinement. A process  $Q$  is said to refine another process  $P$  if the behavior of  $Q$  is in some way a subset of the behavior of  $P$ . For example, if

```

P = (a -> c -> SKIP) [] (b -> SKIP)
Q = a -> SKIP

```

then the sequences of events that  $P$  and  $Q$  can perform are

```

traces(P) = {<>, <a>, <b>, <a, c>}
traces(Q) = {<>, <a>}.

```

Thus  $Q$  is a *trace-refinement* of  $P$ , written  $P \mathbf{[T=]} Q$ . The other standard refinement relations are *stable-failures refinement* and *failures-divergences refinement*. These relations differ from trace-refinement in that they consider not only the traces of each process but also the events each process can refuse to perform at each step of its execution, allowing finer distinctions to be made between processes.

Refinement can be used to define relationships between specification process models and implementation process models. FDR2 checks the validity of such refinement relations. For example, the specification process

```
ValidPayment =
  [] p:Payments @ request_payment.p ->
    receive_payment.p -> ValidPayment
```

requires that any `request_payment` event asking for an amount of money `p` be followed by a `receive_payment` event for the same amount. If we hide all events aside from the `request_payment` and `receive_payment` channels, then checking the refinement assertion

```
S = diff(Events, {|request_payment,
                  receive_payment|})
```

```
assert ValidPayment [T= BookOrder \ S
```

reveals that the refinement does not hold: `BookOrder` will accept any payment, and thus has a traces set that includes sequences of events not in `traces(ValidPayment)`.

### III. MODELING THE PROTOCOL

Modeling FTSP in a way that is suitable for model-checking requires careful use of abstraction to keep the state-space of the model small. We model nodes as purely sequential processes that contain bounded state variables and manipulate abstract time symbols. We model the network as a parallel composition of node processes that abstracts from physical and medium access control layer concerns.

#### A. Nodes

Our FTSP node model underwent numerous revisions as we attempted to reduce the size of the state-space. The final version of the model relies on the following abstractions:

- A simplified execution model that abstracts from TinyOS commands, events, and tasks.
- A model of time-synchronization that abstracts from numerical timer ticks and tracks only the identity of the local time that each node is using as its global time.
- A model of time based on discrete rounds. Only a single time-firing per node may occur in a round. The order of timer-firings within a round is unconstrained.
- State variables that are modeled as bounded counters.
- Sequence numbers that are restricted to the smallest range that still permits synchronization to occur.

Nodes transmit synchronization messages to their neighbours on the `tx` channel, and receive messages on the `rx` channel. Occurrences of `rx` events correspond to `Receive.receive()` events in the FTSP implementation. We do not explicitly model `Timer.fired()` events, although they are implicitly responsible for triggering `tx` events. Most of the complexity of the model is in the logic used to decide whether and what to transmit, and to update the node state in response to received messages.

The structure of the node model is shown in Listing 3. We have tried to maintain the form of the FTSP implementation within our model to make it easier to relate the model to the

code. However, we have not included TinyOS tasks in the model, but instead allow events to produce immediate results. We justify this abstraction on the basis that the FIFO nature of the TinyOS task scheduler ensures the order of events is reflected in the execution order of the event processing tasks.

Within the FTSPnode process, the processes `Receive`, `AddNewEntry`, `TimeSyncMsgSend`, and `SendMsg` correspond to their namesakes in the FTSP implementation. The processes `AwaitTimer` and `TimerFired` represent the two principal modes of FTSP node behavior: waiting for the next timer round, and responding to a timer firing event. Each FTSP node starts in the `AwaitTimer` state (line 10 of Listing 3), with an undefined root node and an initial time estimate drawn from the set `Time = time_t.{0,1,2}`. Here `time_t.0` represents the local time of the root node. We provide two other abstract time symbols so that the non-root nodes are not all set to the same initial time.

The `AwaitTimer` process (Listing 3, line 3) is an external choice between processing a received message or engaging in the `tock` event and moving into the `TimerFired` state:

```
AwaitTimer(rootID, seqNum, heartBeats,
            nEntries, time) =
  Receive(rootID, seqNum, heartBeats,
          nEntries, time, AwaitTimer)
[]
tock ->
  TimerFired(rootID, seqNum, heartBeats,
             nEntries, time)
```

We use synchronization on `tock`<sup>1</sup> to enforce a constraint that no node timer can fire more than twice between any two timer firings on another node. This constraint models the assumption all node timers have similar rates. As explained below, in each round of timer firings we allow the effects of firings on different nodes to occur in any order.

