MODEL CHECKING A SELF-STABILIZING SYNCHRONIZATION PROTOCOL FOR ARBITRARY DIGRAPHS

Mahyar R. Malekpour, Langley Research Center, Hampton, Virginia, USA

mahyar.r.malekpour@nasa.gov

Abstract

This report presents the mechanical verification of a self-stabilizing distributed clock synchronization protocol for arbitrary digraphs in the absence of faults. This protocol does not rely on assumptions about the initial state of the system, other than the presence of at least one node, and no central clock or a centrally generated signal, pulse, or message is used. The system under study is an arbitrary, non-partitioned digraph ranging from fully connected to 1-connected networks of nodes while allowing for differences in the network elements. Nodes are anonymous, i.e., they do not have unique identities. There is no theoretical limit on the maximum number of participating nodes. The only constraint on the behavior of the node is that the interactions with other nodes are restricted to defined links and interfaces. This protocol deterministically converges within a time bound that is a linear function of the self-stabilization period. A bounded model of the protocol is verified using the Symbolic Model Verifier (SMV) for a subset of digraphs. Modeling challenges of the protocol and the system are addressed. The model checking effort is focused on verifying correctness of the bounded model of the protocol as well as confirmation of claims of determinism and linear convergence with respect to the self-stabilization period.

1. Introduction

Synchronization algorithms are essential for managing the use of resources and controlling communication in a distributed system. **Synchronization** of a distributed system is the process of **achieving** and **maintaining** a bounded skew among independent local clocks. A distributed system is said to be self-stabilizing if, from an arbitrary state, it is guaranteed to reach a legitimate state in a finite amount of time and remain in a legitimate state. A legitimate state is a state where all parts in the system are in synchrony. The self-stabilizing distributed-system clock synchronization problem is, therefore, to develop an algorithm (i.e., a protocol) to *achieve* and *maintain* synchrony of local clocks in a distributed system after experiencing system-wide disruptions in the presence of network element imperfections. The **convergence** and **closure** properties address achieving and maintaining network synchrony, respectively. Hereafter in this report, we use the term synchronization to mean self-stabilizing clock synchronization in distributed systems.

A thorough understanding of the synchronization of a distributed system has proven to be elusive for decades. The main challenges associated with distributed synchronization are the complexity of developing a solution and proving the correctness of the solution. The proposed solution must restore synchrony and coordinated operations after experiencing system-wide disruptions in the presence of network element imperfections and, for ultra-reliable distributed systems, in the presence of various faults. A fault is a defect or flaw in a system component resulting in an incorrect state [1][2][3]. Also, addressing network element imperfections is necessary to make a solution applicable to realizable systems. In addition, a proposed solution must be proven to be correct. In the absence of a paper-and-pencil proof, the use of mechanized formal method techniques is a viable alternative.

In [4] a solution is presented for an arbitrary network (digraph) in the absence of faults. The system under study is an arbitrary, non-partitioned digraph ranging from fully connected to 1-connected networks of nodes while allowing for differences in the network elements. This solution does not require any particular information flow nor imposes changes (e.g., embedding a directed spanning tree or rewiring) to the network in order to achieve synchrony. The assumption of an absence of faults is equivalent to the assumption that all faults are detectable. This departure from our previous work at the Byzantine extreme of the fault spectrum [5] is in part because of the niche use and the extra cost associated with the Byzantine faults. Also, using authentication and error detection techniques, it is possible to substantially reduce the effects of variety of faults in the system. Furthermore, the classical definition of a self-stabilizing algorithm assumes generally that there are no faults in the system.

In this report we present model checking efforts in support of the claims of [4]. In particular, this effort encompasses the verification of correctness of a bounded model of the protocol by confirming that a set of candidate systems self-stabilizes from any state. This effort, furthermore, includes the verification of claims of determinism and linear convergence of the bounded model of the protocol with respect to the self-stabilization period. Toward this objective, a number of abstractions and reduction techniques are devised to reduce the state space. The model checking results of the bounded model of the protocol have validated the correctness of the protocol as they apply to the networks with unidirectional and bidirectional links. In addition, the results have confirmed the claims of determinism and linear convergence.

The following sections describe the model checking efforts in detail. In Section 2 we provide a system overview. We present the protocol and its description in Section 3. Modeling specifications and abstractions used in describing a bounded model of this protocol are described in Section 4, where the underlying topology and network models are defined. In Section 5 we enumerate the propositions used and, finally, in Section 6, we present a summary of the model checking results and concluding remarks.

