An Abstraction Technique for Parameterized Model Checking of Leader Election Protocols: Application to FTSP

Ocan Sankur[1](✉) and Jean-Pierre Talpin[2]

[1] CNRS, Irisa, Rennes, France
ocan.sankur@irisa.fr
[2] Inria, Rennes, France

Abstract. We consider distributed timed systems that implement leader election protocols which are at the heart of clock synchronization protocols. We develop abstraction techniques for parameterized model checking of such protocols under arbitrary network topologies, where nodes have independently evolving clocks. We apply our technique for model checking the root election part of the flooding time synchronisation protocol (FTSP), and obtain improved results compared to previous work. We model check the protocol for all topologies in which the distance to the node to be elected leader is bounded by a given parameter.

1 Introduction

One of the apparently simplest services in any loosely-coupled distributed system is the time service. Usually, a client in such a system, e.g. your laptop, simply posts an NTP (network time protocol) request to any registered server and uses the first reply. In many such systems, however, the accuracy and reliability of the time service are critical: clients of traffic and power grids, banking and transaction networks, automated factories and supply plants, acutely depend on a reliable and accurate measure of time.

To make things worse, most cyber-physical system in such distributed networks rely on a quasi-synchronous hypothesis that critically relies on drift and jitter bounds provided by time synchronisation protocols. In a remedy for this Achille's heel of the "Internet of things", fault-tolerant and self-calibrating protocols have been proposed, such as the open source *flooding time synchronisation protocol (FTSP)* of Tiny OS, Google's True Time API, as well as commercial solutions, such as IGS' Real-Time Service. It is critical to provide such services to the 21st Century's Internet as is it to provide proof of their correctness.

Our goal is to develop both modular and scalable verification techniques for time synchronisation protocols. Towards this aim, in this paper, we concentrate on leader election protocols which are at the basis of several time synchronisation protocols where the nodes in the network synchronise their clocks to that of the elected leader. Leader election protocols pose exciting benchmarks and case

A. Legay and T. Margaria (Eds.): TACAS 2017, Part I, LNCS 10205, pp. 23–40, 2017.
DOI: 10.1007/978-3-662-54577-5_2

studies to the verification of distributed systems design. These have been the subject of formal proofs or model-checking, *e.g.* Chang-Robert's algorithm [6,15], and that of Dolev-Klaweh-Rodeh [13,17].

The root election part of FTSP [20], available in open-source in the implementation of Tiny OS, has drawn attention from the formal verification community. Kusy and Abdelwahed [19] model-check FTSP root election using SPIN, showing that a 4-node FTSP network is guaranteed to converge to a single root node. McInnes [21] verifies root-convergence for 7-node models using the FDR2 model checker, and also considers time-convergence properties, *i.e.* whether all nodes agree on the time of the root node. Tan et al. [28] use timed automata to introduce a more realistic simulation model of wireless sensor networks (WSN) with transmission delays and node failures and check the FTSP against these. They identify an error in a scenario where two root nodes fail continuously.

Parameterized Verification. The major issue when model checking such distributed protocols is the state explosion problem due to the large number of nodes in the protocol. Several works have concentrated on given network topologies, for instance, a grid of fixed size, *e.g.* [21]. To model check properties for an arbitrary number of nodes, parameterized verification techniques have been considered. Although the general problem is undecidable [2], decidability has been shown in several cases, by proving cutoffs [14] either on fully connected topologies or particular ones such as rings. Compositional model checking techniques were used in [22] for model checking a cache coherence protocol.

Contributions. We present an abstraction technique for the parameterized verification of distributed protocols with unique identifiers and apply it for model checking the leader election part of the FTSP. Our model for FTSP is more precise compared to the previous works in several aspects. In fact, we consider asynchronous communication between nodes rather than instantaneous broadcasts, and we model the periodically executed tasks as run with local clocks that are subject to imperfections. We were able to model check that a unique leader is elected starting at an *arbitrary* configuration, assuming no fault occurs during this period. This corresponds to checking fault recovery, that is, proving the protocol correct following an arbitrary fault. Thus, if we prove that the leader is elected within N steps in this setting, then following any fault, a unique leader is elected again within N steps in the worst case.

Our parameterized verification algorithm allows us to check FTSP (a) for *arbitrary* topologies in which the maximal distance to the future leader is at most K, (b) where each node evolves under clock deviations whose magnitude can be adjusted, (c) where communication between nodes are either synchronous or asynchronous. As an example, we were able to model check the protocol for $K = 7$ in the synchronous case, and for $K = 5$ in the asynchronous case. Graphs with $K = 7$ include 2D grids with 169 nodes (or 3D grids with 2197 nodes), where the future leader is at the middle. For $K = 5$, these include 2D grids with 81 nodes (and 729 in 3D). Observe that grids of size 60 were considered for simulation in [20], which is out of the reach of previous model checking attempts.

We believe our parameterized verification technique can be adapted to other distributed protocols that work in a similar fashion, *e.g.* [29]. Our project is to extend our technique by integrating non-functional characteristics that have an impact on the accuracy and reliability of these protocols: electronic hazards (inaccuracy in physical clocks fabric), environmental hazards (temperature of clients environment), power hazards (capacity and stability of clients power source). Protocols accounting for such cyber-physical characteristics are being developed in the NSF Roseline and our goal is to prove their correctness.

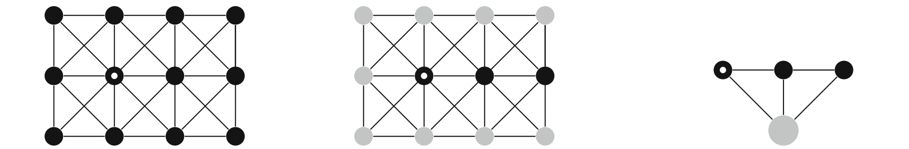
More on Related Work. Our parameterized verification approach is inspired by [7] where an abstraction technique is given for parameterized model checking against *safety* properties in cache coherence protocols. Using the fact that such systems are *symmetric*, the main idea is to isolate a pair of nodes and abstract away other nodes as an abstract environment. In our work, the systems we consider are not symmetric since the nodes have unique identifiers which influence their behaviors and the network topology is arbitrary. We thus deal with these issues in order to lift the technique in our case. Another work introduces a refinement of existential abstraction for parameterized model checking: in [8], an abstraction is obtained by isolating a component, and abstracting away the other nodes by summarizing which control states are occupied by some component, which is similar to counter abstraction [25]. Parameterized verification techniques have been studied for fault-tolerant distributed systems with Byzantine or other types of failures [16]. Such protocols often consider *threshold guards*, which are used to make sure that a given number of messages have been received from different processes. The authors define abstractions on the set of participating nodes with predicates that use these thresholds. This approach is not applicable in our case due to our network topologies, and that the nodes do not use such thresholds. Parameterized verification results on processes with unique identifiers are more rare but decidability was obtained under some restrictions [11].

Overview of the Abstraction Technique. Let us give an overview of our parameterized verification technique. Let us call *future leader* the node that is expected to become the leader. We consider classes of graphs $\mathcal{G}_K$ in which the maximal distance from the future leader is K. We show how to verify the protocol for *all* network topologies in $\mathcal{G}_K$, for given K, essentially in two steps:

1. We apply abstractions on local variables including node identifiers, which reduce the state spaces and renders all nodes anonymous except for the future leader. In fact, the variables storing node ids are mapped to a Boolean domain; encoding whether the node id is that of the future leader or not.
2. We then pick a shortest path of length K from the future leader. We derive an abstract model where all nodes that appear on this path are kept as concrete, but all other nodes have been abstracted away.

For each K, we thus construct a model $\mathcal{A}(K)$ and prove that it is an over-approximation of the protocol on *all* topologies in $\mathcal{G}_K$. We make sure that $\mathcal{A}(K)$ does not depend on the choice of the shortest path; if the property holds on $\mathcal{A}(K)$, it holds on the whole network. The approach is illustrated in Fig. 1.

Clock Deviations. We are interested in protocols where each node executes a periodic action with identical period. However, this period is subject to small



(a) A network with a grid topology. The future root, that is, the node with the smallest ID is shown with a white dot.

(b) We choose a path from the future root to some node

(c) We model all nodes on the path concretely, and summarize the behavior of all other nodes by one abstract node

Fig. 1. Shortest-path abstraction illustrated on a grid topology with $K = 3$.

deviations due to environment and hardware differences. Rather than using real-time verification techniques [1], we use a recent and simple way of modeling behaviors under such conditions. In [12], it is shown that an *approximately synchronous* semantics, where one bounds the progress of each process with respect to that of others, over-approximates the behaviors under bounded clock deviations, which makes it possible to use finite-state model checking techniques and tools.

