Verification of an industrial asynchronous leader election algorithm using abstractions and parametric model checking*

Étienne André[1,2,3], Laurent Fribourg[4], Jean-Marc Mota[5], and Romain Soulat[5]

[1] Université Paris 13, LIPN, CNRS, UMR 7030, F-93430, Villetaneuse, France
[2] JFLI, CNRS, Tokyo, Japan
[3] National Institute of Informatics, Japan
[4] LSV, ENS Paris-Saclay & CNRS & INRIA, U. Paris-Saclay
[5] Thales Research and Technology, Palaiseau, France

Abstract. The election of a leader in a network is a challenging task, especially when the processes are asynchronous, i. e., execute an algorithm with time-varying periods. Thales developed an industrial election algorithm with an arbitrary number of processes, that can possibly fail. In this work, we prove the correctness of a variant of this industrial algorithm. We use a method combining abstraction, the SafeProver solver, and a parametric timed model-checker. This allows us to prove the correctness of the algorithm for a large number p of processes ($p = 5000$).

Keywords: leader election, distributed algorithm, model checking, SaveProver, parameterized verification, parametric timed automata

1 Introduction

Distributed systems, where entities communicate with each other, are booming in our societies. Drones communicating with each other, swarms of various objects, intelligent cars... all may face communication and leadership issues. Therefore, the algorithm that all entities execute should be verified. Thales developed an industrial election algorithm with an arbitrary number of processes, that can possibly fail. We cannot describe the code of the actual algorithm for confidentiality issues. Therefore, we consider a modified variant of the algorithm. This algorithm focuses on the election of a leader in a distributed system with a potentially large number of entities or *nodes* in an *asynchronous* environment. Our main contribution is to perform a formal verification of the algorithm correctness for a large number of nodes. By correctness, we mean the actual election of the leader after a fixed number of rounds.

* This work is partially supported by Institut Farman (ENS Paris-Saclay & CNRS), by the ANR national research program PACS (ANR-14-CE28-0002) and by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST.

We consider here a special form of the general leader election problem [22]: we assume that, in the network, all the processes (or *nodes*) have a specific ID number, and they execute the same code (*symmetry*) in order to agree which ID number is the highest one. In the synchronous context where all processes communicate simultaneously, the problem is often solved using the "Bully algorithm" [18]. In the asynchronous context where each process communicates with a specific period possibly subject to delay variation (jitter), the problem is much more difficult. Periods can be all slightly different from each other, which makes the problem particularly complex. For example, a classical distributed leader election protocol, where the nodes exchange data using broadcasting, was designed by Leslie Lamport [20] in the asynchronous context. The correctness of this algorithm was proved mechanically many times using, e. g., TLA$^+$ tool [21], or, more recently, using the timed model checking tool UPPAAL [10]. However, these automated proofs work only for a small number p of processes, typically for $p \leq 10$. In this paper, we present a technique to prove the correctness of such a distributed leader election using automated tools for a large number of nodes (e. g., $p = 5000$). The principle of the method relies on the abstraction method consisting in viewing the network from the point of view of a specific (but arbitrary) node, say $node_i$, and considering the rest of the nodes of the network as an abstract environment interacting with $node_i$. In this abstract model, two basic properties of the algorithm can be proven. However, in order to prove the full correctness of the leader election algorithm, we will need an auxiliary model, where some timing information is added to (a raw form of) the abstract model. Using this auxiliary timed model, we are able to prove an additional property of the leader election algorithm. Thanks to the three aforementioned properties added as assumptions, we can then prove the full correctness of the leader election algorithm, using the bounded model checker SafeProver [16] on the abstract model.

The leader election algorithm we use is not Lamport's algorithm, but a simple asynchronous form of the Bully algorithm. We consider a specific framework of network structure and asynchronous form of communications. Basically, we assume that:

1. the graph is complete (every node communicate with all the other ones).
2. the communications are instantaneous (the time between the sending of a message and its reception is null), and the nodes exchange data via synchronous one-way unconditional value passing.
3. the processes are *visibly faulty*, i. e., they always execute the generic code of the algorithm, trying to elect the leader when they are non-faulty (mode On), and do nothing when they are faulty (mode Off).

Relationship with Thales' actual algorithm

As mentioned above, for confidentiality issue, we cannot reveal the original algorithm developed at Thales. Nevertheless, it is in essence the same as the one we present. Only the executed code has been modified. In addition, the technique

presented in this paper was designed for and applied to the original algorithm. To summarize, we present exactly the methodology, up to the content of the *UpdateNode* code (that is still similar in spirit).

