Title

Approximate synchrony: An abstraction for distributed almost-synchronous systems

Permalink

https://escholarship.org/uc/item/2nn9n9rs

ISBN

9783319216676

Authors

Desai, A
Seshia, SA
Qadeer, S
et al.

Publication Date

2015

DOI

10.1007/978-3-319-21668-3_25

Approximate Synchrony: An Abstraction for Distributed Almost-Synchronous Systems

Ankush Desai[1], Sanjit A. Seshia[1], Shaz Qadeer[2], David Broman[1,3], John C. Eidson[1]

[1] University of California at Berkeley, CA, USA
[2] Microsoft Research, Redmond, USA
[3] KTH Royal Institute of Technology, Sweden

Abstract. Forms of synchrony can greatly simplify modeling, design, and verification of distributed systems. Thus, recent advances in clock synchronization protocols and their adoption hold promise for system design. However, these protocols synchronize the distributed clocks only within a certain tolerance, and there are transient phases while synchronization is still being achieved. Abstractions used for modeling and verification of such systems should accurately capture these imperfections that cause the system to only be "almost synchronized." In this paper, we present approximate synchrony, a sound and tunable abstraction for verification of almost-synchronous systems. We show how approximate synchrony can be used for verification of both time synchronization protocols and applications running on top of them. We provide an algorithmic approach for constructing this abstraction for *symmetric, almost-synchronous* systems, a subclass of almost-synchronous systems. Moreover, we show how approximate synchrony also provides a useful strategy to guide state-space exploration. We have implemented approximate synchrony as a part of a model checker and used it to verify models of the Best Master Clock (BMC) algorithm, the core component of the IEEE 1588 precision time protocol, as well as the time-synchronized channel hopping protocol that is part of the IEEE 802.15.4e standard.

1 Introduction

Forms of synchrony can greatly simplify modeling, design, and verification of distributed systems. Traditionally, a common sense of time is established using *time-synchronization* (*clock-synchronization*) protocols or systems such as the global positioning system (GPS), network time protocol (NTP), and the IEEE 1588 [20] precision time protocol (PTP). These protocols, however, synchronize the distributed clocks only within a certain bound. In other words, 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, i.e., they are *almost synchronized*.

Distributed protocols running on top of time-synchronized nodes are designed under the assumption that while processes at different nodes make independent progress, no process falls very far behind any other. Figure 1 provides examples of such real world systems. For example, *Google Spanner* [8] is a distributed fault tolerant system that provides consistency guarantees when run on top of nodes that are synchronized using GPS and atomic clocks, wireless sensor networks [28,27] use time synchronized channel hopping (TSCH) [1] as a standard for time synchronization of sensor nodes in the network, and IEEE 1588 precision time protocol (PTP) [20] has been adopted in industrial automation, scientific measurement [22], and telecommunication networks.

Correctness of these protocols depends on having some synchrony between different processes or nodes.

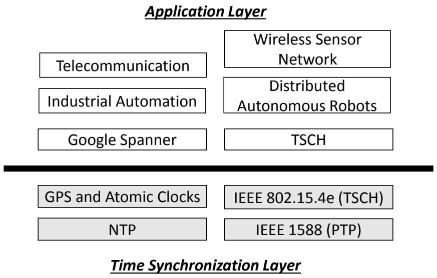

Fig. 1. Almost-synchronous systems comprise an application protocol running on top of a time-synchronization layer.

When modeling and verifying systems that are almost-synchronous it is important to compose them using the right concurrency model. One requires a model that lies somewhere between completely synchronous (lock-step progress) and completely asynchronous (unbounded delay). Various such concurrency models have been proposed in the literature, including *quasi-synchrony* [7,18] and *bounded-asynchrony* [16]. However, we discuss in Sec. 7, these models permit behaviors that are typically disallowed in almost-synchronous systems. Alternatively, one can use formalisms for hybrid or timed systems that explicitly model clocks (e.g., [3,2]), but the associated methods (e.g., [21,17]) tend to be less efficient for systems with a huge discrete state space, which is typical for distributed software systems.

In this paper, we introduce *symmetric, almost-synchronous* (SAS) systems, a class of distributed systems in which processes have symmetric timing behavior. In our experience, protocols at both the application layer and the time-synchronization layer can be modeled as SAS systems. Additionally, we introduce the notion of *approximate synchrony (*AS*)* as a concurrency model for almost-synchronous systems, which also enables one to compute a sound discrete abstraction of a SAS system. Intuitively, a system is approximately-synchronous if the number of steps taken by any two processes do not differ by more than a specified bound, denoted Δ. The presence of the parameter Δ makes approximate synchrony a *tunable* abstraction method. We demonstrate three different uses of the approximate synchrony abstraction:

1. **Verifying time-synchronized systems:** Suppose that the system to be verified runs on top of a layer that guarantees time synchronization throughout its execution. In this case, we show that there is a sound value of Δ which can be computed using a closed form equation as described in Sec. 3.2.
2. **Verifying systems with recurrent logical behavior:** Suppose the system to be verified does not rely on time synchronization, but its traces contain recurrent logical conditions — a set of global states that are visited repeatedly during the protocol's operation. We show that an iterative approach based on model checking can identify such recurrent behavior and extract a value of Δ that can be used to compute a sound discrete abstraction for model checking (see Sec. 4). Protocols verifiable with this approach include some at the time-synchronization layer, such as IEEE 1588 [20].
3. **Prioritizing state-space exploration:** The approximate synchrony abstraction can also be used as a search prioritization technique for model checking. We show in Sec. 6 that in most cases it is more efficient to search behaviors for smaller value of Δ ("more synchronous" behaviors) first for finding bugs.

We present two practical case studies: (i) a time-synchronized channel hopping (TSCH) protocol that is part of the IEEE802.15.4e [1] standard, and (ii) the best master clock (BMC) algorithm of the IEEE 1588 precision time protocol. The former is system where the nodes are time-synchronized, while the latter is the case of a system with recurrent logical behavior. Our results show that approximate synchrony can reduce the state space to be explored by orders of magnitude while modeling relevant timing semantics of these protocols, allowing one to verify properties that cannot be verified otherwise. Moreover, we were able to find a so-called “rogue frame” scenario that the IEEE 1588 standards committee had long debated without resolution (see our companion paper written for the IEEE 1588 community [6] for details).