Incremental Verification. We use an incremental proof technique for model checking $\mathcal{A}(K)$ for increasing values of K, as follows. To check $\mathcal{A}(K+1)$, we first model check $\mathcal{A}(K)$, proving that all nodes eventually agree on the leader. Our abstraction method implies that the first K components in $\mathcal{A}(K+1)$ eventually agree on the leader since their part of the graph belongs to $\mathcal{G}_K$. Thus, to check $\mathcal{A}(K+1)$, we initialize the first K nodes at states where they have agreed on the future leader. This significantly simplifies the verification process.

Overview. Section 2 presents definitions for the formalization of our approach. We describe FTSP in detail in Sect. 3, as well as the abstraction steps explained above, and the incremental verification result. A semi-algorithm for model checking and experimental results on FTSP are presented in Sect. 4.

2 Definitions

Communicating Processes. A *process* is an automaton $\mathcal{A} = (S, s_{\mathsf{init}}, \delta, \Sigma)$ where S are states, $s_{\mathsf{init}} \subseteq S$ are the initial states, and $\delta \subseteq S \times \Sigma \times S$ a transition relation, with alphabet Σ. A transition $(s, a, s') \in \delta$ is also written $\delta(s, a, s')$ or $s \xrightarrow{a} s'$, and we write $s \stackrel{a}{\not\rightarrow}$ to mean that there is no s' such that $\delta(s, a, s')$. We consider predicates that are evaluated on the states of a given process. Let $\mathcal{P}$ be a finite number of predicates where each $p \in \mathcal{P}$ is a subset $p \subseteq S$, representing states in which the predicate is satisfied. We write $s \models p$ if $s \in p$.

We define *simulation* between two processes as follows. Consider process $\mathcal{A} = (S, s_{\mathsf{init}}, \delta, \Sigma)$ with predicates $\mathcal{P}$ and $\mathcal{A}' = (S', s'_{\mathsf{init}}, \delta', \Sigma')$ with predicates $\mathcal{P}'$, an alphabet $\Sigma'' \subseteq \Sigma$, and any function $\alpha : \Sigma'' \rightarrow \Sigma'$. Assume that $\mathcal{P}$ and $\mathcal{P}'$ are in bijection denoted by $p \mapsto p'$ for each $p \in \mathcal{P}$. We say that $\mathcal{A}'$ (Σ'', α)-*simulates* $\mathcal{A}$, written $\mathcal{A} \sqsubseteq_{\Sigma'',\alpha} \mathcal{A}'$ if there exists $R \subseteq S \times S'$ such that $s_{\mathsf{init}} \times s'_{\mathsf{init}} \subseteq R$ and $\forall (s,s') \in R, \forall a \in \Sigma'', t \in S, \delta(s,a,t) \Rightarrow \exists t' \in S', \delta'(s', \alpha(a), t') \wedge (t,t') \in R$, and moreover for all $(s,s') \in R$ and $p \in \mathcal{P}$, $s \models p \Leftrightarrow s' \models p'$. When α is the identity and $\Sigma'' = \Sigma$, this is the usual simulation notion, and we write $\sqsubseteq_{\Sigma''}$. Given a process $\mathcal{A}$, let us define *the mapping of* $\mathcal{A}$ *by* α the process obtained by $\mathcal{A}$ by replacing the transitions δ by $\delta' = \{(s, \alpha(a), s') \mid (s,a,s') \in \delta\}$. It is clear that the mapping $\mathcal{A}'$ (Σ, α)-simulates $\mathcal{A}$.

For any positive integer N, we write $\mathcal{A} \sqsubseteq^N_{\Sigma'',\alpha} \mathcal{A}'$ if there exist $R_1, \ldots, R_N \subseteq S \times S'$ such that $s_{\mathsf{init}} \times s'_{\mathsf{init}} \subseteq R_1$ and for all $1 \leq i \leq N-1$, $\forall (s,s') \in R_i, \forall a \in \Sigma'', t \in S, \delta(s,a,t) \Rightarrow \exists t' \in S', \delta'(s', \alpha(a), t') \wedge (t,t') \in R_{i+1}$; and for all $(s,s') \in R_1 \cup \ldots \cup R_n$, $s \models p \Leftrightarrow s' \models p'$. The latter relation is called *simulation up to* N.

We define a particular alphabet Σ to model synchronization by rendez-vous. Let us fix $n > 0$, and define the set of *identifiers* $\mathsf{Id} = \{1, \ldots, n\}$. Consider also an arbitrary set Msg of message contents. We denote $[1,n] = \{1, \ldots, n\}$. We define the alphabet $\Sigma(\mathsf{Id}, \mathsf{Msg}) = \{i!(j,m) \mid i \in \mathsf{Id}, j \in \mathsf{Id}, m \in \mathsf{Msg}\} \cup \{j?(i,m) \mid i,j \in \mathsf{Id}, m \in \mathsf{Msg}\} \cup \{\tau\}$. We let $\Sigma = \Sigma(\mathsf{Id}, \mathsf{Msg})$. We will later use different sets Id and Msg to define alphabets. Intuitively, the label $i!(j,m)$ means that a process with id i sends message m to process with id j, while $j?(i,m)$ means that process j receives a message m from process i. The special symbol τ is an internal action. For a subset $I \subseteq \mathsf{Id}$, let $\Sigma_I(\mathsf{Id}, \mathsf{Msg}) = \{\tau\} \cup \{i!(j,m), i?(j,m) \in \Sigma(\mathsf{Id}, \mathsf{Msg}) \mid i \in I, j \in \mathsf{Id}, m \in \mathsf{Msg}\}$. These are the actions where the senders and receivers have ids in I. A τ-*path* of $\mathcal{A}$ is a sequence $s_1 s_2 \ldots$ of states such that for all $i \geq 1$, $\delta(s_i, \tau, s_{i+1})$. An *initialized* τ-path is such that $s_1 \in s_{\mathsf{init}}$.

Graphs. To formalize network topologies, we consider undirected graphs. A graph is a pair $G = (V,E)$ with $V = \{1, \ldots, n\}$ and $E \subseteq V \times V$ which is symmetric. Let $\mathcal{G}(n)$ the set of graphs on vertex set $\{1, \ldots, n\}$. In our setting, a node will be identified with a process id. For a graph $G = (V,E)$, and node i, let $\mathcal{N}_G(i) = \{j \in V, (i,j) \in E\}$, the *neighborhood* of i. We define the following subclass of graphs. For any positive number $K \geq 0$, let $\mathcal{G}_K(n)$ denote the set of graphs of $\mathcal{G}(n)$ in which the longest distance between node 1 and any other node is at most K. Here, distance is the length of the shortest path between two nodes.

Asynchronous Product. We now define the product of two processes $\mathcal{A}$ and $\mathcal{A}'$ following CCS-like synchronization [23]. Intuitively, processes synchronize on send $i!(j,m)$ and receive $j?(i,m)$, and the joint transition becomes a τ-transition.

Definition 1. *Consider* $\mathcal{A} = (S, s_{\mathsf{init}}, \delta, \Sigma_J(\mathsf{Id}, \mathsf{Msg}))$ *and* $\mathcal{A}' = (S', s'_{\mathsf{init}}, \delta', \Sigma_{J'}(\mathsf{Id}, \mathsf{Msg}))$ *where* $J, J' \subseteq \{1, \ldots, n\}$ *with* $J \cap J' = \emptyset$. *Let* $G = (V,E) \in \mathcal{G}(n)$. *We define the product* $\mathcal{A}'' = \mathcal{A} \parallel^G \mathcal{A}'$ *as* $(S'', s''_{\mathsf{init}}, \delta'', \Sigma_{J \cup J'})$ *where* $S'' = S \times S'$, $s''_{\mathsf{init}} = s_{\mathsf{init}} \times s'_{\mathsf{init}}$, *and* δ'' *is defined as follows. There are four types of transitions.*

Internal transitions *are defined by* $(s_1, s'_1) \xrightarrow{\tau} (s_2, s'_2)$ *whenever* $\delta(s_1, \tau, s_2) \wedge s'_1 = s'_2$ *or* $\delta'(s'_1, \tau, s'_2) \wedge s_1 = s_2$.

