Formally Proving and Enhancing a Self-Stabilising Distributed Algorithm

Camille Coti\textsuperscript{1}, Charles Lakos\textsuperscript{2}, and Laure Petrucci\textsuperscript{1}

\textsuperscript{1} LIPN, CNRS UMR 7030, Université Paris 13, Sorbonne Paris Cité
99, avenue Jean-Baptiste Clément
F-93430 Villetaneuse, FRANCE
{Camille.Coti,Laure.Petrucci}@lipn.univ-paris13.fr
\textsuperscript{2} Computer Science, University of Adelaide
Adelaide, SA 5005, AUSTRALIA
Charles.Lakos@adelaide.edu.au

Abstract. This paper presents the benefits of formal modelling and verification techniques for self-stabilising distributed algorithms. An algorithm is studied, that takes a set of processes connected by a tree topology and converts it to a ring configuration. The Coloured Petri net model not only facilitates the proof that the algorithm is correct and self-stabilising but also easily shows that it enjoys new properties of termination and silentness. Further, the formal results show how the algorithm can be simplified without loss of generality.

1 Introduction

Goals and contributions. This paper aims at using a formal model and associated verification techniques in order to prove properties of a self-stabilising distributed algorithm. Although the algorithm considered [BCH+09] was shown self-stabilising, the proof was lengthy and cumbersome. Using formal models, in this case Petri nets which are particularly well-suited for such distributed algorithms, provides a much shorter and more elegant proof. It also allows for easily deriving additional properties which were not proven in the past: correctness, termination and silentness. Finally, reasoning on the model leads to a simplification of the algorithm, and its gains in terms of exchanged messages. This simplification, which now appears straightforward is not obvious at all when considering the distributed algorithm alone.

Context. A distributed system consists of a set of processes or processors. Each process \(k\) has a local state \(s_k\), and processes communicate with each other via communication channels that have local outgoing and incoming queues. The configuration, \(C\), of a distributed system is the set of local states of its processes together with the queues of messages that have yet to be sent or received.

Self-stabilising algorithms are distributed algorithms that, starting from any initial configuration and executing the set of possible transitions in any order, make the system converge to a legitimate configuration. This property makes
self-stabilising algorithms suitable for many fields of application, including fault tolerant systems: failures take the system out of its legitimate state and the algorithm is executed to reach a legitimate configuration again. Other properties that may be of interest are the closure property which ensures that once it has reached a legitimate configuration, the system remains in it, and the silent property which states that once it has reached a legitimate state, inter-process communications are fixed. This means that the communication channels hold the same value: processes stop communicating (message queues and communication channels remain empty) or repeatedly send the same message to their neighbours.

It is not always an easy task to prove the aforementioned properties related to self-stabilisation. It is necessary to prove that these properties hold for any possible execution and starting from any initial state. Some techniques exist for this, coming from traditional distributed algorithm techniques, or specific techniques such as attraction and rewriting [BBF99]. Some algorithms can also be formalised using a proof assistant [Cou02].

One technique to prove that a system is self-stabilising with respect to a set of legitimate states consists in considering a norm function which is representative of the state of the system, and proving that this function is integral, positive and strictly decreasing as the algorithm is executed [Gou95].

![Diagram](image-url)

Fig. 1: Execution of the algorithm
Outline of the paper  In this paper, we examine a distributed, self-stabilising algorithm that, given a set of processes interconnected by any tree topology, builds a ring topology between them. This algorithm is interesting since it is a stepping stone to a binomial graph (BMG) configuration which has several desirable properties, such as performance (low diameter and degree) and robustness (in terms of number of processes and communication links that can fail without disconnecting the graph) [ABD07].

Section 2 presents the algorithm and describes its behaviour. In Section 3 we derive a Coloured Petri Net model of the algorithm. The Coloured Petri Net formalism is ideally suited for this task because of its support for concurrency and the variability of sequencing and timing of distributed processes. Further, the graphical representation of Petri Nets can help to highlight the flow of information in the algorithm. This immediately leads to some simplifications which are presented in Section 4, along with certain invariant and liveness properties which can be deduced directly from the model. In Section 5 we prove that the algorithm terminates, effectively presenting a norm function. Then, in Section 6 we prove that the algorithm is correct. As a result of the properties proved above, it is possible to simplify the algorithm further, and this is presented in Section 7. In section 8 we validate our results with an automated tool. Finally, the conclusions are presented in Section 9.