Our abstraction technique can be used with any finite-state model checker. In this paper we implement it on top of the ZING model checker [4], due to its ability to control the model checker’s search using an external scheduler that enforces the approximate synchrony condition.

To summarize, this paper makes the following contributions:

- The formalism of *symmetric, almost synchronous* (SAS) systems and its use in modeling an important class of distributed systems (Sec. 2);
- A tunable abstraction technique, termed *approximate synchrony* (Sec. 2 and 3);
- Automatic procedures to derive values of Δ for sound verification (Sec. 3 and 4);
- An implementation of approximate synchrony in an explicit-state model checker (Sec. 5), and
- The use of approximate synchrony for verification and systematic testing of two real-world protocols, the BMC algorithm (a key component of the IEEE 1588 standard), and the time synchronized channel hopping protocol (Sec. 6).

2 Formal Model and Approach

In this section, we define clock synchronization precisely and formalize the notion of *symmetric almost-synchronous* (SAS) systems, the class of distributed systems we are concerned with in this paper.

2.1 Clocks and Synchronization

Each node in the distributed system has an associated (local) physical clock χ, which takes a non-negative real value. For purposes of modeling and analysis, we will also assume the presence of an ideal (global) reference clock, denoted t. The notation $\chi(t)$ denotes the value of χ when the reference clock has value t. Given this notation, we describe the following two basic concepts:

1. *Clock Skew:* The *skew* between two clocks χ_i and χ_j at time t (according to the reference clock) is the difference in their values $|\chi_i(t) - \chi_j(t)|$.
2. *Clock Drift:* The *drift* in the rate of a clock χ is the difference per unit time of the value of χ from the ideal reference clock t.

Time synchronization ensures that the skew between any two physical clocks in the network is bounded. The formal definition is as below.

Definition 1. *A distributed system is* time-synchronized *(or* clock-synchronized*) if there exists a parameter β such that for every pair of nodes i and j and for any t,*

$$|\chi_i(t) - \chi_j(t)| \leq \beta \tag{1}$$

For ease of exposition, we will not explicitly model the details of dynamics of physical clocks or the updates to them. We will instead abstract the clock dynamics as comprising arbitrary updates to χ_i variables subject to additional constraints on them such as Eqn. 1 (wherever such assumptions are imposed).

Example 1. The IEEE 1588 precision time protocol [20] can be implemented so as to bound the physical clock skew to the order of sub-nanoseconds [22], and the typical clock drift to at most 10^{-4} [20].

2.2 Symmetric, Almost-Synchronous Systems

We model the distributed system as a collection of processes, where processes are used to model both the behavior of nodes as well as of communication channels. There can be one or more processes executing at a node.

Formally, the system is modeled as the tuple $\mathcal{M}_C = (\mathcal{S}, \delta, \mathcal{I}, \text{ID}, \boldsymbol{\chi}, \boldsymbol{\tau})$ where

- $\mathcal{S}$ is the set of discrete states of the system,
- $\delta \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation for the system,
- $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states,
- $\text{ID} = \{1, 2, \ldots, K\}$ is the set of process identifiers,
- $\boldsymbol{\chi} = (\chi_1, \chi_2, \ldots, \chi_K)$ is a vector of local clocks, and
- $\boldsymbol{\tau} = (\tau_1, \tau_2, \ldots, \tau_K)$ is a vector of process timetables. The timetable of the ith process, τ_i, is an infinite vector $(\tau_i^1, \tau_i^2, \tau_i^3, \ldots)$ specifying the time instants according to local clock χ_i when process i executes (steps). In other words, process i makes its jth step when $\chi_i = \tau_i^j$.

For convenience, we will denote the ith process by $\mathcal{P}_i$. Since in practice the dynamics of physical clocks can be fairly intricate, we choose not to model these details — instead, we assume that the value of a physical clock χ_i can vary arbitarily subject to additional constraints (e.g., Eqn. 1).

The kth *nominal step size* of process $\mathcal{P}_i$ is the intended interval between the $(k-1)$th and kth steps of $\mathcal{P}_i$, viz., $\tau_i^k - \tau_i^{k-1}$. The *actual step size* of the process is the actual time elapsed between the $(k-1)$th and kth step, according to the ideal reference clock t. In general, the latter differs from the former due to clock drift, scheduling jitter, etc.

Motivated by our case studies with the IEEE 1588 and 802.15.4e standards, we impose two restrictions on the class of systems considered in this paper:

1. *Common Timetable:* For any two processes $\mathcal{P}_i$ and $\mathcal{P}_j$, $\tau_i = \tau_j$. Note that this does *not* mean that the process step synchronously, since their local clocks may report different values at the same time t. However, if the system is time synchronized, then the processes step "almost synchronously."
2. *Bounded Process Step Size:* For any process $\mathcal{P}_i$, its actual step size lies in an interval $[\sigma^l, \sigma^u]$. This interval is the same for all processes. This restriction arises in practice from the bounded drift of physical clocks.

A set of processes obeying the above restrictions is termed a *symmetric, almost-synchronous* (SAS) system. The adjective "symmetric" refers only to the timing behavior — note that the logical behavior of different processes can be very different. Note also that SAS systems may or may not be running on top of a time synchronization layer, i.e., SAS systems and time-synchronized systems are orthogonal concepts.

Example 2. The IEEE 1588 protocol can be modeled as a SAS system. All processes intend to step at regular intervals called the *announce time interval*. The specification [20] states the nominal step size for all processess as 1 second; thus the timetable is the sequence $(0, 1, 2, 3, \ldots)$. However, due to the drift of clocks and other non-idealities such as jitter due to OS scheduling, the step size in typical IEEE 1588 implementations can vary by $\pm 10^{-3}$. From this, the actual step size of processes can be derived to lie in the interval $[0.999, 1.001]$.