Synchronizing transitions *are defined as* $(s_1, s_1') \xrightarrow{\tau} (s_2, s_2')$ *whenever* $\exists i \in J, j \in J', m \in \mathsf{Msg}$ *with* $i \in \mathcal{N}_G(j)$*, s.t. either* $s_1 \xrightarrow{i!(j,m)} s_2$ *and,* $s_1' \xrightarrow{j?(i,m)} s_2'$*; or,* $s_1' \xrightarrow{j!(i,m)} s_2'$*, and* $s_1 \xrightarrow{i?(j,m)} s_2$*.*

Sending transitions without matching receive *is defined as* $(s_1, s_1') \xrightarrow{i!(j,m)} (s_2, s_2')$ *whenever* $i \in J, j \notin J', m \in \mathsf{Msg}, i \in \mathcal{N}_G(j)$ *s.t. either* $s_1 \xrightarrow{i!(j,m)} s_2, s_1' = s_2'$*; or,* $i \in J', j \notin J, s_1' \xrightarrow{i!(j,m)} s_2', s_1 = s_2$*.*

Receive transitions without matching send *are defined, for all* $i, j \in \mathsf{Id}$ *and* $m \in \mathsf{Msg}$*,* $(s_1, s_1') \xrightarrow{i?(j,m)} (s_2, s_2')$ *whenever* $i \in \mathcal{N}_G(j)$ *and either* $i \in J$*,* $j \notin J', s_1 \xrightarrow{i?(j,m)} s_2, s_1' = s_2'$*, or* $i \in J', j \notin J, s_1' \xrightarrow{i?(j,m)} s_2', s_1 = s_2$*.*

The composition operation $\|^G$ is commutative and associative by definition. We will thus write the product of several processes as $\mathcal{A}_1 \|^G \ldots \|^G \mathcal{A}_n$, or $\|^G_{i=1\ldots n} \mathcal{A}_i$.

Predicates and LTL Satisfaction. We will use LTL for our specifications [24] which use the predicates $\mathcal{P}$ we consider for our model. We assume the reader is familiar with this logic, and refer to [10,24] otherwise. We just need the *eventually* (F), and *globally* (G) modalities. Given an LTL formula ϕ, we write $\mathcal{A} \models \phi$ if all initialized τ-paths satisfy ϕ.

Abstractions and Simulation. A *label abstraction function* is defined by $\alpha : \mathsf{Id} \to \mathsf{Id}^\sharp$, and $\alpha : \mathsf{Msg} \to \mathsf{Msg}^\sharp$[1]. This function is uniquely extended to $\Sigma(\mathsf{Id}, \mathsf{Msg})$ by $\alpha(\tau) = \tau, \quad \alpha(i!(j,m)) = \alpha(i)!(\alpha(j), \alpha(m))$, and $\alpha(i?(j,m)) = \alpha(i)?(\alpha(j), \alpha(m))$. We will see examples of label abstractions later in this paper.

Lemma 1. *Let* $\mathcal{A}_i = (S_i, s^i_{\mathsf{init}}, \delta_i, \Sigma_{J_i}(\mathsf{Id}, \mathsf{Msg}))$ *for* $i \in [1,n]$*, with pairwise disjoint* $J_i \subseteq \mathsf{Id}$*, and* $G \in \mathcal{G}(m)$ *with* $\cup_i J_i \subseteq \{1, \ldots, m\}$*. Consider a label abstraction function* α*, s.t.* $\alpha(J_i) \cap \alpha(J_j) = \emptyset$ *for all* $i \neq j \in [1,n]$*; and mappings* $\mathcal{A}_i'$ *of* $\mathcal{A}_i$ *by* α *so that* $\mathcal{A}_i \sqsubseteq_{\Sigma_{J_i}(\mathsf{Id},\mathsf{Msg}),\alpha} \mathcal{A}_i'$*. Then,* $\|^G_{i=1\ldots n} \mathcal{A}_i \sqsubseteq_{\{\tau\}} \|^G_{i=1\ldots n} \mathcal{A}_i'$*.*

Notice that when $A \sqsubseteq_{\{\tau\}} B$, all LTL formulas that hold in B also hold in A (see *e.g.* [3]) since simulation implies trace inclusion. Thus, to prove that A satisfies a given property, it suffices to verify B.

An abstraction can also be obtained by relaxing the graph G.

Lemma 2. *Consider* $\mathcal{A}_i = (S_i, s^i_{\mathsf{init}}, \delta_i, \Sigma_{J_i}(\mathsf{Id}, \mathsf{Msg}))$ *for* $i \in [1,n]$*, where* $J_i \subseteq \mathsf{Id}$ *are pairwise disjoint, and* $G, G' \in \mathcal{G}(m)$ *where* $\cup_i J_i \subseteq \{1, \ldots, m\}$*. We write* $G = (V, E)$ *and* $G' = (V, E')$*. If* $E \subseteq E'$*, then* $\|^G_{i=1\ldots n} \mathcal{A}_i \sqsubseteq_{\{\tau\}} \|^{G'}_{i=1\ldots n} \mathcal{A}_i$*.*

Approximate Synchrony. We recall the results of [12] where a finite-state scheduler is defined for concurrent processes which run a periodic action with an approximately equal period. This is the case in FTSP since all nodes run processes that wake up and execute an action with an identical nominal period T. Since each node is executed on a distinct hardware with a local clock, the

[1] Both are denoted α. Formally, α can be defined on the disjoint union of these sets.

observed period is only approximately equal to T. Thus, some nodes can execute faster than other nodes. In our model, we would like to include different interleavings that can be observed due to clock rate changes. Let us assume that the actual period lies in the interval $[\sigma^l, \sigma^u]$ (which contains T). However, not all interleavings between processes can be observed. In particular, if $|\sigma^u - \sigma^l|$ is small, the periods of different processes will be close, so they will be *approximately synchronous*: within one period of a process, another process cannot execute several periods. This restricts considerably the interleavings to be considered for model checking. Following [12], we define a scheduler that generates at least all interleavings that can be observed during the first N periods, when the clock rates are within a given interval.

We give two schedulers to model such approximately periodic behaviors. We will later instantiate these again for the particular case of FTSP. Let us consider $\mathcal{A}_1, \ldots, \mathcal{A}_n$, and an additional process $\mathcal{S}$ which will be used to schedule processes $\mathcal{A}_i$. Let us add a label $\texttt{tick}_i?$ to each $\mathcal{A}_i$, and $\{\texttt{tick}_i!\}_{1 \leq i \leq n}$ to $\mathcal{S}$; this models the periodic task of the node i.[2] Let us assume that all states of $\mathcal{A}_i$ accept a transition with $\texttt{tick}_i?$

Real-Time Scheduler. We define a *concrete* scheduler $\mathcal{S}_t$ which describes the executions generated by local clocks. We define $\mathcal{S}_t$ with an infinite state space, $S_{\mathcal{S}} = [0, \sigma_u]^n$, where the i-th component is the elapsed time since the latest execution of $\texttt{tick}_i?$ in process $\mathcal{A}_i$. We allow two kinds of transitions that alternate. There are *time elapse* transitions $(t_1, \ldots, t_n) \xrightarrow{\tau} (t'_1, \ldots, t'_n)$ if for some $d \geq 0$, $\forall 1 \leq i \leq n$, $t'_i = t_i + d$, and $\forall 1 \leq i \leq n, t'_i \leq \sigma_u$. Second, we have the transition $(t_1, \ldots, t_n) \xrightarrow{\texttt{tick}_i!} (t'_1, \ldots, t'_n)$ where $t'_j = t_j$ for all $j \neq i$ and $t'_i = 0$ if $t_i \in [\sigma_l, \sigma_u]$. Thus, $\mathcal{S}_t$ describes the executions where each process is executed with a period that varies within $[\sigma_l, \sigma_u]$.

Abstract Scheduler. Although the scheduler $\mathcal{S}_t$ above describes the behaviors we are interested in, its state space is continuous, and one would need a priori timed or hybrid automata to model it precisely. In this work, we prefer using finite-state model checking techniques for better efficiency, thus we now describe a simple abstraction of $\mathcal{S}_t$ using finite automata.