2 The algorithm

In this paper, we focus on a distributed, self-stabilising algorithm that, given a set of processes interconnected by any tree topology, builds a ring topology between them. This algorithm was originally described in [BCH+09] and we reproduce the pseudocode in algorithm 1.

This algorithm is meant to establish a fault-tolerant, scalable communication infrastructure. For instance, it can be used to support the execution of parallel processes [BGL00] such as the processes of MPI programs [For94]. Robust, fault-tolerant run-time environments [FD02] are necessary to support middleware-level, automatic fault tolerance [BBC+02] or application-level fault-tolerance [GL04,BDDL09,PLP98]. In order to be efficient at large scale, tree topologies are often preferred [APM06]. However, trees are not robust and connectivity can be lost if intermediate processes fail. Resilient extensions of trees introduce additional inter-process connections, such as k-sibling trees [AFB+06] or redundant storage of connection information [AM]. The Binomial Graph topology (BMG) is scalable and allows efficient communications [Mas77,BHK+97] while being robust. More precisely, a BMG made of $N$ processes has a diameter in $O(\log N)$, therefore a message is routed between two processes in at most $O(\log N)$ hops. It has a degree in $O(\log N)$, which means that every process needs to handle $O(\log N)$ open connections. Every node sees the other nodes along a binomial tree: therefore, efficient (in terms of number of messages) collective communications can be implemented directly on top of a BGM. Its node connectivity and its link connectivity are both in $O(\log N)$, which means that $O(\log N)$ nodes or
Algorithm 1: self-stabilising algorithm that builds a ring from any tree

Constants:
- Parent : ID /* empty if I am the root of the tree */
- Children : List(ID) /* empty if I am a leaf process */
- Id : ID /* my own identifier */

Output:
- Pred : ID /* both empty start-up time */
- Succ : ID

Initialisation:
1 - Children ≠ ∅ → /* I have children: send a F_Connect message to my leftmost child. */
   Send (F_Connect, Id) to Succ ;
2 - Children = ∅ → /* I am a leaf process */
   Send (Info, Id) to Parent ;

Run:
3 - Recv (F_Connect, I) from p → /* I have received an FC message. From whom? */
   if p ∈ Children then
     let q = next(p, Children) ;
     if q ≠ ⊥ then
       Send (Ask_Connect, I) to q ;
     else if Parent ≠ ⊥ then
       Send (Info, I) to Parent ;
       Pred = I ;
       Send (B_Connect, Id) to I ;
       end if
     end if
   end if
4 - Recv (Info, I) from p → /* I have received an Info message. From whom? */
   if p ∈ Children then
     let q = next(p, Children) ;
     if q ≠ ⊥ then
       Send (Ask_Connect, I) to q ;
     else if Parent ≠ ⊥ then
       Send (Info, I) to Parent ;
       Pred = I ;
       Send (B_Connect, Id) to I ;
     end if
   end if
5 - Recv (Ask_Connect, I) from p → /* I am being asked to connect to a leaf process. */
   Pred = I ;
   Send (B_Connect, Id) to I ;
6 - Recv (B_Connect, I) from p → /* I received a B_Connect; here is my successor. */
   Succ = I ;

$O(\log N)$ network connections can fail without it topology to be disconnected [ABD07].

Processes are spawned on a set of resources and each process is connected to the process that spawned it. This results in a tree topology that matches the shape of the spawning operation (figure 1(a)).

To connect with another process, a process needs to know the remote process’s communication information (e.g. an IP address and a port). This communication information and the name of a process in a naming system (e.g. a rank) form the identity of this process. In the initial state, processes know their own identity, that of their parent and of their children. The algorithm allows for concurrent or even parallel propagation of the identity of leaf processes in order to establish connections between some leaves and some intermediate processes. The algorithm sets pointers to reflect the node order of depth-first traversal. It achieves this by propagating information bottom-up from the leaves, thereby allowing for parallel execution. Besides, inter-process communications are asynchronous.