Traces and Segments. A *timed trace* (or simply *trace*) of the SAS system $\mathcal{M}_C$ is a timestamped record of the execution of the system according to the global (ideal) time reference t. Formally, a timed trace is a sequence $h_0, h_1, h_2, \ldots$ where each element h_j is a triple $(s_j, \boldsymbol{\chi}_j, t_j)$ where $s_j \in \mathcal{S}$ is a discrete (global) state at time $t = t_j$ and $\boldsymbol{\chi}_j = (\chi_{1,j}, \chi_{2,j}, \ldots, \chi_{K,j})$ is the vector of clock values at time t_j. For all j, at least one process makes a step at time t_j, so there exists at least one i and a corresponding $m_i \in \{0, 1, 2, \ldots\}$ such that $\chi_{i,j}(t_j) = \tau_i^{m_i}$. Moreover, processes step according to their timetables; thus, if any $\mathcal{P}_i$ makes its m_ith and l_ith steps at times t_j and t_k respectively, for $m_i < l_i$, then $\chi_{i,j}(t_j) = \tau_i^{m_i} < \tau_i^{l_i} = \chi_{i,k}(t_k)$. Also, by the bounded process step size restriction, if any $\mathcal{P}_i$ makes its m_ith and $m_i + 1$th steps at times t_j and t_k respectively (for all m_i), $|t_k - t_j| \in [\sigma^l, \sigma^u]$. Finally, $s_0 \in \mathcal{I}$ and $\delta(s_j, s_{j+1})$ holds for all $j \geq 0$ with the transition into s_j occuring at time $t = t_j$.
A *trace segment* is a (contiguous) subsequence $h_j, h_{j+1}, \ldots, h_l$ of a trace of $\mathcal{M}_C$.

2.3 Verification Problem and Approach

The central problem considered in this paper is as follows:

Problem 1. Given an SAS 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.

One way to model $\mathcal{M}_C$ would be as a hybrid system (due to the continuous dynamics of physical clocks), but this approach does not scale well due to the extremely large discrete state space. Instead, we provide a sound discrete abstraction $\mathcal{M}_A$ of $\mathcal{M}_C$ that preserves the relevant timing semantics of the ‘almost-synchronous’ systems. (Soundness is formalized in Sec. 3).
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 sections.

3 Approximate Synchrony

We now formalize the concept of *approximate synchrony (AS)* and explain how it can be used to generate a discrete abstraction of almost-synchronous distributed systems. Approximate synchrony applies to both (segments of) traces and to systems.

Definition 2. *(Approximate Synchrony for Traces) A trace (segment) of a SAS system $\mathcal{M}_C$ is said to satisfy* approximate synchrony *(is* approximately-synchronous*) with parameter Δ if, for any two processes $\mathcal{P}_i$ and $\mathcal{P}_j$ in $\mathcal{M}_C$, the number of steps N_i and N_j taken by the two processes in that trace (segment) satisfies the following condition:*

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

Although this definition is in terms of traces of SAS systems, we believe the notion of approximate synchrony is more generally applicable to other distributed systems also. An early version of this definition appeared in [10].

The definition extends to a SAS system in the standard way:

Definition 3. *(Approximate Synchrony for Systems) A SAS system $\mathcal{M}_C$ satisfies* approximate synchrony *(is* approximately-synchronous*) with parameter Δ if* all traces *of that system satisfy approximate synchrony with parameter Δ.*

We refer to the condition in Definition 3 above as the *approximate synchrony (AS) condition* with parameter Δ, denoted AS(Δ). For example, in Fig. 2, executing step 5 of process *P1* before step 3 of process *P2* violates the approximate synchrony condition for $\Delta = 2$. Note that Δ quantifies 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.

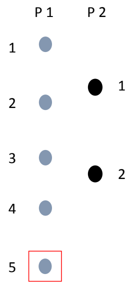

Fig. 2. AS(Δ) violated for $\Delta = 2$

3.1 Discrete Approximate Synchrony Abstraction

We now present a discrete abstraction of a SAS system. The key modification is to (i) remove the physical clocks and timetables, and (ii) include instead an explicit scheduler that constrains execution of processes so as to satisfy the approximate synchrony condition AS(Δ).

Formally, given a SAS system $\mathcal{M}_C = (\mathcal{S}, \delta, \mathcal{I}, \text{ID}, \boldsymbol{\chi}, \boldsymbol{\tau})$, we construct an Δ*-abstract model* $\mathcal{M}_A$ as the tuple $(\mathcal{S}, \delta^a, \mathcal{I}, \text{ID}, \rho_\Delta)$ where ρ_Δ is a scheduler process that performs an asynchronous composition of the processes $\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_K$ while enforcing AS(Δ). Conceptually, the scheduler ρ_Δ maintains state counts N_i of the numbers of steps taken by each process $\hat{\mathcal{P}}_i$ from the initial state.[4] 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 processes. The abstract model $\mathcal{M}_A$ changes its configuration according to its transition function δ^a where $\delta^a((s, N), (s', N'))$ iff (i) $\delta(s, s')$ and (ii) $N'_i = N_i + 1$ if ρ_Δ permits $\mathcal{P}_i$ to make a step and $N'_i = N_i$ otherwise.

In an initial state, all processes $\mathcal{P}_i$ are enabled to make a step. At each step of δ^a, ρ_Δ enforces the approximate synchrony condition by only enabling $\mathcal{P}_i$ 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 $s_0, s_1, s_2, \dots$ 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 defined above.

Note that 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 $\mathcal{M}_C$) is also an execution trace of the former (abstract model $\mathcal{M}_A$). 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 physical clocks of each process as real-valued variables as described in Sec. 2. The executions of this model can be represented as *timed traces* (sequences of timestamped states). On the other hand, in the

[4] The inclusion of step counts may seem to make the model infinite-state. We will show in Sec. 5 how the model checker can be implemented without explicitly including the step counts in the state space.

Δ-abstract model, processes are interleaved asynchronously while respecting the approximate synchrony condition stated in Definition 3. 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] — i.e., the traces must be identical with respect to the discrete states.

3.2 Computing Δ for Time-Synchronized Systems

We now address the question of computing a value of Δ such that the resulting $\mathcal{M}_A$ is a sound abstraction of the original SAS system $\mathcal{M}_C$. We consider here the case when $\mathcal{M}_C$ is a system running on a layer that guarantees time synchronization (Eqn. 1) from the initial state. A second case, when nodes are not time-synchronized and approximate synchrony only holds for segments of the traces of a system, is handled in Sec. 4.