For each process i, and time t, let us denote by $N_i(t)$ the number of transitions $\texttt{tick}_i?$ that was executed in $\mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n \parallel \mathcal{S}_t$ up to time t. We define the *abstract scheduler* $\mathcal{S}_a(\Delta)$ on a finite state-space, given integer Δ, which ensures that, at any time point t, for all pairs of processes i, j, we have $|N_i(t) - N_j(t)| \leq \Delta$. Intuitively, $\mathcal{S}_a(\Delta)$ describes the behaviors in which a fast process can execute at most Δ periods within one period of a slow process. Notice that $\mathcal{S}_a(\Delta)$ can be defined simply by counting the number of times each process has executed $\texttt{tick}_i?$ One can actually use bounded counters in $[0, \Delta]$; in fact, it is sufficient to keep the relative values of $N_i(t)$ with respect to the smallest one, so $\mathcal{S}_a(\Delta)$ can be defined as a finite automaton.

[2] These labels can actually be defined within $\Sigma(\mathsf{Id}, \mathsf{Msg})$ by adding a special message content $\texttt{tick}$ to Msg, and setting $\texttt{tick}_i! = (n+1)!(i, \texttt{tick})$ where $n+1$ is the identifier of $\mathcal{S}$. We will write them simply as $\texttt{tick}_i?$ and $\texttt{tick}_i!$ to simplify the presentation.

The intuition behind $\mathcal{S}_a(\varDelta)$ is that, given the bounds $[\sigma_l, \sigma_u]$ on the observable periods, all interleavings up to some length N under $\mathcal{S}_t$ are also present in $\mathcal{S}_a(\varDelta)$. That is, $\mathcal{S}_a(\varDelta)$ over-approximates $\mathcal{S}_t$ for finite executions. We will show how one can choose N. Let us denote $\texttt{Ticks} = \{\texttt{tick}_i!\}_{1 \leq i \leq n}$. We have the following correspondance between $\mathcal{S}_t$ and $\mathcal{S}_a$:

Lemma 3 (*[12]***).** *Consider* $\varDelta > 0$*, and interval* $[\sigma_l, \sigma_u]$*. Let* N_f *be the minimal integer satisfying the following constraints:* $N_f \geq N_s, N_f - N_s > \varDelta, \sigma_l N_f + \sigma_u \leq \sigma_u N_s$*, and* $N_f, N_s \geq 1$*. Then, we have* $\mathcal{S}_t \sqsubseteq_{Ticks}^{N_f - 1} \mathcal{S}_a(\varDelta)$*.*

In the above lemma, N_f represents the number of steps performed by the fastest processes, and N_s is that of the slowest processes. Minimizing N_f means that we look for the earliest step where $N_f - N_s > \varDelta$ holds, so that the simulation holds up to $N_f - 1$ steps. Hence, we can use $\mathcal{S}_a(\varDelta)$ for model checking rather than $\mathcal{S}_t$ for N steps, where N is determined by $\varDelta$ and σ_l, σ_u.

3 Parameterized Model Checking of FTSP

In the FTSP, each node has a unique identifier, and the nodes dynamically elect the node with the least id as the *root*. The root regularly sends messages to its neighbors, which forward it to their own neighbors and so on. These messages contain time information which is used by the nodes to adjust their clocks. If the root node fails, that is, stops transmitting messages, then other nodes eventually time out and declare themselves as roots, and the protocol makes sure that a unique root is eventually elected if no more faults occur during a period of time.

More precisely, each node has an identifier `ID`, and executes the periodic action `send`, depicted in Fig. 2 in which it increments a "heart beat" counter `b`. This counter is reset to 0 if the node receives a certain message via the `receive` function: this can happen either when the node first hears about a node with a smaller ID `ri` than the currently known one, stored in `r`, or when the currently known root sends a *new* message with a larger sequence number `si` than that of the latest message `s`. The sequence numbers are used to distinguish new messages from the old ones that originate from a root node; a lexicographic order is used so that smaller root IDs with higher sequence numbers are preferred. A node declares itself root if the counter `b` exceeds the threshold `FTO`; and it only broadcasts messages if it is root, or if it has received at least `LIM` messages from some root. We refer the reader to [20] for the details on FTSP.

Both functions `send` and `receive` are executed atomically. Thus, the effects of each function on local variables are self-explanatory. The operation $o!!(r,s)$ means broadcast: it is a system call to broadcast the message (r,s) to all the neighbors of the node. This operation is non-blocking: when the function `send` returns, the node sends the message to each neighbor in an arbitrary order. We assume the broadcast data is stored in a variable `m` which takes values from the set $\{\bot\} \cup 2^{\mathsf{Id}} \times \mathsf{Msg}$. Here $\bot$ means that there is no ongoing broadcast, and a pair (I, m) means that processes with ids in I are still to receive the message m. That is, the operation $o!!(r, s)$ actually just assigns the value (r, s) to local variable `m`.

The node can receive messages and execute `receive` before its own broadcast is over. We just make the following assumption on broadcasts, which is justified by the fact that the typical period of the `send` events is about 30 s [20].

Assumption: Any broadcast started by a node is completed before the node executes the next `send` event.

3.1 Concrete Model

We fix a graph $G \in \mathcal{G}(n)$ with n nodes, and set $\mathsf{Id} = \{1, \ldots, n\}$, and $\mathsf{Msg} = \mathsf{Id} \times \mathbb{N}$. In Msg, the first component of a message is the ID of the root node which has generated the message (and not the ID of the node that forwards the message), while the second component is the sequence number. Each process $\mathcal{A}_i$ is a node in the protocol in which the variable `ID` is i, and executes functions `receive` and `send` of Fig. 2. We define $\mathcal{A}_i = (S_i, s^i_{\mathsf{init}}, \delta_i, \Sigma_{\{i\}}(\mathsf{Id}, \mathsf{Msg}))$, with $S_i = V_i \times (2^n \cup \{\bot\})$ where V_i are the set of valuations for all local variables. For any variable `a`, and state $s \in S_i$, we write $s(\mathtt{a})$ for the value of `a` in s (we also write $v(\mathtt{a})$ for $v \in V_i$). The second component of a state $s \in S_i$ denotes whether the process is currently broadcasting: if it is $\bot$, there is no broadcast occurring and $s(\mathtt{m}) = \bot$; if it is $I \subseteq 2^{\mathsf{Id}}$, then message $s(\mathtt{m})$ is to be received by processes in I. We denote by $s[\mathtt{a} \leftarrow a]$ the state obtained from s by assigning a to `a`.

Since each function is executed atomically, in $\mathcal{A}_i$, a single transition corresponds to an uninterrupted execution of `send` or `receive`, or to a communication. For any $m \in \mathsf{Msg}$, let us define the relation $\mathtt{receive}_i(m) \subseteq V_i \times V_i$ (resp. `send`) as $(v, v') \in \mathtt{receive}_i(m)$ (resp. $(v, v') \in \mathtt{send}_i$) if, and only if there is an execution of this function from state v to state v', when the node ID is i. These relations are functions since $\mathtt{receive}_i$ and $\mathtt{send}_i$ are deterministic; however, subsequent abstractions will transform these into nondeterministic programs, thus we will obtain relations instead of functions. Thus, δ_i is defined as follows:

```
#define MAX   6 /* MAX_ENTRIES       */
#define LIM   3 /* ENTRY_SEND_LIMIT*/
#define MIN   2 /* IGNORE_ROOT_MSG */
#define FTO   8 /* ROOT_TIMEOUT      */
extern int ID   /* TOS_NODE_ID       */
#define NIL 255

void receive (byte ri, byte si) {
  if(ri < r && !(b < MIN && r==ID))
      || (ri == r && si - s > 0){
      r = ri;
      s = si;
      if(r < ID){b = 0;}
      if(e < MAX){e++;}
  }
}
```

```
byte b; /* heartBeats */
byte e; /* numEntries */
byte r; /* outgoingMsg.rootID */
byte s; /* outgoingMsg.seqNum */
chan o; /* Output channel */

void send () {
  if(b >= FTO){
    if(r == NIL){ s = 0; }
    else { b = 0; s++; }
    r = ID
  }
  b++;
  if(r == ID){ o !! (r, s); s++; }
  else if(e >= LIM){ o !! (r, s) }
}
```

Fig. 2. Pseudocode of the main send and receive functions in FTSP

$$
\begin{aligned}
&(v,\bot) \xrightarrow{\texttt{tick}_i?} (v',\mathcal{N}_G(i)) \Leftrightarrow (v,v') \in \texttt{send}_i \wedge v'(\texttt{m}) \neq \bot,\\
&(v,\bot) \xrightarrow{\texttt{tick}_i?} (v',\bot) \Leftrightarrow (v,v') \in \texttt{send}_i \wedge v'(\texttt{m}) = \bot,\\
&(v,\emptyset) \xrightarrow{\texttt{tock}_i?} (v[\texttt{m} \leftarrow \bot],\bot),\\
&(v,I) \xrightarrow{j?(i,m)} (v',I) \Leftrightarrow (v,v') \in \texttt{receive}_i(m) \wedge j \in \mathcal{N}_G(i),\\
&(v,I) \xrightarrow{i!(j,m)} (v,I\setminus\{j\}) \Leftrightarrow m = v(\texttt{m}) \neq \bot \wedge j \in I,
\end{aligned}
$$

where the last two lines are defined for all $I \in \{\bot\} \cup 2^{\mathsf{Id}}$.