2. System Overview

We consider a system of pulse-coupled entities (e.g., oscillators, pacemaker cells) pulsating periodically at regular time intervals. We model the system as a set of nodes that represent the pulse-coupled entities and a set of communication links that represent their interconnectivity. The underlying topology considered here is a network of $K \geq 1$ nodes that exchange messages through a set of communication links. Nodes are anonymous, i.e., they do not have unique identities. All nodes are assumed to be good, i.e., actively participate in the synchronization process and correctly execute the protocol. The communication links are assumed to connect a set of source nodes to a set of destination nodes with a source node being different than a destination node. All communication links are assumed to be good, i.e., reliably transfer data from their source nodes to their destination nodes. The nodes communicate with each other by exchanging broadcast messages. Broadcast of a message by a node is realized by transmitting the message, at the same time, to all nodes that are directly connected to it. The communication network need not guarantee any relative order of arrival of a broadcast message at the receiving nodes. There is neither a central system clock nor an externally generated global pulse or message at the network level. The communication links and nodes can behave arbitrarily provided that eventually the system adheres to the protocol assumptions (Section 3.4).

2.1. *Drift Rate (ρ) And The Logical Clock (LocalTimer)*

Each node is driven by an independent, free-running local physical oscillator (i.e., the phase is not controlled in any way) and a logical-time clock (i.e., a counter), denoted *LocalTimer*, which locally keeps track of the passage of time and is driven by the local physical oscillator. An **oscillator tick**, also called a **clock tick**, is a discrete value and the basic unit of time in the network. An ideal oscillator has zero drift rate with respect to real-time, perfectly marking the passage of time. Real oscillators are characterized by non-zero drift rates with respect to real-time. The oscillators of the nodes are assumed to have a known bounded drift rate, ρ, which is a small constant with respect to real-time, where ρ is a unitless non-negative real value and is expressed as $0 \leq \rho << 1$. The maximum drift of the fastest *LocalTimer* over a time interval of t is given by $(1+\rho)t$. The maximum drift of the slowest *LocalTimer* over a time interval of t is given by $(1/(1+\rho))t$. Therefore, the **maximum relative drift** of the fastest and slowest nodes with respect to each other over a time interval of t is given by $\delta(t) = ((1+\rho) - 1/(1+\rho))t$.

2.2. *Communication Delay (D), Network Imprecision (d), And γ*

The communication latency between the nodes is expressed in terms of the minimum event-response delay, D, and network imprecision, d. These parameters have units of real time clock ticks. A message transmitted at real time t_0 is expected to arrive at all destination nodes, be processed, and subsequent messages are generated within the time interval of $[t_0+D, t_0+D+d]$. Communication between independently clocked nodes is inherently imprecise. The network imprecision, d, is the maximum time difference among all receivers of a message from a transmitting node with respect to real time. These two parameters are assumed to be bounded such that $D \geq 1$ and $d \geq 0$ and both have discrete values with units of real time clock tick. The communication latency, denoted γ, is expressed in terms of D and d, and is constrained by $\gamma = (D+d)$ and so has units of real time clock ticks.

2.3. *Topology (T)*

A communication link, or simply link, is an edge in the graph representing a direct physical connection between two nodes. A path is a logical connection between two nodes consisting of one or more links. A path-length is the number of links connecting any two nodes. The general topology, T, considered is a strongly connected directed graph (digraph) consisting of K nodes, where each node is connected to the graph by at least one link, there is a path from any node to any other node, and the links are either unidirectional or bidirectional. Furthermore, we assume there is no direct link from a node to itself, i.e., no self-loop, and there are no multiple links directly connecting any two nodes in any one direction.

We use the terms network and graph interchangeably. The following graph specific terms are used in the subsequent sections.

- L, an integer value, is the number of links denoting the largest **loop** in the graph, i.e., the maximum value of the longest path-lengths from a node back to itself visiting the nodes along the path only once (except for the first node which is also the last node).
- W, an integer value, is the number of links signifying the **width** or diameter of the graph, i.e., the maximum value of the shortest path connecting any two nodes.

For digraphs of size $K > 1$, L and W are bounded by $2 \leq L \leq K$ and $1 \leq W \leq K - 1$.

3. The Protocol

In this section we enumerate protocol assumptions, properties, parameters, and describe the protocol in pseudo-code. The general form of the distributed synchronization problem, S, is defined by the following septuple [4].

$S = (K, T, D, d, \rho, P, F)$

In other words, the distributed synchronization problem is a function of the number of nodes (K), network topology (T), communication delay (D), communication imprecision (d), oscillator drift rate (ρ), synchronization period (P), and number of faults (F), respectively. The solution to this problem is a protocol with convergence and closure properties, at a minimum, as discussed subsequently in this section. However, in this protocol we do not deal with faults.