Consider a SAS system in which the physical clocks are always synchronized to within β, i.e., Equation 1 holds for all time t and β is a tight bound computed based on the system configuration. Intuitively, 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. The main result of this section is that SAS systems that are time-synchronized are also approximately-synchronous, and the value of Δ is given by the following theorem.

Theorem 1. *Any SAS system $\mathcal{M}_C$ satisfying Equation 1 is approximately-synchronous with parameter $\Delta = \left\lceil \frac{\beta}{\sigma^l} \right\rceil$.* *(Proof in [12])*

Suppose the abstract model $\mathcal{M}_A$ is constructed as described in Sec. 3.1 with Δ as given in Theorem 1 and σ^l is the lower bound of the step size defined in Sec. 2.2. Then as a corollary, we can conclude that $\mathcal{M}_A$ is a sound abstraction of $\mathcal{M}_C$: every trace of $\mathcal{M}_C$ satisfies AS(Δ) and hence is a trace of $\mathcal{M}_A$ after untiming.

Example 3. The *Time-Synchronized Channel Hopping* (TSCH) [1] protocol is being adopted as a part of the low power Medium Access Control standard IEEE802.15.4e. It can be modeled as a SAS system since it has a time-slotted architecture where processes share the same timetable for making steps. The TSCH protocol has two components: one that operates at the application layer, and one that provides time synchronization, with the former relying upon the latter. We verify the application layer of TSCH that assumes that nodes in the system are always time-synchronized within a bound called the "guard time" which corresponds to β. Moreover, in practice, β is much smaller than σ^l and thus Δ is typically 1 for implementations of the TSCH.

4 Systems with Recurrent Logical Conditions

We now consider the case of a SAS system that does not execute on top of a layer that guarantees time synchronization (i.e., Eqn. 1 may not hold). We identify behavior of certain SAS systems, called *recurrent logical conditions*, that can be exploited for abstraction and verification. Specifically, even though AS(Δ) may not hold for the system for any finite Δ, it may still hold for *segments* of every trace of the system.

Definition 4. *(Recurrent Logical Condition) For a SAS system $\mathcal{M}_C$, a* recurrent logical condition *is a predicate $logicConv$ on the state of $\mathcal{M}_C$ such that $\mathcal{M}_C$ satisfies the LTL property* $\mathbf{G}\ \mathbf{F}\ logicConv$.

Our verification approach is based on finding a finite Δ such that, for every trace of $\mathcal{M}_C$, segments of the trace between states satisfying $logicConv$ satisfy AS(Δ). This property of system traces can then be exploited for efficient model checking.

We begin with an example of a recurrent logical condition case in the context of the IEEE 1588 protocol (Sec. 4.1). We then present our verification approach based on inferring Δ for trace segments via iterative use of model checking (Sec. 4.2).

4.1 Example: IEEE 1588 protocol

The IEEE 1588 standard [20], also known as the *precision time protocol (PTP)*, enables precise synchronization of clocks over a 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 whose purpose is two-fold: (i) to elect a unique *grandmaster clock* that is the best clock in the network, and (ii) to find a unique *spanning tree* in the network with the grandmaster clock at the root of the tree. The combination of a grandmaster clock and a spanning tree constitutes the global stable configuration known as *logical convergence* that corresponds to the recurrent logical condition. The second phase, the time synchronization phase, uses this stable configuration to synchronize or correct the physical clocks (more details in [20]).

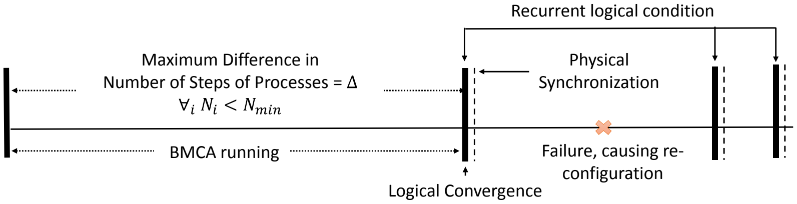
Recurrent logical condition
Maximum Difference in
Number of Steps of Processes = Δ
$\forall_i\ N_i < N_{min}$
Physical
Synchronization
BMCA running
Failure, causing re-configuration
Logical Convergence

Fig. 3. Phases of the IEEE 1588 time-synchronization protocol

Figure 3 gives an overview of the phases of the IEEE 1588 protocol execution. The distributed system starts executing the first phase (e.g., the BMC algorithm) from an initial configuration. Initially, the clocks are not guaranteed to be synchronized to within a bound β. However, once logical convergence occurs, the clocks are synchronized shortly thereafter. Once the clocks have been synchronized, it is possible for a failure at a node or link to break clock synchronization. The BMC algorithm operates continually, with the goal of ensuring that, if time synchronization is broken, the clocks will be re-synchronized. Thus, a typical 1588 protocol execution is structured as a (potentially infinite) repetition of the two phases: logical convergence, followed by clock synchronization. We exploit this *recurrent* structure to show in Sec. 4.2 that we can compute a value of Δ obeyed by segments of any trace of the system. The approach operates by iterative model checking of a specially-crafted temporal logic formula.

Note that the time taken by the protocol to logically converge depends on various factors including network topology and clock drift. In Sec. 6, we demonstrate empirically that the value of Δ depends on the number of steps (length of the segment) taken by BMCA to converge which in turn depends on factors mentioned above.