Notice that we separate the execution of the body of the two functions and the broadcast operations. A broadcast operation is completed between the $\texttt{tick}_i?$ and $\texttt{tock}_i?$ events. Hence, the broadcast can be interrupted with a receive event, but another send event cannot be executed before the broadcast is complete, which conforms to our assumption above. The role of $\texttt{tick}_i$ and $\texttt{tock}_i$ signals will be clear in the next paragraph where the schedulers are defined. The initial states are the set of all valuations since we assume that the network starts in an arbitrary configuration. Now, $\|_{i=1\ldots n}^{G} \mathcal{A}_i$ defines the protocol on the given topology G. It remains to define the schedulers.

Schedulers and Two Communication Semantics. We define schedulers which determine when each process can execute its `send` event, and how the communication is modeled. We sketch our schedulers with two communication models.

Synchronous Communication. In the first model, we assume that communication between the sender and *all* receivers occur simultaneously. So, one step consists in a node executing `send` followed by all its neighbors immediately receiving the message by executing `receive`. This is the *synchronous communication model* as considered in previous works [19,21,28].

To implement synchronous communication, we introduce the signal $\texttt{tock}_i!$, and force the whole communication initiated by node i to happen uninterrupted between $\texttt{tick}_i!$ and $\texttt{tock}_i!$ signals. We define $\mathcal{S}_{t,\mathsf{syn}}$ by modifying the real-time scheduler $\mathcal{S}_t$ defined above by requiring that each $\texttt{tick}_i!$ is immediately followed by a corresponding $\texttt{tock}_i!$, and by disallowing any other $\texttt{tick}_j!$ inbetween. We also define $\mathcal{S}_{a,\mathsf{syn}}^{\mathsf{ftsp}}(\Delta)$ from $\mathcal{S}_a(\Delta)$ using the alternating $\texttt{tick}_i$ and $\texttt{tock}_i$ signals.

Asynchronous Communication. The second type of schedulers we define implement asynchronous communication, and is more faithful to the real behavior e.g. in the TinyOS implementation. In this setting, both events `send` and `receive` are still atomic, but the broadcast is concurrent: while the sender is broadcasting the message to its neighbors, other nodes can execute their own `send` action or receive other messages. We call this the *asynchronous communication model.*

We define $\mathcal{S}_{t,\mathsf{asyn}}$ by adding to $\mathcal{S}_t$ self-loops labeled by $\texttt{tock}_i!$ to all states for all $i \in \mathsf{Id}$. (Note that $\texttt{tock}_i!$ signals are useless here, but we keep them so that both schedulers have a uniform interface). We define the scheduler $\mathcal{S}_{a,\mathsf{asyn}}^{\mathsf{ftsp}}(\Delta)$ similarly, by adding self-loop $\texttt{tock}_i!$ to all states of $\mathcal{S}_a(\Delta)$.

The next developments are independent from the communication model.

Complete Model and Property to be Verified. Given a graph $G \in \mathcal{G}(n)$ let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ denote the processes thus defined, and write $\mathcal{A}(G) = \|_{i=1\ldots n}^{G} \mathcal{A}_i$. We let $\mathcal{M}_{\bowtie}^{\mathsf{conc}}(G) = \mathcal{A}(G) \parallel \mathcal{S}_{t,\bowtie}$, for $\bowtie \in \{\mathsf{syn}, \mathsf{asyn}\}$, which is the *concrete* protocol under the real-time scheduler $\mathcal{S}_t$ defined above. This model defines the behaviors we would like to verify. For each $i \in \mathsf{Id}$, let us add a counter $\mathtt{c}_i$ to the model that counts the number of times $\mathtt{tick}_i!$ is executed, and define $\mathtt{c} = \max_i \mathtt{c}_i$, which will be used in the specifications.

The property we want to check is that all nodes eventually agree on a common root. Let FRID denote the constant 1, which stands for the $f$uture $r$oot id. In fact, according to the protocol, $\mathcal{A}_1$ is expected to become the root since it has the least id. We will call $\mathcal{A}_1$ the *future root*. Define P_i as the set of states in which the local variable `r` of process i has value FRID. We consider the property $\mathcal{P}(N) = \mathsf{F}(\mathtt{c} \le N \wedge \wedge_{i=1}^{n} P_i)$ for some N. Thus, along all executions, before any process has executed more than N $\mathtt{tick}_i$'s, all processes agree on FRID to be the root. Thus, our goal is to show that $\mathcal{M}_{\bowtie}^{\mathsf{conc}}(G) \models \mathcal{P}(N)$ for some $N > 0$. By Lemma 3, given Δ, it suffices to find $N > 0$ for each $\bowtie \in \{\mathsf{syn}, \mathsf{asyn}\}$, such that $\mathcal{A}(G) \parallel \mathcal{S}_{a,\bowtie}^{\mathsf{ftsp}}(\Delta) \models \mathcal{P}(N)$.

3.2 Abstractions on Individual Nodes

We now present the abstraction steps we use before model checking. We will abstract our variables and statements involving these using *data abstraction*: we map the domain of the variables to a smaller set, and redefine the transitions using *existential abstraction* so that the abstract program is an over-approximation in the sense that the original process is simulated by the existential abstraction. This is a standard abstraction technique; we refer the reader to [9] for details.

More precisely, the applied abstraction steps are the following.

1. Add a redundant variable `imroot` that stores the value of the predicate `r == ID`, that is, whether the node is currently root.
2. Relax the behaviors of both functions in the case $\mathtt{r} \neq \mathtt{FRID} \wedge \mathtt{ri} \neq \mathtt{FRID} \wedge \mathtt{ID} \neq \mathtt{FRID}$ by abstracting the variables `s` and `e` away (*i.e.* we assume their values change arbitrarily at any time).
3. Map the variables `r` and `ri` in the abstract domain $\{\mathtt{FRID}, \mathtt{NRID}\}$ in each node. Also map `b` to the bounded integer domain $\{0, \mathtt{FTO}\}$, `e` to $\{0, \ldots, \mathtt{LIM}\}$.

The resulting pseudocode is shown in Fig. 3. Here, the value $\perp$ represents *any value*, which make any comparison operation nondeterministic. The constant NRID we introduce stands for *non-root id*, and is an abstract value that represents all ids different than FRID.

Note that the second step always yields an over-approximation, independently from the if-then-else condition chosen to separate the concrete and abstract cases in Fig. 3. In fact, the concrete case is identical to the original code, while the abstract case is an over-approximation by data abstraction. In Fig. 3, the abstractions of the predicates on variables `r` and `ri` are

```
#define LIM 3 /* ENTRY_SEND_LIMIT */
#define MIN 2 /* IGNORE_ROOT_MSG */
#define FTO 8 /* ROOT_TIMEOUT */
#define NIL 255
extern int ID; /* TOS_NODE_ID */
#define FRID 0 /* FUTURE ROOT ID */
#define NRID 1 /* Abstract ID for
               all other nodes > FRID */

void receive (byte ri, byte si) {
  /* Concrete case */
  if (r == FRID || ri ==
FRID || ID == FRID){
    if (“ri < r” && !(b < MIN && imroot)
        || “ri == r” && si - s > 0 ){
      r = ri;
      s = si;
      imroot = (ID == FRID);
      if (“r < ID” ) b = 0;
      if (e < LIM) e++;
    }
  } else {
  /* Abstract case */
     if (“ri < r” && !(b < MIN && imroot))
         || (“ri == r” && *){
         r = ri;
         s = ⊥;
         imroot = “r == ID”;
         if (“r < ID” ) b = 0;
         e = ⊥;
}}}
```

```
byte b; /* heartBeats */
byte e; /* numEntries */
byte r; /* outgoingMsg.rootID */
byte s; /* outgoingMsg.seqNum */
chan i, o; /* IO channels */
byte imroot; /* Predicate: r == ID */

void send () {
  /* Concrete case */
  if(r == FRID || ID == FRID){
    if(b >= FTO){
      if (“r == NIL”) s = 0;
      if (“r != ID”) { b = 0; s++; }
      r = ID;
      imroot = 1;
    }
    b++;
    if(imroot){ o !! (r, s); s++; }
    else if(e >= LIM){ o !! (r, s); }
  } else {
  /* Abstract case */
    if(b >= FTO){
      if(“r != ID”) { b = 0; s = ⊥; }
      r = ID;
      imroot = 1;
    }
    if (b < FTO) b++;
    if(imroot){ o !! (r, *); s = ⊥; }
    else if(*){ o !! (r, *); }
}}
```