The occurrence of a `tock` event leads the model into the `TimerFired` process (Listing 3, line 4). This process provides an external choice between processing a received message or transmitting a beacon message:

```
TimerFired(rootID, seqNum, heartBeats,
            nEntries, time) =
  Receive(rootID, seqNum, heartBeats,
          nEntries, time, TimerFired)
[]
TimeSyncMsgSend(rootID, seqNum,
                 heartBeats, nEntries, time)
```

All nodes synchronize on `tock` and enter the `TimerFired` state at the same time, and all nodes offer the same choice between message transmission and reception. As a result, the order of transmissions is resolved nondeterministically. The arbitrary order of transmissions in each timer round models the effects of clock-skew, differences in node start-time, other mismatches between timers, and Medium Access Control (MAC) layer resolution of transmission conflicts.

The `TimeSyncMsgSend` process (Listing 3, line 5) makes use of a bounded counter abstraction. The `heartBeats`

<sup>1</sup>The name “tock” is traditionally used in discrete-time CSP models because in CSP theory “tick” refers to a special event that indicates successful process termination.

Listing 3. Top-level structure of the FTSP node model

```

1 FTSPnode (nodeID) =
2 let
3   AwaitTimer (rootID, seqNum, heartBeats, nEntries, time) = ...
4   TimerFired (rootID, seqNum, heartBeats, nEntries, time) = ...
5   TimeSyncMsgSend (rootID, seqNum, heartBeats, nEntries, time) = ...
6   SendMsg (rootID, seqNum, heartBeats, nEntries, time) = ...
7   Receive (rootID, seqNum, heartBeats, nEntries, time, Next) = ...
8   AddNewEntry (rootID, seqNum, heartBeats, nEntries, time, mt, Next) = ...
9 within
10  AwaitTimer (UNDEFINED_NODE, 0, 0, 0, init_time)

```

variable is only used in comparisons against `ROOT_TIMEOUT`. The number of rounds it takes `heartBeats` to reach `ROOT_TIMEOUT` is important, but once `heartBeats` reaches `ROOT_TIMEOUT` its actual value does not matter. We therefore use the bounded increment function

```

binc(x, MAX) = if x < MAX then (x + 1)
              else MAX

```

to increment `heartBeats` instead of the unbounded version used in the FTSP implementation. This abstraction keeps the range of `heartBeats` small, reducing the model state-space.

The `TimeSyncMsgSend` process is:

```

TimeSyncMsgSend (rootID, seqNum, heartBeats,
                 nEntries, time) =
let
  heartBeats' = binc (heartBeats, ROOT_TIMEOUT)
within
  if rootID != UNDEFINED_NODE
  then SendMsg (rootID, seqNum, heartBeats,
               nEntries, time)
  else
    if heartBeats' >= ROOT_TIMEOUT
    then SendMsg (nodeID, 0, heartBeats',
                 nEntries, time)
    else AwaitTimer (rootID, seqNum,
                     heartBeats', nEntries,
                     time)

```

The decision logic parallels that of the FTSP implementation (see Listing 2): the node transitions to `SendMsg` if it has a root, declares itself root and then transitions to `SendMsg` if it has been without a root for `ROOT_TIMEOUT` beacon periods, and otherwise increments `heartBeats` and returns to waiting for the next timer firing.

The `SendMsg` process (Listing 3, line 6) is responsible for the final decision on whether to transmit a message.

```

SendMsg (rootID, seqNum, heartBeats,
         nEntries, time) =
let SendMsgAux = ... within
  if (rootID != nodeID
      and heartBeats >= ROOT_TIMEOUT)
  then SendMsgAux (nodeID, sqinc (seqNum),
                  0, nEntries, time)
  else SendMsgAux (rootID, seqNum,
                  heartBeats, nEntries, time)

