Approximate Synchrony: An Abstraction for Distributed Time-Synchronized Systems

Ankush Desai
David Broman
John Eidson
Shaz Qadeer
Sanjit A. Seshia

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-136
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-136.html

June 30, 2014

Approximate Synchrony: An Abstraction for Distributed Time-Synchronized Systems

Ankush Desai, David Broman, John C. Eidson, and Sanjit A. Seshia
{ankush, broman, eidson, sseshia}@eecs.berkeley.edu
University of California, Berkeley

Shaz Qadeer
qadeer@microsoft.com
Microsoft Research

***Abstract*—Time synchronization plays a central role in the design of reliable distributed embedded systems. However, the clocks of nodes that are time-synchronized are only guaranteed to be equal within a certain tolerance. Thus, when modeling and verifying distributed protocols that involve or rely upon time synchronization, abstractions are needed that accurately capture the notion of systems being "almost synchronized." In this paper, we present the concept of approximate synchrony, a modeling and verification abstraction for time-synchronized systems. Approximate synchrony is a sound and tunable abstraction. We have implemented approximate synchrony as a part of a model checker and used it to verify the Best Master Clock (BMC) algorithm, the core component of IEEE 1588 precision time protocol and the time-synchronized channel hopping protocol that is part of the IEEE 802.15.4e standard.**

I. Introduction

Many distributed systems, especially in the cyber-physical systems domain, require coordination based on a common sense of time across all nodes [6]. Examples may be found in safety-critical aircraft and industrial control systems, and scientific applications such as the operation of the CERN particle accelerator. Traditionally, a common sense of time is established using systems or protocols such as the global positioning system (GPS), network time protocol (NTP), and the IEEE 1588 [15] precision time protocol (PTP) that synchronize the clocks of the nodes in the distributed system. These protocols are commonly referred to as *time-synchronization* or *clock-synchronization* protocols. It is important to note that all of these protocols do not guarantee that the time-synchronized nodes in the distributed system step synchronously. Instead, the guarantee is that the difference in the time values of the clocks of the various nodes of the system is bounded.

Formal modeling and verification of time-synchronized systems must accurately capture this notion of the nodes being "almost synchronized." In essence, each node has a clock that moves at a variable (but bounded) rate. At any time point, clocks of different nodes can have differing values, but time synchronization ensures that those values are within a specified offset of each other. Given this setting, one approach is to model hybrid systems [2], with continuous variables representing clocks. However, state-of-the-art hybrid verification tools, such as SpaceEx [12], do not scale when the discrete state space of the model is large. An alternative is to encode models of multirate time systems as timed automata, which can be verified in real-time model checking tool, such as UPPAAL [16]. Although there exist several techniques in the literature to model multirate systems as timed automata [9], [14], [19], all these approaches approximate real valued clocks as integer clocks. Therefore, it is natural to employ an alternative approach that uses suitable discrete abstractions of distributed systems. For instance, one approach is to use *full asynchrony*, assuming that all possible interleavings of actions at different nodes is possible. However, this abstraction retains no properties of time synchronization, and consequently can be too coarse to prove properties and too inefficient for state space exploration, as it considers too many spurious interleavings. There exist also several work on untimed abstractions for capturing asynchrony; for instance, multiclock Esterel [18], the quasi-synchronous approach [8], [13], and bounded asynchrony [11]. However, none of these abstraction techniques capture all aspects of the clock dynamics in a time-synchronized system.

In this paper, we introduce the notion of *approximate synchrony (*AS*)*, an abstraction technique for modeling time-synchronized distributed systems where components execute "almost synchronously." Based on parameters characterizing clock rates and offsets between clocks, we show how to compute a bound Δ such that the number of steps of any two components do not differ by more than Δ. As a consequence, we can create a purely discrete model of the system by composing components asynchronously under the approximate synchrony bound Δ. Since the bound Δ can vary from system to system, we term approximate synchrony a *tunable* abstraction. Additionally, we derive machine-checkable conditions for the soundness of this abstraction. We show through case studies that approximate synchrony is an effective abstraction method in practice: it greatly reduces the state space that a finite-state model checker must explore to prove desired safety and liveness properties, or find bugs.

We apply modeling and verification based on approximate synchrony to two practical case studies: (i) the best master clock (BMC) algorithm of the IEEE 1588 protocol, and (ii) a time-synchronized channel hopping protocol that is part of the IEEE802.15.4e [1] standard. The latter protocol requires the nodes to be time-synchronized *a priori* in order to operate correctly; we show how the approximate synchrony abstraction captures enough timing information to prove its correctness. On the other hand, the BMC algorithm, which is the first phase of the 1588 time synchronization protocol, does not itself require time synchronization for correctness. However, we show that we can use approximate synchrony along with the fact that time synchronization is achieved *a posteriori* to substantially speed up the verification of the BMC algorithm by model checking.

Our abstraction technique can be used with any finite-state model checker. However, in this paper we implement it on top of the ZING model checker [4]. One of the attractive features of ZING is the ability to control the model checker's search using an external scheduler that enforces the approximate synchrony condition. ZING comes with a modeling language called P [10], which provides an easy mechanism to specify protocols and facilitates not only verification, but also code generation from the formally-verified model.

To summarize, this paper makes the following contributions:

- A tunable abstraction technique, termed *approximate synchrony*, for formal modeling and verification of time-synchronized distributed systems (Sec. III and IV),
- A formalization of the conditions under which approximate synchrony is sound, along with a automatic procedure to

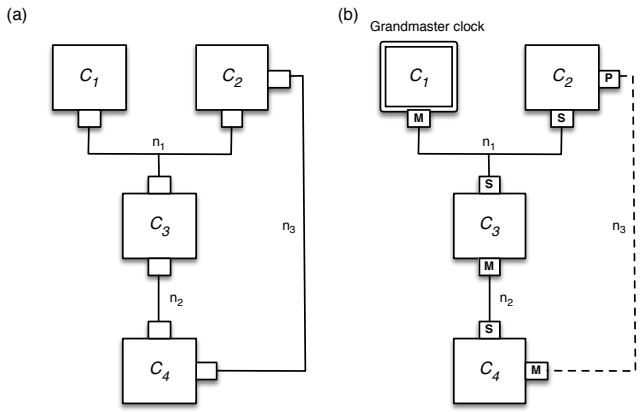

Fig. 1: Fig. (a) shows four clocks C_1, C_2, C_3, and C_4, connected using three networks n_1, n_2, and n_3. Fig. (b) depicts the resulting master-slave synchronization hierarchy after executing the BMC algorithm. The dashed line indicates that the link is not used in the spanning tree.

derive a sound value of Δ for two important classes of protocols (Sec. IV), and

- Formal modeling and verification of two important distributed protocols for embedded systems, the BMC algorithm (a central component of the IEEE 1588 standard), and the time synchronized channel hopping protocol (Sec. V and VI).

II. Motivation

In this section, we provide an overview of two motivating case studies. The first case study concerns verification of the best master clock algorithm in the IEEE 1588 precision timed protocol [15], where clocks are not (initially) synchronized, but the drift of clocks are bounded. This protocol is representative of a class we term *a posteriori time-synchronized*, since it forms the first phase of a time synchronization protocol. The second case study concerns time-synchronized channel hopping (TSCH) that is part of the IEEE802.15e protocol [1]. This latter case study shows an example where the correctness properties are proven for an *a priori* time-synchronized system.

A. IEEE 1588 Precision Time Protocol

The IEEE 1588 standard [15], also known as the *precision timed protocol (PTP)*, is a distributed protocol that enables precise synchronization of clocks over a communication network. The protocol consists of two parts: the *best master clock (BMC)* algorithm and a *time synchronization phase*. The BMC algorithm is a distributed algorithm and its purpose is twofold: (i) to elect one *grandmaster clock* that is the best clock in the network, and (ii) to find a unique spanning tree in a network, where the grandmaster clock is the root of the tree. Thus, the goal of the BMC algorithm can be characterized as *convergence* to a particular *stable configuration*, comprising agreement on network topology and leader (grandmaster clock). The time synchronization phase uses the spanning tree to synchronize the time of all clocks in the network against the grandmaster clock. In this case study, we are focusing on the correctness of the BMC algorithm, not the time synchronization phase.

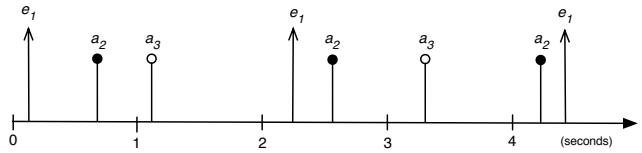

Fig. 2: The figure shows periodic state decision events e_1 for clock C_1 and announce messages a_2 and a_3 received from clocks C_2 and C_3, respectively.