Fig. 3. After the second and third steps of the abstraction. The behavior of `receive` is relaxed when `r != FRID` or the received message `(ri,si)` is such that `ri != FRID`. Similarly, the behavior of `send` is relaxed when `r != FRID` and `ID != FRID`. For both functions, we redefine the behaviors of the protocol by disregarding the variables `e` and `s`. The updates and tests on these variables become completely non-deterministic. In particular, nodes in such states can send more often messages with arbitrary sequence numbers. Then, the variables `r,ri` are mapped to the domain $\{\texttt{FRID}, \texttt{NRID}\}$. The variable `b` is mapped to $\{0, 1, \ldots, \texttt{MAX}\}$, and `e` to $\{0, 1, \ldots, \texttt{LIM}\}$.

represented in quotes. They represent non-deterministic transitions as follows. The comparison relation becomes non-deterministic: we have $\texttt{FRID} < \texttt{NRID}$ and $\texttt{FRID} = \texttt{FRID}$, but, for instance, a comparison between `NRID` and `NRID` can yield both true and false. As an example, “`r == ri`” stands for `r = FRID && ri = FRID || r = NRID && ri = NRID && *`, $*$ being a nondeterministic Boolean value.

Let $S_i' = V_i' \times (2^n \cup \{\bot\})$ where V_i' is the set of valuations of node variables (with given id i), with the abstract domains we have described. Let us define the relations $\texttt{receive}_i' \subseteq V_i' \times V_i'$ and $\texttt{send}_i' \subseteq V_i' \times V_i'$, similarly as before, e.g. $(s, s') \in \texttt{receive}_i'$ if, and only if there is an execution of $\texttt{receive}_i'$ from s yielding s'. Let $\mathcal{A}_i'$ denote the process defined just like $\mathcal{A}_i$ in Subsect. 3.1 but using the new relations $\texttt{receive}_i'$ and $\texttt{send}_i'$. We state the relation between $\mathcal{A}_i$ and $\mathcal{A}_i'$ using a label abstraction function α. We let α be the identity over Id, and set $\mathsf{Msg}^\sharp = \{\texttt{FRID}, \texttt{NRID}\} \times (\mathbb{N} \cup \{\bot\})$ with $\alpha((k, s)) = (\texttt{FRID}, s)$ if $k = \texttt{FRID}$, and $\alpha((k, s)) = (\texttt{NRID}, \bot)$ otherwise.

Lemma 4. *For all i, $\mathcal{A}_i \sqsubseteq_{\Sigma_i(\mathsf{Id},\mathsf{Msg}),\alpha} \mathcal{A}_i'$.*

By Lemma 1, it follows that $\|_{i=1\ldots n}^{G} \mathcal{A}_i \sqsubseteq_{\{\tau\}} \|_{i=1\ldots n}^{G} \mathcal{A}_i'$.

3.3 Abstraction on Network Topology: Shortest-Path Abstraction

Recall that our model has a network topology $G \in \mathcal{G}_K(n)$. Consider an arbitrary shortest path $\mathcal{A}_{i_1}\mathcal{A}_{i_2}\ldots\mathcal{A}_{i_m}$ with $m \leq K$, where $i_1 = 1$. Let $C = \{i_2, \ldots, i_m\}$, that is, all nodes on this path but the future root. Define $O = \mathsf{Id} \setminus C$. Let us relax the graph $G = (V, E)$ into $G' = (V, E')$ by $E' = E \cup O \times O \cup O \times C \cup C \times O$. Thus, we render the graph complete within O, and add all edges between O and C. Let us write $\mathcal{A}'_C = \|^{G'}_{i\in C} \mathcal{A}'_i$, and $\mathcal{A}'_O = \|^{G'}_{i\in O} \mathcal{A}'_i$. By Lemma 2, these are over-approximations of the products defined for G.

We define $\mathcal{A}''_O$ as a single-state process with alphabet $\Sigma_O(\mathsf{Id}, \mathsf{Msg}^\sharp)$ which can send any message to any other node. We clearly have $\mathcal{A}'_O \sqsubseteq_{\Sigma_O(\mathsf{Id},\mathsf{Msg}^\sharp)} \mathcal{A}''_O$.

We now get rid of the identifiers outside $C \cup \{1\}$ by defining a label abstraction function $\alpha' : \mathsf{Id} \to \mathsf{Id}^\sharp$ with $\mathsf{Id}^\sharp = C \cup \{\mathcal{O}, 1\}$ where $\mathcal{O}$ is a fresh symbol. We let $\alpha'(i) = i$ for all $i \in C \cup \{1\}$, and $\alpha'(i) = \mathcal{O}$ for all $i \in O \setminus \{1\}$. So, all nodes outside $C \cup \{1\}$ are merged into one identifier $\mathcal{O}$. Let $\mathcal{B}_O$ be the mapping of $\mathcal{A}''_O$ by α', and $\mathcal{B}_C$ that of $\mathcal{A}'_C$, so that we have $\mathcal{A}'_O \sqsubseteq_{\Sigma_O(\mathsf{Id},\mathsf{Msg}^\sharp)} \mathcal{A}''_O \sqsubseteq_{\Sigma_O(\mathsf{Id},\mathsf{Msg}^\sharp),\alpha'} \mathcal{B}_O$ and $\mathcal{A}'_C \sqsubseteq_{\Sigma_C(\mathsf{Id},\mathsf{Msg}^\sharp),\alpha'} \mathcal{B}_C$.

We need to adapt the scheduler so that it does not keep track of the offset of the processes represented by $\mathcal{O}$. Let $\mathcal{S}'^{\mathsf{ftsp}}_{a,\mathsf{syn}}(\Delta)$ and $\mathcal{S}'^{\mathsf{ftsp}}_{a,\mathsf{asyn}}(\Delta)$ defined similarly as before which track the offsets of all nodes in $C \cup \{1\}$, but have a self-loop with label $\mathtt{tick}_{\mathcal{O}}!$ at all states. We thus have $\mathcal{S}^{\mathsf{ftsp}}_{a,\bowtie}(\Delta) \sqsubseteq_{\mathtt{Ticks},\alpha'} \mathcal{S}'^{\mathsf{ftsp}}_{a,\bowtie}(\Delta)$ for both $\bowtie \in \{\mathsf{syn}, \mathsf{asyn}\}$.

By Lemmas 1–2, $\mathcal{A}'_O \|^G \mathcal{A}'_C \|^G \mathcal{S}^{\mathsf{ftsp}}_{a,\bowtie}(\Delta) \sqsubseteq_{\{\tau\},\alpha'} \mathcal{B}_O \|^{G'} \mathcal{B}_C \|^{G'} \mathcal{S}'^{\mathsf{ftsp}}_{a,\bowtie}(\Delta)$.

We need another abstraction to obtain a finite model: The variable $\mathtt{s}$ is a priori unbounded in each process; however, the only applied operations are incrementation (by $\mathtt{FRID}$ only), assignment, and comparison. Therefore, we can shift the values so that the minimal one is always 0; thus limiting the maximal value that is observed. We modify our process to map these variables to a finite domain $\{0, 1, \ldots, \mathtt{SeqMax}, \bot\}$ and *normalize* their values after each transition: we make sure that at any step, the values taken by $\mathtt{s}$ at all nodes define a set $X \cup \{\bot\}$ for some $0 \in X \subseteq \{0, 1, \ldots, \mathtt{SeqMax}\}$.

We summarize all the steps of the abstractions as follows. Given graph $G \in \mathcal{G}_K(n)$, a path π of length K from node 1, let $\mathcal{M}^{\mathsf{abs}}_{\bowtie}(G, \pi, \Delta) = \mathcal{B}_O \|^{G'} \mathcal{B}_C \|^{G'} \mathcal{S}'^{\mathsf{ftsp}}_{a,\bowtie}(\Delta)$ where $\bowtie \in \{\mathsf{syn}, \mathsf{asyn}\}$.