After its verification using the techniques we present here, the original algorithm has been implemented in C, and is nowadays running in one of the Thales products. This product embeds a standard processor (in the line of Intel X86), with some limited RAM, hard drive, Ethernet ports, etc.

Related work

The method proposed here makes use of several powerful techniques such as counter abstraction, bounded model checking and parametric timed model checking for verifying distributed fault-tolerant algorithms, similarly to what has been recently described, e. g., in [19]. As said in [19]: "Symmetry allows us to change representation into a *counter representation* (also referred to as 'counter abstraction'): (...) Instead of recording which process is in which local state, we record for each local state, how many processes are in this state. Thus, we need one counter per local state ℓ, hence we have a fixed number of counters. A step by a process that goes from local state ℓ to local state ℓ' is modeled by decrementing the counter associated with ℓ and incrementing the counter associated with ℓ'. When the number p of processes is fixed, each counter is bounded by p." The work described in [19] makes use of SMT solvers [15] in order to perform finite-state model checking of the abstracted model.

Our work can be seen as a new application of such techniques to (a variant of) an industrial election algorithm. Another originality is to combine counter abstraction, bounded model checking, with *parametric timed* model checking.

In an orthogonal direction, the verification of identical processes in a network, i. e., a unknown number of nodes running the same algorithm, has been studied in various settings, notably in the long line of work around regular model checking [17,11], and in various settings in the *timed* case [3,1,2]. However, the focus of that latter series of works is on decidability, and they do not consider real-world algorithms, nor do they have tools implementing these results.

Finally, the line of works around the Cubicle model-checker [14,12,13] performs parameterized verification of cache memory protocols, that is also parameterized in the number of processes. However, timing parameters are not present in these works.

Outline

The rest of the paper is organized as follows. Section 2 introduces the variant of the leader election algorithm we consider. Section 3 presents our direct verification method for a small number of nodes. Section 4 presents our abstraction-based verification for a much larger number of nodes. Section 5 concludes the manuscript and outlines perspectives.

Table 1: Constants (in ms)

Constant	Value
$\mathsf{period}_{\mathsf{min}}$	49
$\mathsf{period}_{\mathsf{max}}$	51
$\mathsf{jitter}_{\mathsf{min}}$	-0.5
$\mathsf{jitter}_{\mathsf{max}}$	0.5

2 An asynchronous leader election algorithm

Thales recently proposed a leader election algorithm. This simple leader election algorithm is based on the classical Bully algorithm originally designed for the synchronous framework [18]. Basically, all nodes have an ID (all different), and the node with the largest ID must be elected as a leader. This algorithm is asynchronous. As usual, each node runs the same version of the code. We cannot describe the code of the actual algorithm for confidentiality issues, and we therefore consider and prove a modified variant of Thales' original algorithm, described throughout this section.

2.1 Periods, jitters, offset

The system is a fixed set of p nodes $\mathcal{N} = \{node_1, \dots, node_p\}$, for some $p \in \mathbb{N}$. Each node $node_i$ is defined by:

1. its integer-valued *ID* $node_i.id \in \mathbb{N}$,
2. its rational-valued *activation period* $node_i.per \in [\mathsf{period}_{\mathsf{min}}, \mathsf{period}_{\mathsf{max}}]$,
3. its rational-valued *first activation time* $node_i.start \in [0, node_i.per]$ (which can be seen as an *offset*, with the usual assumption that the offset must be less than or equal to the period), and
4. its rational-valued *jitter* values represent a delay variation for each period belonging to $[\mathsf{jitter}_{\mathsf{min}}, \mathsf{jitter}_{\mathsf{max}}]$, which is a static interval defined for all nodes and known beforehand.

Observe that all periods are potentially different (even though they are all in a fixed interval, and each of them remains constant over the entire execution), which makes the problem particularly challenging. In contrast, the jitter is different at each period (this is the common definition of a jitter), and the jitter of node i at the jth activation is denoted by $jitter_i^j$. The jth activation of node $node_i$ therefore takes place at time $t_i^j = t_i^{j-1} + node_i.per + jitter_i^j$. We have besides: $t_i^0 = node_i.start$.

The concrete values for the static timing constants are given in Table 1.

Example 1. Assume the system is made of three nodes. Assume $node_1.per = 49$. Recall that a period is an arbitrary *constant* in a predefined interval. Assume $node_1.start = 0$.

Assume $node_2.per = 51$ and $node_2.start = 30$.

Table 2: Jitter values for Example 1

	$jitter^1$	$jitter^2$	$jitter^3$
Node 1	0.5	−0.5	0.5
Node 2	0	0.1	0
Node 3	0.1	0.3	0.5

Assume $node_3.per = 49$ and $node_3.start = 0.1$.

