Formal Verification of an Industrial Distributed Algorithm: an Experience Report

Nikolai Kosmatov[0000−0003−1557−2813], Delphine Longuet[0000−0002−8394−276X], and Romain Soulat[0000−0003−4431−5250]

Thales Research and Technology, Palaiseau, France

Abstract. Verification of distributed software is a challenging task. This paper reports on modeling and verification of a consensus algorithm developed by Thales. The algorithm has an arbitrary number of processes (nodes), which can possibly fail and restart at any time. Communications between nodes are periodic, but completely asynchronous. The goal of this algorithm is that, after a given amount of time since the last status change, the network of nodes agrees on a list of working nodes. Our verification approach is based on modeling both the source code of the algorithm and the possible interleavings of executions. We present how we were able to scale up to 100 processes using the rely-guarantee based technique. Some of the initially expected properties did not hold, and generated counter-examples helped to fix and prove them. We also successfully verified other consensus algorithms at Thales with the same approach. We describe our experiments on applying several model-checking tools and a symbolic execution tool, and present some lessons learned.

Keywords: distributed algorithm verification, consensus algorithms, symbolic model checking, rely-guarantee, symbolic execution

1 Introduction

Distributed software is largely used nowadays. The advance of Internet of Things (IoT) devices and autonomous systems promises their ever-growing use in the next generations of industrial systems. Design and verification of distributed algorithms for distributed software remain a very active research topic since several decades. Many efficient algorithms were proposed [17,18], and important impossibility results were established [11].

Distributed algorithms include in particular *consensus protocols*, where the processes (nodes) of a network, executing the same code, have to come to the agreement on some data. One example is *leader election protocols*, where several processes have to choose a unique leader. In the synchronous context, the nodes exchange information on the common clock signal, while in the asynchronous one, they can communicate at different moments. A well-known synchronous leader election protocol is the so-called *Bully algorithm* [12] in which the node with the highest ID is elected as a leader after several rounds. In the asynchronous context, a leader election protocol was described by Leslie Lamport [15]. It was formally proved correct using several tools, in particular, in TLA$^+$ [16], or, more recently,

in the timed model checking tool Uppaal [6]. However, the existing automated proofs were performed only for a small number of processes, typically ≤ 10.

The verification of a given protocol is tightly linked to the considered setting: synchronous or asynchronous, specific fault models, expected properties, values of periods of message exchanges, the degree of possible period variations and communication delays in a given system, etc. While many protocols were proposed in the literature, the industrial practice shows that their assumptions and properties do not always correspond to those of the target real-life system, and some specific (variants of) algorithms can be needed. Of course, as soon as the initial assumptions are modified, each new algorithm should be verified again.

Thales designed several distributed consensus algorithms where an arbitrary number of processes are run and can possibly fail and restart at any moment. Thus, each node can be on or off, but these statuses can change. Communications between nodes are periodic, but completely asynchronous. The goal of these algorithms is that, after a given amount of time since the last status modification, the network of nodes agrees on a list of working nodes. Thus, the final property of interest states that after a given number of activations, each node identifies the same set of nodes as working, that is, the consensus is reached. They also ensure some partial consistency properties after a smaller amount of time, which express step-by-step progress of the algorithm towards the desired final property.

This paper presents an experience report on formal verification of a distributed consensus algorithm developed at Thales. This experience was mainly realized by the formal methods group of Thales Research and Technology, with participation of other Thales engineers. The algorithm itself is not detailed in the paper: first, it is broadly inspired by the existing algorithms, and second, the precise algorithm cannot be revealed due to confidentiality reasons. We describe the methodology used to model and formally prove the initial algorithm as well as some similar algorithms.

Our verification approach combines several ideas from previous work, in particular, on modeling the interleavings using a simulation loop to represent the whole system (e.g. [7]), and the technique of rely-guarantee [13] that allows us to focus on the behavior of a given node rather than the whole system. We first modeled the source code of the algorithm and simulated the possible interleavings of executions, and used synchronous model checkers to verify the model. Possible variations of communication periods and communication delays in the target system are simulated using *jitters*, slightly modifying the activation times of the nodes. However, this technique does not allow us to prove the algorithm for a large number of nodes. Then we created a second, abstracted model focusing on the execution of a unique node and modeling the behavior of other nodes by assumptions, and attempting to prove that each of them is indeed guaranteed by the current node. This allowed the proof to scale up to 100 processes, and to validate the C code against those assumptions. We also verified other consensus algorithms at Thales with the same approach. Some of the initially expected properties did not hold, and generated counter-examples helped to fix and prove

them. We describe our experiments on applying several model-checking tools and a symbolic execution tool, and present some lessons learned.

This paper builds on a previously published case study [5] on verification of an industrial algorithm at Thales using the SafeProver tool [10]. This work extends it with a clearer presentation of the methodology, additional experiments with two other verification tools, Cbmc [9] and Klee [8], and some lessons learned. Recent case studies for similar algorithms confirmed the applicability of this work.

Outline. The paper is organized as follows. Section 2 gives an overview of the target system and algorithm. The verification methodology is presented in Sect. 3. Our experiments using different verification tools are reported in Sect. 4. Finally, Section 5 concludes the paper with some lessons learned and future work.

2 Presentation of the System and the Algorithm

2.1 Overview of the System

For confidentiality reasons, we cannot disclose the exact real-life system and algorithm, but we believe it does not prevent from understanding our approach and results. In particular, system parameters were modified in the paper but the consistency of the presented algorithm and results was of course preserved.