The BMC algorithm is distributed, meaning that there is no central node that coordinates the execution of the algorithm. Consider Fig. 1(a) that depicts four devices with separate clocks C_1, C_2, C_3, and C_4, that are connected using three networks n_1, n_2, n_3. Fig. 1(b) depicts the final result after executing BMC. A tree is formed where C_1 is the root (the grandmaster). The parent/child relationships are defined using the states of the ports: master (M) and slave (S) indicate parent and child, respectively. Note also that the cycle between C_2, C_3, and C_4 is broken by disabling the link between C_2 and C_4, by specifying one of the ports as passive (P).

Each port in the network operates logically as a state machine, determining (somewhat simplified) if it is a master port, a slave port, or a passive port. During execution of the BMC algorithm, each port executes periodically at *state decision events* to exchange messages, where the (slightly varying) period is termed the *announce interval*. These events are fired by timers defined by each individual local clock. Because all clocks can start at different states and be drifting away from each other, there is no guarantee that the clocks will be synchronized. The only assumption that can be made is that the clock drift is bounded. Such a bound is specified by the IEEE 1588 standard. Consider Fig. 2 that shows an example where state decision events e_1 at clock C_1 are fired periodically and *announce messages* a_2 and a_3 are received from clocks C_2 and C_3, respectively. Announce messages are used by the BMC algorithm to inform the clocks in the network about clock characteristics and to communicate the current best clock; it is the main mechanism used for forming the spanning tree and electing the grandmaster clock.

There are several sources of non-determinism during the BMC phase. Firstly, note for instance that in Fig. 2 the state decision events e_1 occurs with a period of 2 seconds, but are drifting slightly for every event. The rate of the drift is bounded, but the clock skew (the difference of time between two clocks) may increase over time. Secondly, the length of an announce interval can vary within a tolerance of $\pm 30\%$ (see section 9.5.8 in [15]). Note for instance how announce messages a_2 and a_3 appears at different times, and how the jitter caused by sending these messages (e.g., because of internal queues and protocol stacks) can result in variation of the number of messages received between two consecutive events; a_2 appears once between the first two events, but twice between the second two events.

The challenge we consider in this case study is to verify the correctness of a central aspect of the BMC algorithm: for a specific topology, we verify that the BMC algorithm converges to one specific grandmaster clock. The non-determinism of when announce messages are received and when periodic events occur make the model checking problem particularly challenging. In this paper, we address the problem of how to model such non-determinism, by providing an analytic solution that abstract away the real-time aspect of the BMC

algorithm and transform the model checking problem into an untimed model. In this case we see that events and announce messages are "almost synchronous", where non-determinism is introduced by bounded clock rates, jitter when sending messages, and by unknown initial clock states.

B. Time-Synchronized Channel Hopping

The *time-synchronized channel hopping* (TSCH) [1] protocol is being adopted as a part of the low power Medium Access Control (MAC) standard IEEE802.15.4e. It has a *time-slotted* architecture and time-slots are grouped into scheduled-superframe which repeats over time. A global schedule instructs each node on what time-slot to transmit/receive data to/from which node. The TSCH protocol makes the strong assumption that the nodes in the system are time-synchronized within a bound called the 'guard' time. Hence, nodes can wake up just before start of the time-slot allotted by the schedule and remain in sleep mode otherwise. In the absence of precise time-synchronization, the time-slots across nodes would not be aligned within the guard bound and hence nodes will fail to communicate successfully during the allotted slot.

Nodes keep track of time-slots using timers maintained by local clocks. Over a duration of time because of the drift in clocks, nodes may get *desynchronized*. A central server computes a global schedule to ensure that nodes always synchronize at least once within the threshold period after which they would be desynchronized. Nodes synchronize on receiving messages from the master node, hence successful communication with the master node periodically is essential and should be ensured by the schedule.

The TSCH standard provides no recommendation on building the schedule. It is the responsibility of the central server to compute the right schedule given the worst-case clock drift and the environmental assumptions. Over-synchronization by communicating more frequently than required may keep all nodes synchronized, but is not desirable because of power constraints. The challenge is to verify the reliability property that given a network deployment, worst-case drift, lossy channels, and a global schedule can all nodes in the system be always synchronized. The assumption is that the nodes are time-synchronized and the property to check is that the protocol extended with the schedule ensures that the nodes remain synchronized.

III. Formal Model and Approach

In this section, we present the formalism used for modeling time-synchronized distributed systems, as well as terminology used when formalizing the notion of approximate synchrony in Sec. IV. The section concludes with a sketch of our verification strategy.

A. Nodes, Clocks, Channels, and Processes

The overall distributed system is a networked composition of concurrent components. We refer to each concurrent component as a *node*. Nodes are modeled as concurrent composition of *processes*. Communication channels that connect nodes are also modeled as processes, but these are not associated with any single node. The simplest model of a channel is an *instantaneous, error-free, first-in-first-out (FIFO) queue*. In this model, messages are delivered instantaneously, without loss or corruption, in FIFO order. More general channel models are also possible by modeling message delays, reordering, or losses using non-deterministic modeling.

Each node has an associated physical clock χ. χ can be modeled as a local state variable associated with the node taking non-negative real values. There are three concepts associated with clocks in a distributed system that are central in this paper:

1. *Clock Skew:* The *skew* between two clocks χ_i and χ_j is the difference in their values $|\chi_i - \chi_j|$.
2. *Clock Drift:* The *drift* in the rate of a clock χ is the difference per unit time of the value of χ from an ideal reference clock.
3. *Clock Offset:* The initial *offset* between two clocks χ_i and χ_j is the difference between their values at time 0 of an ideal reference clock.

Informally speaking, time synchronization ensures that the skew between any two physical clocks in the network is bounded. More formally,

Definition 1. *(Time/Clock Synchronization) A distributed system comprising K nodes with physical clocks $\chi_1, \chi_2, \ldots, \chi_K$ is said to be* time synchronized *if there exists a parameter ξ such that for any two nodes i and j in the system, $|\chi_i - \chi_j| \leq \xi$.*

Example 1. *The IEEE 1588 precision time protocol can have bounds on the physical clock skew on the order of $10s$ of nanoseconds or even less [17]. The typical clock drift is about 10^{-5}. The initial offset between clocks at different nodes can vary widely and there is typically no prescribed bound on its value.*

We formalize a process as a *transition system* that steps on the tick of a "process clock." The process clock can be different for different processes on the same node. Even for a single process, the size of a clock tick can vary over time. Formally, a process $\mathcal{P}$ is a tuple $(\mathcal{S}, \delta, \mathcal{I}, \tau)$ where $\mathcal{S}$ denotes the set of (discrete) states, $\delta \subseteq \mathcal{S} \times \mathcal{S}$ denotes the transition relation, $\mathcal{I}$ denotes the set of initial states, and τ denotes the *process clock*, modeled as a real-valued variable. Although processes typically have inputs and outputs, for simplicity we do not separate them from state in the formal representation. We will further assume, for simplicity, that the individual processes on a single node are deterministic, implying that the transition relation for each such process is in fact a function. This is the case for both practical protocols we have considered in this paper, and the assumption is not fundamental to the theoretical results of this paper. Thus, the only sources of non-determinism are the timing behavior of processes (which determines their interleaving) and the network channels. Finally, note that τ is the *only* continuous state variable in the process.

B. Timing Behavior

Each process $\mathcal{P}_i$ steps (executes) on the tick of its local process clock τ_i. However, note that the size of a step by $\mathcal{P}_i$ (the time between ticks of the process clock) is not a constant. We now define the dynamics of τ_i.

First, note that the step size of a process $\mathcal{P}_i$ executing at node $\mathcal{N}_i$ is determined in part by the physical clock χ_i at $\mathcal{N}_i$. χ_i can be modeled as a real-valued variable obeying the following differential inclusion:

$$\dot{\chi_i} \in [1 - \alpha, 1 + \alpha], \quad \text{where } \alpha \in [0, 1) \tag{1}$$

Note that if $\alpha = 0$, then the physical clock tracks real time (the ideal time reference) perfectly. However, this does not hold in general due to clock drift.

Example 2. *Consider a network where the maximum drift at any node is* 10^{-5}*. Then, the dynamics of a physical clock at each node can be modeled as* $\dot{\chi}_i \in [0.99999, 1.00001]$.

Recall that processes make steps at the ticks of their process clock, but that their *step sizes* — the time intervals between those ticks — is variable. However, motivated by our case studies with the IEEE 1588 and 802.15.4e standards, it is a common requirement that all processes step at regular intervals of nominal size. For instance, a periodic process $\mathcal{P}_i$ might intend to make steps at time instants $\tau_i = 0, 1, 2, 3, \ldots$, but variation in the dynamics of τ_i between ticks might make the step sizes differ slightly from 1. Typically, the process clock is implemented using a timer that is set to expire after a certain amount of time elapses. Since this timer is ultimately based on the physical clock χ_i, the dynamics of τ_i is a function of that of χ_i. Moreover, this function is not the same for all processes executing on the same node. This is especially the case for software implementations of protocols, due to, for example, operating system scheduling jitters, which can introduce minor variations in the step sizes of different processes.