Also assume the jitters for the first three activations of the nodes given in Table 2.

We therefore have $t_1^0 = 0$, $t_1^1 = 49.5$, $t_1^2 = 97.5$, $t_1^3 = 147.5$, $t_2^0 = 30$, $t_2^1 = 81$, $t_2^2 = 132.1$, $t_2^3 = 183$, $t_3^0 = 0.1$, $t_3^1 = 48.6$, $t_3^2 = 98.4$, $t_3^3 = 147.6$. The first activations of the nodes are depicted in Fig. 1. Due to both uncertain periods and the jitters, it can happen that, between two consecutive activations of a node, another node may not be activated at all: for example, between t_3^0 and t_3^1, node 1 is never activated, and therefore node 3 will not receive a message from node 1 during this interval. Conversely, between two consecutive activations of a node, another node is activated twice: for example, between t_3^1 and t_3^2, node 1 is activated twice (i. e., t_1^1 and t_1^2), and therefore node 3 may receive two messages from node 1 during this interval.

Finally note that, in this example, the number of activations since the system start for nodes 1 and 3 is always the same at any timestamp, up to a difference of 1 (due to the jitters) because they have the same periods. In contrast, the number of activations for node 2 will be smaller than that of nodes 1 and 3 by an increasing difference, since node 2 is slower (period: 51 instead of 49). This phenomenon does not occur when periods are equal for all nodes, and makes this setting more challenging.

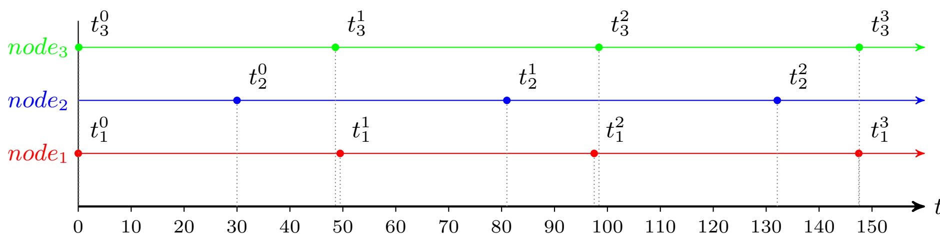

Fig. 1: Activation of three nodes with uncertain periods and jitters

Remark 1. The rest of this paper assumes the constant values given in Table 1. However, our method remains generic for constants of the same order of magnitude. Here, the variability of the periods is reasonably limited (around 4 %). A variability of more than 40 % will endanger the soundness of our method, as our

Algorithm 1: $UpdateNode(i)$

```
1  if node_i.EvenActivation then
2      allMessages ← ReadMailbox()
3      higherIDreceived ← false
4      foreach message ∈ allMessages do
5          if message.SenderID > node_i.id then
6              state_next ← Follower
7              higherIDreceived ← true
8      if ¬ higherIDreceived then
9          if node_i.state = Follower then
10             state_next ← Candidate
11         else if node_i.state = Candidate then
12             state_next ← Leader
13         else if node_i.state = Leader then
14             state_next ← Leader
15     node_i.state ← state_next
16 node_i.EvenActivation ← ¬node_i.EvenActivation
17 message = {node_i.id; node_i.state}
18 Send_To_All_Network(message)
```

upcoming assumption that between any three consecutive activations of a node, all nodes execute at least once, would not hold anymore.

2.2 IDs, modes, messages

We assume that all the IDs of the nodes in the network are different. Each node executes the same code. Each node has the ability to send messages to all the nodes in the network, and can store (at least) one message received from any other node in the network. Nodes are either in mode On and execute the code at each activation time, or do nothing when they are in mode Off. (This models the fact that some nodes in the network might fail.) A node in mode On is in one of the following states:

- Follower: the node is not competing to become leader;
- Candidate: the node is competing to become leader;
- Leader: the node has declared itself to be the leader.

Each transmitted message is of the form: $message = (SenderID, state)$ where $state$ is the state of the sending node.

2.3 The algorithm

At each new activation, $node_i$ executes the code given in Algorithm 1. In short, if the Boolean flag $node_i.EvenActivation$ (which we can suppose being initially

arbitrary) is true, then the code line 1–line 15 is executed. In this code, the node first reads its mailbox, and checks whether any message contains a higher node ID than the node ID (line 3–line 7) and, if so, sets itself as a follower (line 6). If no higher ID was received, the node "upgrades" its status from follower to candidate (line 10), from candidate to leader (line 12), or remains leader if already leader (line 14).