The system is composed of several identical computing nodes. They can perform various tasks and receive a part of the workload. The nodes are fully interconnected, which means that any node can send messages to any other node in the network, for example, to communicate computation results. On top of those messages, periodically, each node sends to all other nodes a special kind of message indicating that the sender is still alive and providing some additional data. Our distributed algorithm uses these messages in order for each node to be able to compute a correct list of all working nodes in the network. This has several uses: workload balancing, clock synchronization, leader election, etc.

More formally, the system is a fixed set of p nodes $\mathcal{N} = \{node_1, \ldots, node_p\}$, $p \in \mathbb{N}$, given with integer timing constants $\mathsf{period}_{\mathsf{min}}$, $\mathsf{period}_{\mathsf{max}}$, $\mathsf{jitter}_{\mathsf{min}}$, $\mathsf{jitter}_{\mathsf{max}}$, $\mathsf{msgDelay}_{\mathsf{min}}$, $\mathsf{msgDelay}_{\mathsf{max}}$. Each node $node_i$ is a record containing the following fields:

1. a unique integer-valued *ID*: $id \in \mathbb{N}$,
2. an integer-valued *activation period*: $per \in [\mathsf{period}_{\mathsf{min}}, \mathsf{period}_{\mathsf{max}}]$,
3. an integer-valued *first activation time*: $start \in [0, per[$ (which can be seen as an *offset*, with the usual assumption that the offset is less than the period),
4. a flag *failure*, $failure \in \{\bot, \top\}$, which corresponds to the state of the node, $\top$ means that the node has failed, $\bot$ means that the node is running,
5. a *receive failure* flag, $rcvFailure \in \{\bot, \top\}$, indicating a failure in the capacity to receive messages ($\top$ means that the node cannot receive messages, $\bot$ means that the node can receive messages), and
6. a *send failure* flag, $sndFailure \in \{\bot, \top\}$, indicating a failure in the capacity of the node to send messages.
7. a field *state* represents the set of internal variables of the node,

Constant	Value
$\mathsf{period}_{\mathsf{min}}$	49
$\mathsf{period}_{\mathsf{max}}$	51
$\mathsf{jitter}_{\mathsf{min}}$	-0.5
$\mathsf{jitter}_{\mathsf{max}}$	0.5
$\mathsf{msgDelay}_{\mathsf{min}}$	0
$\mathsf{msgDelay}_{\mathsf{max}}$	0

Node	*per*	*start*	$jitter_i^1$	$jitter_i^2$	$jitter_i^3$
$node_1$	49	0	0.5	-0.5	0.2
$node_2$	51	30	0	0.1	0
$node_3$	49	0.1	0.1	-0.5	0.5

Fig. 1. (a) Static constants (in ms), and (b) values chosen for the nodes in Example 1.

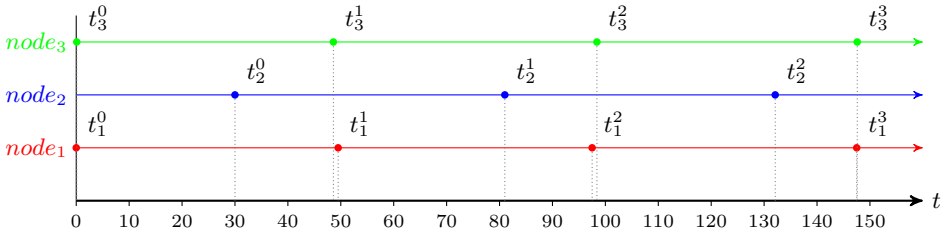

Fig. 2. Activation times (in ms) of the three nodes of Example 1.

8. a field *networkView* indicates the definitive belief of the node on the list of fully working nodes in the network.

In addition, we use the following constants to model uncertainties (time variations) in the system execution:

1. an integer $jitter_i^j \in [\mathsf{jitter}_{\mathsf{min}}, \mathsf{jitter}_{\mathsf{max}}]$, indicating a delay in the execution of $node_i$ for the j-th activation,
2. an integer $msgDelay_{ki}^j \in [\mathsf{msgDelay}_{\mathsf{min}}, \mathsf{msgDelay}_{\mathsf{max}}]$ giving a delay in message transmission between $node_k$ and $node_i$ for the j-th activation of $node_i$.

The j-th activation of node $node_i$ occurs at time $t_i^j = t_i^{j-1} + node_i.per + jitter_i^j$ for $j > 0$. We set besides: $t_i^0 = node_i.start$.

In all our models we use values in microseconds for periods, activation times, jitters, message delays, etc., in order to be as close as possible to a real-life execution and to be able to model all possible interleavings, while still using integer values, easier to read and often better supported by tools. (In the examples in this paper, we use milliseconds just to make the constants more readable.)

Note that the periods of different nodes can be different, which makes the problem particularly challenging. While the period remains constant over the node's entire execution, its effective activation time can still be modified by a jitter, varying at each activation (this is the common definition of a jitter).

The static timing parameters are defined by the target system constraints. In this paper, we use the (anonymized) values given in Fig. 1a.

Example 1. Assume the system is made of three nodes. The periods, start times, and jitters for the first three activations of the nodes are given in Fig. 1b.

We therefore have $t_1^0 = 0, t_1^1 = 49.5, t_1^2 = 98, t_1^3 = 147.2, t_2^0 = 30, t_2^1 = 81, t_2^2 = 132.1, t_2^3 = 183.1, t_3^0 = 0.1, t_3^1 = 49.2, t_3^2 = 97.7, t_3^3 = 147.2$. The first activations

Algorithm 1: Pseudo-code of function $UpdateNode(i)$

```
1 if node_i.EvenActivation then
2 |   allMessages ← ReadMessages(i)
3 |   nodeState ← ComputeState(allMessages, nodeState)
4 |   message ← ComputeMessage(nodeState)
5 if ¬node_i.sndFailure then
6 |   SendToAllNetwork(message, currentTime)
7 node_i.EvenActivation ← ¬node_i.EvenActivation
```

of the nodes are depicted in Fig. 2. Due to both uncertain periods and jitters, it can happen that, 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, between two consecutive activations, node 3 may receive two messages from node 1, while node 1 receives no new message from node 3.