Each node is driven by an independent logical-time clock, i.e., *LocalTimer*. The clocks need to be periodically synchronized due to their inherent drift with respect to each other. In order to achieve synchronization, the nodes communicate by exchanging **Sync** messages. The periodic synchronization after achieving the initial synchrony is referred to as the **resynchronization process** whereby all nodes reengage in the synchronization process. A node is said to **time-out** when its *LocalTimer* reaches its maximum value. The resynchronization process begins when the first node (fastest node) times-out and transmits a *Sync* message and ends after the last node (slowest node) transmits a *Sync* message. For $\rho << 1$, the fastest node cannot time-out again before the slowest node transmits a *Sync* message [4].

A node consists of a **synchronizer** and a set of **monitors**. A *Sync* message is transmitted either as a result of a resynchronization timeout, or when a node receives *Sync* message(s) indicative of other nodes engaging in the resynchronization process. The messages to be delivered to the destination nodes are deposited on communication links. Although the network level measurements are real values, locally and at the node level, all protocol parameters have discrete values with the time-based terms having units of real time clock ticks. The discretization is for practical purposes in implementing and model checking of the protocol. The following definitions and terms are used in the description and operation of the protocol.

- The **resynchronization period**, denoted P, has units of real time clock ticks and is defined as the upper bound on the time interval between any two consecutive resets of the *LocalTimer* by a node.
- **Drift per *t***, denoted $\delta(t)$, has units of real time clock ticks and is defined as the maximum amount of drift between any two nodes for the duration of t, $\delta(t) \geq 0$. In particular:
  - Drift per D, denoted $\delta(D)$, for the duration of one D, $\delta(D) \geq 0$.
  - Drift per γ, denoted $\delta(\gamma)$, for the duration of one γ, $\delta(\gamma) \geq 0$.
  - Drift per P, denoted $\delta(P)$, for the duration of one period P, $\delta(P) \geq 0$.
- The **graph threshold**, T_S, is based on a specified graph topology and has units of real time clock ticks (see Section 3.1).
- The guaranteed precision or simply **precision** of the network, π, $0 \leq \pi < P$, has units of real time clock ticks and is defined as the guaranteed achievable precision among all nodes.
- The **convergence time**, denoted C, has units of real time clock ticks and is defined as the bound on the maximum time it takes for the network to converge, i.e., to achieve synchrony.
- **Precision between *LocalTimers*** of any two adjacent nodes N_i and N_j denoted by Δ_{ij} and has units of real time clock ticks.
- The **initial synchrony** is a state of the network and the earliest time when the precision among all nodes, upon convergence, is within π. The initial synchrony occurs at time C_{Init}.
- The **initial precision** among *LocalTimers* of all nodes, Δ_{Init}, has units of real time clock ticks and, for all $t \geq C_{Init}$, is defined as a measure of the precision of the network immediately after a resynchronization process.
- The **initial guaranteed precision** among *LocalTimers* of all nodes, $\Delta_{InitGuaranteed}$, has units of real time clock ticks and, for all $t \geq C$, is defined as a measure of the precision of the network immediately after a resynchronization process.

3.1. *The Graph Threshold (T_S)*

When a node receives a *Sync* message, except during a predefined window, referred to as the **ignore window**, it accepts the *Sync* message and undergoes the resynchronization process where it resets its *LocalTimer* and relays the *Sync* message to others. The ignore window provides a means for the protocol to stop the endless cycle of resynchronization processes triggered by the follow up *Sync* messages. We bound the ignore window to [D, T_S). The lower bound is due to the minimum event-response delay, D, and the upper bound, referred to as the graph threshold, T_S, is a function of a specified graph topology and the maximum delay for a *Sync* message to return to the originating node after traversing the graph.

3.2. *Sync Message And Its Validity*

In order to achieve synchrony, the nodes communicate by exchanging *Sync* messages[1]. When the system is in synchrony, the protocol overhead is at most one message per resynchronization period P. Assuming physical-layer error detections are dealt with separately, the reception of a *Sync* message is indicative of its validity in the value domain. The protocol performs as intended when the timing requirements of the messages from every node are satisfied. However, in the absence of faults, the reception of a *Sync* message is indicative of its validity in the value and time domains. A valid *Sync* message is discarded after it is relayed to the synchronizer and has been kept for one local clock tick.

3.3. *The Monitor, The Synchronizer, And Protocol Functions*

A node consists of a **synchronizer** and a set of **monitors**. To assess the behavior of other nodes, a node employs as many monitors as the number of nodes it is directly connected to with one monitor for each source of incoming messages. A node neither uses nor monitors its own messages. A monitor

[1] Since only one message type is used for the operation of this protocol, a single bit suffices.

keeps track of the activities of its corresponding source node. Specifically, a monitor reads, evaluates, validates, and stores the last valid message it receives from that node. Upon conveying the valid message to the local synchronizer, a monitor disposes of the valid message after it has been kept for one local clock tick. The functions *ValidateMessage()* and *ConsumeMessage()*, Figure 1, are used by the monitors. The function *ValidSync()* is used by the synchronizer.

```
ValidateMessage():
if (incoming message = Sync) then
    {Message is valid, Store it.}
ConsumeMessage():
if (stored message timer ≥ 1 tick) then
    {Message is invalid, Clear it.}

ValidSync():
if (number of stored messages > 0) then
    return true,
else
    return false.
```

Figure 1. The protocol functions.