```

where `sqinc(x) = (x+1) % (MAX_SEQNUM+1)` wraps back to 0 when incrementing past `MAX_SEQNUM`. Within `SendMsg` the auxiliary process `SendMsgAux` handles the details of

message transmission. If the decision is made to transmit, a message containing the current root, sequence number and time symbol is sent on the `tx` channel. If the transmitting node is a root node, it increments the sequence number.

```

SendMsgAux (rootID, seqNum, heartBeats,
            nEntries, time) =
let
  heartBeats' = binc (heartBeats, ROOT_TIMEOUT)
within
  if (nEntries < ENTRY_SEND_LIMIT
      and rootID != nodeID)
  then AwaitTimer (rootID, seqNum, heartBeats',
                  nEntries, time)
  else
    tx ! rootID.seqNum.time ->
    let
      seqNum' = if rootID == nodeID
                then sqinc (seqNum)
                else seqNum
    within
      AwaitTimer (rootID, seqNum', heartBeats',
                  nEntries, time)

```

The `Receive` process (Listing 3, line 7) is initiated by an `rx` event, and proceeds by examining the message content to determine whether the message requires further processing:

```

Receive (rootID, seqNum, heartBeats,
        nEntries, time, Next) =
rx ? mr.msn.mt -> -- ProcessMsg
  if (mr < rootID
      and not (heartBeats < IGNORE_ROOT_MSG
              and rootID == nodeID))
  then AddNewEntry (mr, msn, heartBeats,
                  nEntries, time, mt,
                  Next)
  else
    if (mr == rootID
        and newer (seqNum, msn))
    then AddNewEntry (rootID, msn,
                    heartBeats, nEntries,
                    time, mt, Next)
    else Next (rootID, seqNum, heartBeats,
                nEntries, time)

```

The FTSP implementation tests for new sequence numbers using the properties of unsigned 8-bit arithmetic (Listing 1, lines 10–11). Our model uses a much smaller sequence number range, so we cannot use the same arithmetic trick to test sequence numbers. Our `newer` function (Listing 4) generalizes the technique used in the FTSP implementation to sequence number ranges other than `[0, 255]`.

Listing 4. Generalized function for determining sequence number newness

```

1 newer(s1, s2) = -- True if s2 newer than s1
2   let
3     d = s2 - s1
4     MID = (MAX_SEQNUM + 1)/2
5   within
6     (d < MID and d > 0) or (d < -MID)

```

The last element of the `FTSPnode` model is `AddNewEntry` (Listing 3, line 8). This process updates the node regression-table state. Since our model abstracts from numerical time values and calculation of global-time estimates, we do not store a table of time entries. Instead, we maintain a bounded count of the number of entries added to the table.

```

AddNewEntry(rootID, seqNum, heartBeats,
            nEntries, time, mt, Next) =
let
  heartBeats' = if rootID < nodeID
                then 0 else heartBeats
within
  if (nEntries >= ENTRY_VALID_LIMIT
      or rootID == nodeID) and (mt != time)
  then Next(rootID, seqNum, heartBeats',
            0, init_time) -- Clear table
  else -- Add entry
    let nEntries' = binc(nEntries,
                        ENTRY_VALID_LIMIT)
        within
          Next(rootID, seqNum, heartBeats',
              nEntries', mt)

```

### B. The Network

The network model is built from `FTSPnode` processes. We model a grid topology in which each node communicates with the eight surrounding nodes (see Fig. 1). This topology is identical to that used in Kusy and Abdelwahed’s work and the FTSP testing reported by Maróti *et al.* [1].

Our network model is built on the following assumptions:

- Message transmission is effectively instantaneous.
- Messages are received simultaneously by all recipients.
- All links are fault-free.
- MAC-layer arbitration is adequately modeled by non-deterministic interactions between nodes that are prepared to both transmit and receive.

The network model is a parallel composition indexed over the set `Nodes = {0..(NUM_NODES-1)}`:

```
Net = (|| i:Nodes @ [Alpha(i)] Element(i))
```

where `Alpha(i)` is the interface of the  $i$ th grid element, and `Element(i)` is constructed from an `FTSPnode` process by assigning a unique node identifier, `ID(i)`, and renaming the `tx` and `rx` channels.