Finally note that the number of activations of different nodes can increase at a different speed. The number of activations since the system start for nodes 1 and 3 having the same periods remains roughly the same at any timestamp: initially equal to at most 1, the difference can change very slowly (with the values of Fig. 1a, due to jitters, it can increase by 1 after at least j activations such that $(49-0.5)(j+1) < (49+0.5)j$, i.e. $j > 48.5$). The numbers of activations for node 2 and nodes 1 and 3 can evolve with a more rapidly increasing difference.

Nodes can fail and restart. We consider in our models 5 failure modes:

- F1: A node can stop and flush its internal memory. When restarting, the node will believe it is alone in the network until it receives messages from the other nodes. This corresponds to a node shutting down.
- F2: A node can stop and keep its internal memory. When restarting, the node will have the same state as before stopping, it will still assume the network in the same state as when it stopped, until it receives messages that will contradict this belief. This corresponds to a node freezing.
- F3: A node can stop sending and receiving messages. This corresponds to a disconnection from the network.
- F4/F5: A node can stop receiving (resp., sending) messages but still be able to emit (resp. receive) messages. This corresponds to a partial disconnection.

2.2 Overview of the Algorithm

We briefly describe how we modeled the node operation in this part. We assume that $currentTime$, the timestamp of the node execution, is a global variable that can be accessed by any node. (In practice, the view of the time by the node can be slightly imprecise. This imprecision is covered in the model by jitters and message delays.)

At each activation, each $node_i$ executes a code similar to the one given in Algorithm 1. Due to asynchronicity, the main part of the code treating received

messages is executed only once every two activations (when the boolean flag $node_i.EvenActivation$ is true, cf. lines 1–4), while a working node sends a message at each activation. This ensures that if a node is present in the network, then any other node has received at least one message from it (cf. Fig. 2 for an example of when it is necessary).

When $node_i.EvenActivation$ is true, the node first reads the content of its mailbox (line 2). Then it computes its new state from the messages it received (line 3), and a message to be sent to the rest of the network (line 4). The code of these two parts is confidential. Finally, at every iteration, if the sending capacity has not failed, it sends a message to the rest of the network (using the $SendToAllNetwork$ function, cf. lines 5–6). The message is either the one that has just been computed, or the repetition of the old message but with the current timestamp. It then swaps the flag $EvenActivation$.

3 Methodology

One of our objectives was to develop a methodology and models that could be read by software developers not specialized in formal verification, and hence, must be as straightforward as possible. The first model, M_{sim}, simulating the whole network, is presented in Sect. 3.1. However, its verification was not able to scale up to the real number of nodes so we developed an abstract model, M_{abs}, presented in Sect. 3.2, that uses the rely-guarantee and abstraction techniques. The models were specified in a simple subset of an imperative C-like language (and were adapted to the input language of the tools used in our experiments). We present their simplified pseudo-code versions in this paper.

3.1 Modeling the Whole System by Simulation

The model M_{sim} simulates a network of a fixed, constant number of p processes. It explicitly represents all nodes, with their associated periods, first activation times, local memories, and mailboxes of received messages. A pseudo-code of M_{sim} is given in Algorithm 2. The mailbox of each node is represented as a list containing the messages received from every other node (or nothing, if such messages were not received).

In the initialization phase, we assume (cf. lines 1–3 in Algorithm 2) that all variables are initialized by arbitrary values. It is equivalent to be initialized by a call to an uninterpreted function $nondet()$ (for which, by abuse of notation, the same function name will be used to return data of the relevant type). Necessary constraints (e.g. on the periods, cf. line 7) are introduced by the assume clause. The assumption on line 4 states that all node IDs are different. In this way, a symbolic initial state is created. (This notion of a symbolic initial state was earlier used to solve a challenge by Thales [19], also featuring uncertain periods.) The variable $Activation_i$ is used to store how many times $node_i$ has been executed.

The code of function $UpdateNode(i)$ is given by Algorithm 1. To take into account possible receiving failures and delays, the $ReadMessages(i)$ function selects the messages received by $node_i$ to consider (unless it lost the ability to

Algorithm 2: Pseudo-code of model M_{sim} simulating p nodes

```
   // Network initialization for nodes i=1,2,...,p
 1 State : node_i.state
 2 Integer : node_i.id, node_i.per, node_i.start, Activation_i, nextActivationTime_i
 3 Boolean :
     node_i.EvenActivation, node_i.failure, node_i.rcvFailure, node_i.sndFailure
 4 Assume : AllDifferent(node_i.id | i ∈ 1,...,p)  // Identifiers are unique
 5 foreach i ∈ {1,...,p} do
 6 |   Activation_i ← 0
 7 |   Assume : period_min ≤ node_i.per ≤ period_max
 8 |   Assume : 0 ≤ node_i.start < node_i.per
 9 |   nextActivationTime_i ← node_i.start
   // Mailbox initialization
10 ...
   // Main algorithm simulating the network execution
11 while true do
12 |   i ← indexMin(nextActivationTime)  // Node with the smallest time
13 |   if ¬node_i.failure then
14 |   |   UpdateNode(i)  // Execute the node
15 |   |   Activation_i ← Activation_i + 1
16 |   jitter ← nondet()  // Choose an arbitrary value for a new jitter
17 |   Assume : jitter_min ≤ jitter ≤ jitter_max  // in the considered bounds
18 |   nextActivationTime_i ← nextActivationTime_i + node_i.per + jitter
19 |   Assert : P_1 ∧ ··· ∧ P_α  // Partial and final properties
```