Finally (and this code is executed at every iteration), the node swaps the Boolean flag *EvenActivation* (line 16), prepares a message with its ID and current state (line 17) and sends this message to the entire network (line 18). We assume that the *Send_To_All_Network* function sends a message to all nodes—including the sender.

We can see that the significant part of the code (line 1–line 15) is only executed once every two activations (due to Boolean test $node_i.EvenActivation$). This is enforced in order to ensure that each node executes the code after receiving at least one message from all the other nodes (in mode On). However, note that each node sends a message at each iteration.

The order of magnitude of the constants in Table 1 gives the immediate lemma.

Lemma 1. *Assume a node i and activation times t_i^j and t_i^{j+2}. Then in between these two activations, node i received at least one message from all nodes.*

Proof. From Table 1 and Algorithm 1.

Remark 2. For different orders of magnitudes, we may need to execute the code once every more than two activations. For example, if we set $\mathsf{jitter}_{\mathsf{min}} = -25$ and $\mathsf{jitter}_{\mathsf{max}} = 25$ in Table 1, the code should be executed every three activations for our algorithm to remain correct.

2.4 Objective

We first introduce the following definitions.

Definition 1 (round). *A* round *is a time period during which all the nodes that are* On *have sent at least one message.*

Definition 2 (cleanness). *A round is said to be* clean *if during its time period no node have been switched from* On *to* Off *or from* Off *to* On*.*

The correctness property that we want to prove is:

"**When, following a preliminary clean round, 4 new clean rounds occur, the node with the highest ID is recognized as the leader by all the nodes in modes On of the network.**"

This property is denoted by (P) in the following.

Remark 3 (fault model). Our model does allow for faults but, according to Definition 2, only prior to the execution of the algorithm. That is, once it has started, all nodes remain in On or Off during its entire execution. If in reality there is a fault during the execution, it suffices to consider the execution of the algorithm at the next clean round.

Algorithm 2: SafeProver code for model M

```
1  Activation[1,...,p] ← [0,...,0]
   // Network initialization
2  foreach i ∈ {1,...,p} do
3      node_i.id ∈ ℕ
4      node_i.per ∈ [period_min; period_max]
5      node_i.start ∈ [0; node_i.per]
6      node_i.state ∈ {Follower, Candidate, Leader}
7      node_i.EvenActivation ∈ {true, false}
8      node_i.mode ∈ {On, Off}
9      nextActivationTime(i) ← node_i.start
   // Mailboxes initializations
10 foreach i ∈ {1,...,p} do
       // Arbitrary mailbox initialization with a message from oneself
11     node_i.mailbox ← [(node_i.id, Follower)]
12 foreach i ∈ {1,...,p} do
13     foreach j ∈ {1,...,p} do
14         if node_j.mode = On then
15             node_i.mailbox.enqueue(message_j)
   // Main algorithm
16 while true do
17     i ← arg min(nextActivationTime)
18     if node_i.mode = On then
19         UpdateNode(i)
20         Activation(i) ← Activation(i) + 1
21         jitter ∈ [jitter_min, jitter_max]
22         nextActivationTime(i) ← nextActivationTime(i) + node_i.per + jitter
```

3 Direct verification of the leader election algorithm

In this section, we first verify our algorithm for a fixed number of processes.

We describe here the results obtained by SafeProver on a model M representing directly a network of a fixed, constant number of p processes (without abstraction); for a small number p of nodes, we thus obtain a simple proof of the correctness of the algorithm. The model includes explicitly a representation of each node of $\mathcal{N}$ as well as their associated periods, first activation times, local memories, and mailboxes of received messages. The code is given in Algorithm 2. The mailbox is represented as a queue, initially filled with a message from oneself.[6]

During the initialization declaration, we set everything as free variables (with some constraints, e. g., on the periods) in order to have no assumptions on the

[6] An initial empty mailbox would do as well but, in the actual Thales system, this is the way the initialization is performed.

Table 3: Computation times

Nodes	Time (s)
$p = 4$	66.65
$p = 5$	215.61
$p = 6$	time out (> 3600)

state of the network at the beginning. This ensures that this model is valid whatever happened in the past, and this can be seen as a symbolic initial state: this notion of symbolic initial state was used to solve a challenge by Thales [23], also featuring uncertain periods. We also fully initialize the mailboxes of all the nodes since we are assuming that we are right after a clean round. The variable *Activation* is used as a variable to store how many times a node has been executed after the last clean round. The code of called function $UpdateNode(i)$ corresponds exactly to Algorithm 1.