The renaming used in defining `Element(i)` maps `tx` to the  $i$ th broadcast channel, and `rx` to the broadcast channels of the node’s neighbours (see Fig. 2):

```

Element(i) =
  FTSPnode(ID(i)) [[tx <- broadcast.i,
                    rx <- broadcast.j
                    | j <- neighbours(i)]]

```

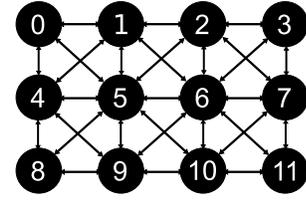


Fig. 1. Example 4 × 3 grid network

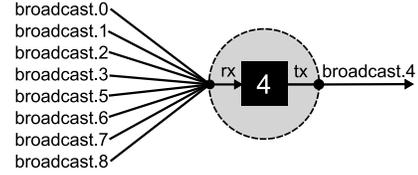


Fig. 2. Renaming node 4 in a 3 × 3 grid

After renaming, the interface of each element contains `tock`, on which all nodes synchronize, and the broadcast channels of the element and its neighbours:

```

Alpha(i) = union({| tock, broadcast.i |},
                { broadcast.j.m
                  | j <- neighbours(i),
                    m <- Message })

```

Synchronization on the broadcast channels in `Alpha(i)` allows two-way communication between `Element(i)` and its neighbours.

## IV. CHECKING THE MODEL

We specified and checked five properties of the FTSP network model:

- 1) **Time Progress:** the model allows only a finite number of events between two ticks of the global clock.
- 2) **Single Transmission per Timer Round:** the model produces no more than one transmission from each node in each round of timer-firings.
- 3) **Root Convergence:** all nodes always eventually agree on a single global root.
- 4) **Time Convergence to Root Time:** all nodes always eventually synchronize to the root local time.
- 5) **Time Convergence:** all nodes always eventually synchronize to some single global time.

The first two properties are sanity checks to ensure that the model behaves as intended. The check for time progress is a standard check on discrete-time process models. It ensures that the model does not include behaviors that circumvent the constraints of the time model. The check on the number of transmissions in each timer round ensures that the `tock`-based constraints on relative timer rates work as intended.

The last three properties are the aspects of FTSP that it was our goal to check. We list two time convergence properties because the first was what we originally set out to check, while the second is what we found to hold. We discuss FTSP’s time convergence guarantees further in Section IV-B.

We first describe the specification of each property in CSP, and then discuss our model-checking results.

## A. Property Specifications

We specify all five properties using refinement assertions.

1) *Time Progress*: We express time-progress as a failures-divergences refinement assertion stating that if all events aside from `tock` are hidden then the observable behavior of the model should be an infinite series of `tock` events:

```
TimeProgress = tock -> TimeProgress
assert TimeProgress [FD=
  Net \ diff(Events, {tock})
```

In other words, the `Net` process should never be able to enter a state in which it either stops completely, or performs an infinite number of non-`tock` events. This is a standard approach for specifying time progress in CSP [5].

2) *Single Transmission*: We express the single-transmission property as a trace-refinement assertion

```
assert OneTxPerRound [T= Net
```

where the traces of `OneTxPerRound` contain no more than one broadcast event per node between any two `tock` events. Our definition of `OneTxPerRound` is:

```
OneTxPerRound = let
  Transmit({}) = tock -> Transmit(Nodes)
  Transmit(N) =
    [] n:N @ broadcast.n ? _ ->
      Transmit(diff(N, {n}))
    [] tock -> Transmit(Nodes)
within Transmit(Nodes)
```

3) *Root Convergence*: We specify root-convergence by using an auxiliary tester process that runs in parallel with the network model and tracks the network state by observing all broadcast events. The tester generates an infinite sequence of `success` events if the network converges to the expected root node and remains in that state. We express root-convergence as the failures-divergences refinement assertion

```
Convergence = success -> Convergence
assert Convergence [FD=
  ( Net [||broadcast||] RootConvTester )
  \ diff(Events, {success, error})
```

The tester process is:

```
RootConvTester = let
  Test(Y, {}) =
    ([] n:Y @ broadcast.n.ROOT_NODE ? _
     -> success -> Test(Y, {}))
    [][[] n:Y, r:diff(Nodes, {ROOT_NODE})
     @ broadcast.n.r ? _ -> error
     -> Test(diff(Y, {n}), {n})]
  Test(Y, N) =
    ([] n:union(Y, N)
     @ broadcast.n.ROOT_NODE ? _
     -> Test(union(Y, {n}), diff(N, {n})))
    [][[] n:union(Y, N),
     r:diff(Nodes, {ROOT_NODE})
     @ broadcast.n.r ? _
     -> Test(diff(Y, {n}),
             union(N, {n}))]
within Test({}, Nodes)
```

where `Y` is the set of nodes that currently indicate their root is `ROOT_NODE = 0`, and `N` is the set of nodes that have not yet signaled the correct root.