In the algorithm, ID denotes the set of possible process identities. List(ID) is an ordered list of identities; First(L) returns the first element in the list L, and next(e, L) returns the element following e in L. If there is no such element, these functions return ⊥, which is also used to denote a non-existing information.

During the first step, each non-leaf process sends an F_Connect (or FC for short) message to its oldest (i.e. leftmost) child (rule 1). Thus, each process that receives an FC message is connected to its predecessor (rule 3). Concurrently,
each leaf process sends its identity to its parent in an Info message (rule 2 and figure 1(b)).

Then each time a process receives an Info message from one of its children (rule 4), it forwards this information to the younger sibling as an Ask_Connect message (or AC, see figure 1(c)). The message forwarded here contains the identity of the rightmost leaf process of its older sibling. Therefore, the child that receives the AC message will then be able to send a B_Connect (or BC) message (rule 5) to the rightmost leaf process of its older (i.e. left) sibling. As a consequence, each process that receives a BC is connected to its predecessor (rule 6). The root process receives the identity of the rightmost leaf of the tree from its youngest (i.e. rightmost) child and establishes a BC connection with it.

Eventually, all the leaves are connected to another process (figure 1(d), where the colours relate to the different transitions sequences of Section 6), and the set of FC and BC connections forms the ring.

3 A Coloured Petri Net Model

Algorithm 1 was proved to be self-stabilising in [BCH+09], but the proof was lengthy and cumbersome. Moreover, the algorithm enjoys additional properties that were not established before, such as being silent.

Therefore, we build a formal model of the algorithm which provides the following features:

– a graphical representation for an easier and better understanding of the system’s behaviour;
– a global view of the system states at a glance, allowing to focus on the flow of messages;
– facilities for property analysis and formal reasoning.

A coloured Petri net (CPN) model [Jen92,Jen94] is thus designed, capturing the flow of information and the individual actions in its graph, and the actual essential values in the data it manipulates. For the sake of readability, it is presented in 3 parts (Figures 2(a), 2(c) and 2(b)) corresponding to the initialisation phase, the core of the algorithm, and the termination phase, respectively. Should it be presented in a single figure, the places with the same name would be fused together. Also note that the arcs connected to place Messages (and the associated arc inscriptions) are coloured according to the messages they handle. This has no formal meaning but enhances readability, which is particularly useful when illustrating message flow in the proofs.

3.1 Data Types Declaration

The type declarations in Figure 3(a) show all data types that are to be used in the CPN. First, the processes are identified as members of a set Proc of processes (of the form \{P1,\ldots\}). This set also includes a particular fake process that is used to denote that the parent or child of a process does not exist (when it is respectively the root or a leaf). This corresponds to \bot in the algorithm. Type
2Proc is a pair of process names. Then MessType describes all four types of messages: FC, AC, BC, and Info. A message also contains a process identifier, its sender and its receiver.

The algorithm we model makes use of the tree topology with parent and child relation plus the next child in a parent’s list. To model this, we use triples consisting of the parent, the child, and the number of the child in the list of children. The fake child is always the last one in the list, thus denoting its end.

For example, the tree in Figure 1(a) is modelled by the set of triples:

\{(fake, P0, 1), (fake, fake, 2), (P0, P1, 1), (P0, P2, 2), (P0, fake, 3), (P1, P3, 1), (P1, P4, 2), (P1, fake, 3), (P2, P5, 1), (P2, fake, 2), (P3, P6, 1), (P3, P7, 2), (P3, P8, 3), (P3, fake, 4), (P4, P9, 1), (P4, fake, 2), (P5, fake, 1), (P6, fake, 1), (P7, fake, 1), (P8, fake, 1), (P9, fake, 1)\}.

Proc = set of processes ∪ \{fake\};
2Proc = Proc x Proc;
MessType = \{FC, AC, BC, Info\};
Mess = MessType x Proc x Proc x Proc;
TreeStructure = Proc x Proc x Int;

(a) Type declarations

Tree = initial topology

c, f, I, p, q, r: Proc;
m, n: Int;

(c) Variables and constants declarations

Fig. 2: Model corresponding to Algorithm 1.

Fig. 3: CPN declarations
3.2 Initialisation Phase

The initial phase of the algorithm is modelled by the CPN of Figure 2(a). It describes all the necessary places with their initial marking (in italics) of the type described in the places declarations in Figure 3(b). Figure 3(c) shows the types of variables used in arc expressions as well as the initial Tree topology constant.