Lemma 5. *For all $n, K > 0$, and all $G \in \mathcal{G}_K(n)$, let π be any shortest path from node 1. Let C be the nodes of π except 1, and $O = [1, n] \setminus C$. We have, for all $\bowtie \in \{\mathsf{syn}, \mathsf{async}\}$, $\mathcal{M}^{\mathsf{conc}}_{\bowtie}(G) \sqsubseteq_{\{\tau\}} \mathcal{M}^{\mathsf{abs}}_{\bowtie}(G, \pi, \Delta)$.*

Notice that in $\mathcal{M}^{\mathsf{abs}}_{\bowtie}(G, \pi, \Delta)$, all node ids are in the set $\{\mathtt{FRID}, \mathtt{NRID}\}$. Thus, given two different paths π, π', $\mathcal{M}^{\mathsf{abs}}_{\bowtie}(G, \pi, \Delta)$ and $\mathcal{M}^{\mathsf{abs}}_{\bowtie}(G, \pi', \Delta)$ are identical up to the renaming of their channel numbers since both models still contain labels of the form $i!(j, m)$ and $i?(j, m)$. However, these numbers i, j only define the topology and do not affect the behaviors. Let us state this formally as follows:

Lemma 6. *For all $K, n > 0$, graph $G \in \mathcal{G}_K(n)$, and paths π, π' of same length from node 1, we have $\mathcal{M}^{\mathsf{abs}}_{\bowtie}(G, \pi, \Delta) \sqsubseteq_{\{\tau\}} \mathcal{M}^{\mathsf{abs}}_{\bowtie}(G, \pi', \Delta)$.*

From the above lemma, it follows that for verification purposes (against LTL), the model $\mathcal{M}^{\mathsf{abs}}_{\bowtie}(G, \pi, \varDelta)$ is actually independent of the chosen path π, but only depends on the length of π. For each $K > 0$, let us pick one such model with $|\pi| = K$ and name it $\mathcal{M}^{\mathsf{abs}}_{\bowtie}(K, \varDelta)$. Then, we have $\mathcal{M}^{\mathsf{abs}}_{\bowtie}(G, \pi, \varDelta) \sqsubseteq_{\{\tau\}} \mathcal{M}^{\mathsf{abs}}_{\bowtie}(K, \varDelta)$ for all $G \in \mathcal{G}_K(n)$ and $\varDelta > 0$. It follows that model checking a property in $\mathcal{M}^{\mathsf{abs}}_{\bowtie}(K, \varDelta)$ proves it on all graphs $G \in \mathcal{G}_K(n)$ and all paths π.

In the rest, w.l.o.g. let us assume that $C = \{2, \ldots, K\}$. Our goal is to check $\mathcal{M}^{\mathsf{abs}}_{\bowtie}(K, \varDelta) \models \mathcal{P}^K(N)$ for some N, where $\mathcal{P}^K(N) = \mathsf{F}(\mathsf{c} \leq N \wedge \bigwedge_{i=1}^{K} P_i)$.

3.4 Incremental Verification Technique and Refinement

We explain an incremental approach to model-check our system for successive values of K. Intuitively, we assume that we have proved the root election property for K, and we want to prove it for $K + 1$. For K, if we prove the property is *persistent*, that is, holds forever after some point in time, then, we can prove the property for $K + 1$ as follows: initialize the first K nodes in $\mathcal{M}^{\mathsf{abs}}_{\bowtie}(K + 1, \varDelta)$ to a state in which they agree on the future root, and the $K + 1$-th node in an arbitrary state; then verify the property for the last process only:

Lemma 7. *Consider processes $\mathcal{R}_1, \ldots, \mathcal{R}_n$, and $\mathcal{S}_1, \ldots, \mathcal{S}_n$. For some graph G, let $\mathcal{R}(K) = \|^{G}_{i=0\ldots K} \mathcal{R}_i$. Assume that $\mathcal{R}(K + 1) \; \|^G \; \mathcal{S}_{K+1} \sqsubseteq_\tau \mathcal{R}(K) \; \|^G \; \mathcal{S}_K$ for all K. Consider predicate Q_i for each $\mathcal{R}_i$, and define $Q(K) = \wedge_{i=1}^{K} Q_i$. Consider $\mathcal{R}'(K+1)$ obtained from $\mathcal{R}(K+1)$ by restricting the states to $Q(K)$ (That is, we remove all states outside and all transitions that leave outside this set.), where the initial state set is $Q(K)$. We have that $\mathcal{R}(K) \parallel \mathcal{S}_K \models \mathsf{FG}Q(K) \wedge \mathcal{R}'(K+1) \parallel \mathcal{S}_{K+1} \models \mathsf{FG}Q_{K+1}$ implies $\mathcal{R}(K + 1) \parallel \mathcal{S}_{K+1} \models \mathsf{FG}Q(K + 1)$.*

Here is how we apply the above lemma in our case. Let $\mathcal{R}_i = \mathcal{A}'_{i+1}$ for $i = 1 \ldots K$. We write $S_K = {\mathcal{S}'}^{\mathsf{ftsp}}_{a,\mathsf{asyn}}(\varDelta)$ defined for $C = \{1, \ldots, K\}$ in Sect. 3.3, and let $\mathcal{S}_K = \mathcal{A}''_{O_K} \parallel S_K$ with $O_K = \mathsf{Id} \setminus \{2, \ldots, K\}$. We have $\mathcal{A}''_{O_{K+1}} \sqsubseteq_{\varSigma(O_{K+1}, \mathsf{Msg}^\sharp)} \mathcal{A}''_{O_K}$ and $S_{K+1} \sqsubseteq_{\mathtt{Ticks}} S_K$ by definition, so these processes do satisfy the relation $\mathcal{R}(K + 1) \parallel \mathcal{S}_{K+1} \sqsubseteq_\tau \mathcal{R}(K) \parallel \mathcal{S}_K$. As properties to satisfy, we consider $Q_i = P_i \wedge \mathsf{b}_i \leq \mathtt{FTO} - 1 \wedge \mathsf{e} \geq \mathtt{LIM}$, and also define $Q(K, N) = (c \leq N \wedge \wedge_{i=1}^{K} Q_i)$. Notice that $Q(K, N)$ implies $\mathcal{P}^K(N)$.

Assume we have proved the following statements: ${\mathcal{M}'}^{\mathsf{abs}}_{\bowtie}(1, \varDelta) \models \mathsf{FG}Q(1, n_1)$, ${\mathcal{M}'}^{\mathsf{abs}}_{\bowtie}(2, \varDelta) \models \mathsf{FG}Q(2, n_2)$, $\ldots$, ${\mathcal{M}'}^{\mathsf{abs}}_{\bowtie}(K - 1, \varDelta) \models \mathsf{FG}Q(K - 1, n_{k-1})$, and ${\mathcal{M}'}^{\mathsf{abs}}_{\bowtie}(K, \varDelta) \models \mathsf{FG}Q(K, n_k)$. Then, by the previous lemma, $\mathcal{M}^{\mathsf{abs}}_{\bowtie}(K, \varDelta) \models \mathsf{FG}Q(K, N)$ for $N = n_1 + \ldots + n_k$, which means $\mathcal{P}^K(N)$. Note that the last property to be proven can also be chosen as $\mathsf{F}P_K$ which might be satisfied earlier than $\mathsf{FG}Q(K, n_k)$.

Non-interference Lemma. The first verification attempts reveal a spurious counter-example: the non-deterministic process $\mathcal{B}_O$ can send a node in C a message (r, s) with $r = \mathtt{FRID}$ and s large enough so that the node will ignore all messages for a long period of time, causing a timeout; this causes the violation of $\mathsf{FG}P_i$. However, intuitively, a node should not be able to send a message with r = `FRID` with a newer sequence number than what has been generated by the root itself. Following [7], we use *guard strengthening*: We require that all messages that come from the process $\mathcal{B}_O$ must satisfy that either $r \neq$ `FRID` or s is at most equal to the variable `s` of the root. Let this condition be ϕ. We thus constrain the transitions of our model to satisfy ϕ, which eliminates this spurious counter-example. Property ϕ is also called a *non-interference lemma* [7]. However, we also need to actually prove ϕ. As it turns out, one can prove ϕ on the very same abstraction obtained by strenghtening. The works [7,18] explain why the apparently circular reasoning is correct. We do not detail this technique further since this is now a well-known result; see also [5,27].