4) *Time Convergence to Root Time*: As with the root-convergence specification, we use a tester-based refinement assertion to specify that the network eventually converges to a global time identical to the local time of the root node (node 0). The difference between the two specifications is the tester process, which in this case is:

```
TimeConvToRootTester = let
  Test(Y, {}) =
    ([] n:Y @ broadcast.n ?_ ?_ ! time_t.0
     -> success -> Test(Y, {}))
    [][[] n:Y, t:diff(Time, {time_t.0})
     @ broadcast.n ?_ ?_ ! t
     -> error
     -> Test(diff(Y, {n}), {n})]
  Test(Y, N) =
    ([] n:union(Y, N)
     @ broadcast.n ?_ ?_ ! time_t.0
     -> Test(union(Y, {n}), diff(N, {n})))
    [][[] n:union(Y, N),
     t:diff(Time, {time_t.0})
     @ broadcast.n ?_ ?_ ! t
     -> Test(diff(Y, {n}),
             union(N, {n}))]
within Test({}, Nodes)
```

The tester process signals `success` once all nodes are broadcasting messages containing the root local time.

5) *Time Convergence*: The time-convergence property is more general than the time-convergence-to-root property. It requires only that the network converge to *some* global time. That time does not have to be the local time of the root. Our specification for this property is similar to that used to specify time-convergence-to-root. However, the general time-convergence tester runs several `Test` processes in parallel:

```
TimeConvTester = let ... within
  [||broadcast||] t:Time
  @ Test({}, Nodes, t)
```

Each `Test` process is similar to the single `Test` process within `TimeConvToRootTester`, but each tests for convergence to a different time. If at least one `Test` process reaches a converged state `TimeConvTester` will produce an infinite stream of `success` events.

## B. Results

We used FDR2 to check the properties defined in section IV-A for grid networks ranging in size from 2 to 7 nodes. We attempted checking a network larger than 7 nodes, but found the state-space to be so large that after 5 hours the checking process had made no obvious progress. Fig. 3 shows the network configurations we examined. For each configuration we ran separate checks with the root-node located at the edge of the network, and located in the middle of the network. We also checked each configuration with node identifiers assigned in sequence, and with identifiers assigned so that consecutive node identifiers were not adjacent. In all cases we found that, given a value of `MAX_SEQNUM` greater than twice the network radius, the FTSP network always converges to the root and to a single global time. The results of our model-checking are summarized in Table I.

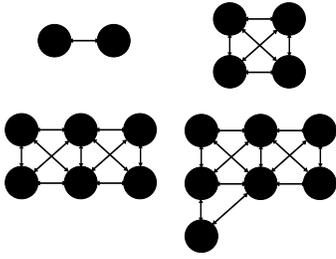


Fig. 3. The network configurations examined in this analysis

TABLE I

RESULTS FOR NETWORKS WITH  $\text{MAX\_SEQNUM} > 2 \text{NetworkRadius}$

	2 nodes	4 nodes	6 nodes	7 nodes
Time progress	✓	✓	✓	✓
Single transmission	✓	✓	✓	✓
Root convergence	✓	✓	✓	✓
Time convergence to root	×	×	×	×
Time convergence	✓	✓	✓	✓

1) *Unexpected Time Convergence Result:* Our first specification for time-convergence assumed that the eventual global time is always identical to the local time of the root. But checking that specification shows that even a 2-node network does not always converge to the global time we expect. Listing 5 is a counterexample trace that shows how the global time may end up being something other than the root local time. During network startup node 1 reaches `heartBeats >= ROOT_TIMEOUT`, declares itself root, and broadcasts a beacon message before the node 0 timer fires (line 6). Because node 0 receives the message before it declares itself the root, its internal `rootID` is still undefined (`0xFFFF`), and the node incorporates the received time information into its regression table, offset and skew. Since the regression table, offset, and skew values are not cleared when the node becomes the root node in the next round (line 8), the global time maintained by the root has a permanent offset and skew from the root local time.

Initially, we thought the behavior we observed might be a bug in the FTSP design. However, upon re-examining the FTSP design [1] we determined that this behavior is caused by the way the protocol is designed to ensure smooth transitions in global time when new root nodes are added to

Listing 5. Counterexample trace of events at the intended root

```

1 tock
2 tock
3 tock
4 tock
5 tock
6 rx.1.0.time_t.2
7 tock
8 tx.0.1.time_t.2
9 tock
10 tx.0.2.time_t.2
11 ...