4.2 Iterative Algorithm to Compute $\varDelta$-Abstraction for Verification

Given a SAS system $\mathcal{M}_C$ whose traces have a recurrent structure, and an LTL property Φ, we present the following approach to verify whether $\mathcal{M}_C$ satisfies Φ:

1. *Define recurrent condition:* Guess a *recurrent logical condition*, $logicConv$, on the global state of $\mathcal{M}_C$.
2. *Compute $N_{\min}$:* Guess an initial value of $\varDelta$, and compute, from parameters σ^l, σ^u of the processes in $\mathcal{M}_C$, a number $N_{\min}$ such that the AS($\varDelta$) condition is satisfied on all trace segments where no process makes $N_{\min}$ or more steps. We describe the computation of $N_{\min}$ in more detail below.
3. *Verify if $\varDelta$ is sound:* Verify using model checking on $\mathcal{M}_A$ that, every trace segment that starts in an initial state or a state satisfying $logicConv$ and ends in another state in $logicConv$ satisfies AS($\varDelta$). This is done by checking that no process makes $N_{\min}$ or more steps in any such segment. Note that verifying $\mathcal{M}_A$ in place of $\mathcal{M}_C$ is sound as AS($\varDelta$) is obeyed for up to $N_{\min}$ steps from *any* state. Further details, including the LTL property checked, are provided below.
4. *Verify $\mathcal{M}_C$ using $\varDelta$:* If the verification in the preceding step succeeds, then a model checker can verify Φ on a discrete abstraction $\widehat{\mathcal{M}}_A$ of $\mathcal{M}_C$, which, similar to $\mathcal{M}_A$, is obtained by dropping physical clocks and timetables, and enforcing the AS($\varDelta$) condition to segments *between visits to logicConv*. Formally, $\widehat{\mathcal{M}}_A = (\mathcal{S}, \widehat{\delta^a}, \mathcal{I}, \textsc{id}, \rho_\varDelta)$ where $\widehat{\delta^a}$ differs from δ^a only in that for a configuration (s, N), $N'_i = 0$ for all i if $s' \in logicConv$ (otherwise it is identical to δ^a).
   However, if the verification in Step 3 fails, we go back to Step 2 and increment $\varDelta$ and repeat the process to compute a sound value of $\varDelta$.

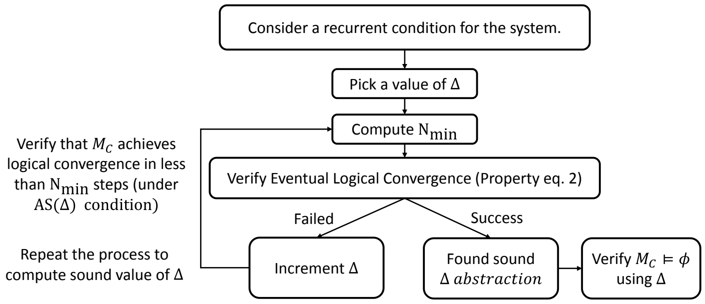
Consider a recurrent condition for the system.
Pick a value of Δ
Compute N$_{\min}$
Verify that M_C achieves logical convergence in less than N$_{\min}$ steps (under AS(Δ) condition)
Verify Eventual Logical Convergence (Property eq. 2)
Failed
Success
Repeat the process to compute sound value of Δ
Increment Δ
Found sound Δ *abstraction*
Verify $M_C \vDash \phi$ using Δ

Fig. 4. Iterative algorithm for computing $\varDelta$ exploiting logical convergence

Figure 4 depicts this iterative approach for the specific case of the BMC algorithm. We now elaborate on Steps 2 and 3 of the approach.

Step 2: Computing $N_{\min}$ for a given $\varDelta$. Recall from Sec. 2.2 that the actual step size of a process lies in the interval $[\sigma^l, \sigma^u]$. 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 3 is always satisfied if $N_f - N_s \leq \varDelta$. We wish to find the smallest number of steps taken by the fastest process when AS($\varDelta$) is violated. We denote this value as $N_{\min}$, and obtain it by formulating and solving a linear program.

Suppose first that $\mathcal{P}_s$ and $\mathcal{P}_f$ begin stepping at the same time t. Then, since the time between steps of $\mathcal{P}_f$ is at least σ^l and that between steps of $\mathcal{P}_s$ is at most σ^u, the total elapsed must be at least $\sigma_l N_f$ and at most $\sigma^u N_s$, yielding the inequality $\sigma^l N_f \leq \sigma^u N_s$.

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

$$\sigma^l N_f + \sigma^u \leq \sigma^u N_s$$

We can now set up the following integer linear program (ILP) to solve for $N_{\min}$:

$$\min N_f \quad s.t.$$

$$N_f \geq N_s, \qquad N_f - N_s > \Delta, \qquad \sigma^l N_f + \sigma^u \leq \sigma^u N_s, \qquad N_f, N_s \geq 1$$

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

Example 4. For the IEEE 1588 protocol, as described in Sec. 2.2, the actual process step sizes lie in $[0.999, 1.001]$. Setting $\Delta = 1$, solving the above ILP yields $N_{\min} = 1502$.

Step 3: Temporal Logic Property. Once $N_{\min}$ is computed, we verify on the discrete abstraction $\mathcal{M}_A$ whether, from any state satisfying $\mathcal{I} \vee logicConv$, the model reaches a state satisfying $logicConv$ in less than $N_{\min}$ steps. This also verifies that all traces in the BMC algorithm satisfy the recurrent $logicConv$ property and the segments between $logicConv$ satisfy AS(Δ). We perform this by invoking a model checker to verify the following LTL property, which references the variables N_i recording the number of steps of process $\mathcal{P}_i$:

$$(\mathcal{I} \vee logicConv) \implies \mathbf{F}\left[logicConv \wedge \left(\bigwedge_i (0 < N_i < N_{\min})\right)\right] \qquad (2)$$

We show in Sec. 5 how to implement the above check without explicitly including the N_i variables in the system state. Note that it suffices to verify the above property on the discrete abstraction $\mathcal{M}_A$ constrained by the scheduler ρ_Δ because we explore no more than $N_{\min}$ steps of any process and so $\mathcal{M}_A$ is a sound abstraction. The overall soundness result is formalized below.

Theorem 2. *If the abstract model $\mathcal{M}_A$ satisfies Property 2, then all traces of the concrete model $\mathcal{M}_C$ are traces of the model $\widehat{\mathcal{M}}_A$ (after untiming)* *(Proof in [12])*

In Sec. 6, we report on our experiments verifying properties of the BMC algorithm by model checking the discrete abstract model $\widehat{\mathcal{M}}_A$ as described above.

5 Model Checking with Approximate Synchrony

We implemented approximate synchrony within ZING [4], an explicit state model checker. ZING performs a "constrained" asynchronous composition of processes, using an external scheduler to guide the interleaving. Approximate synchrony is enforced by an external scheduler that explores only those traces satisfying AS(Δ) by scheduling, in each state, only those processes whose steps will not violate AS(Δ).