3.4. Protocol Assumptions

The following are protocol assumptions. 1) $K \geq 1$. 2) All nodes correctly execute the protocol. 3) All links correctly transmit data from their sources to their destinations. 4) T is a non-partitioned, strongly connected digraph. 5) $0 \leq \rho << 1$. 6) A message sent by a node will be received and processed by all other nodes within γ, where $\gamma = (D + d)$ And 7) The initial values of the variables of a node are within their corresponding data-type range, although possibly with arbitrary values.

3.5. The Self-Stabilizing Distributed Clock Synchronization Problem

To simplify the presentation of this protocol, it is assumed that all time references are with respect to an initial real time t_0, where $t_0 = 0$, and for all $t \geq t_0$ the system operates within the *protocol assumptions*. The maximum difference in the value of *LocalTimer* for all pairs of nodes at time t, $\Delta_{Net}(t)$, is determined by the following equation that accounts for the variations in the values of the *LocalTimer* across all nodes.

$r = \lceil (W + 1)(\gamma + \delta(\gamma)) \rceil$,
$LocalTimer_{min}(x) = min\ (N_i.LocalTimer(x))$, and
$LocalTimer_{max}(x) = max\ (N_i.LocalTimer(x))$, for all i.

$$\Delta_{Net}(t) = min\ ((LocalTimer_{max}(t) - LocalTimer_{min}(t)),\ (LocalTimer_{max}(t - r) - LocalTimer_{min}(t - r))).$$

The following symbols were defined earlier and are listed here for reference:

- P denotes the resynchronization period, $P > 0$.
- C denotes a bound on the maximum convergence time,
- $\Delta_{Net}(t)$, for real time t, is the maximum difference of values of the *LocalTimers* of any two nodes (i.e., the relative clock skew) for $t \geq t_0$, and
- π, the synchronization precision, is the guaranteed upper bound on $\Delta_{Net}(t)$, for all $t \geq C$.

To show that a protocol is self-stabilizing, it has to be proven that there exist C and π such that the following self-stabilization properties hold.

1.	**Convergence:**	$\Delta_{Net}(C) \leq \pi$, $0 \leq \pi < P$
2.	**Closure:**	For all $t \geq C$, $\Delta_{Net}(t) \leq \pi$
3.	**Congruence:**	For all nodes N_i, for all $t \geq C$, $(N_i.LocalTimer(t) = \gamma)$ implies $\Delta_{Net}(t) \leq \pi$.
4.	**Liveness:**	For all $t \geq C$, *LocalTimer* of every node sequentially takes on at least all integer values in $[\gamma, P - \pi]$.

3.6. The Self-Stabilizing Distributed Clock Synchronization Protocol For Arbitrary Digraphs

The protocol, executed by all nodes, is presented in Figure 2 and consists of a synchronizer and a set of monitors which execute once every local clock tick.

```
Synchronizer:
E1: if (ValidSync() and (LocalTimer < D))
        LocalTimer := $\gamma$,
E2: elseif ((ValidSync() and (LocalTimer $\geq T_S$))
        LocalTimer := $\gamma$,
        Transmit Sync,
E3: elseif (LocalTimer $\geq P$)      // time-out
        LocalTimer := 0,
        Transmit Sync,
E4: else
        LocalTimer := LocalTimer + 1.

Monitor:
case (message from the corresponding node)
{Sync:
            ValidateMessage()
      Other:
            Do nothing.
} // case
ConsumeMessage()
```

Figure 2. The self-stabilizing clock synchronization protocol for arbitrary digraphs.

The following is a list of protocol parameters when all links are bidirectional.

$T_S \geq (L+2)(\gamma + \delta(\gamma))$
$P \geq 3T_S$, for $\rho = 0$
$P \geq 3(T_S + \delta(T_S))$, for $L = K$ and $\rho > 0$
$P \geq \max((2K + 1)(\gamma + \delta(\gamma)), 3(T_S + \delta(T_S)))$, for $L = f(T)$ and $\rho > 0$

The following is a list of protocol parameters for digraphs, i.e., when at least one link is unidirectional.

$T_S \geq (K+2)(\gamma + \delta(\gamma))$
$P \geq K(T_S + \delta(T_S))$

Regardless of the types of links in the network, the following is a list of protocol measures.