receive messages, i.e. $node_i.rcvFailure$ is true). For each other node $node_k$, it checks when the latest message from $node_k$ has been sent. If the message has been sent since less than $\mathsf{msgDelay}_{\mathsf{min}}$, we consider that it has not yet reached $node_i$, and we use the previous message from $node_k$. If the latest message has been sent a long time ago (since more than $\mathsf{msgDelay}_{\mathsf{max}}$), we consider it has reached $node_i$ and we use this message. If the message has been sent between $\mathsf{msgDelay}_{\mathsf{min}}$ and $\mathsf{msgDelay}_{\mathsf{max}}$, the message may or may not have reached the node, in that case we consider one of the two last messages to model both possible cases.

The main simulating loop of Algorithm 2 chooses $node_i$ to execute next as the node having the smallest $nextActivationTime$ (line 12). If $node_i$ did not fail, it is executed and its activation counter incremented (lines 13–15). Then a jitter is chosen and the next activation time computed (lines 16–18).

The final consensus achievement property P_α we want to prove for working nodes since the last node status change is of the form:

$$\begin{aligned} \forall k \in \{1,\dots,p\},\, (\, node_k \in networkState \,\wedge Activation_k \geq \alpha\,) \\ \Rightarrow node_k.networkView = networkState \end{aligned} \qquad (P_\alpha)$$

where $networkState$ is the list of all currently working nodes (i.e. without any failure), and $\alpha \in \mathbb{N}$ is a parameter depending on the system (in our system,

$\alpha = 7$). Thus, to prove the consensus is reached, the loop on line 11 should execute each node at least α times. The consensus preservation (for any $j \geq \alpha$) can be proved using k-induction (or by checking in some way that the system returns to the same symbolic state as in the beginning). Our algorithm goes through several partial consistency properties before reaching the consensus. Such a partial property P_j, $1 \leq j < \alpha$, is of the form:

$$\forall k \in \{1, \ldots, p\}, \, (\, node_k \in networkState \, \wedge Activation_k \geq j \,) \Rightarrow ... \qquad (P_j)$$

The right-hand side is a property of $node_k$'s state. It is not necessarily different for all j. For the M_{sim} model of our algorithm, only P_2, P_5 and of course P_7 are stronger than a previous one, so we will focus on them in our experiments and call them *key properties*. In our algorithm, properties P_j can also be expressed in terms of the message sent by $node_k$ at its j-th activation.

As suggested in Fig. 2, the number of different executions can grow very fast with the number of nodes and the execution length. A rough lower bound for the number of different execution orderings of M_{sim} can be computed as $(p!)^n$, where n is the number of activations of a node in the considered executions (so $n \geq \alpha$). The target Thales system is comprised of roughly 20 nodes, and the consensus is supposed to be reached after 7 activations of each node. Hence the number of execution paths of M_{sim} is greater than $(20!)^7 \approx 5 \cdot 10^{128}$.

As we will see in Sect. 4, model M_{sim} allows us to prove our algorithm for a small number of nodes. It is very useful to debug the implementation and iteratively check properties P_j or find counter-examples that are easily readable and understandable by the algorithm developers. To prove the same algorithm for a larger number of nodes, we need to abstract some behaviors.

3.2 Modeling one Node with an Abstraction of the System

We now explain how we construct an abstract model M_{abs} (cf. Algorithm 3). It is (manually) deduced from the original model M_{sim}. The idea is to model the system as one node $node_i$ (the node of interest) interacting with the rest of the network. The model activates only $node_i$, which receives messages from the other nodes. The behavior of the other nodes is defined only by assumptions.

M_{abs} also abstracts away the timing information contained in M_{sim} and also ensures it by suitable assumptions, called P_{timed}. To infer and verify these assumptions, we consider a simple auxiliary model M_T of M_{sim} which merely contains relevant timing information. We then use a parametric timed model checker to infer properties on possible execution interleavings of nodes.

In our methodology, we use M_{abs}, with the integration of P_{timed} as assumptions, to iteratively prove properties P_j, $1 \leq j \leq \alpha$, on the (state and) messages of $node_i$. To prove P_l, we assume in addition P_j, $1 \leq j < l$, for all other nodes. Each time a new property P_l is proven, we add it as an assumption on the messages sent to $node_i$ by other nodes. After several steps, when $l = \alpha$, the proven property P_l is the consensus property P_α (cf. lines 14, 18 in Algorithm 3).

We now describe models M_{abs} and M_T in more detail.

Algorithm 3: Pseudo code for abstract model M_{abs} (for a proof of P_l)

```
   // Initialization for nodes k=1,2,...,p
 1 Integer : node_k.id, Activation_k
 2 Boolean : node_k.failure, node_k.rcvFailure, node_k.sndFailure
 3 Assume : AllDifferent(node_k.id | k ∈ 1,...,p)  // Identifiers are unique
   // Initialization for node i
 4 Assume : i ∈ {1,...,p}
 5 Activation_i ← 0
 6 Boolean : node_i.EvenActivation
   // Main loop iteratively activating node i
 7 while true do
 8     Mailbox ← ∅  // Model possible messages from other nodes
 9     for k ∈ {1,...,p} \ {i} do
10         message_k ← nondet()
11         if ¬node_i.rcvFailure ∧ ¬node_k.sndFailure then
12             Mailbox ← Mailbox ∪ message_k
13         Activation_k ← Activation_k + |nondet()| // An increasing value
14     Assume : P_timed ∧ P_1 ∧ ··· ∧ P_{l−1} // Assume up to P_{l−1} for all nodes
15     if ¬node_i.failure then
16         UpdateNode(i)
17     Activation_i ← Activation_i + 1
18     Assert : (P_1 ∧ ··· ∧ P_l)|_{node_i}  // Prove properties up to P_l for node_i
```

Abstract Model M_{abs} and Proof of Properties P_j. This model considers only the activations of the node under study, $node_i$. The rest of the network is abstracted by the messages contained in the mailbox of $node_i$. Every other node $node_k$ ($k \neq i$) can have any state and send any message at any activation, provided it respects the assumptions (cf. line 14). Its behavior does not directly rely on its parity ($node_k.EvenActivation$), what the nodes sent previously, or what $node_i$ is sending. Thus, only some of the node fields are used in this model.