Let us consider how the dynamics of τ_i depends on that of χ_i. Abusing notation slightly, suppose that the steps of a process $\mathcal{P}$ are at values of its process clock τ denoted by $\tau^1, \tau^2, \tau^3, \ldots$. Suppose that the interval between values τ^k and τ^{k-1} is set using a timer to expire after δt time units, measured using the physical clock χ on that node. Further, assume that J denotes the jitter (because of software) in determining the size of that interval. Then, we have, for every step k of $\mathcal{P}$:

$$\tau^k - \tau^{k-1} = J + \int_0^{\delta t} \dot{\chi}(t)\,\mathrm{d}t \tag{2}$$

In general, the jitter J can vary over time and for each process, but for brevity of notation we hide this dependence. Typically, one can derive bounds on J, say $J \in [J_l, J_u]$. From these bounds and Equations 1 and 2, we can infer that

$$J_l + \delta t(1-\alpha) \leq \tau^k - \tau^{k-1} \leq J_u + \delta t(1+\alpha) \tag{3}$$

We can rewrite the above equation as, for every step k,

$$\tau^k - \tau^{k-1} \in [\tau_i^0 - \epsilon, \tau_i^0 + \epsilon] \quad \text{where } \epsilon \in [0, \tau_i^0) \tag{4}$$

where $\tau_i^0 = \delta t + \frac{J_l + J_u}{2}$ denotes the *nominal step size* of the process. The drift of the process clock is denoted by $\epsilon = \alpha \cdot \delta t + \frac{J_u - J_l}{2}$.

Example 3. *In the IEEE 1588 specification [15], the interval between ticks of the process clock is termed as the* announce interval*. The size of this interval(δt) across all nodes, is* 1 *second. The typical drift assumed on physical clocks is* $\pm 10^{-5}$*. While the allowed bound on jitter is* $\pm 30\%$ *of announce interval, a more typical value is* $\pm 10^{-3}$ *sec; i.e.,* $J_u = -J_l = 10^{-3}$*. Thus, the nominal step size* τ_i^0 *is the same as the announce interval,* 1 *second. Further, the drift in the process clock is bounded by* $\epsilon = 10^{-3} + 10^{-5}$*. Therefore, the step size of processes in typical IEEE 1588 implementations can be derived to lie in the interval* $[0.99899, 1.00101]$.

When we consider distributed systems comprising processes with *symmetric timing behavior* (they step on the same values of τ_i), synchronization of physical clocks implies a synchronization of process clocks. We therefore extend the definition of time synchronization from physical clocks to process clocks for such systems. If the distributed system is time-synchronized, there exists a parameter β such that for all i and j,

$$|\tau_i - \tau_j| \leq \beta \tag{5}$$

Example 4. *In the time-synchronized channel hopping (TSCH) protocol in IEEE 802.15.4e, all nodes step on a common notion of time slot. If the physical clocks of all nodes are time-synchronized, then the process clocks will also be time-synchronized.*

The initial value of τ_i also depends on whether the system is time-synchronized in the initial state. If so, for simplicity, we assume that in the initial state $\tau_i = 0$ for all i. If not, then we assume the initial value of τ_i is non-deterministically chosen from the interval $[0, \theta]$ where the parameter θ determines the maximum initial offset between process clocks.

C. Composition and Traces

The network topology determines the structural form of composition of processes corresponding to nodes and channels. Some of the more commonly used simple topologies include a linear chain, a star, and a ring [17]. Given a network topology and the processes modeling various nodes and channels that form the network, one can compose together those processes to obtain the overall model of the system.

The overall model of the distributed system $\mathcal{M}_C$ is obtained as the *synchronous* composition of processes $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_K$. (See Appendix IX-B for a definition of synchronous composition.) Let δ^c_{full} denote the transition relation for the full composed concrete model. A *timed trace* of the composed system is a timestamped record of the execution of the system according to the global time reference mentioned above. More formally, a timed trace is a sequence $h_0, h_1, h_2, \ldots$ where each element h_j is a triple $(s_j, \vec{\tau}_j, t_j)$ where $s_j \in \mathcal{S}_1 \times \mathcal{S}_2 \times \ldots \times \mathcal{S}_K$ is a discrete (global) state, $\vec{\tau}_j = (\tau_{1,j}, \tau_{2,j}, \ldots, \tau_{K,j})$ is the vector of process clock values, $\delta^c_{\text{full}}(s_j, s_{j+1})$ holds for all $j \geq 0$, and the transition into s_j occurs at time t_j.

D. Verification Problem and Approach

The main problem considered in this paper is as follows:

Problem 1. *Given a distributed system* $\mathcal{M}_C$ *modeled as above, and a linear temporal logic (LTL) property* Φ *with propositions over the discrete states of* $\mathcal{M}_C$*, verify whether* $\mathcal{M}_C$ *satisfies* Φ.

A brief introduction to LTL is provided in Appendix IX-A. One way to model $\mathcal{M}_C$ would be as a hybrid system (due to the continuous dynamics of process clocks), but this approach currently does not scale well due to the extremely large discrete state space.

Our approach to solve this problem is based on computing a sound discrete abstraction $\mathcal{M}_A$ of $\mathcal{M}_C$ that preserves the relevant timing semantics of the 'almost-synchronous' systems. There are two phases in our approach:

1. *Compute Abstraction Parameter:* Using parameters of $\mathcal{M}_C$ (relating to clock dynamics), we compute a parameter Δ characterizing the "approximate synchrony" condition, and use Δ to generate a sound abstract model $\mathcal{M}_A$.
2. *Model Checking:* We verify the temporal logic property Φ on the abstract model using finite-state model checking.

The key to this strategy is the first step, which is the focus of the following section.

IV. Approximate Synchrony

We now formalize the concept of *approximate synchrony (AS)*, a tunable timing abstraction for time-synchronized distributed systems. We also present algorithmic techniques to compute a sound AS abstraction.

A. Definition

We construct an abstract model $\mathcal{M}_A$ from $\mathcal{M}_C$ along with a *scheduler* that decides when each process gets to execute. In essence, the scheduler performs a form of *asynchronous* composition of processes, where at each step, any subset of processes can execute provided their execution does not violate a certain condition known as the *approximate synchrony* condition. The condition depends on a parameter Δ which is computed based on the parameters appearing in the definition of the dynamics of the process clock variables. Appendix IX-B includes the definition of asynchronous composition; note that it permits simultaneous steps.

We now define the notion of approximate synchrony.

Definition 2. *(Approximate Synchrony) Two processes $\mathcal{P}_i$ and $\mathcal{P}_j$ are said to execute in* approximate synchrony *(are approximately synchronous) with parameter Δ, if, the number of steps N_i and N_j taken by the two processes from an initial state always satisfy the following condition:*

$$|N_i - N_j| \leq \Delta$$

□

If a system composed of multiple processes is such that, for some Δ, any pair of processes is approximately synchronous with parameter Δ, then we say that the system is *approximately synchronous* with parameter Δ, or, more succinctly, that the system is Δ-abstract; We also say that the system is formed using *approximately synchronous composition*. We will refer to the condition in Definition 2 above as the *approximate synchrony (AS) condition* with parameter Δ, denoted AS(Δ).

Note that Δ is a parameter that characterizes the "approximation" in approximate synchrony. For example, for a perfectly synchronous system $\Delta = 0$, since processes step at the same time instants. For a fully asynchronous system, $\Delta = \infty$, since one process can get arbitrarily ahead of another.

B. Approximate Synchrony Abstraction

We model an approximately synchronous system by explicitly including a scheduler that constrains when processes can execute. Formally, a Δ-abstract model $\mathcal{M}_A$ is an asynchronous composition of abstract processes $\hat{\mathcal{P}}_1, \hat{\mathcal{P}}_2, \ldots, \hat{\mathcal{P}}_K$ with a constraining scheduler ρ_Δ. where each $\hat{\mathcal{P}}_i$ is formed from the corresponding concrete process $\mathcal{P}_i$ by simply dropping the process clock τ; i.e., $\hat{\mathcal{P}}_i = (\mathcal{S}_i, \delta_i, \mathcal{I}_i)$. The set of possible states for the composed abstract processes is $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2 \times \ldots \times \mathcal{S}_K$. The scheduler ρ_Δ maintains as state counts N_i of the numbers of steps taken by each process $\hat{\mathcal{P}}_i$ from the initial state.[1] A configuration of $\mathcal{M}_A$ is a pair (s, N) where $s \in \mathcal{S}$ and $N \in \mathbb{Z}^K$ is the vector of step counts of the K abstract processes. The abstract model $\mathcal{M}_A$ changes its configuration according to its transition function δ^a_{full} where $\delta^a_{\text{full}}((s, N), (s', N'))$ iff (i) $\delta^c_{\text{full}}(s, s')$ and (ii) $N'_i = N_i + 1$ if ρ_Δ permits $\hat{\mathcal{P}}_i$ to make a step and $N'_i = N_i$ otherwise.