$C_{Init} = 2P + K(\gamma + \delta(\gamma))$
$\Delta_{Init} \leq (K - 1)(\gamma + \delta(\gamma))$
$C = C_{Init} + \lceil \Delta_{Init} / \gamma \rceil P$
$Wd \leq \Delta_{InitGuaranteed} \leq W(\gamma + \delta(\gamma))$, for all $t \geq C$
$\pi = \Delta_{InitGuaranteed} + \delta(P) \geq 0$, for all $t \geq C$, and $0 \leq \pi < P$

A trivial solution is when $P = 0$. Since $P > T_S$ and the *LocalTimer* is reset after reaching P (worst-case wraparound), a trivial solution is not possible.

4. Verification Model

There are two general formal methods approaches for the verification of the correctness of a protocol: **theorem proving** and **model checking**. Verification via theorem proving requires a deductive proof of the protocol. Verification via model checking is based on specific scenarios and generally limited to a subset of the problem space. In this report we focus on the model checking approach for its ease, feasibility, and quick examination of a subset of the problem space while attempting a more comprehensive proof via theorem proving.

In this section, we present the details of the model checking efforts by describing models of the system components, their data structures, and the modeling simplification and abstractions techniques employed in the mechanical verification of the protocol. The Symbolic Model Verifier (SMV) was used in modeling of this protocol on a PC with 4GB of memory running Linux [6]. SMV's language description and modeling capability provide relatively easy translation from the pseudo-code. SMV semantics is synchronous composition, where all assignments are executed in parallel and synchronously. Thus, a single step of the resulting model corresponds to a step in each of the components.

A matter of concern in model checking is the ease of encoding the algorithm and assumed environment in the language of the model checker. In model checking, the state explosion, i.e., the time and space required to run the model checker, grows rapidly and eventually becomes infeasible as the size and complexity of the model grows. Thus, abstraction must be employed with respect to the size of the model and real-time delays. The algorithm described in this report is fairly subtle and must cope with many kinds of timing behaviors. Model checking has been used to explore and verify distributed algorithms but faces certain difficulties [7][8][9][10]. One of the foremost challenges is a realistic representation of time as a continuous variable.

As we elaborated earlier in this report, although the network level measurements are real values, locally and at the node level, all parameters are discrete. Since continuous time model is impracticable, we looked for an abstraction employing discrete time. Also, although we cannot yet prove the soundness of this abstraction, our decision to use a discrete model for time was critical to our ability to undertake this verification effort.

4.1. *Modeling Communication Links*

An explicit model of the communication link requires a separate entity (SMV module) with its own local memory, at a minimum, to store and forward a message. This approach would readily exhaust the available 4GB memory even for small values of K and render the model checking effort ineffective. To reduce state space, links are implicitly modeled and the outgoing message is kept within the transmitting node long enough for the receiving nodes to sample it.

4.2. *Modeling Monitors*

A monitor keeps track of activities of its corresponding source node and manages message validity. Recall that we assume physical-layer error detections are dealt with separately and so, receiving a *Sync* message is indicative of its validity in the value and time domains. In other words, we analyze the system at the point where the valid messages arrive at the *Synchronizer* of the node. Since we assume no faulty nodes are present, an explicit model of the monitors is not necessary. Instead, and to reduce the state space, monitors are implicitly modeled at the receiving nodes.

4.3. *Modeling Nodes*

The synchronizer describes the collective behavior of the node utilizing assessment results from its monitors. The local measures within each node are used to keep track of timing of the self-stabilization events. Although the protocol parameters are defined with respect to real time, ultimately, in implementations they have to be translated into discrete values. Discretization of the protocol parameters is performed using the ceiling operation. In this protocol, all local variables and watchdog timers are discretized and represented by integer values. These local variables are, therefore, measured with respect to the local clock.

A parameterized node, *NodeType*, is introduced that executes the protocol and consists of local variables. The *NodeType*'s data structure consists of *Monitors*, *Synchronizer*, and *MessageOut*. The *Synchronizer* in turn consists of *LocalTimer* which represents the duration of time since the node has gone through the resynchronization process. The *MessageOut* element represents the out going message of the node. The range of values that these elements can hold are as follows.

LocalTimer	$= \{0 .. P\}$
MessageOut	$= \{NONE, Sync\}$

In the SMV implementation, the parameters T_S and P are customized for each node and are passed on to the node as input parameters (Section 4.6). The set of unidirectional inputs/outputs links of the *NodeType* module in SMV, $InputMessages_j/OutputMessages_h$, specify the input/output links and source/destination of the messages, respectively. Together, they define the network topology. Because of the message validity assumptions and implicit model of the monitors, the related protocol functions are implemented at the *NodeType*. These functions examine the number of available messages at the transmitting node utilizing implicit model of the communication links. The function *ValidSync()* is an *or* operation over the set of input messages to node N_i.

$$ValidSync() = OR\ (Node_j.MessageOut),\ i \neq j$$

4.4. *Modeling Communication Delays*

Since we have assumed absence of malicious faulty nodes, the nodes react to each other's messages within γ and the minimum event-response delay, D, and the network imprecision, d, do not play distinctive roles in the synchronization process. In other words, the effects of D and d in the synchronization process are incorporated in γ. This assertion is not true in the presence of malicious faulty nodes. These parameters, however, directly contribute to the guaranteed precision of the network.