The property (P) we want to prove is formalized as:

$$\big(\forall i \in \{1, \ldots, p\}, Activation(i) \geq 4\big) \;\Rightarrow$$
$$\big(\forall j \in \{1, \ldots, p\}, j \neq maxId : node_j.state = \mathsf{Follower}$$
$$\wedge\; node_{maxId}.state = \mathsf{Leader}\big)$$

with $maxId = \arg\max(\{node_i.id \mid node_i.mode = \mathsf{On}\}_{i \in \{1,\ldots,p\}})$. Using this model and SafeProver [16], we obtain the proof of (P) with the times tabulated in Table 3.[7] While this method allows us to formally prove the leader election for up to 5 nodes, SafeProver times out for larger number of nodes. This leads us to consider another method to prove the correctness of our algorithm for larger numbers.

4 Abstraction-based method

We now explain how to construct an abstract model $\widehat{M}$ of the original model M. This model $\widehat{M}$ clusters together all the p processes, except the process $node_i$ under study (where i is arbitrary, i. e., a free variable); $\widehat{M}$ also abstracts away the timing information contained in M. We then use SaveProver to infer two basic properties P1 and P2 for $\widehat{M}$.

In a second phase, we consider an auxiliary simple (abstract) model T of M which merely contains relevant timing information; we then use a parametric timed model checker to infer a third property (P3) for T. The parametric timed model checker is required due to the *uncertain* periods, that can have any value in $[\mathsf{period}_{\mathsf{min}}, \mathsf{period}_{\mathsf{max}}]$ but remain constant over the entire execution.

In the third phase, we consider again the model $\widehat{M}$, and integrate P1–P3 to SafeProver as assumptions, which allows us to infer a fourth property P4. The

[7] All the experiments reported in this paper have been run on a machine with two Intel® Xeon® CPU E5-2430 at 2.5 GHz, with 164 GiB of RAM and running a Debian 9 Linux distribution.

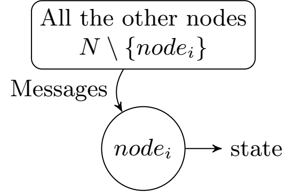

Fig. 2: Scheme of model $\widehat{M}$ with node i under study interacting with the cluster of all the other nodes

properties P1 and P4 express together a statement equivalent to the desired correctness property P of the leader election algorithm. The advantage of reasoning with abstract models $\widehat{M}$ and T rather than directly to M, is to prove P for a large number p of processes.

We now describe our method step by step in the following.

4.1 Abstract model $\widehat{M}$ and proof of P1-P2

The idea is to model the system as one node $node_i$ (the node of interest) interacting with the rest of the network: $node_i$ receives messages from the other nodes which are clustered into a single abstract process (see Fig. 2). In the abstract model $\widehat{M}$, each node can take any state at any activation, with no regards to the parity ($node_i.EvenActivation$), what has been previously sent, what $node_i$ is sending. We only consider the activation of $node_i$. The rest of the network is abstracted by the messages contained in the mailbox of $node_i$. Since we assume that at least one clean round has passed, we always have a message from a working node in the mailbox. The code is given in Algorithm 3. (Note its analogy with the SafeProver code of Algorithm 2 for M.) The first four lines define free variables.

The list of assumptions that the solver can make on the messages received is denoted by $List_of_assumptions_on_message_j$. This list is initially empty, and augmented with "guarantees" (a.k.a. "proven properties") on the other nodes as they are iteratively generated by the solver. The first proven properties are:

- P1: $(Activation(j) \geq 2 \ \wedge \ node_j.id \neq maxId) \Rightarrow \ node_j.state = \mathsf{Follower}$
- P2: $(Activation(j) \geq 2 \ \wedge \ node_j.id = maxId)$
  $\Rightarrow node_j.state \in \{\mathsf{Candidate}, \mathsf{Leader}\}$

In these properties, j is a free variable: therefore, it is "fixed" among one execution, but can correspond to any of the node IDs. The two properties state that, after two rounds, a node which has not the largest ID is necessarily a follower (P1), or a candidate or a leader if it has the largest ID (P2). As said before, P1 and P2 are then integrated to $List_of_assumptions_on_message_j$.

$Guarantee_to_prove$ contains iteratively P1, then P1 and P2, then P1, P2 and P4.