[1]The inclusion of step counts may seem to make the model infinite-state. We will show in Sec. V how the model checker can be implemented without explicitly including the step counts in the state space.

In the initial state, all processes $\hat{\mathcal{P}}_i$ are enabled to make a step. At each step of δ^a_{full}, ρ_Δ enforces the approximate synchrony condition by only enabling $\hat{\mathcal{P}}_i$ to step iff that step does not violate AS(Δ). Behaviors of $\mathcal{M}_A$ are *untimed traces*, i.e., sequences of discrete (global) states $s_0, s_1, s_2, \ldots$ where $s_j \in \mathcal{S}$, s_0 is an initial (global) state, and each transition from s_j to s_{j+1} is consistent with δ^a_{full} defined above.

Approximate synchrony is a *tunable timing abstraction*. Larger the value of Δ, more conservative the abstraction. The key question is: for a given system, what value of Δ constitutes a *sound* timing abstraction, and how do we automatically compute it? Recall that one model is a sound abstraction of another if and only if every execution trace of the latter (concrete model) is also an execution trace of the former (abstract model). In our setting, the Δ-abstract and concrete models both capture the protocol logic in an identical manner, and differ only in their timing semantics. The concrete model explicitly models the process clocks of each process as real-valued variables whose dynamics can be described as in Sec. III. The executions of this model can be represented as *timed traces* (sequences of timestamped states). On the other hand, in the Δ-abstract model, processes are interleaved asynchronously while respecting the approximate synchrony condition stated in Definition 2. An execution of the Δ-abstract model is an *untimed trace* (sequences of states). We equate timed and untimed traces using the "untiming" transformation proposed by Alur and Dill [3]; in essence, the traces must be identical with respect to the discrete states.

Next, we address this question of computing a sound abstraction. Two cases are considered: (i) protocols such as TSCH that run on top of a layer of time-synchronization and require a priori time synchronization, and (ii) protocols such as the BMC algorithm that are themselves part of a time synchronization protocol and do not rely on a priori time synchronization. Both classes of protocols share three characteristics:

(a) The nominal step size of a process is constant across all processes, denoted by τ^0;
(b) The process clocks have bounded drift ϵ, and
(c) The processes step at the same sequence of values of their respective clocks.[2]

C. A Priori Time-Synchronized Systems

The first case for deriving Δ is for protocols that, in addition to the two properties listed in Sec. IV-B, satisfy the following condition:

(d) They begin operation after the distributed system has reached a time-synchronized state. Recall from Sec. III, Equation 5, that this implies the skew between any two process clocks τ_i and τ_j must be bounded by a parameter β.

This condition holds for the time-synchronized channel hopping (TSCH) protocol in the IEEE 802.15.4e standard.

Suppose $\mathcal{P}_i$ is specified to step at the following sequence of values of τ_i: $\tau_i^1, \tau_i^2, \tau_i^3, \ldots$. Similarly, $\mathcal{P}_j$ must step at values $\tau_j^1, \tau_j^2, \tau_j^3, \ldots$ of τ_j. Since the processes step on the same sequence of clock values, we know further that $\tau_i^l = \tau_j^l$ for all $l = 1, 2, 3, \ldots$. However, at any time t (the ideal time

[2]Since the clocks of different processes in general have different values, the processes will not in general step synchronously.

reference), $\tau_i(t)$ differs from $\tau_j(t)$ by at most β. Thus, $\mathcal{P}_i$ and $\mathcal{P}_j$ make their l^{th} step within β time units of each other.

If $\beta > 0$, then $\Delta \geq 1$ since two processes are not guaranteed to step at the same time instants, and so the number of steps of two processes can be off by at least one. In the general case, the value of Δ is given by the following theorem.

Theorem 1. *If the dynamics of process clocks in a distributed system $\mathcal{M}_C$ obey Equations 4 and 5, and the system satisfies conditions (a), (b), (c), and (d) above, then processes of $\mathcal{M}_C$ obey the approximate synchrony condition with*

$$\Delta = \lceil \frac{\beta}{\tau^0 - \epsilon} \rceil$$

Proof: Consider two arbitrary processes $\mathcal{P}_i$ and $\mathcal{P}_j$. We show that $|N_i - N_j| \leq \lceil \frac{\beta}{\tau^0 - \epsilon} \rceil$ at all time points.

Consider an arbitrary time point t according to an ideal time reference. Without loss of generality, assume $N_i(t) > N_j(t)$ (i.e., that $\mathcal{P}_i$ has made more steps than $\mathcal{P}_j$) and that $\mathcal{P}_j$ has performed a step at time t. We seek to bound the number of additional steps that $\mathcal{P}_i$ has made over $\mathcal{P}_j$.

Since $\mathcal{P}_i$ and $\mathcal{P}_j$ step at the same values of their respective clocks, it must be the case that $\tau_i > \tau_j$. Due to time synchronization, we also have $\tau_i - \tau_j \leq \beta$. Further, the step size of $\mathcal{P}_i$ is bounded below by $\tau^0 - \epsilon$. Thus, the number of additional steps $\mathcal{P}_i$ could have taken at time t over $\mathcal{P}_j$ is bounded above by

$$\lceil \frac{\tau_i - \tau_j}{\tau^0 - \epsilon} \rceil \leq \lceil \frac{\beta}{\tau^0 - \epsilon} \rceil$$

Thus, $|N_i - N_j| \leq \lceil \frac{\beta}{\tau^0 - \epsilon} \rceil$ at time t, for any t. This yield the desired value of Δ.

■

Thus, if the abstract model $\mathcal{M}_A$ is constructed as the Δ-abstraction of $\mathcal{M}_C$ with Δ as given in Theorem 1, then $\mathcal{M}_A$ is a sound abstraction of $\mathcal{M}_C$: every trace of $\mathcal{M}_C$ satisfies AS(Δ).

In practice, if β is much smaller than $\tau^0 - \epsilon$, then $\Delta = 1$. This is the case for the TSCH protocol.

D. A Posteriori Time-Synchronized Systems

We now consider the second scenario in which we are verifying a protocol that forms part of a time-synchronization scheme. More precisely, the protocol comprises the first phase of a time synchronization scheme, in which all nodes agree on a stable network configuration that is then used to synchronize clocks. We refer to this agreement as *logical convergence*. The goal is to verify that logical convergence is attained within a finite time bound (and maintained thereafter).

In this case, the analysis of Sec. IV-C does not apply as the distributed system may not be time-synchronized when the protocol *begins operation*. However, we show in this section that, even in this case, for the specific verification task of proving that logical convergence is attained, we can derive a sound approximate synchrony abstraction. Our approach is motivated by the BMC algorithm of the IEEE 1588 protocol, and also applies to other similar time synchronization protocols.

Example 5. *As described in Sec. II, IEEE 1588 uses the BMC algorithm to logically converge on a unique spanning tree with the best clock at its root. Nodes in the 1588 network have a common nominal step size called Announce Interval of 1 second; thus, we can model each process as* intending *to step at time $0, 1, 2, 3, \ldots$. (Due to imperfect process clocks, the processes will end up stepping at slightly different time instants.) After the logical convergence of BMC algorithm the time-synchronization phase of the protocol is performed that synchronizes all the physical clocks. Generally, the time from logical convergence of BMC to all the clocks being physically synchronized is approx. 30 seconds, which is approximately equivalent to 30 steps of a process.*

For simplicity in the main body of this paper, we will assume that the physical clocks are instantaneously synchronized after logical convergence is attained; Appendix IX-C discusses the (minor) modification to our theory when this is not the case.

We now present an iterative approach to construct a sound Δ-abstract model of a protocol satisfying conditions (a), (b), and (c) for the purpose of verifying logical convergence of the protocol. Our abstraction method operates in the following steps:

1. First, from Δ and ϵ, we compute a number $N_{\min}$ such that if we can prove the Δ-abstract model $\mathcal{M}_A$ achieves logical convergence before any process makes $N_{\min}$ steps, then this implies the same for $\mathcal{M}_C$. We describe this derivation of $N_{\min}$ in Sec. IV-D1 below.
2. Second, we verify using model checking on the Δ-abstract model that the system achieves logical convergence before any process makes more than $N_{\min}$ steps.

If the second (verification) step succeeds, then we can conclude that $\mathcal{M}_C$ achieves logical convergence within $N_{\min}$ steps. Otherwise, we increment Δ and try Steps 1 and 2 again. When we terminate,[3] we have a value $N_{\min}$ such that logical convergence is guaranteed to be attained when any process reaches $N_{\min}$ steps. Under the assumption that physical time synchronization is instantaneous, from this time point in the trace, the analysis of Sec. IV-C holds and a corresponding value of Δ, say Δ', can be derived accordingly. We then update Δ as $\Delta \leftarrow \max(\Delta, \Delta')$ and use the resulting value to compute the abstract model $\mathcal{M}_A$. We formalize the soundness argument in Theorems 2 and 3 below.