Section 4 described an iterative approach to verify whether a Δ-abstract model of a protocol is sound. The soundness proof depends on verifying Property 2. A naïve approach for checking this property would be to include a local variable N_i in each process as part of the process state to keep track of the number of steps executed by each process, causing state space explosion. Instead, we store the values of N_i for each i external to the system state, as a part of the model checker explorer.

The algorithm in Fig. 5 performs systematic bounded depth first search for a state $s_{initial}$, belonging to the set of all possible initial states. To check whether all traces of length $N_{\min}$ satisfy eventual logical convergence under AS(Δ) constraint, we enforce two bounds: first, the final depth bound is $(N_{\min} + \Delta)$ and second, in each state a process is enabled only if executing that process does not violate AS(Δ). If a state satisfies $logicConv$ then we terminate the search along that path.

The BoundedDFS function is called recursively on each successor state and it explore only those traces that satisfy AS(Δ). If the steps executed by a process is N_{min} then the $logicConv$ monitor is invoked to assert if $s' \models logicConv$ (i.e. we have reached logical convergence state) and if the assertion fails we increment the value of Δ as described in Sec. 4.2. $N_{\min}$ and Δ values are derived as explained in Sec. 4.2.

```
**var** $StateTable : Dictionary\langle State, List\langle int\rangle\rangle$;
$BoundedDFS(s : State)$ {
    **var** $i : int,\ s' : State,\ steps' : List\langle int\rangle$;
    $i := 0$;
    **while** ($i <$#Processes($s$)){
        $steps'$ :=IncElement($i, StateTable[s]$);
        **if** $\neg$ CheckASCond($steps'$)
            $\vee\ steps'[i] > (N_{\min} + \Delta)$
            $\vee\ s \models logicConv$ **then**
            **continue** ;
        $s'$ :=NextState($s, i$);
        **if** $steps'[i] = N_{min}$ **then**
            $assert(s' \models logicConv)$;
        **if** $s' \notin Domain(StateTable)$
            $\vee \neg(steps' \geq_{pt} StateTable[s'])$ **then**
            $StateTable[s'] := steps'$;
            BoundedDFS($s'$);
        $i := i + 1$; } }

Verify() {
    $StateTable[s_{initial}] = newList\langle int\rangle$;
    BoundedDFS($s_{initial}$);}
```

Fig. 5. Algorithm for Verification of Property 2

$StateTable$ is a map from reachable state to the tuple of steps with which it was last explored. $steps'$ is the vector of number of steps executed by each process and is stored as a list of integers. #Processes(s) returns the number of enabled processes in the state s. IncElement(i, t) increments the i^{th} element of tuple t and returns the updated tuple. CheckASCond($steps'$) checks the following condition that $\forall s_1, s_2 \in steps'\ |s_1 - s_2| \leq \Delta$.

To avoid re-exploring a state which may not lead to new states, we do not re-explore a state if it is revisited with $steps'$ greater than what it was last visited with. The operator $\geq_{pt}$ does a pointwise comparison of the integer tuples. We show in the following section that we are able to obtain significant state space reduction using this implementation.

6 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 in IEEE 1588 and (2) the time synchronized channel hopping protocol in IEEE 802.15.4e.
- Evaluate if we can verify properties that cannot be verified with full asynchrony (either by reducing state space or by capturing relevant timing constraints).
- Evaluate approximate synchrony as an iterative bounding technique for finding bugs efficiently in almost-synchronous systems.

6.1 Modeling and Experimental Setup

We model the system in P [11], a domain-specific language for writing 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]. 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. Faults and message losses in the protocol are modeled as non-deterministic choices.

Protocol	Temporal Property	Description
BMCA	$\mathbf{F}\ \mathbf{G}\ (logicConv)$	Eventually the BMC algorithm stabilizes with a unique spanning tree having the grandmaster at its root. The system is said to be in $logicConv$ state when the system has converged to the expected spanning tree.
TSCH	$\bigwedge_{i \in n} \mathbf{G}(\neg desynched_i)$	A node in TSCH is said to be *desynched* - if it fails to synchronize with its master within the threshold period. The desired property of a correct system is that the nodes are always synchronized.

Table 1. Temporal properties verified for the case studies

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

6.2 Verification and Testing using Approximate Synchrony

We applied approximate synchrony in three different contexts : (1) Time synchronized Channel Hopping protocol (*time synchronized system*) (2) Best Master Clock Algorithm in IEEE 1588 (*exploiting recurrent logical condition*) (3) Approximate Synchrony as a bounding technique for finding bugs.

Verification of the TSCH Protocol. Time Synchronized Channel Hopping (TSCH) is a Medium Access Control scheme that enables low power operations in wireless sensor network using time-synchronization. It makes an assumptions that the clocks are always time-synchronized within a bound, referred to as the 'guard' time in the standard. The low power operation of the system depends on the sensor nodes being able to maintain synchronization (desynchronization property in Table 1). A central server broadcasts the global schedule that instructs each sensor node when to perform operations. Whether the system satisfies the desynchronization property depends on this global schedule, and the standard provides no recommendation on these schedules.

We modeled the TSCH as a SAS system and used Theorem 1 to calculate the value of Δ [5]. We verified the desynchronization property (Table 1) in the presence of failures like message loss, interference in wireless network, etc. For the experiments we considered three schedules (1) round-robin: nodes are scheduled in a round robin fashion, (2) shared with random back-off: all the schedule slots are shared and conflict is resolved using random back-off (3) Priority Scheduler: nodes are assigned fixed priority and conflict is resolved based on the priority.

We were able to verify if the property was satisfied for a given topology under the global schedule, and generated a counterexample otherwise (Table 2) which helped the TSCH system developers in choosing the right schedules for low power operation. Using sound approximate synchrony abstraction (with $\Delta = 1$), we could accurately capture the "almost synchronous" behavior of the the TSCH system.