Algorithm 3: SafeProver code for abstract model $\widehat{M}$

```
1  Assume i ∈ {1, ..., p}
2  Assume node_i.id ∈ ℕ
3  Assume node_i.state ∈ {Follower, Candidate, Leader}
4  Assume node_i.EvenActivation ∈ {true, false}
5  Activation(i) ← 0
6  while true do
7      for j ∈ {1, ..., p} do
8          message_j ∈ {node_j.id} × {Follower, Candidate, Leader}
9          Assume: List_of_assumptions_on_message_j
10         node_i.mailbox.enqueue(message_j)
11     UpdateNode(i)
12     Activation(i) + +
13     Guarantee_to_prove
```

4.2 Abstract model T and proof of P3

To represent the timed abstract model T of M, we use an extension of the formalism of timed automata [5], a powerful extension of finite-state automata with clocks, i. e., real-valued variables that evolve at the same time. These clocks can be compared with constants when taking a transition ("guards"), or to remain in a discrete state ("invariants"). Discrete states are called *locations*. Timed automata were proven successful in verifying many systems with interactions between time and concurrency, especially with the state-of-the-art model-checker Uppaal [10]. However, timed automata cannot model and verify arbitrary periods: while it is possible to model a different period at each round, it is not possible to first fix a period once for all (in an interval), and then use this period for the rest of the execution. We therefore use the extension "parametric timed automata" [6,8] allowing to consider *parameters*, i. e., unknown constants (possibly in an interval). IMITATOR [9] is a state-of-the-art model checker supporting this formalism.

In our method, the timed abstract model T of M is a product of two similar parametric timed automata representing the node i under study and a generic node j belonging to $\mathcal{N} \setminus \{i\}$ respectively. Each parametric timed automaton contains a single location. The parametric timed automaton corresponding to $node_i$ uses an activation period per_i that we model as a parameter. Indeed, recall that the period belongs to an interval: taking a value in the interval at each round would not be correct, as the period would not be constant. This is where we need parameters in our method. In addition, we constrain this parameter per_i to belong to $[\mathsf{period}_{\mathsf{min}}, \mathsf{period}_{\mathsf{max}}]$. Each automaton has its own clock c_i that is used to measure how much time has passed since the last activation. Each

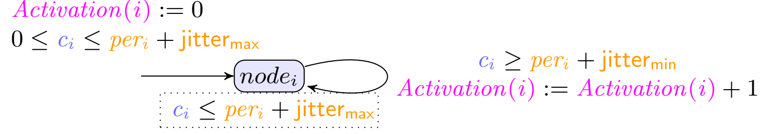

Fig. 3: Component 1 of timed model T

automaton has a discrete variable[8] $Activation(i)$ which is initialized at 0 and is used to count the number of activations for this node. We give the constraint on c_i at the beginning that $c_i \in [0, per_i + \mathsf{jitter}_{\mathsf{max}}]$. An activation can occur as soon as c_i reaches $per_i + \mathsf{jitter}_{\mathsf{min}}$. This is modeled by the guard $c_i \geq per_i + \mathsf{jitter}_{\mathsf{min}}$ on the transition that resets c_i and increment $Activation(i)$. An activation can occur as long as c_i is below or equal to $per_i + \mathsf{jitter}_{\mathsf{max}}$. This is modeled by the invariant $c_i \leq per_i + \mathsf{jitter}_{\mathsf{min}}$ on the unique location of the automaton. This invariant forces the transition to occur when c_i reaches its upper bound. This parametric timed automaton is represented in Fig. 3.[9] The other component representing the cluster of the rest of the nodes is modeled similarly as a generic component $node_j$.

For nodes $node_i$ and $node_j$, the property that we want to specify corresponds in the direct model M (without abstraction) of Section 3 to:

– $(Activation(i) \leq 13 \ \wedge \ Activation(j) \leq 13)$
$$\Rightarrow |\ Activation(i) - Activation(j)\ | \leq \ 2.$$

In our timed abstract model T, such a property becomes:

– (P3): $\forall i \in \{1, \dots, p\}\ Activation(j) \leq 13 \Rightarrow$
$-2 \leq Activation(j) - Activation(i) \leq 1.$

where $Activation(i)$ denotes the number of activations of node i since the last clean round.

The value "13" has been obtained experimentally: smaller numbers led the algorithm to fail (the property was not satisfied). Intuitively, it consists in the number of activations by which we are sure the leader will eventually be elected.

The proof of P3 is obtained by adding to the model an observer[10] automaton checking the value of the discrete variables $Activation(i)$ and $Activation(j)$, which goes to a *bad* location when the property is violated. The property is then verified by showing that the *bad* location is not reachable. For the values

[8] Discrete variables are global Boolean- or integer-valued variables, that can be read or written in transition guards. If their domain is finite they are syntactic sugar for a larger number of locations.

[9] The color code is that of IMITATOR automated LaTeX outputs: clocks are in blue, parameters in orange, and discrete variables in pink.

[10] An observer is an additional automaton that can synchronize with the system (using synchronized actions, clocks or discrete variables values), without modifying its behavior nor blocking it. See e. g., [4,7].