1) Computing $N_{\min}$: Recall that the nominal step size of a process is τ^0, and that the step size of any process is at least $\tau^0 - \epsilon$, and at most $\tau^0 + \epsilon$.

Let $\mathcal{P}_f$ be the fastest process (the one that makes the most steps from the initial state) and $\mathcal{P}_s$ be the slowest (the fewest steps). Denote the corresponding number of steps by N_f and N_s respectively. Then the approximate synchrony condition in Definition 2 is always satisfied if $N_f - N_s \leq \Delta$. We wish to find the smallest number of steps taken by the fastest process before AS(Δ) is violated. We denote this value as $N_{\min}$, and obtain it by formulating and solving a linear program.

Before we state the linear program, we introduce one of its key components. At the time point where $\mathcal{P}_s$ has just completed N_s steps (and $\mathcal{P}_f$ has finished N_f steps), the sum of the step sizes of $\mathcal{P}_f$ must be less than the corresponding sum for $\mathcal{P}_s$ The former can be bounded below by $(\tau^0 - \epsilon)N_f$ and the latter can be bounded above by $(\tau^0 + \epsilon)N_s$. If $\mathcal{P}_s$ and $\mathcal{P}_f$ start making steps at the same instant of time, then their

[3] Termination of this procedure is an open theoretical question at this point; in practice, however, a single iteration of the loop has sufficed in all our experiments.

sums of step sizes are equal, yielding the inequality:

$$(\tau^0 - \epsilon)N_f \leq (\tau^0 + \epsilon)N_s$$

However, processes need not begin making steps simultaneously. Since each process must make its first step at least $\tau^0+\epsilon$ seconds into its execution, the maximum initial offset between processes is $\tau^0 + \epsilon$. The smallest value of N_f occurs when the fast process starts $\tau^0 + \epsilon$ time units after the slowest one, yielding the inequality:

$$(\tau^0 - \epsilon)N_f + (\tau^0 + \epsilon) \leq (\tau^0 + \epsilon)N_s$$

Given the above analysis, we can set up the following integer linear program (ILP) to solve for $N_{\min}$:

$$\begin{aligned} &\min N_f \\ &s.t. \\ &N_f \geq N_s \qquad N_f - N_s > \Delta \\ &(\tau^0 - \epsilon)N_f + (\tau^0 + \epsilon) \leq (\tau^0 + \epsilon)N_s \qquad N_f, N_s \geq 1 \end{aligned} \tag{6}$$

$N_{\min}$ is the optimal value of this ILP. In effect, it gives the fewest steps any process can take (smallest value of N_f) before the approximate synchrony condition AS(Δ) is violated.

Example 6. *Suppose $\epsilon = 10^{-3}$ and $\tau^0 = 1$, as is the case for the IEEE 1588 protocol. Setting $\Delta = 1$, solving the above ILP yields $N_{\min} = 1001$.*

The quantity $N_{\min}$ helps connect the Δ-abstract model $\mathcal{M}_A$ with the concrete model $\mathcal{M}_C$ as formalized in the following theorem.

Theorem 2. *Suppose the Δ-abstract model $\mathcal{M}_A$ reaches logical convergence from any initial state before any process has made $M < N_{\min}$ steps, then the concrete model $\mathcal{M}_C$ also reaches logical convergence from any initial state before any process has taken M steps.*

Proof: By the formulation of the ILP 6, for any timed trace of $\mathcal{M}_C$, if we consider a prefix of the trace in which no process makes more than $N_{\min}$ steps, then that prefix is (after untiming) also a prefix of a trace in $\mathcal{M}_A$ (since the AS(Δ) condition holds for that prefix). If the Δ-abstract model $\mathcal{M}_A$ reaches logical convergence from any initial state before any process has made $M < N_{\min}$ steps, then it means that for any trace of $\mathcal{M}_A$, logical convergence is attained in a prefix of that trace in which no process makes more than M steps. Since this set of trace prefixes includes all trace prefixes of $\mathcal{M}_C$, it follows that $\mathcal{M}_C$ also attains logical convergence before any process has taken M steps. ■

2) Temporal Logic Property: Once $N_{\min}$ is computed, we must verify whether the system (abstract model) achieves logical convergence in less than $N_{\min}$ steps. We perform this by invoking a model checker to verify the following LTL property, which references the variables N_i recording number of steps of process $\mathcal{P}_i$:

$$\mathbf{G}\left[\left(\bigvee_i (N_i = N_{\min} - 1)\right) \implies logicConv\right] \tag{7}$$

If the verification passes, then we can conclude that from any initial state of the abstract model — where all processes are enabled to step — one can reach a logical convergence before any process makes $N_{\min}$ steps. If not, either logical convergence does not hold, or we need to increase Δ; using a model checker allows us to distinguish between these two cases by replaying the counterexample on the concrete model $\mathcal{M}_C$ and increasing Δ only if the counterexample is spurious.

Note that LTL Property 7 is equivalent to

$$\bigwedge_i \left[\mathbf{G}\left((N_i = N_{\min} - 1) \implies logicConv\right)\right]$$

and therefore, one can decompose the check into checking corresponding properties on individual processes. When two processes are symmetric (in functionality and network connectivity), only one need be checked.

We conclude this section with the overall soundness result.

Theorem 3. *If the Δ-abstract model $\mathcal{M}_A$ satisfies the LTL Property 7, then all traces of the concrete model $\mathcal{M}_C$ are traces of the abstract model $\mathcal{M}_A$ (after untiming).*

Proof: From Theorem 2, since the Δ-abstract model $\mathcal{M}_A$ satisfies the LTL Property 7, the concrete model $\mathcal{M}_C$ also reaches logical convergence from any initial state before any process has taken $N_{\min}$ steps. Assuming instantaneous time synchronization, and the update to Δ based on the analysis of Sec. IV-C ($\max(\Delta, \Delta')$, as described above), the remainder of the trace of $\mathcal{M}_C$ (after logical convergence) also satisfies the AS(Δ) condition. Thus, every trace of $\mathcal{M}_C$ is a trace of $\mathcal{M}_A$ (after untiming). ■

V. Model Checking with Approximate Synchrony

In this section we describe the approach of model checking with approximate synchrony as a *scheduler*. We also provide an overview of the model checking algorithm used for verifying soundness of AS abstraction for a given Δ.

A. Approximate Synchrony as a Scheduler

Verification using approximate synchrony is independent of any specific model checker. The model checker does an asynchronous composition of all the processes, where each process is modeled as a transition system. Approximate synchrony can then be implemented as a separate *scheduler* process that enforces the AS(Δ)-constraints (Definition 2). This scheduler process represents the *constraining* scheduler ρ_Δ described in section IV-B.

Implementation: We used ZING an explicit state model checker for verification of our case studies. The explorer in ZING supports guided-search based on an external-scheduler. The AS scheduler (implemented as a ZING external scheduler) systematically explores only those traces that satisfy AS(Δ). It keeps track of the number of steps performed by each process and schedules only those processes that do not violate the AS(Δ) condition. Implementing AS scheduler as an external scheduler reduces the state space, as compared to including the scheduler process as a part of the composed model. We implement a small part of the scheduler as a part of system state for soundness of the verification (See Appendix IX-D).

B. Checking Soundness of Δ Abstract Model

Section IV-D describes an approach to verify whether an Δ abstract model of a protocol is sound. The soundness proof depends on verifying property 7. A naive approach for checking this property would be to include a local variable N_i in each process as part of the process state. We found that the model checker cannot prove property 7 using this naive approach because of state space explosion. To mitigate state space explosion, the N_i information corresponding to each process is maintained external to the system state, and as a part of the explorer of the model checker.

```
typedef Stp List⟨int⟩;
var N : int;
var Q_statetable : Dictionary⟨S, Stp⟩;

1: CheckSoundNess(s : S) {
2:     var i : int;
3:     var s' : S;
4:     var steps' : Stp;
5:     i := 0;
6:     while (i < #P(s)){
7:         steps' := IncElement(i, Q_statetable[s]);
           if (ρ_Δ(steps')) ∨
8:                 steps'[i] > (N + Δ) then
9:             continue ;
10:        s' := NextState(s, i);
11:        if steps'[i] = N_min then
12:            Check(logicConv(s'));
           if (s' ∉ Domain(Q_statetable)) ∨
13:                ¬(steps' ≥_pt Q_statetable[s']) then
14:            Q_statetable[s'] := steps';
15:            CheckSoundNess(s');
16:        i := i + 1 } };
17:    }
18: }
19: Verify() {
20:    Q_statetable[s_initial] = new Stp();
21:    CheckSoundNess(s_initial);
22: }
ρ_Δ(steps') ≡ ∀s_1 ∈ steps' ∀s_2 ∈ steps'(|s_1 − s_2| ≤ Δ)
```