Algorithm 3 first initializes (non-deterministically) the useful fields of all nodes (lines 1–3), then those of the node under study $node_i$ (lines 4–6). The loop on lines 7–18 iteratively activates $node_i$ (lines 15–17) if it did not fail. Prior to that, it constructs a mailbox with any possible messages from the other nodes (lines 8–12) that can communicate with $node_i$ (cf. line 11). These messages have to respect the assumptions on line 14.

Notice that the number of activations $Activation_k$ of any other node $node_k$ is not tightly linked to that of $node_i$. The constraint for $Activation_k$ to be increasing follows from line 13, while having an acceptable difference with $Activation_i$ will follow from the assumption P_{timed}, as explained below. Thus, property P_{timed} will force some of the assumed properties P_j to constrain the messages received by $node_i$ from a working node $node_k$ in the case when $Activation_k \geq j$ follows from P_{timed}.

We iteratively prove the properties P_l for the messages sent by $node_i$, starting by $l = 1$. Once a new property is proved, we add it as an assumption on the messages sent by the other nodes. To prove property P_l $(1 \leq l \leq \alpha)$ for $node_i$, we assume that all other nodes respect the properties for smaller numbers of activations $P_j,\ 1 \leq j < l$, along with P_{timed} (cf. line 14). The assertion on line 18 requires to prove P_l for $node_i$ in addition to the previously proven $P_j, 1 \leq j < l$ (that are already true at this step if the proof is done iteratively). The notation $P_j|_{node_i}$ means that the property is reduced to $node_i$ (by removing the quantification and taking $k = i$ in the definition (P_j)).

In this way, if the assertion on line 18 is proved for $node_i$, we can deduce that P_l holds for any node and any execution, thanks to a non-deterministic choice of the node under study and symbolic states of all other nodes.

Interestingly, we observed that due to abstraction, the consensus property in the abstract model M_{abs} for our algorithm was not reached after $\alpha = 7$ activations as in the simulating model M_{sim}, but after $\alpha = 8$ activations. This extra delay was not an issue for system developers, a rigorous proof being more important. Therefore, we use $\alpha = 8$ in the specification and verification of M_{abs}. The other key properties P_2 and P_5 were still true for the same j.

Abstract Model M_T and Proof of Property P_{timed}. We give in this section only a very brief overview of the model M_T. More detail about it can be found in [5].

Our goal is to establish a property P_{timed} relating the numbers of executions of two nodes for the given system parameters. It has the following form:

$$\begin{array}{l} \forall i, k \in \{1, \ldots, p\},\ \mathit{Activation}_i \leq \beta \\ \quad \Rightarrow\ |\ \mathit{Activation}_i - \mathit{Activation}_k\ |\ \leq\ \gamma \end{array} \qquad (P_{timed})$$

for some $\beta, \gamma \in \mathbb{N}$. The value γ depends on the activation periods, the maximal jitter values, and of course the value of β. The value β must be sufficient in order to cover executions long enough to prove all properties P_j, $1 \leq j \leq \alpha$ needed to reach the consensus property. Example 1 illustrated some situations where the difference between such numbers of activations can (slightly) increase. In model M_{abs} of our target system, since we need at least $\alpha = 8$ activations of each node to reach a consensus, we can take $\beta = 8$ and have the bound $\gamma = 2$.

Given the current number of activations of $node_i$, this property allows us to deduce information on a possible number of activations of $node_k$. To prove it, we use a timed abstract model M_T of M_{sim}. It relies on an extension of the formalism of timed automata [1], a powerful extension of finite-state automata with clocks, i.e. real-valued variables that evolve at the same time. 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 [6]. 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* [2,3] allowing to consider *parameters*, i.e. unknown constants (possibly in an interval).

IMITATOR [4] is a state-of-the-art model checker supporting this formalism. The timed abstract model M_T of M_{sim} is a product of two similar parametric timed automata representing the node under study $node_i$ and a generic node $node_k$.

Thanks to a more abstract view of the system in the model M_{abs}, where other nodes and timing constraints were abstracted away and replaced by assumptions relating the numbers of activations of nodes, and thanks to an iterative proof of partial properties P_j, the proof for M_{abs} scaled up to larger numbers of nodes.

4 Experiments with Various Tools

To perform our experiments, we selected three tools: SAFEPROVER [10], CBMC [9], and KLEE [8]. Our experiments were not specifically aimed at comparing the tools or judging their potential. Their goal was rather to report the results that industrial engineers without an advanced knowledge of these tools can obtain when using them on a real-life distributed algorithm.