Table 4: Computation times

Nodes	Time (s)
$p = 500$	13.34
$p = 1000$	45.95
$p = 5000$	623.46

of the timing constants in Table 1, IMITATOR proves P3 (by showing the non-reachability of the bad location) in 12 s. Recall that, thanks to our assumption on the number of nodes, we only used two nodes in the input model for IMITATOR.

In the next part, we show how the addition of P3 as an assumption in the original abstract model allows to prove the desired property P for a large number of nodes.

4.3 Proof of P using P1–P3 as assumptions

In addition to P1-P2, we now put P3 ($Activation(j) \in [Activation(i) - 1; Activation(i)+2]$) as an element of $List_of_assumptions_on_message_j$ used in the SafeProver code of $\widehat{M}$ (see Algorithm 3). SafeProver is then able to generate the following property:

$$P4 : (Activation(i) \geq 4 \ \wedge \ node_i.id = maxId) \Rightarrow node_i.state = \mathsf{Leader}$$

Property P4 states that the node with the highest ID will declare itself as Leader after at most 4 activations. Besides, property P1 states that a node, not having the highest ID, is in the state Follower within at most 2 activations. Properties P1 and P4 together thus express a statement equivalent to the desired correctness property P. The global generation of properties (P1), (P2) and (P4) by SafeProver takes the computation times tabulated in Table 4. As one sees, the computation time is now smaller by an order of magnitude than the ones given in Table 3, thus showing the good scalability of our method.

Remark 4. Note that verifying the model for 5,000 nodes also gives a guarantee for any smaller of nodes. Indeed, we can assume that an arbitrary number of nodes are in mode Off, and remain so, which is equivalent to a smaller number of nodes.

4.4 Discussion

Soundness We briefly discuss the soundness of the algorithm. First, note that the assumptions used above have been validated by the system designers (i. e., those who designed the algorithm). Second, SafeProver validated the assumptions, i. e., proved that they were not inconsistent with each other (which would result in an empty model).

Now, the abstraction used in Section 4.2, i. e., to consider only two nodes, is the one which required most human creativity. Let us briefly discuss its soundness. Our abstraction allows to model the sending of any message, which includes the actual message to be sent in the actual system. The fact that a message was necessarily received in the actual system between two (real) executions of the node under study is given by the fact that all nodes necessarily execute at least once in the last two periods (see Lemma 1). Of course, this soundness is only valid under our own assumptions on the variability of the period, considering the constants in Table 1: if the period of one node is 1 while the other is 100, our framework is obviously not sound anymore.

Parametric vs. parametrized model checking As shown in Table 4, we verified the model for a *constant* number of nodes. This comes in contrast with the recent work on parameterized verification (e. g., [2,12,13]). However, while these latter consider a parameterized number of nodes, they consider *non-parametric* timed models; in contrast, we need here parametric timed models to be able to represent the uncertainty on the periods. Combining both worlds (parameterized number of nodes with parametric timed models) would be of interest—but remains a very challenging objective.

5 Conclusion

We described a method combining abstraction, SafeProver and parametric timed model-checking in order to prove the correctness of a variant of an asynchronous leader election algorithm designed by Thales. Our approach can efficiently verify the leader election after a fixed number of rounds for a large number p of processes (up to $p = 5000$). The method relies on the construction of two abstract models $\widehat{M}$ and T of the original model M. Although it is intuitive, it could be interesting to prove formally that each abstraction is *correct* in the sense that it *over-approximates* all the behaviors of M.

Perspectives

Many variants of the algorithm can be envisioned (loss of messages, non-instantaneous transmission, non-complete graph topology, . . .). The fault model could also be enriched. It will then be also interesting to propose extensions of our abstraction-based method to prove the correctness of such extensions.

The correctness of the method relies on the order of magnitude of the constants used (Remark 1). For different intervals, it might be necessary to both adapt the algorithm (read messages only every k activations) but also the assumptions used in the proof using abstraction, a manual and possibly error-prone process. A more general verification method would be desirable.

In addition, the number of activations in our correctness property ("after 13 activations, the leader is elected") was obtained using an incremental verification (values of up to 12 all gave concrete counterexamples). As a future work, we

would like to automatically infer this value too, i. e., obtaining the minimal value of activations before a leader is guaranteed to be elected.

Finally, adding probabilities to model the fault of nodes will be of interest.

Acknowledgment

We thank anonymous reviewers for very useful remarks and suggestions.