Fig. 3: Algorithm for Verification of Property 7

Implementation: The algorithm in Fig. 3 performs bounded depth first search (DFS) with the final bound on value of *steps* as $(N + \Delta)$ (where $N = N_{min} - 1$). The final bound is considered as $(N + \Delta)$ so as to explore all processes till atleast N steps. It systematically explores the AS transition system described in section IV-B. A variable of type $\mathcal{S}tp$ is used to store a tuple of steps executed by all processes. N and Δ values are derived using Equation 6. $\mathcal{Q}_{statetable}$ is a map from reachable state ($\mathcal{S}$) to the tuple of steps ($\mathcal{S}tp$) with which it was last explored. $CheckSoundNess$ is called recursively on all the successors of a state in DFS order. $\#\mathcal{P}(s)$ returns the number of processes in the current state s. $IncElement(i, t)$ increments the ith element of tuple t and returns the updated tuple. $NextState(s, i)$ returns the next state from s after executing the ith process. At line 8 we check if executing the current process would violate the AS(Δ) (ρ_Δ) or the final bound. Thus, ensuring that we explore only those traces that satisfy AS(Δ). If the steps executed is N then the $logicConv$ monitor is invoked to check if $s' \models logicConv$ (we have reached logical convergence state). As an optimization, to avoid re-exploring a state which may not lead to new states, we don't re-explore a state if it is revisited with $steps'$ greater than what it was last visited. The operator $\geq_{pt}$ does a pointwise comparison of the integer tuples. $steps$ in our algorithm represents N_i in equation 7 and it is a part of the explorer, which is external to the system state and hence we avoid state space explosion caused by the naive approach.

VI. Evaluation

In this section, we present our empirical evaluation of the approximate synchrony abstraction, guided by the following goals:

- Verify two real-world standards protocols: (1) the best master clock algorithm and (2) the time synchronized channel hopping protocol.
- Evaluate how effective the approximate synchrony abstraction is in reducing the state space for model checking.
- Evaluate our approach as a bug finding technique.

We first describe the temporal properties verified for both the case studies, followed by an explanation of our modeling approach. Finally, we present the experimental results in support of the afore-mentioned goals.

A. Temporal Properties of Case Studies

Section II provided an overview of both the case studies and also described the desired properties for the correctness of these protocol. In this section we present the temporal properties verified for the BMC algorithm and the TSCH protocol.

Best Master Clock Algorithm: BMC algorithm is a complex distributed protocol with the property that eventually it stabilizes with a unique spanning tree having the grandmaster at its root (under certain assumptions mentioned in detail in section VI). The system is said to be in $logicConv$ state when the system has converged to the expected spanning tree. The convergence property is:

$$\mathbf{F}\,\mathbf{G}\,(logicConv) \tag{8}$$

It states that the system eventually stabilizes in the $logicConv$ state.

Time Synchronized Channel Hopping: Nodes in the TSCH network should remain time-synchronized, they synchronize on receiving messages from their master. A node is said to be *desynchronized* – if it fails to synchronize with its master within the threshold period. Once desynchronized, the node has to perform a complete resynchronization which consumes a lot of power and should be avoided. To avoid desynchronization, the property that the system should satisfy is:

$$\bigwedge_{i \in nodes} \mathbf{G}(\neg desynchronized_i) \tag{9}$$

It states that all nodes are always synchronized.

B. Modeling of Protocols in P

P [10] is a domain-specific language for writing asynchronous event-driven protocols. A protocol model in P is a collection of state machines interacting with each other via asynchronous events or messages. The P compiler generates a model for systematic exploration by ZING [4], an explicit state model checker. P also provides ways of writing LTL properties as monitors that are synchronously composed with the model. Both the case studies, the BMC algorithm and the TSCH protocol, are modeled using P. Each node in the protocol is modeled as a separate P state machine. The timeouts in the protocol are modeled as non-deterministic choice. Message losses and message delays are also modeled as non-deterministic choice where messages are non-deterministically dropped or delayed. ZING provides easy mechanisms to bound the total number of messages lost or timeouts by bounding the appropriate choice operators. We did not model node/link failures in the network.

C. Experimental Setup

All the experiments were performed on Intel Xeon ES-2440, 2.40GHz (12 cores/24 threads) with 160 GB of memory running 64-bit Windows server OS. ZING explorer can exploit parallelism as its iterative depth-first search algorithm is completely parallelized. All timing results reported in this section

Network Topology (#Nodes)	Safety Property							Convergence Property with *bounded* Fairness Assumption						
	Fully Asynchronous Model			Model with Approximate Synchrony				Fully Asynchronous Model			Model with Approximate Synchrony			
	States Explored	Time (h:mm)	Property Proved	Δ	States Explored	Time (h:mm)	Property Proved	States Explored	Time (hh:mm)	Property Proved	Δ	States Explored	Time (hh:mm)	Property Proved
Linear(5)	1.2 E+9	7:12	Yes	1	9.5 E+5	0:35	Yes	9.1 E+11	13:11	Yes	1	7.2 E+8	8:34	Yes
Star(5)	2.4 E+10	9:40	Yes	1	5.8 E+5	0:54	Yes	8.3 E+11	12:44	Yes	1	4.4 E+7	5:23	Yes
Random(5)	9.19 E+9	9:01	Yes	2	5.5 E+6	1:44	Yes	4.1 E+13*	*	No	2	1.8 E+9	9:21	Yes
Ring(5)	7.1 E+12*	*	No	1	4.8 E+7	3:44	Yes	1.3 E+13*	*	No	1	8 E+9	8:34	Yes
Linear(7)	1.4 E+13*	*	No	1	4.6 E+7	3:05	Yes	1.2 E+13*	*	No	1	1.0 E+8	6:21	Yes
Star(7)	1.1 E+13*	*	No	2	3.7 E+8	5:06	Yes	2.1 E+12*	*	No	2	3.3 E+10	13:34	Yes
Ring(7)	3.3 E+12*	*	No	2	6.8 E+8	8:04	Yes	1.7 E+13*	*	No	2	2.1 E+10	11:11	Yes

** denotes end of exploration as model checker ran out of memory*

TABLE I: Verification of Safety and Liveness property of BMC model.

are when ZING is run with 24 threads. We use the number of states explored and the time taken to explore them as the comparison metric.

D. Verification of the BMC Algorithm

We generated various verification instances by changing the configuration parameters such as number of nodes, clock characteristics, and the network topology. The results in Table I for the BMC algorithm are for 5 and 7 nodes in the network with linear, star, ring, and random topology. The Δ values (1 and 2) used for verification of these configurations were derived by using the sound approach described in section IV-D. We found that for Random(5), Ring(7) and Star(7) the Δ value required is 2 (for others Δ = 1) because BMC model took longer to converge for these configurations and hence nodes would diverge beyond Δ = 1 (BMC for these configurations converges before $N_{min} = 2002$ steps for $\Delta = 2$).

Verification of Safety and Convergence properties: P by default checks a *receptiveness* safety property: that there is an outgoing transition for the dequeued event in the current state for each process. We also verified certain safety properties of BMC model expressed as monitors in P. For checking the convergence property (property 8) we composed the model with *bounded* fairness assumptions, the BMC model can converge only in the presence of bounded message delay and bounded message loss. Our *bounded* fairness assumption is based on strong fairness for scheduling and also bounds the number of timeouts in the system which in turn bounds the message loss/delay in the system.

Table I shows the total number of states explored and time taken by the model checker for proving the safety and convergence property. For demonstrating the state space reduction obtained because of approximate synchrony we also conducted the experiments with complete asynchronous composition, exploring all possible interleavings. The complete asynchronous model is simple to implement but fails to prove the properties for most of the topologies. It runs out of memory on the server because of the large state space. Approximate synchrony abstraction is orders of magnitude faster as it explores reduced states space under the Δ bound.

An upshot of our approach is that we are the first to prove that the BMC algorithm in IEEE 1588 achieves logical convergence to a unique stable state for some interesting configurations. This was possible because of the *sound and tunable* approximate synchrony abstraction. Although experiments with 5/7 nodes may seem small, it is common to have networks of this size in practice, e.g., in industrial automation where one has small teams of networked robots on a factory floor.

Network Topology (#Nodes)	Round-Robin Scheduler			Shared with CSMA		
	States Explored	Time (h:mm)	Property Satisfied	States Explored	Time (h:mm)	Property Satisfied
Linear(5)	4.4 E+4	0:20	Yes	1.2 E+2*	0:03	No
Random(5)	3.6 E+2*	0:01	No	6.2 E+3*	0:12	No
Mesh(5)	1.7 E+7	4:05	Yes	9.1 E+6	2:01	Yes

** denotes the states explored before generating the counter example*

TABLE II: Verification of TSCH with different Schedulers

E. Verification of TSCH

We modeled three different topologies for the TSCH network: linear, random, and mesh with 5 nodes. We verified the desynchronization property (property 9) in the presence of network characteristics like message loss, interference in wireless network, etc using a *failure* machine that injects bounded number of these failures in the system. Given the failure model, network of nodes, and a global schedule, we verified whether the desynchronization property holds for the configuration. Recall that the correctness of the protocol depends on the global schedule that instructs each node when to perform operations. Since the standard provides no recommendation on the schedule, a system designer may want to experiment with different schedules that satisfy the property but also consume minimum power. For the experiments we considered two schedules (1) round-robin (2) shared with CSMA, derived considering the threshold period of 30 time-slots (Appendix IX-D). Table II present the results for TSCH with different schedulers. We were able to verify if the property was satisfied for a given topology under the global schedule, and generated a counterexample otherwise. Thus, approximate synchrony (with $\Delta = 1$) could accurately capture the "almost synchronous" behavior of the a priori time synchronized TSCH system.