An explicit model of D and d requires more memory to store and delay a message both in the node and the communication link modules. These explicit models would exponentially increase state space. Recall that all system parameters are discretized to local ticks. Therefore, an increase of one local tick in the communication delays directly increases the value of all other timing parameters. As a result, this approach would readily exhaust the available 4GB memory even for small values of K and render the model checking effort ineffective. To further minimize state space, D and d are chosen to be at their minimum values of 1 and 0 clock ticks, respectively. As a result, γ is at its minimum value of 1 clock tick. This simplification, consequently, implies that the local oscillators of the nodes are in phase with each other but it does not imply that the nodes are synchronized with each other.

4.5. Modeling Clocks and Timers

Each node has a logical clock, *LocalTimer*, that locally keeps track of time. This logical clock is used in measuring the self-stabilization precision, π, across the nodes from an external view of the system. A single clock per node suffices to advance a nodes's *LocalTimer*. Since $\gamma = 1$ clock tick, a single clock suffices to advance all *LocalTimers*. To further minimize the state space, all timers, *LocalTimers* and *GlobalClock* (Section 4.7), are incremented once per model checker cycle. The SMV cycle, therefore, binds the whole system together, providing a means for advancing the *GlobalClock* and the *LocalTimer* at the nodes and providing an external view of the system at any time. Although the use of SMV cycle, along with $\gamma = 1$ clock tick, does not imply synchrony at the nodes, it does imply that the nodes are in phase with each other at the local oscillator level. However, due to the inherent non-deterministic execution of a model in the model checker, the order of execution of the nodes is not predetermined. Since there is no control over the order of transmission of messages and the start of execution of the nodes at each model checker cycle, the nodes potentially broadcast and receive messages out of order of issuance.

4.6. Modeling Drift

In a realizable distributed system the clocks drift with respect to real time and each other. As a result, any viable solution has to account for the clock drift rate, ρ. An explicit model of ρ would require dealing with real values. Dealing either with real values or their equivalent integer values for ρ increases the state space drastically.

To reduce state space, we have employed the **implicit drift model** (IDM) as described in [11] to model ρ implicitly. In IDM approach, instead of explicitly specifying the drift rate for a node's local oscillator and determining the node's drift on a clock tick base, we determine the node's effective period based on the drift rate and pass the effective period to the node. Thus, each node will have its unique synchronization period with the proper amount of drift incorporated. In this approach the effective synchronization period is directly applied to the nodes with at least one node being the slowest and another the fastest in the system with their maximum relative drift being $\delta(P)$. One advantage of this modeling technique is that it drastically reduces state space. Another advantage is that when a node's behavior is not influenced by the behavior of other nodes for duration of time, the model checking time can advance to the end of that time interval[2]. Thus, the IDM substantially improves the model checking performance.

We apply the IDM approach to all parameters that are based on time including γ, T_S, and P. The amount of drift applied to a particular parameter is linearly proportional to its value. Since typically $\rho << 1$ and γ is very small, the effect of ρ during γ is negligible, i.e., $\delta(\gamma) = 0$. Also, since all parameters are locally defined as integers, we set T_S and P to large enough values, beyond their minimum values, to guarantee proportional presence of the effect of drift in T_S and P in the nodes.

As mentioned earlier, the use of SMV cycle, along with $\gamma = 1$ clock tick, imply that the nodes are in phase with each other at the local oscillator level. However, applying the IDM implies that the nodes are out of phase with each other at the *LocalTimer* level. Due to the inherent non-deterministic execution of a model in the model checker, the order of execution of the nodes is not predetermined, there is no control over the order of transmission of messages and the start of execution of the nodes at each model checker cycle, thus, the nodes potentially broadcast and receive messages out of order of issuance. As a result, we believe our modeling techniques and abstractions properly capture the intended properties of a realizable system.

4.7. Modeling Network

Model checking is conducted on a given network consisting of a set of nodes that are instances

[2] The concept of advancing time has been used in hardware description language (e.g., VHDL and Verilog) simulation tools for decades.

of the *NodeType* and are interconnected to reflect a desired topology. A single step of the resulting model corresponds to a step in each of the components. A global clock, *GlobalClock*, is introduced to measure passage of time from the beginning of the operation and with respect to the real time and from the perspective of an external observer. The *GlobalClock* is used to measure the convergence time, *C*, and is incremented once per model checker cycle. The synchronization properties are examined at the network level and provide an external view of the system. The properties examined to verify the claims of the protocol are described in Section 5.