4.1 Experiments with SafeProver

We describe here the results obtained by a commercial SMT solver called SAFEPROVER [10] designed by the SafeRiver company. SAFEPROVER is designed to perform model-checking on Mathworks Simulink designs. It is a symbolic synchronous model checker, which fits with our modelings. However, it can be used with other input languages. In our case, we elected to use the Imperative Common Language (ICL) also designed by SafeRiver. ICL has some constraints that fitted with the use-case: all loops have to be statically bounded, all types and array sizes must be known statically. It forced us to use arrays for incoming messages, but this was coherent with the original implementation of the algorithm. SAFEPROVER uses several steps of proven model simplification, as outlined in [10], before sending the resulting model to a bit-blasting algorithm. The proof can then be performed by a choice of several SAT solvers.

We chose this language and tool as it is very close to a regular programming language for the algorithmic parts and offers the possibility to specify assumptions and assertions.

Models. We performed the proof of both models, M_{sim} and M_{abs}, with all failure modes. For these models, we ran SAFEPROVER on the proof of correction of P_7 (resp. P_8). We obtained that:

- The algorithm was correct when nodes could fail completely, or fail to send messages (F1, F2, F3, and F5);
- The algorithm was not correct if nodes could fail to receive messages, while still emitting new messages to the network (F4).

Results. For these models we ran SAFEPROVER on a full proof using k-induction. Contrary to the other tools tested for these experiments, it means that if a property is proven, it is true for all possible infinite executions. In order to establish this property, SAFEPROVER first performs a Bounded Model Checking (BMC) step with k steps and then tries to prove the k-induction step. In all

(a)

#nodes	$p = 3$	$p = 4$	$p = 5$
variant	Correct	Correct	Correct
time	59.5 s	95m48 s	TO
result	✓	✓	—

(b)

#nodes	$p = 3$	$p = 18$	$p = 42$	$p = 100$
variant	Correct	Correct	Correct	Correct
time	0.29 s	9.74 s	5min12 s	15min30
result	✓	✓	✓	✓

Fig. 3. Experiments with SafeProver for correct properties with all failure modes for models (a) M_{sim}, and (b) M_{abs}. TO means a timeout (set to 2 hours).

our experiments, the hardest and most time-consuming part of the proof was the k-induction step. The tool did not report the time for the BMC part of the proof, so we are unable to provide it for a clearer comparison between the tools. We believe that these proof times are still interesting because it shows that this approach is also viable for industrial proof. Times in Fig. 3 are given for the models with all possible failure modes. It includes the proof when the algorithm is correct (failure modes F1, F2, F3, and F5) and the generation of counter-examples when it was not (F4). SafeProver gave good results in terms of proof times and scaled up to $p = 100$ nodes for M_{abs}.

Next Steps. SafeProver offers the possibility to work on Mathworks Simulink sheets. Simulink[1] allows to model multidomain dynamical systems and automatically generate the code. It is widely used in Thales entities for algorithm design. We plan on working on a framework that would allow engineers to develop their algorithm in Simulink, and have an automatic generation of a provable model. Counter-examples would be given as a test case in Simulink so that the engineer can correct their design.

4.2 Experiments with CBMC

Cbmc is a bounded model-checker for C programs [9]. We chose to work on C code because the modelling language used for the initial model is very close to C so the translation was quite straightforward, which allowed to compare tools on similar models. We chose Cbmc since it is a well-known state-of-the-art model-checker for C programs. We thought it was well fitted for this experiment since the core of the problem is naturally bounded: the consensus is reached after a given number of executions, so the proof only requires a fixed number of iterations. For a complete proof, one also needs to prove that once a consensus is reached, it is preserved by the following executions. Bounded model-checking is not able to prove such a property stated on infinite executions, but it helped us gain confidence in it by proving it for very long executions.

Models. In a first set of experiments, we worked on simpler variants of the two models M_{sim} et M_{abs}, where we do not consider message delays nor possible failures. For these models, we ran Cbmc on the following properties:

- P_j^{err}, with $j = 1, 4, 6$ for M_{sim} and $j = 1, 4, 7$ for M_{abs}: an erroneous version of the key property P_{j+1} where the same property is stated after j executions instead of $j + 1$. This leads to the production of a counter-example.

[1] See https://fr.mathworks.com/products/simulink.html.

#nodes	$p=3$				$p=4$				$p=5$			
variant	P_1^{err}	P_4^{err}	P_6^{err}	Correct	P_1^{err}	P_4^{err}	P_6^{err}	Correct	P_1^{err}	P_4^{err}	P_6^{err}	Correct
time	0.25 s	0.95 s	6.27 s	37.75 s	0.33 s	1.52 s	53.98 s	14 m52 s	0.57 s	8.4 s	2 m27 s	TO
result	CE	CE	CE	✓	CE	CE	CE	✓	CE	CE	CE	—
RDP	0.13 s	0.74 s	5.76 s	37.49 s	0.20 s	1.25 s	53.18 s	14 m51 s	0.35 s	7.95 s	2 m26 s	TO
#vars	30,898	70,488	96,542	87,797	47,765	111,735	153,802	143,204	68,448	162,716	224,677	212,182
#clauses	110,966	256,340	351,902	319,867	172,625	408,119	562,800	523,886	261,475	627,154	867,407	819,048

Fig. 4. Experiments on erroneous and correct versions of model M_{sim} simulating all nodes without failures with CBMC. TO means a timeout (set to 2 hours). RDP stands for runtime decision procedure.