```

an FTSP network. The unexpected behavior results when the order of node timer firings causes the intended root node to effectively join the FTSP network slightly later than the other nodes. Since the guarantee provided by FTSP is not that the global time will have a specific value, but rather that the network will synchronize to *some* global time, small offsets from the root local time do not compromise the operation of the FTSP network. If necessary, offsets from the root time can be avoided turning on the intended root node at least one beacon period before the other nodes in the network.

2) *Limit on FTSP Network Size:* During development of our model, we discovered that the range of possible sequence numbers places a limit on the size of an FTSP network. We found that for a sequence number range  $[0, \text{MaxSequenceNumber}]$ , an FTSP network will only converge to a single root if

$$\text{MaxSequenceNumber} > 2 \text{NetworkRadius}.$$

Networks that fail to meet this condition appear to converge to a single root and global time within a radius corresponding to their maximum sequence number, but are not guaranteed to produce synchronized nodes outside that radius.

The cause of the synchronization failure is the finite range of sequence numbers, which forces eventual repetition of sequence numbers. Although FTSP takes steps to ensure that sequence number repetition does not cause problems, if the sequence number range is too small conditions can still arise in which new and old sequence numbers cannot be distinguished, and a new message will be incorrectly ignored. For example, in a 3-node, radius-2 network using a  $[0, 3]$  sequence number range, the sequence of events shown in Listing 6 leads to problems. In the first round, node 1 transmits (line 2) before it has received an incremented sequence number from the root (line 3), leaving node 2 with a sequence number of 1 (line 4). In the second round, the root transmits the incremented sequence number to node 1 (line 6), which then relays that number to node 2 (line 7). As a result, node 2 observes a jump in the sequence number from 1 to 3. However, the value of `newer(1, 3)` (Listing 4) is `False` for a sequence number range  $[0, 3]$ , so node 2 treats the message as one containing old data (line 8).

To ensure that synchronization occurs, the range of sequence numbers must be large enough to allow the `newer` function to identify new messages even when the sequence number jumps by an amount equal to the number of hops

Listing 6. Trace showing a node ignoring a new message

```

1 ...
2 broadcast.1.0.1.time_t.0
3 broadcast.0.0.2.time_t.0
4 broadcast.2.0.1.time_t.0
5 tock
6 broadcast.0.0.3.time_t.0
7 broadcast.1.0.3.time_t.0
8 broadcast.2.0.1.time_t.0
9 ...

```

from the root to the network edge. This requires that the maximum sequence number exceed twice the network radius. The current FTSP implementation uses 8-bit sequence numbers, giving a sequence number range  $[0, 255]$  and a corresponding network size limit of  $NetworkRadius \leq 127$ .

3) *Execution Time*: We ran FDR2 on two computers, a 1.5 GHz PowerPC with 1 GB of RAM, and a 2.4 GHz Intel Core 2 Duo with 2 GB of RAM. On the PowerPC we checked all five assertions for a 2-node network in 65 seconds, and for a 4-node network in 31 minutes. On the Core 2 Duo the execution times for checking all five assertions were:

- 2-node network: 10 seconds
- 4-node network: 4 minutes
- 6-node network: 37 minutes
- 7-node network:  $\sim 2$  hours

For a 2-node network FDR2 reports checking 184 states with 241 transitions, while a 7-node network involves checking 6,575,314 states with 17,629,537 transitions.

## V. RELATED WORK

Leader election is a classic problem in distributed systems design. It has been studied by numerous of researchers, and several leader election protocols have been the subject of formal proofs or model-checking. For example: Lynch's text on distributed algorithms [8] collects a variety of leader election protocols for different kinds of networks, and provides proofs of their effectiveness; a model of the Dolev/Klawe/Rodeh protocol for leader election in unidirectional rings is included as an example with the SPIN model checker [9]; Romijn [10] used SPIN and Xtl to model-check the HAVi leader-election protocol. Similarly, time synchronization protocols in general [11], and time synchronization protocols for sensor networks in particular [12], have been widely studied. However, to our knowledge the only work on formal verification of any aspect of FTSP is the work by Kusy and Abdelwahed [4] on model-checking FTSP root election using SPIN.