F. Bug finding using AS

We also evaluated whether approximate synchrony enables us to find bugs faster. A few realistic bugs were injected into our BMC model, to capture both safety and liveness violations. We then compared the number of states explored and time taken to find these bugs. Table III shows that by using the approximate synchrony abstraction, we are able to find these bugs orders of magnitude faster. The reason for this is that approximate synchrony explores reduced state space, whereas complete asynchrony explores a lot of spurious interleaving. For example, for Liveness_Bug_1, with complete asynchrony we failed to uncover it, but it was found with AS.

Buggy Models	Complete Asynchrony		with AS	
	States Explored	Time (h:mm)	States Explored	Time (h:mm)
Safety_Bug_1	2.4 E+7	3:23	1.7 E+4	0:10
Safety_Bug_2	5.3 E+8	4:11	9.5 E+4	0:22
Liveness_Bug_1	*	*	2.2E+8	4:49

$* \rightarrow$ *failed to find the bug and search ran out of memory*

TABLE III: Finding Bugs Faster with AS

VII. Related Work

The related work can be categorized into two parts: (i) existing approaches for *modeling* time-synchronized systems using timed models, and (ii) existing *untimed abstractions* that are similar to approximate synchrony, but used in different domains.

Modeling: The choice of modeling formalism greatly influences the verification approach. Time-synchronized systems can be modeled as a hybrid system [2]. However, it is important to note that, unlike traditional hybrid systems examples from the domain of control, the discrete part of the state space for these protocols is very large. Due to this we observed that leading hybrid systems verification tools, such as SpaceEx [12], cannot explore the entire state space.

There has been work on modeling protocols similar to IEEE 1588 using real-time formalisms such as *timed automata* [3], where the derivatives of all continuous-time variables are equal to one. One of the more established tools for modeling real-time systems using timed automaton is UPPAAL [16]. The current official version of UPPAAL (version 4.0) does not, however, explicitly support modeling of *mutlirate time systems* [2]; that is, systems with skewed clocks where clocks proceed at different rates. There exist, however, techniques for approximating multirate clocks by making the clock skew part of the model. For instance, Huang *et al.* [14] propose the use of *integer clocks* on top of UPPAAL models. Such clocks are basically integer variables that are periodically updated by a global pulse generator. Daws and Yovine [9] show how multirate timed systems can be abstracted into timed automata. Vaandrager and Groot [19] models a clock that can proceed with different rate by defining a clock model consisting of one location and one self transition (more details in Appendix IX-F).

Such models do not completely represent multirate time systems, but an approximation. By contrast, our approach algorithmically constructs abstractions that can be refined to be more precise by tuning the value of Δ, and results in an untimed model that can be directly checked by a finite-state model checker. Consequently, for the systems we consider, our approach do not suffer from any approximation on integer clocks and we do not need to resort to advanced real-time model checkers such as UPPAAL.

Untimed abstractions: There have been numerous efforts devoted towards capturing the asynchrony in synchronous systems [7], [13]. Multiclock Esterel [18] and CRP [5] are similar extensions to the synchronous language Esterel. Multiclock Esterel provides language extensions to partition clocks into two categories: those that tick simultaneously and those that can have unbounded skew and drift. In time-synchronized systems there is a guarantee of a fixed bound which is captured by approximate synchrony but cannot be captured by these abstractions. The *quasi-synchronous* [8], [13] approach, which is based on Lustre-Scade, is designed for processes communicating over shared memory that are periodic and have almost *same* period. They do capture the clock skew and non-determinism because of clock jitters, but do not capture bounded clock drift (which may lead to bounded variable progress). *Bounded asynchrony* is another approach to restricting the degree of asynchrony with applications to biological systems [11] and cannot be applied directly to time-synchronized systems. Processes in bounded asynchrony synchronize after each step and are interleaved asynchronously between two steps, hence inappropriate for modeling the non-deterministic, but bounded progress drift between processes.

Approximate synchrony is designed to capture all the imperfections in clock dynamics in time-synchronized system — *bounded clock skew*, *clock drift*, and *initial offset*. It is a sound and tunable abstraction, directly usable for verification. Approximate synchrony with $\Delta = 1$ is equivalent to quasi-synchrony and bounded asynchrony with some minor variation in the way systems are modeled.

VIII. Conclusions

We have introduced the notion of approximate synchrony as an abstraction technique for protocols operating on time-synchronized systems, along with techniques to compute sound abstractions. The approach is demonstrated by efficiently verifying properties of two IEEE standard protocols.

References

[1] 802.15.4e 2012. IEEE standard for local and metropolitan area networks-part 15.4: Low-rate wireless personal area networks (LR-WPANs) amendment 1: MAC sublayer. 2012.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical computer science*, 1995.

[3] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 1994.

[4] T. Andrews, S. Qadeer, S. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Proceedings of CAV*. 2004.

[5] G. Berry, S. Ramesh, and R. Shyamasundar. Communicating reactive processes. In *Proceedings of POPL*, 1993.

[6] D. Broman, P. Derler, and J. C. Eidson. Temporal issues in cyber-physical systems. *Journal of Indian Institute of Science*, 2013.

[7] P. Caspi. Embedded control: From asynchrony to synchrony and back. In *Embedded Software*, 2001.

[8] P. Caspi, C. Mazuet, and N. R. Paligot. About the design of distributed control systems: The quasi-synchronous approach. In *SAFECOMP*. 2001.

[9] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with Kronos. In *Proceedings of RTSS*, 1995.

[10] A. Desai, V. Gupta, E. K. Jackson, S. Qadeer, S. K. Rajamani, and D. Zufferey. P: Safe asynchronous event-driven programming. In *Proceedings of PLDI*, 2013.

[11] J. Fisher, T. A. Henzinger, M. Mateescu, and N. Piterman. Bounded asynchrony: Concurrency for modeling cell-cell interactions. In *FMSB*. 2008.

[12] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. Spaceex: Scalable verification of hybrid systems. In *CAV*, 2011.

[13] N. Halbwachs and L. Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *Proceedings of ACSD*, 2006.

[14] X. Huang, A. Singh, and S. A. Smolka. Using Integer Clocks to Verify the Timing-Sync Sensor Network Protocol. In *Proceedings of NFM*, 2010.

[15] IEEE Instrumentation and Measurement Society. *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, 2008.

[16] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on STTT*, 1997.

[17] M. Lipinski, T. Wlostowski, J. Serrano, P. Alvarez, J. Gonzalez Cobas, A. Rubini, and P. Moreira. Performance results of the first white rabbit installation for cngs time transfer. In *Proceedings of ISPCS*, 2012.

[18] B. Rajan and R. Shyamasundar. Multiclock esterel: a reactive framework for asynchronous design. In *IPDPS*, 2000.

[19] F. W. Vaandrager and A. de Groot. Analysis of a biphase mark protocol with Uppaal and PVS. *Formal Aspects of Computing*, 2006.

IX. Appendix

A. Linear Temporal Logic

Given a finite set of atomic propositions Σ, formulas in linear temporal logic (LTL) are constructed as per the following grammar:

$$\psi ::= p \,|\, \neg\psi \,|\, \psi \vee \psi \,|\, \mathbf{X}\psi \,|\, \psi\mathbf{U}\psi$$

where $p \in \Sigma$ is an atomic proposition, $\mathbf{X}$ is the temporal operator *next* and $\mathbf{U}$ is the temporal operator *until*. Other temporal operators can be derived using these two temporal operators and Boolean operators, for example, "eventually ψ" as $\mathbf{F}\psi = \mathbf{true}\mathbf{U}\psi$ and "globally ψ" as $\mathbf{G}\psi = \neg\mathbf{F}\neg\psi$.

B. Synchronous/Asynchronous Composition

Our use of synchronous composition is standard. Given two processes $\mathcal{P}_i = (\mathcal{S}_i, \delta_i, \mathcal{I}_i, i, \tau_i)$ and $\mathcal{P}_j = (\mathcal{S}_j, \delta_j, \mathcal{I}_j, j, \tau_j)$ the synchronous composition is the process $\mathcal{P} = (\mathcal{S}_i \times \mathcal{S}_j, \delta, \mathcal{I}_i \times \mathcal{I}_j, \text{ID}, (\tau_i, \tau_j))$ where $\delta \doteq \delta_i \wedge \delta_j$ (interpreting transition relations as Boolean formulas in the usual way).