#nodes	$p=3$				$p=18$				$p=42$			
variant	P_1^{err}	P_4^{err}	P_7^{err}	Correct	P_1^{err}	P_4^{err}	P_7^{err}	Correct	P_1^{err}	P_4^{err}	P_7^{err}	Correct
time	0.32 s	0.33 s	0.41 s	0.34 s	2.19 s	2.35 s	2.42 s	2.85 s	8.07 s	8.65 s	9.81 s	11.05 s
result	CE	CE	CE	✓	CE	CE	CE	✓	CE	CE	CE	✓
RDP	0.13 s	0.14 s	0.17 s	0.14 s	0.88 s	0.97 s	1.08 s	1.18 s	2.20 s	2.79 s	3.81 s	4.28 s
#vars	38,615	38,606	38,597	38,594	197,795	197,786	197,777	197,774	452,483	452,474	452,465	452,462
#clauses	116,705	116,411	116,009	115,851	578,390	577,736	576,434	575,856	1,317,086	1,315,856	1,313,114	1,311,864

Fig. 5. Experiments on erroneous and correct versions of the abstract model M_{abs} without failures with CBMC. TO means a timeout (set to 2 hours).

– P_7 (resp. P_8): each node executed at least 7 (resp. 8) times knows the actual list of working nodes in M_{sim} (resp. M_{abs}). The proof should be successful.

In a second step we extended the two models to all possible failures (except those preventing a consensus to be reached). On these last models, we only ran CBMC on P_7 for M_{sim} (resp. P_8 for M_{abs}), to be able to compare the proof times to those of SAFEPROVER.

Results. On model M_{abs} without failures, all properties are falsified or proved in less than 25s for a size of network ranging from 3 to 64 nodes (see Fig. 5). On model M_{sim} without failures, the exponentially growing complexity of the model prevents the termination of proofs for the more complex properties, except for a very small number of nodes (see Fig. 4).

On models with failures, proofs are much more difficult. For model M_{sim}, only the model with 3 nodes is proven in less than 2 hours (with 4 nodes the proof takes almost 6 hours). And even for model M_{abs}, the property is proven in less than 2 hours only on models with less than 22 nodes. Results are shown in Fig.6. This seems to make SAFEPROVER much more efficient on these models.

One may notice that for all the experiments, the total proof time is mainly the time of the runtime decision procedure (RDP), except for model M_{abs} without failures, where the time needed to build the formula takes two thirds of the total proof time. The complexity of the BMC problem is expressed in terms of the number of clauses and variables of the formula sent to the SAT solver.

The main advantage of CBMC in an industrial setting is to work directly on C code. Moreover, the tool is quite easy to handle and the default parameters seem to be sufficient for the proof of user-defined properties. It requires very few adaptations of the initial code to be run so the time needed to make ones first proof is quite short. One may regret that, with the command line interface, default counter-examples are not very easy to read, and that it is not possible to directly replay them.

(a)

#nodes variant	$p = 3$ Correct	$p = 4$ Correct
time	4 min20 s	TO
result	✓	—
RDP	4 min19 s	TO
#vars	305,431	527,197
#clauses	986,574	1,713,053

(b)

#nodes variant	$p = 3$ Correct	$p = 18$ Correct	$p = 22$ Correct	$p = 23$ Correct
time	2.36 s	9 min2 s	55 min47 s	TO
result	✓	✓	✓	—
RDP	1.63 s	8 min46 s	55 min19 s	TO
#vars	371,265	2,007,225	2,436,945	2,543,945
#clauses	1,143,416	6,080,111	7,350,811	7,665,476

Fig. 6. Experiments with CBMC on correct versions of models (a) M_{sim} with failures, and (b) M_{abs} with failures. TO means a timeout (set to 2 hours).

#nodes	$p = 2$			$p = 3$			$p = 4$			$p = 5$		
variant	P_1^{err}	P_4^{err}	P_6^{err}	P_1^{err}	P_4^{err}	P_6^{err}	P_1^{err}	P_4^{err}	P_6^{err}	P_1^{err}	P_4^{err}	P_6^{err}
time	0.6s	7s	48s	0.8s	12m33s	TO	1.15s	TO	TO	2.1s	TO	TO
result	CE	CE	CE	CE	CE	—	CE	—	—	CE	—	—
#instr.	2,299	595,279	4,231,836	4,820	51,662,006	?	12,288	?	?	44,118	?	?

Fig. 7. Experiments on erroneous versions of model M_{sim} simulating all nodes with KLEE. For correct versions, the tool timed out. TO means a timeout (set to 2 hours).

Next steps. A BMC proof of the original (real-life) code could be tried, thanks to the knowledge we gained on the tool.

4.3 Experiments with KLEE

In the last set of experiments, we used KLEE [8], a popular dynamic symbolic execution tool. It explores program paths using a combination of concrete and symbolic execution, offers several strategies, and generates test cases for a given C program, possibly enriched with assume and assert statements.

Models. We used KLEE on the same models and properties as in the first set of experiments with CBMC (see Sect. 4.2).

Results. The results for M_{sim} and M_{abs} are shown, resp., in Fig. 7 and 8. They show that KLEE was able to generate counter-examples for both models, but for the most complex properties it was possible only for 2 nodes. The execution was stopped at a first assertion failure so we do not have data on a complete session, but we report the number of instructions KLEE explored before a counter-example was found. They show the combinatorial complexity of the models. On the correct versions and for some properties for more nodes, KLEE timed out (and also reported that the memory cap was exceeded, so many paths were dropped).

5 Lessons Learned and Perspectives

Our experience shows that despite the existence of several verified consensus algorithms, *industrial users often need to verify a specific algorithm (variant)*

#nodes	$p = 2$			$p = 3$			$p = 4$			$p = 5$		
variant	P_1^{err}	P_4^{err}	P_7^{err}	P_1^{err}	P_4^{err}	P_7^{err}	P_1^{err}	P_4^{err}	P_7^{err}	P_1^{err}	P_4^{err}	P_7^{err}
time	0.3s	27s	17m53s	0.5s	26m21s	TO	1s	60m53s	TO	1.7s	1h25m	TO
result	CE	CE	CE	CE	CE	—	CE	CE	—	CE	CE	—
#instr.	2,213	2,039,992	57,289,534	6,739	63,235,648	?	21,582	88,588,978	?	62,962	109,841,990	?