4 Algorithm and Experimental Results

Semi-Algorithm for FG *Properties with Optimal Bound Computation.* Model checking algorithms consist in traversing the state space while storing the set of visited states so as to guarantee termination. However, this set can sometimes become prohibitively large. We introduce a simple semi-algorithm for properties of type FG p where we do not store all states: at iteration i, we just store the states reachable in i steps exactly, and only if all these satisfy p, do we start a fixpoint computation from these states. The resulting semi-algorithm is more efficient and allows us to find the smallest i in one shot.

We implemented the semi-algorithm in NuSMV 2.5.4[3]. We model-checked the property $\mathcal{P}(N)$, and computed the bounds N using our semi-algorithm. The models are initialized in an arbitrary state, so our results show that the network recovers from any arbitrary fault within the given time bound. We summarize our results in Fig. 4. We distinguish the best values for N in both communication models for different values of K. Missing rows mean timeout of 24 h.

We observe that the time for the root election N differs in the two communication semantics. This value is higher in the asynchronous case since it contains all behaviors that are possible in the synchronous case. Observe that the largest network topology that had been model checked for FTSP contained 7 nodes [21] with synchronous communication. In our case, we prove the property for $K = 7$, which means that it holds on two dimensional grids with 169 nodes (13×13) when the root is at the middle (and three-dimensional grids with 2197 nodes), and 49 nodes if it is on a corner (and 343 nodes in three dimensions). In the asynchronous case, we prove the property for $K = 5$, *e.g.* 2D grids of size 81 nodes where the root is at the middle.

	synchronous		asynchronous	
K	N	time	N	time
1	8	0s	8	0s
2	14	1s	14	1s
3	23	1s	25	28s
4	35	3s	39	130s
5	54	16s	63	65mins
6	67	76s		
7	107	13mins		

Fig. 4. Verification results for the property $\mathcal{P}(N)$, obtained with the semi-algorithm. For each K and communication model, the best derived N is given.

[3] The source code and models are available at https://github.com/osankur/nusmv/tree/ftsp.

This implies the following bounds on clock rates: by Lemma 3, for $N = 107$, property $\mathcal{P}(N)$ is guaranteed on $\mathcal{M}_{\bowtie}^{\mathsf{conc}}(G)$ for all $G \in \mathcal{G}_K$ and $[\sigma_l, \sigma_u] = [29.7, 30.3]$ which is the case when the clock rates are within 1 ± 10^{-2}.

5 Conclusion

We presented an environment abstraction technique inspired from [7] for processes with unique identifiers, arbitrary network topologies, and drifting clocks. We introduced an incremental model checking technique, and gave an efficient semi-algorithm that can compute bounds for the eventually properties in one shot. We applied our technique to model check the root election part of FTSP and obtained significant improvements over previous results.

An important future work will be to automatize the presented abstraction method. Several steps of our abstractions, such as data abstractions, can easily be automatized with minimal user intervention. We would like to go further following [5], and consider automatic abstractions of the network topology.

Our aim is to address the case of more elaborate time synchronisation protocols based on interval methods, such as Sugihara and Gupta's [26] that are able to implement TSP in WSN without making assumptions on bounded drift, but simply on precise thermal and cristal oscillator specifications of the WSN hardware. We would like to obtain formal bounds on time precision guaranteed by a protocol under various assumptions on environment.

We believe our shortest-path abstraction technique can be used to verify different distributed protocols in which an information is forwarded in layers through the network such as [4,29]. An interesting future work would be to consider protocols that construct a spanning tree of the network in which case shortest paths would be replaced by a richer subgraph of the network topology.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoret. Comput. Sci. **126**(2), 183–235 (1994)
2. Apt, K.R., Kozen, D.C.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. **22**(6), 307–309 (1986)
3. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT press, Cambridge (2008)
4. Bakhshi, R., Bonnet, F., Fokkink, W., Haverkort, B.: Formal analysis techniques for gossiping protocols. ACM SIGOPS Oper. Syst. Rev. **41**(5), 28–36 (2007)
5. Bingham, J.: Automatic non-interference lemmas for parameterized model checking. In: Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design, FMCAD 2008, Piscataway, NJ, USA, pp. 11:1–11:8. IEEE Press (2008)
6. Chang, E., Roberts, R.: An improved algorithm for decentralized extrema-finding in circular configurations of processes. Commun. ACM **22**(5), 281–283 (1979)

7. Chou, C.-T., Mannava, P.K., Park, S.: A simple method for parameterized verification of cache coherence protocols. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 382–398. Springer, Heidelberg (2004). doi:10.1007/978-3-540-30494-4_27
8. Clarke, E., Talupur, M., Veith, H.: Proving ptolemy right: the environment abstraction framework for model checking concurrent systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 33–47. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_4
9. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. (TOPLAS) **16**(5), 1512–1542 (1994)
10. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
11. Delzanno, G., Sangnier, A., Traverso, R.: Parameterized verification of broadcast networks of register automata. In: Abdulla, P.A., Potapov, I. (eds.) RP 2013. LNCS, vol. 8169, pp. 109–121. Springer, Heidelberg (2013). doi:10.1007/978-3-642-41036-9_11
12. Desai, A., Seshia, S.A., Qadeer, S., Broman, D., Eidson, J.C.: Approximate synchrony: an abstraction for distributed almost-synchronous systems. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 429–448. Springer, Cham (2015). doi:10.1007/978-3-319-21668-3_25
13. Dolev, D., Klawe, M., Rodeh, M.: An o (n log n) unidirectional distributed algorithm for extrema finding in a circle. J. Algorithms **3**(3), 245–260 (1982)
14. Emerson, E.A., Namjoshi, K.S.: Reasoning about rings. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1995, pp. 85–94. ACM, New York (1995)
15. Garavel, H., Mounier, L.: Specification and verification of various distributed leader election algorithms for unidirectional ring networks. Sci. Comput. Program. **29**(1), 171–197 (1997)
16. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In: FMCAD, pp. 201–209 (2013)
17. Fredlund, L., Groote, J.F., Korver, V.: Formal verification of a leader election protocol in process algebra. Theoret. Comput. Sci. **177**(2), 459–486 (1997)
18. Krstic, S.: Parameterized system verification with guard strengthening and parameter abstraction. In: Automated Verification of Infinite State Systems (2005)
19. Kusy, B., Abdelwahed, S.: FTSP protocol verification using SPIN, May 2006
20. Maróti, M., Kusy, B., Simon, G., Lédeczi, A.: The flooding time synchronization protocol. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, SenSys 2004, pp. 39–49. ACM, New York (2004)
21. McInnes, A.I.: Model-checking the flooding time synchronization protocol. In: IEEE International Conference on Control and Automation, ICCA 2009, pp. 422–429, December 2009
22. McMillan, K.L.: Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In: Margaria, T., Melham, T. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 179–195. Springer, Heidelberg (2001). doi:10.1007/3-540-44798-9_17
23. Milner, R.: A Calculus of Communicating Systems. Springer, New York (1982)
24. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57, October 1977

25. Pnueli, A., Xu, J., Zuck, L.: Liveness with (0,1, ∞)- counter abstraction. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 107–122. Springer, Heidelberg (2002). doi:10.1007/3-540-45657-0_9
26. Sugihara, R., Gupta, R.K.: Clock synchronization with deterministic accuracy guarantee. In: Marrón, P.J., Whitehouse, K. (eds.) EWSN 2011. LNCS, vol. 6567, pp. 130–146. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19186-2_9
27. Talupur, M., Tuttle, M.R.: Going with the flow: parameterized verification using message flows. In: Formal Methods in Computer-Aided Design, FMCAD 2008, pp. 1–8, November 2008
28. Tan, L., Bu, L., Zhao, J., Wang, L.: Analyzing the robustness of FTSP with timed automata. In: Proceedings of the Second Asia-Pacific Symposium on Internetware, Internetware 2010, pp. 21:1–21:4. ACM, New York (2010)
29. Vasudevan, S., Kurose, J., Towsley, D.: Design and analysis of a leader election algorithm for mobile ad hoc networks. In: Proceedings of the 12th IEEE International Conference on Network Protocols, ICNP 2004, pp. 350–360. IEEE (2004)

Springer

http://www.springer.com/978-3-662-54576-8

Tools and Algorithms for the Construction and Analysis
of Systems
23rd International Conference, TACAS 2017, Held as
Part of the European Joint Conferences on Theory and
Practice of Software, ETAPS 2017, Uppsala, Sweden,
April 22-29, 2017, Proceedings, Part I
Legay, A.; Margaria, T. (Eds.)
2017, XXIV, 609 p. 152 illus., Softcover
ISBN: 978-3-662-54576-8