Our definition of asynchronous composition is slightly different from the usual in that we allow multiple processes to step simultaneously. In effect, given processes $\mathcal{P}_i$ and $\mathcal{P}_j$ as given above, the asynchronous composition differs from synchronous composition in that it is also possible for just one process to step:

$$\begin{aligned}\delta((s_i, s_j), (s'_i, s'_j)) \doteq &(\delta_i(s_i, s'_i) \wedge \delta_j(s_j, s'_j)) \vee \\ &(\delta_i(s_i, s'_i) \wedge s_j = s'_j) \vee \\ &(s_i = s'_i \wedge \delta_j(s_j, s'_j))\end{aligned}$$

This definition generalizes to K processes in the standard way, allowing any subset of processes to step simultaneously at a given point in time.

In order to enforce the approximate synchrony condition, we permit steps consistent with asynchronous composition as above provided the AS condition in Definition 2 is not violated.

C. Non-Zero Time to Physical Clock Synchronization

For a time synchronization protocol such as IEEE 1588, there typically exists an integer L2P such that, if logical convergence is achieved, the clocks of the nodes will be time-synchronized within L2P steps of any process. L2P is derived from a real-time bound on the amount of time between achieving logical convergence and synchronizing the physical clocks, by dividing this real-time bound by the smallest step size of any process.

As an example, for the IEEE 1588 protocol the time from logical convergence of BMC to all the clocks being physically synchronized is approximately 30 seconds. With the nominal step size (announce interval) of 1 second and $\epsilon = 10^{-3}$, the value of L2P is $\lceil \frac{30}{0.999} \rceil = 31$.

We can account for this delay to physical clock synchronization by replacing $N_{\min}$ in the LTL Property 7 with $N_{\min} - \text{L2P}$. The rest of the theoretical analysis remains unchanged.

D. Implementation of AS as scheduler

Section V gives an overview of how we implemented the AS scheduler (ρ_Δ) as an external ZING scheduler. Hence the entire logic of which process should be executed in the current state is evaluated and enforced externally and not add any more states to the existing state space (SS).

We used explicit state model checker with state caching which means that if a state is already explored then it is not re-explored when visited again. Consider two cases (1) when the entire scheduler ρ_Δ is part of the composed system ($SS = (PS \times SchedS)$, where PS is the set of program state and $SchedS$ is the set of states of ρ_Δ scheduler), this is when ρ_Δ is included as a process in the program model (2) when the scheduler ρ_Δ is implemented as a external scheduler without adding any scheduler state ($SS = PS$). In the first case since scheduler state is a part of the system state and hence we would not miss any possible combination of program state and scheduler state. In the second case since scheduler state is not maintained, we miss soundness because we might visit the same program state with different scheduler state (which can mean that different out-going transitions are enabled) and hence the state should be re-explored with the new scheduler state. But because of state-caching only program state the explorer assumes that all possible transition from this state are explored and hence we don't re-explore it and can miss reachable state.

The fix for this is that we maintain minimal information as a part of the system state that distinguishes the program state when it is visited with different scheduler state ($SS = (PS \times minSchedS)$, note that $minSched$ is not the set of states of ρ_Δ scheduler). The way we do it is by storing the steps executed by each process modulo Δ and this is enough to ensure all combination of program state and scheduler state are explored. And the complex logic of evaluating which process to execute next and enforcing AS condition is still in the external scheduler. We did not add new scheduler process in the system which does save a lot of states.

E. Parameters for Experiments

1) BMC Algorithm: Using the set of Equations 6, and the values of $\epsilon = 10^{-3}$ we get for :

- $\Delta = 1$ $N_{min} = 1001$
- $\Delta = 2$ $N_{min} = 2002$

2) TSCH: In TSCH network, all the nodes are assumed to start communicating at the start of the time-slot. To tolerate some desynchronization the receivers start listening a small time duration before the start of time-slot and keeps listening sometime after. This duration is called the 'guard' time (T_g). Typical T_g value is $1ms$. Consider the system being equipped with 60ppm crystals then two nodes can drift by $120\mu s$. The synchronization period is τ_{sp} and is calculated using the equation 10. Which means that the clocks desynchronize $8s$ after it last communicated. For safety we consider that the nodes should communicate every $3s$. If a step in the model corresponds to 1 time-slot and the time-slot size if $100ms$ then the number of steps between two periodic resynchronization is $N_{period} = 30$

$$\tau_{sp} = \frac{T_g}{drift} \tag{10}$$

Hence N_{period} represents the threshold period referred to in the evaluations section. Also, the round robin scheduler cycles

over all the nodes in the network periodically. Shared with CSMA have only shared slots in them and use CSMA protocol to resolve conflict.

F. UPPAAL Multirate Clock Modeling

Vaandrager and Groot [19] propose one possible approach of modeling clocks with variable rate in a timed automata. In this approach, a clock model consists of one location and one self transition. The location has an invariant $x \leq max$ and the transition has a guard $x \geq min$, where x is a timer represented as an UPPAAL clock with fixed rate of 1, and min and max are the minimal and maximal number of time units that can elapse between each clock tick, respectively. Each time the transition is taken, the clock ticks and x is reset to 0. The min and max parameters are bounding the clock rates, similar to a multirate clock, but clock rates are specified using integers rather than reals. For instance, if time is measured in milliseconds and $max = 1010$ and $min = 990$, the timer x will count to between 990 and 1010 before one tick occurs.

G. Temporal Properties of the BMC Algorithm

An instance of the model checking problem is a triple $(\mathbf{P}, \Gamma, \Phi)$ where $\mathbf{P}$ is the set of processes modeling nodes and channels, Γ is the system configuration, and Φ is the temporal logic property to be verified. In this section, we describe the latter two elements.

Configuration. Configuration Γ is a tuple $(C, Rank, NT)$ where C represents the set of clocks (nodes) in the system (either ordinary or boundary clock), $Rank$: $C \rightarrow N$ is a function that assigns a rank to each clock in the system based on its characteristics, and NT: $C \rightarrow 2^C$ is a function defining the network topology of the clocks in the system (chain, ring, star, etc.). The characteristics of a clock is determined based on features described in the 1588 standard [15]. The function $Rank$ is meant to capture the clock quality and provides a total ordering among different clocks in the system. During verification, we vary the parameters of Γ to generate interesting configurations.

Properties. We present safety and convergence properties for BMC algorithm here. Our properties are defined using two functions eGM and $eCut$ where $eGM : C \rightarrow C$ is a function that given a clock in the system returns its *expected grandmaster clock* in the logically synchronized state. Similarly, $eCut$ is a set of edges in the network graph that partition it into different trees if there are multiple grandmasters. To encode this property, each clock C_i maintains a local variable $ParentGM_i$ in which it stores its current GM. Initially, $ParentGM_i$ is *null* and eventually as the clocks exchange messages and converge to a stable state, $ParentGM_i$ should point to the expected grandmaster $eGM(C_i)$. Let $logicConv$ be the correct logically synchronized state of the system, in which it must stabilize. Formally, we consider a state as $logicConv$ state if it satisfies the following property :

$$(\forall_i ParentGM_i = eGM(C_i)) \wedge (currCut = eCut) \quad (11)$$

The above property specifies that in $logicConv$ state all the nodes should have the correct grandmaster information and the network should be correctly partitioned into trees based on the number of grandmasters in the system.

Convergence Property: The convergence property of the system can be specified as

$$\mathbf{F}\,\mathbf{G}\,(logicConv) \quad (12)$$

It states that the system eventually stabilizes in the $logicConv$ state.

Safety Properties: The protocol is said to be in a $metaStable$ state if the number of distinct values of the variable $ParentGM$ is equal to the number of distinct grandmasters in the system. We capture this formally using a special function termed $range$:

$$(|range(ParentGM)| = |range(eGM)|) \Leftrightarrow metaStable \quad (13)$$

Given this definition, the BMC algorithm should satisfy the following property:

$$\mathbf{G}(metaStable \rightarrow (\forall_i ParentGM_i \in range(eGM))) \quad (14)$$

Property 14 specifies that if the system is in $metaStable$ state then $ParentGM$ of all the nodes should point to a valid grandmaster node. This enforces a safety check that in all intermediate $metaStable$ states of the protocol, the protocol always chooses a valid node as the grandmaster node.

H. Temporal Properties of TSCH

Let N_{period} be the allowed period calculated by the central server based on the worst case drift of clocks in the system. Each node in the network is modeled as a P state machine. Each state machine i has a local variable $lastSynchronized_i$ that keeps track of when the node was last synchronized. $lastSynchronized_i$ is incremented each time the state machine performs a step (which in our case is one time-slot). If a node receives message from its master then it synchronizes using that message and sets $lastSynchronized_i$ to 0. To verify that given the model of the system and the schedule does it satisfy the property that no node is ever desynchronized, we check the following LTL property.

$$\mathbf{G}(\forall_i (lastSynchronized_i \leq N_{period})) \quad (15)$$