References

1. Abdulla, P.A., Delzanno, G., Rezine, O., Sangnier, A., Traverso, R.: On the verification of timed ad hoc networks. In: Fahrenberg, U., Tripakis, S. (eds.) Proceedings of the 9th International Conference Formal Modeling and Analysis of Timed Systems (FORMATS 2011). Lecture Notes in Computer Science, vol. 6919, pp. 256–270. Springer (2011)
2. Abdulla, P.A., Delzanno, G., Rezine, O., Sangnier, A., Traverso, R.: Parameterized verification of time-sensitive models of ad hoc network protocols. Theoretical Computer Science 612, 1–22 (2016)
3. Abdulla, P.A., Jonsson, B.: Model checking of systems with many identical timed processes. Theoretical Computer Science 290(1), 241–264 (2003)
4. Aceto, L., Bouyer, P., Burgueño, A., Larsen, K.G.: The power of reachability testing for timed automata. In: Arvind, V., Ramanujam, R. (eds.) Proceedings of the 18th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1998). Lecture Notes in Computer Science, vol. 1530, pp. 245–256. Springer (1998)
5. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
6. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: Kosaraju, S.R., Johnson, D.S., Aggarwal, A. (eds.) Proceedings of the twenty-fifth annual ACM symposium on Theory of computing (STOC 1993). pp. 592–601. ACM, New York, NY, USA (1993)
7. André, É.: Observer patterns for real-time systems. In: Liu, Y., Martin, A. (eds.) Proceedings of the 18th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2013). pp. 125–134. IEEE Computer Society (Jul 2013)
8. André, É.: What's decidable about parametric timed automata? International Journal on Software Tools for Technology Transfer (2019), to appear
9. André, É., Fribourg, L., Kühne, U., Soulat, R.: IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In: Giannakopoulou, D., Méry, D. (eds.) Proceedings of the 18th International Symposium on Formal Methods (FM 2012). Lecture Notes in Computer Science, vol. 7436, pp. 33–36. Springer (2012)
10. Behrmann, G., David, A., Larsen, K.G.: A tutorial on uppaal. Lecture Notes in Computer Science, vol. 3185, pp. 200–236. Springer (2004)
11. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Emerson, E.A., Sistla, A.P. (eds.) Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000). Lecture Notes in Computer Science, vol. 1855, pp. 403–418. Springer (2000)

12. Conchon, S., Declerck, D., Zaïdi, F.: Compiling parameterized X86-TSO concurrent programs to cubicle-W. In: Duan, Z., Ong, L. (eds.) Proceedings of the 19th International Conference on Formal Engineering Methods (ICFEM 2017). Lecture Notes in Computer Science, vol. 10610, pp. 88–104. Springer (2017)
13. Conchon, S., Declerck, D., Zaïdi, F.: Parameterized model checking modulo explicit weak memory models. In: Laleau, R., Méry, D., Nakajima, S., Troubitsyna, E. (eds.) Proceedings of the Joint Workshop on Handling IMPlicit and EXplicit knowledge in formal system development (IMPEX) and Formal and Model-Driven Techniques for Developing Trustworthy Systems (FM&MDD), IMPEX/FM&MDD 2017. EPTCS, vol. 271, pp. 48–63 (2017)
14. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaïdi, F.: Cubicle: A parallel SMT-based model checker for parameterized systems – tool paper. In: Madhusudan, P., Seshia, S.A. (eds.) Proceedings of the 24th International Conference on Computer Aided Verification (CAV 2012). Lecture Notes in Computer Science, vol. 7358, pp. 718–724. Springer (2012)
15. De Moura, L., Bjørner, N.: Satisfiability modulo theories: Introduction and applications. Communications of the ACM 54(9), 69–77 (Sep 2011)
16. Étienne, J.F., Juppeaux, É.: Safeprover: A high-performance verification tool. ACM SIGAda Ada Letters 36(2), 47–48 (May 2017)
17. Fribourg, L., Olsén, H.: Reachability sets of parameterized rings as regular languages. Electronic Notes in Theoretical Computer Science 9, 40 (1997)
18. García-Molina, H.: Elections in a distributed computing system. IEEE Transactions on Computers 31(1), 48–59 (1982)
19. Konnov, I.V., Veith, H., Widder, J.: What you always wanted to know about model checking of fault-tolerant distributed algorithms. In: Ershov Memorial Conference. Lecture Notes in Computer Science, vol. 9609, pp. 6–21. Springer (2015)
20. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems 16(2), 133–169 (1998)
21. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
22. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996)
23. Sun, Y., André, É., Lipari, G.: Verification of two real-time systems using parametric timed automata. In: Quinton, S., Vardanega, T. (eds.) Proceedings of the 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2015) (Jul 2015)