5. Propositions

Computational tree logic (CTL), a temporal logic, is used to express properties of a system in this context. In CTL formulas are composed of **path quantifiers**, *E* and *A*, and **temporal operators**, *X*, *F*, *G*, and *U* [12]. In this section the claims of convergence, closure, and congruence properties as well as the claims of maximum convergence time and determinism of the protocol are examined. Although in the description of the protocol convergence and closure properties are stated separately, they are examined via one CTL proposition. This proposition also expresses the claims of determinism and linear convergence. Validation of this general CTL proposition requires examination of a number of underlying propositions. In particular, since $\Delta_{LocalTimer}(t)$ is defined in terms of the *LocalTimer* of the nodes, examination of the properties that described proper behavior of the *LocalTimer* take precedence. The variable *ElapsedTime* is used in these properties and is defined here.

ElapsedTime = (GlobalClock ≥ ConvergenceTime) ;

The *GlobalClock* is a measure of elapsed time from the beginning of the operation and with respect to the real time, i.e., external view. The *ElapsedTime* is indicative of the *GlobalClock* reaching its target maximum value of *ConvergenceTime*.

Proposition *SystemLiveness*: This property addresses the liveness property of the system by examining whether or not time advances and the amount of time elapsed, *ElapsedTime*, has advanced beyond the predicted convergence time, *ConvergenceTime*.

```
AF (ElapsedTime)
```

Proposition *ConvergenceAndClosure*: This proposition encompasses the criteria for the convergence and the closure properties as well as the claims of maximum convergence time and determinism. This proposition specifies whether or not the system will converge to the predicted precision after the elapse of convergence time, *ElapsedTime*, and whether or not it will remain within that precision thereafter. This and subsequent properties are expected to hold.

The proper value of the *AllWithinPrecision* is determined by measuring the difference of maximum and minimum values of the *LocalTimers* of all nodes for the current tick and in conjunction with the result from the previous $(W+1)\gamma$ ticks. The expected difference of *LocalTimers* is the predicted precision bound.

```
-- Determinism Property
AF (ElapsedTime) ∧
-- Convergence Property
AG (ElapsedTime  → AllWithinPrecision) ∧
-- Closure Property
AG ((ElapsedTime ∧ AllWithinPrecision) →
      AX (ElapsedTime ∧ AllWithinPrecision))
```

To eliminate trivial results and false positives, the following proposition is examined and the expected result is a false value. This property specifies that after the elapse of convergence time, *ElapsedTime*, whether or not the system will not converge or if it converges, whether or not it drifts apart beyond the expected precision bound.

```
AF (ElapsedTime) ∧
AG (ElapsedTime  → AllWithinPrecision) ∧
AG ((ElapsedTime ∧ AllWithinPrecision) →
     EX (¬AllWithinPrecision))
```

Proposition *Congruence*: This property specifies the criteria for the congruence property of the protocol. Unlike the convergence and closure properties that provide system view from the perspective of an external viewer, the congruence property provides a local view from the perspective of a node by providing the necessary and sufficient conditions for the node to locally determine whether or not the system has converged. The congruence property is essential in integration of this underlying self-stabilization protocol with higher level protocols in the system. This property is described with respect to only one node, namely *Node_1*. Since all nodes are identical, due to symmetry, the result of the proposition equally applies to other nodes.

$$\begin{array}{l} AF\ (ElapsedTime) \wedge \\ AG\ ((ElapsedTime \wedge (Node\_1.LocalTimer = \gamma)) \\ \quad \rightarrow AX\ (ElapsedTime \wedge AllWithinPrecision)) \end{array}$$

Proposition *ProtocolLiveness*: This property specifies the criteria for the liveness property of the protocol. This property examines whether or not a node takes on all discrete values within an expected range. Since all nodes are identical, due to symmetry, this property is described with respect to only one node, namely *Node_1*.

$$\begin{array}{l} AF\ (ElapsedTime) \wedge \\ AG\ (((ElapsedTime) \wedge (Node\_1.LocalTimer = i)) \\ \quad \rightarrow AX\ ((Node\_1.LocalTimer = i) \mid \\ \qquad (Node\_1.LocalTimer = i+1))) \wedge \\ AG\ (((ElapsedTime) \wedge (Node\_1.LocalTimer = P)) \\ \quad \rightarrow AX\ (Node\_1.LocalTimer = 0)) \\ For\ all\ i = \gamma\ ..\ (P - \pi) \end{array}$$

6. Results And Conclusion

Since in the protocol we do not limit the size of the network, *K*, model checking of all possible digraphs for all *K*, even for idealized scenarios ($d = 0$, $\rho = 0$), is simply impossible. Model checking of all possible topologies for a given *K* is also a daunting task. Given the limited resources available and to circumvent state space explosion, we had to limit the network size. Nevertheless, to verify our claims of the correctness of the protocol, we have model checked all possible digraphs for smaller *K*. Additionally, we were able to model check some topologies for larger *K*. Table 1 is a list of the model checked networks with their sizes and corresponding number of topologies while bounding the drift to $0 \leq \rho \leq 0.2$. Each row of the table corresponds to a given *K* and two types of topologies considered with the number of model checked graphs of the possible total combinations for the corresponding topology type in its column. Sample SMV codes are available on my webpage.