At the start, there is no message, all processes (except the fake one) are ready to start. No process knows a possible successor or predecessor in the ring, hence is associated with the fake process in the Pred and Succ places. The tree topology to be processed is described by the constant Tree.

Transition T1 models rule 1 in the algorithm. A process p with c as first child is processed, sending an FC message with its identity to this child, and updating its successor information with this child’s identity.

Every leaf process p executes rule 2 of the algorithm, as depicted by transition T2. It is a leaf if described by (p, fake, 1) in the tree topology, and it is the child number n of some (non-fake) parent process f (f, p, n) in the tree. It then sends an Info message with its identity to its parent.

Fig. 4: Simplified model
3.3 Main Phase

The CPN in Figure 2(c) describes the core part of the algorithm, i.e. the processing of FC and Info messages. Transition T3 handles an FC message, as rule 3 of the algorithm by updating the predecessor information of the receiver of the message. Rule 4 is decomposed into 3 transitions corresponding to the different possible receiver configurations:

T4a: relative to receiver r, the sending child has a next sibling q to whom the received information is forwarded as an AC message;

T4b: relative to receiver r, the sending child is the last in the list (i.e. the youngest sibling) and the receiving node is not the root (i.e. it has a parent q which is not the fake one) to which it forwards the Info message;

T4c: relative to receiver r, the sending child is the last in the list (i.e. the youngest sibling) and the receiver is the root (i.e. it has no parent—its parent is the fake one). It updates its predecessor with the information I received and sends a BC message with its own identity to process I.

3.4 Termination Phase

Finally, the termination phase, shown in Figure 2(b) handles the AC and BC messages, using transitions T5 and T6 respectively. In case of an AC message, the predecessor information of receiver r is updated with the content I of the message. It also sends a BC message to this process I with its identifier r. When a BC message is handled, only the successor information of the receiver r is updated with the identity I carried by the message.

4 A Simplified Coloured Petri Net Model

In this section, we first simplify the CPN model, which then makes it easier to exhibit its invariant properties.

4.1 The Simplified Model

The simplified CPN is given in Figures 4(a), 4(c), 4(b). First, both places Pred and Succ initially contained bogus values associated with each process (i.e. (p,fake) for a process p), which was then discarded to be replaced by the actual values. In the new net, these two places are initially empty, and the only operation now consists in putting in these places the predecessor and successor values produced.

Second, an FC message is only produced by transition T1, and thus has the form (FC,p,p,c). Thus the information I carried by the message is the identity of the sender process p, I=p. Hence, transition T3 (the only one for the reception of FC messages) is modified by using p only.
4.2 Invariants of the simplified CPN

The Petri net model allows us to identify various properties of the system. Of interest here are the place invariants [Jen92, Jen94]. These identify weighted sums of tokens which remain invariant under the firing of all transitions. We use projection functions such as $\pi_2$ to select the second element of a token which has a tuple value, and $\pi_{2,4}$ to select the second and fourth elements, to form a pair. We also use a function notation to select elements of a particular type, thus $Messages(FC)$ is the set of FC messages in place $Messages$.

It is possible that some of these invariant properties could be extracted automatically from the model by a suitable tool, while others could at least be checked automatically. These properties may then be of assistance in proving more involved results for our system.

When verifying properties of a modelled system, it is important to validate the model, i.e. show that it is an accurate model. In this regard, we note that the CPN model does not introduce any new information, but it does make explicit certain values (like the sender and receiver of a message) which then make it easier to prove the invariant properties. Another important issue for validating a distributed algorithm is that the model does not have one process inadvertently accessing information that is local to another process. In the case of our model, we need to ensure that each firing of a transition is relevant to only one process and does not access information local to another process. We note the following properties of each transition:

- $T_1$ fires for process $p$ and accesses its first child and generates its $Succ$ entry.
- $T_2$ fires for process $p$ and accesses its parent.
- $T_3$ fires for process $r$ and generates its $Pred$ entry.
- $T_{4a}$ fires for process $r$ and accesses its children.
- $T_{4b}$ fires for process $r$ and accesses its parent and its children.
- $T_{4c}$ fires for process $r$ and checks that it is the root and generates its $Pred$ entry based on the received message.
- $T_5$ fires for process $r$ and generates its $Pred$ entry based on the received message.
- $T_6$ fires for process $r$ and generates its $Succ$ entry based on the received message.