[5] For system of nodes under consideration, the maximum clock skew, $\epsilon = 120\mu s$ and nominal step size of $100ms$, the value of $\Delta = 1$

Verification of BMC Algorithm											
Network Topology (#Nodes)	Safety Property							Convergence Property			
	Fully Asynchronous Model			Model with Approximate Synchrony				Model with Approximate Synchrony			
	States Explored	Time (h:mm)	Property Proved	Δ	States Explored	Time (h: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	1	5.3 E+8	6:33	Yes
Star(5)	2.4 E+10	9:40	Yes	1	5.8 E+5	0:54	Yes	1	4.1 E+7	5:10	Yes
Random(5)	9.19 E+9	9:01	Yes	2	5.5 E+6	1:44	Yes	2	1.8 E+9	9:10	Yes
Ring(5)	7.1 E+12*	*	No	1	4.8 E+7	3:44	Yes	1	8 E+9	8:04	Yes
Linear(7)	1.4 E+13*	*	No	1	4.6 E+7	3:05	Yes	1	1.0 E+8	6:21	Yes
Star(7)	1.1 E+13*	*	No	2	3.7 E+8	5:06	Yes	2	3.3 E+10	13:34	Yes
Ring(7)	3.3 E+12*	*	No	2	6.8 E+8	8:04	Yes	2	2.1 E+10	11:11	Yes
Random(6)	1.1 E+13*	*	No	3	5.7 E+9	6:00	Yes	3	1.3 E+10	10:34	Yes
Random(7)	1.1 E+13*	*	No	3	8.1 E+8	7:11	Yes	3	9.9 E+10	10:11	Yes

Verification of TSCH Protocol									
Network Topology (#Nodes)	Round-Robin Scheduler			Shared with CSMA			Priority Scheduler		
	States Explored	Time (h:mm)	Property Satisfied	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	2.4E +3$^{\#}$	0:09	No
Random(5)	3.6 E+2$^{\#}$	0:05	No	6.2 E+3$^{\#}$	0:12	No	1.9E +6	0:35	Yes
Mesh(5)	1.7 E+7	4:05	Yes	9.1 E+6	2:01	Yes	9.3 E+5	0:31	Yes

** denotes end of exploration as model checker ran out of memory, # denotes property violated and counter example is reported*

Table 2. Verification results using Approximate Synchrony.

Verification of BMC Algorithm. The BMC algorithm is a core component of the IEEE 1588 precision time protocol. It is a distributed fault tolerant protocol where nodes in the system perform operations periodically to converge on a unique hierarchical tree structure, referred to as the *logical convergence* state in Sec. 4. Note that the convergence property for BMCA holds *only in the presence of almost synchrony* — it does not guarantee convergence in the presence of unbounded process delay or message delay. Hence, it is essential to verify BMC using the right form of synchrony.

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 2 for the BMC algorithm are for 5 and 7 nodes in the network with linear, star, ring, and random topologies. The Δ value used for verification of each of these configurations was derived by using the iterative approach described in Sec. 4.2. The results demonstrate that the value of Δ required to construct the sound abstraction varies depending on network topology, and clock dynamics. Table 2 shows the total number of states explored and time taken by the model checker for proving the safety and convergence property (Table 1) using the sound Δ-abstract model. Approximate synchrony abstraction is orders of magnitude faster as it explores the reduced state-space. BMCA algorithm satisfies safety invariant even in the presence of complete asynchrony. For demonstrating the efficiency of using approximate synchrony we also conducted the experiments with complete asynchronous composition, exploring all possible interleaving (for safety properties). The complete asynchronous model is simple to implement but fails to prove the properties for most of the topologies.

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, networks of this size do occur in practice, e.g., in industrial automation where one has small teams of networked robots on a factory floor.

Endlessly circulating (rogue) frames in IEEE 1588: The possibility of an endlessly circulating frame in a 1588 network has been debated for a while in the standards committee. Using formal model of BMC algorithm under approximate synchrony, we were able to reproduce a scenario were rogue frame could occur. Existence of a rogue frame can lead to network congestion or cause the BMC algorithm to never converge. The counter example was cross-validated using simulation and is described in detail in [6]. It was well received by the IEEE 1588 standards committee.

Buggy Models	Iterative Depth Bounding with Random Search			Non-Iterative AS			Iterative AS		
	Depth	States Explored	Time (h:mm)	Δ	States Explored	Time (h:mm)	Δ	States Explored	Time (h:mm)
BMCA_Bug_1	51	1.4 E+3	0:05	2	1.1 E+3	0:04	0	2.1 E+2	0:02
BMCA_Bug_2	64	5.9 E+5	0:15	2	6.1 E+4	0:14	0	1.6 E+3	0:04
BMCA_Bug_3	101	9.4 E+7	0:45	3	3.3 E+5	0:17	1	9.1 E+2	0:05
ROGUE_FRAME_Bug_1	44	3.9 E+5	0:18	2	9.7 E+6	0:29	1	5.6 E+4	0:12
ROGUE_FRAME_Bug_2	87	4.4 E+4	0:09	2	2.1 E+3	0:05	1	1.1 E+3	0:03
SPT_Bug_1	121	8.4 E+8	1:05	3	8.1 E+4	0:11	0	5.5 E+2	0:04

Table 3. Iterative Approximate Synchrony with bound Δ for finding bugs faster.

Approximate Synchrony as a Search Prioritization Technique. Approximate synchrony can also be used as a bounding technique to prioritize search. We collected buggy models during the process of modeling the BMC algorithm and used them as benchmarks, along with buggy instance of the Perlman's Spanning Tree Protocol [24] (SPT). We used AS as an iterative bounding technique, starting with $\Delta = 0$ and incrementing Δ after each iteration. For $\Delta = 0$, the model checker explores only synchronous system behaviors. Increasing the value could be considered as adding bounded asynchronous behaviors incrementally. Table 3 shows comparison between iterative AS, non-iterative AS with fixed value of Δ taken from Table 2 and iterative depth bounding with random search. Number of states explored and the corresponding time taken for finding the bug is used as the comparison metric. Results demonstrate that most of the bugs are found at small values of Δ (hence iterative search is beneficial for finding bugs). Some bugs like the rogue frame error, that occur only when there is asynchrony were found with minimal asynchrony in the system ($\Delta = 1$). These results confirm that prioritizing search based on approximate synchrony is beneficial in finding bugs. Other bounding techniques such as delay bounding [15] and context bounding [23] can be combined with approximate synchrony but this is left for future work.