Table 1. Model checked networks.

K	**Topology** (all links bidirectional)	**Topology** (digraphs)
2	1 of 1	1 of 1
3	2 of 2	5 of 5
4	6 of 6	83 of 83
5	21 of 21	Single Directed Ring, 2 Variations of Doubly Connected Directed Ring
6	112 of 112	-
7	Linear*	Linear*
7	Star*	Star*
7	Fully Connected*	Fully Connected*
7 (3×4)	Fully Connected Bipartite*	Fully Connected Bipartite*
7	Combo	4 of 4
7	Grid	-
7	Full Grid	-
9 (3×3)	Grid	-
15	Star*	Star*
20	Star*	Star*

* For Linear, Star, and Fully Connected (Complete/ Bipartite) the links are bidirectional.

A bounded model of *A Self-Stabilizing Distributed Clock Synchronization Protocol For Arbitrary Digraphs* is model checked using SMV where, for a set of digraphs, the entire state space is examined and verified to self-stabilize from an arbitrary state. This SMV model checking effort was performed on a PC with 4GB of memory running Linux. We described modeling concepts by abstracting the problem to discrete time and for realizable systems. The model checking results have confirmed the correctness of the protocol as they apply to the networks with unidirectional and bidirectional links as described earlier (Section 2.3). Also, the results indicate that the protocol is applicable to realizable systems and practical applications. In addition, the results confirmed the claims of determinism and linear convergence with respect to the synchronization period. Because of the model checking results, we conjecture that the protocol solves the general case of this problem for all $K \geq 1$ and is applicable to realizable systems and practical applications. Furthermore, this model checking effort has shown that, at a minimum, a deterministic solution for this problem exists.

References

[1] Girault, A.; Rutten, E.: *Modeling Fault-tolerant Distributed Systems for Discrete Controller Synthesis*, Electronic Notes in Theoretical Computer Science, vol. 133, pp. 81-100, 2005.

[2] Torres-Pomales, W; Malekpour, M.R.; Miner, P.S.: *ROBUS-2: A fault-tolerant broadcast communication system*, NASA/TM-2005-213540, March 2005.

[3] Butler, R.: *A primer on architectural level fault tolerance*, NASA/TM-2008-215108, February 2008.

[4] Malekpour, M.R.: *A Self-Stabilizing Synchronization Protocol For Arbitrary Digraphs*, in The 17th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), December 2011, to appear.

[5] Malekpour, M.R.: *A Byzantine-Fault Tolerant Self-Stabilizing Protocol for Distributed Clock Synchronization Systems*, Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS06), November 2006.

[6] http://www-2.cs.cmu.edu/~modelcheck/smv.html

[7] Steiner, W.; Rushby, J.; Sorea, M.; Pfeifer, H.: *Model Checking a Fault-Tolerant Startup Algorithm: From Design Exploration To Exhaustive Fault Simulation*, The International Conference on Dependable Systems and Networks (DSN'04), 2004.

[8] Lönn, H.; and Pettersson, P.: *Formal verification of a TDMA protocol start-up mechanism*, In Pacific Rim International Symposium on Fault-Tolerant Systems, pages 235–242, Taipei, Taiwan, Dec. 1997. IEEE Computer Society.

[9] Malekpour, M.R.: *Verification of a Byzantine-Fault-Tolerant Self-Stabilizing Protocol for Clock Synchronization*, IEEE Aerospace Conference, March 2008.

[10] Steiner, W.; Dutertre, B.: *Automated Formal Verification of the TTEthernet Synchronization Quality*, 3rd NASA Formal Method Symposium, April 2011.

[11] Malekpour, M.R.: *Model Checking A Self-Stabilizing Distributed Clock Synchronization Protocol for Arbitrary Digraphs*, NASA/TM-2011-217152.

[12] Clarke, E.M.; Emerson, E.A.: *Design and synthesis of synchronization skeletons using branching time temporal logic*, In Logic of Programs: Workshop, Yorktown Heights, NY, May 1981, LNCS 131. Springer, 1981.

31st Digital Avionics Systems Conference
October 14-18, 2012