Fig. 8. Experiments on erroneous and correct versions of the abstract model M_{abs} with KLEE. For correct versions, the tool timed out. TO means a timeout (set to 2 hours).

that precisely fits their needs. The reasons can be grouped into two categories. System developers can be reluctant to changes and prefer to obtain a proof of the existing legacy algorithm, or adapted only with minor changes. Second, especially in embedded systems where performance is a key factor, the existing algorithms do not always meet the constraints of the target system, e.g., memory size, network usage, computational time, non-interference with other computations, relevant fault models and robustness constraints, the level of possible variations of the activation or communication times.

Generic verification methodologies applicable to large families of similar algorithms can be very helpful in this context. The existence of advanced verification tools and a large record of verification efforts in the area makes it possible today to suggest such methodologies for industrial engineers. After verifying one algorithm, the engineer may need to adapt it to a new system and to verify again. The present paper describes such a methodology for a family of consensus algorithms and reports on its application to some of them. The criteria for acceptance of the methodology include the capacity to perform the proof, the possibility to analyze the real-life code or a model as close as possible to the code, and to produce and easily read counter-examples.

As is often the case in distributed algorithms, the models of consensus algorithms we considered have *a high combinatorial complexity,* due to several free variables in the initial state and lots of possible interleavings. It is rapidly increasing with the number of nodes and executions. Therefore, we focused on symbolic tools: symbolic model checking and symbolic execution. Timed model checking did not seem to be a suitable candidate for two reasons. The timed model checkers we experimented with were enumerative. When experimenting with IMITATOR [4], we filled the 160GB RAM memory of the server before ending the proof. The second reason is the modeling language expected to be close to the code, and the need to easily generate and read counter-examples without having to invest too much time. Thus, Timed Petri nets or automata were not considered as suitable candidates to perform the whole study, but their application should be further investigated in the future. We restricted the usage of timed model checkers to one property, P_T, in order to perform the proof on the abstract model M_{abs}. The underlying model was simple enough to be reviewed.

Symbolic model checking tools we used (SAFEPROVER and CBMC) appeared to be very powerful both for finding counter-examples and proving the correct version of the algorithm. In particular, the support of bit operations was particularly useful to achieve better results thanks to a compact bit-level encoding of data in our models: the results became much better than for an earlier, naive array-based version.

We have also verified algorithms where message transmission can be delayed, using the SAFEPROVER tool and applying the same methodology. While this change did increase the time for the proof to be completed, it did not significantly change the number of nodes for which the proof worked.

Symbolic execution using KLEE also proved to be useful to detect counter-examples for small numbers of nodes and executions. Due to the combinatorial

explosion of the number of paths, in this case study we were not able to use it to explore all paths in order to show the absence of errors on the correct models. KLEE was convenient for producing readable counter-examples since it directly handles the C code.

Abstracting the system model using the rely-guarantee based technique was essential for the proof to scale for a large number of nodes. While the proof of the properties on the complete model M_{sim} was successful for smaller numbers of nodes ($p < 10$), it ran out of time and memory for bigger numbers of nodes which were required in the target systems. The rely-guarantee based approach, dating back to the work of Jones [13] and well-established today, solved this issue for the family of algorithms we faced in this work.

Future Work. Future work directions include the application of the methodology to other industrial algorithms, proof of the assumptions for the real-life C code using deductive verification (using e.g. FRAMA-C [14]) and experiences using other verification tools (model checking and symbolic execution). More generally, collecting the engineers' needs and experience related to verification of distributed algorithms at Thales and supporting them in their verification work remains a priority for the formal methods group of Thales Research and Technology.

Acknowledgment. The authors are grateful to Etienne André, Laurent Fribourg and Jean-Marc Mota for their contribution to the previous case study [5], as well as to the anonymous reviewers for their useful comments.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**, 183–235 (1994)
2. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: STOC. ACM (1993)
3. André, É.: What's decidable about parametric timed automata? Int. J. Softw. Tools Technol. Transf. **21**(2), 203–219 (2019)
4. André, É., Fribourg, L., Kühne, U., Soulat, R.: IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In: FM. LNCS, vol. 7436. Springer (2012)
5. André, É., Fribourg, L., Mota, J., Soulat, R.: Verification of an industrial asynchronous leader election algorithm using abstractions and parametric model checking. In: VMCAI. LNCS, vol. 11388. Springer (2019)
6. Behrmann, G., David, A., Larsen, K.G.: A tutorial on uppaal. In: SFM. LNCS, vol. 3185. Springer (2004)
7. Blanchard, A., Kosmatov, N., Lemerre, M., Loulergue, F.: Conc2Seq: A Frama-C plugin for verification of parallel compositions of C programs. In: SCAM. IEEE (2016)
8. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI (2008)
9. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. LNCS, vol. 2988. Springer (2004)
10. Étienne, J.F., Juppeaux, É.: Safeprover: A high-performance verification tool. ACM SIGAda Ada Letters **36**(2), 47–48 (2017)

11. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)
12. García-Molina, H.: Elections in a distributed computing system. IEEE Transactions on Computers **31**(1), 48–59 (1982)
13. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983)
14. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. **27**(3), 573–609 (2015)
15. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems **16**(2), 133–169 (1998)
16. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc. (2002)
17. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc. (1996)
18. Raynal, M.: Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach. Springer (2018)
19. Sun, Y., André, É., Lipari, G.: Verification of two real-time systems using parametric timed automata. In: WATERS (2015)