7 Related Work

The concept of *partial synchrony* has been well-studied in the theory of distributed systems [14,13,25]. There are many ways to model partial synchrony depending on the type of system and the end goal (e.g., formal verification). Approximate synchrony is one such approach, which we contrast against the most closely-related work below.

Hybrid/Timed Modeling: The choice of modeling formalism greatly influences the verification approach. A time-synchronized system 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 [17], cannot explore the entire state space.

There has been work on modeling timed protocols using real-time formalisms such as *timed automata* [3], where the derivatives of all continuous-time variables are equal to one. While tools based on the theory of timed automata do not explicitly support modeling and verification of multi-rate timed systems [21], there do exist techniques for approximating multirate clocks. For instance, Huang *et al.* [19] propose the use of *integer clocks* on top of UPPAAL models. Daws and Yovine [9] show how multirate timed

systems can be over-approximated into timed automata. Vaandrager and Groot [29] models a clock that can proceed with different rate by defining a clock model consisting of one location and one self transition. Such models only approximately represent multirate time systems. 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 *sound* untimed model that can be directly checked by a finite-state model checker.
Synchrony and Asynchrony: There have been numerous efforts devoted towards mixing synchronous and asynchronous modeling. Multiclock Esterel [26] and communicating reactive processes (CRP) [5] extend the synchronous language Esterel to support a mix of synchronous and asynchronous processes. *Bounded asynchrony* is another such modeling technique with applications to biological systems [16]. It can be used to model systems in which processes can have different but *constant* rates, and can be interleaved asynchronously (with possible stuttering) before they all synchronize at the end of a global "period." Approximate synchrony has no such synchronizing global period. The *quasi-synchronous (QS)* [7,18] approach is designed for communicating processes that are *periodic* and have almost the *same* period. QS [18] is defined as "Between any two successive activations of one period process, the process on any other process is activated either 0, 1, or at most 2 times". As a consequence, in both quasi-synchrony and bounded asynchrony, the difference of the absolute number of activations of two different processes can grow unboundedly. In contrast, the definition of AS does not allow this difference to grow unbounded.

8 Conclusion

This paper has introduced two new concepts: a class of distributed systems termed as *symmetric, almost-synchronous* (SAS) systems, and *approximate synchrony*, an abstraction method for such systems. We evaluated applicability of approximate synchrony for verification in two different contexts: (i) application-layer protocols running on top of time-synchronized systems (TSCH), and (ii) systems that do not rely on time synchronization but exhibit recurrent logical behavior (BMC algorithm). We also described an interesting search prioritization technique based on approximate synchrony with the key insight that, prioritizing synchronous behaviors can help in finding bugs faster.

In this paper, we focus on verifying protocols that fit the SAS formalism defined in Sec. 2.2. While other protocols whose behavior and correctness relies on using values of timestamps do not natively fit into the SAS formalism, they can be abstracted using the suitable methods (e.g., using a state variable to model a local timer for a process whose value is incremented on each step of that process — with approximate synchrony the timer values across different processes will not differ by more than Δ). Evaluating such abstractions for protocols like Google Spanner and others that use timestamps would be an interesting next step.

Acknowledgments. The first and second authors were supported in part by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. The fourth author was supported in part by the Swedish Research Council (#623-2013-8591) and the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by IBM and United Technologies).

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. K. 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, A. Desai, J. C. Eidson, and S. A. Seshia. Endlessly circulating messages in IEEE 1588-2008 systems. In *Proceedings of the 8th International IEEE Symposium on Precision Clock Synchronization for Measurement, Control and Communication (ISPCS)*, September 2014.
7. P. Caspi, C. Mazuet, and N. R. Paligot. About the design of distributed control systems: The quasi-synchronous approach. In *SAFECOMP*. 2001.
8. J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of OSDI 2012*.
9. C. Daws and S. Yovine. Two examples of verification of multirate timed automata with Kronos. In *Proceedings of RTSS*, 1995.
10. A. Desai, D. Broman, J. Eidson, S. Qadeer, and S. A. Seshia. Approximate synchrony: An abstraction for distributed time-synchronized systems. Technical Report UCB/EECS-2014-136, University of California, Berkeley, Jun 2014.
11. 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.
12. A. Desai, S. A. Seshia, S. Qadeer, D. Broman, and J. Eidson. Approximate synchrony: An abstraction for distributed almost-synchronous systems. Technical Report UCB/EECS-2015-158, EECS Department, University of California, Berkeley, May 2015.
13. D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, Jan. 1987.
14. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 1988.
15. M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded scheduling. In *Proceedings of POPL*, 2011.
16. J. Fisher, T. A. Henzinger, M. Mateescu, and N. Piterman. Bounded asynchrony: Concurrency for modeling cell-cell interactions. In *FMSB*. 2008.
17. 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.
18. N. Halbwachs and L. Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *Proceedings of ACSD*, 2006.
19. X. Huang, A. Singh, and S. A. Smolka. Using Integer Clocks to Verify the Timing-Sync Sensor Network Protocol. In *Proceedings of NFM*, 2010.
20. IEEE Instrumentation and Measurement Society. *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, 2008.
21. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on STTT*, 1997.
22. 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.
23. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of PLDI*, 2007.
24. R. Perlman. An algorithm for distributed computation of a spanning tree in an extended LAN. In *Proceedings of SIGCOMM*, 1985.

25. S. Ponzio and R. Strong. Semisynchrony and real time. In A. Segall and S. Zaks, editors, *Distributed Algorithms*, volume 647 of *Lecture Notes in Computer Science*, pages 120–135. Springer Berlin Heidelberg, 1992.
26. B. Rajan and R. Shyamasundar. Multiclock esterel: a reactive framework for asynchronous design. In *IPDPS*, 2000.
27. B. Sundararaman, U. Buy, and A. D. Kshemkalyani. Clock synchronization for wireless sensor networks: A survey. *Ad Hoc Networks (Elsevier*, 2005.
28. A. Tinka, T. Watteyne, and K. Pister. A decentralized scheduling algorithm for time synchronized channel hopping. In *Second International Conference on Ad-Hoc Networks (ADHOCNETS)*, pages 201–216. Springer, 2010.
29. F. W. Vaandrager and A. de Groot. Analysis of a biphase mark protocol with Uppaal and PVS. *Formal Aspects of Computing*, 2006.