Kusy and Abdelwahed's study of FTSP root election was the original inspiration for the work presented here. Their model-checking analysis showed that a 2-node FTSP network is guaranteed to converge to a single root node. However, they encountered difficulties with state-explosion for networks larger than 2-nodes, and were unable to verify root-convergence for those larger networks. Kusy and Abdelwahed's model-checking was carried out on a 2.6 GHz Intel Pentium IV with 512 MB of RAM. Verification of root-convergence for their 2-node model involved checking on the order of  $10^7$  states and  $2 \times 10^7$  transitions, and took around 15 minutes. The model-checking reported in this paper was carried out on both a 1.5 GHz PowerPC with 1 GB of RAM, and a 2.4 GHz Intel Core 2 Duo with 2 GB of RAM. Verification of root-convergence for our 2-node model involved checking 184 states and 241 transitions, and took under 1 minute even on the slower processor. The substantially smaller state-space of our FTSP model makes it feasible to check root-convergence for networks larger than 2 nodes, and also to extend the model to allow checks of previously unanalyzed time-convergence properties.

## VI. CONCLUSIONS

We modeled the Flooding Time Synchronization Protocol in CSP, using various abstractions to restrict the size of the model state-space. In particular, we used bounded abstractions of key variables to accurately model FTSP behavior while requiring far fewer distinct states than previous models. The resulting reductions in state-space size enabled us to check models of much larger FTSP networks than have previously been analyzed, and to study properties of FTSP that have not previously been examined. Although the maximum model size of 7 nodes is smaller than most FTSP deployments, it is sufficient to allow investigation of the behavior of multi-hop topologies. The model is available at <http://coweb.elec.canterbury.ac.nz/cda/uploads/ftsp.csp>.

Our results show that the FTSP network configurations we checked correctly converge to a single root node, and agree on a global time. Our results also show that there is a limit on the size of correctly functioning FTSP networks proportional to the size of the sequence number datatype. Although our results are not a proof that FTSP works for all topologies, they do provide increased confidence that FTSP works as intended, and more insight into FTSP behavior.

Our FTSP model assumes that links and nodes do not fail. We have begun work on a model that incorporates failures in order to explore the fault-tolerance of FTSP networks. Our model also assumes that all nodes are activated within the same beacon period. A more general model would allow nodes to join the network at arbitrary times.

## REFERENCES

- [1] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi, "The flooding time synchronization protocol," in *Proc. of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys'04)*. ACM, 2004, pp. 39–49.
- [2] B. Kusy, "FTSP - TinyOS documentation wiki," August 2008. [Online]. Available: <http://docs.tinyos.net/index.php/FTSP>
- [3] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and yield in a volcano monitoring sensor network," in *Proc. of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. Berkeley, CA, USA: USENIX Assoc., 2006, pp. 381–396.
- [4] B. Kusy and S. Abdelwahed, "FTSP protocol verification using SPIN," Institute for Software Integrated Systems, ISIS technical report ISIS-06-704, May 2006. [Online]. Available: [http://www.isis.vanderbilt.edu/sites/default/files/Abdelwahed\\_S.5.0\\_2006\\_FTSP\\_Proto.pdf](http://www.isis.vanderbilt.edu/sites/default/files/Abdelwahed_S.5.0_2006_FTSP_Proto.pdf)
- [5] A. W. Roscoe, *The Theory and Practice of Concurrency*. Englewood Cliffs, NJ, USA: Prentice Hall, 1998.
- [6] A. I. McInnes, "Using CSP to model and analyze TinyOS applications," in *Proc. of the 16th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'09)*. IEEE Computer Society, April 2009, pp. 79–88.
- [7] P. Gardiner *et al.*, *Failures-Divergences Refinement: FDR2 User Manual*, Formal Systems (Europe) Ltd, 2005.
- [8] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.
- [9] G. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Reading, MA, USA: Addison-Wesley, 2003.
- [10] J. M. Romijn, "Model checking the HAVi leader election protocol," CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, Tech. Rep., 1999.
- [11] U. Schmid, "An annotated bibliography on clock synchronization in distributed systems," Technische Universität Wien, Dept. of Automation, Technical Report TR 183/1-45, December 1994.
- [12] K. Römer, P. Blum, and L. Meier, "Time synchronization and calibration in wireless sensor networks," in *Handbook of Sensor Networks: Algorithms and Architectures*, I. Stojmenovic, Ed. John Wiley & Sons, 2005, pp. 199–237.