Having convinced ourselves that the model accurately captures a distributed system, we now consider the properties of the model.

**Property 1** $\text{InitP} + \pi_1(Succ) + \pi_2(Messages(Info)) + \pi_3(Messages(AC)) + \pi_4(Messages(BC)) = \text{Proc} \setminus \{\text{fake}\}$

*Proof.* Initially, we have no messages and $\text{InitP} = \text{Proc} \setminus \{\text{fake}\}$. Then, we can consider each transition in turn:

- **$T_1$** The initialisation of a parent removes an item from $\text{InitP}$ and adds a $Succ$ entry with the same identity.
- **$T_2$** The initialisation of a leaf removes an item from $\text{InitP}$ and adds an $Info$ message with the relevant identity.
- **$T_{4a}$** This consumes an $Info$ message and generates an $AC$ message with the same identity.
- **$T_{4b}$** This consumes one $Info$ message and generates another with the same identity.
T4c This consumes an Info message and generates a matching BC message.
T5 This consumes an AC message and generates a matching BC message for the
destination given by the identity.
T6 This consumes a BC message and adds a Succ entry for the receiver.

Property 2 \( \text{succ} + \pi_{4,2}(\text{Messages}(\text{BC})) = \pi_{3,4}(\text{Messages}(\text{FC})) + \pi_{2,1}(\text{Pred}) \)

Proof. Initially, there are no messages and places Succ and Pred are empty. Subsequently, we consider the relevant transitions in turn:
T1 The setting of Succ is paired with the generation of an FC message.
T3 The consumption of an FC message is paired with the addition of a Pred entry.
T4c The setting of Pred is paired with the generation of a BC message.
T5 The setting of Pred is paired with the generation of a BC message.
T6 The consumption of a BC message is paired with the addition of a Succ entry.

4.3 Liveness of the simplified CPN

We now summarise some liveness properties of the CPN.

Property 3 For any tree with at least two nodes, either transition T1 or transition T2 can fire for every node. Thus, InitP will eventually be empty.

Proof. A tree with at least two nodes will have a root and at least one leaf. Thus:
1. T1 can fire for every node which is not a leaf, i.e. for every node which has a non-fake child.
2. T2 can fire for every leaf, i.e. for every node which has a non-fake parent.

Property 4 All messages can eventually be uniquely consumed.

Proof. We consider the different kinds of messages in turn:
FC The only constraint on the consumption of FC messages by transition T3 is that the identity and the source of the message are the same. This is guaranteed by the generation of FC messages in transition T1.
Info Every Info message can be consumed by one of the transitions T4a, T4b or T4c. Transition T4a can consume Info messages from node p for parent r provided p has a younger sibling. Transition T4b can consume Info messages from node p for parent r provided p has no younger sibling and r has a (non-fake) parent. Transition T4c can consume Info messages from node p for parent r provided p has no younger sibling and r is the root.
AC There is no constraint on the consumption of AC messages by transition T5.
BC There is no constraint on the consumption of BC messages by transition T6.

Note that in each case, exactly one transition can consume each kind of message.

Property 4 guarantees that the algorithm is silent. Once a legitimate configuration has been reached (i.e. once the ring has been established), there is no pending message held in the communication channels of the system. A silent
algorithm is an algorithm in which, upon a certain point of its execution, the contents of the communication channels remain the same [DGS99]. In our case, no message is sent between processes once the system has stabilised, as mentioned in the introduction of this paper.

5 Algorithm Termination

Definition 1. We define the weight of the state as follows:

- for each node prior to sending its first message: weight(node) = 3 + depth(node)
- for each node after sending its first message: weight(node) = 0
- for each FC message: weight(FC) = 1
- for each BC message: weight(BC) = 1
- for each AC message: weight(AC) = 2
- for each Info message: weight(Info) = 3 + depth(target)

Then the total weight of the state is given by: Weight = \sum_{x \in \text{node} \cup \text{msg}} weight(x).

Note that the weight of a state is always positive if there are any nodes yet to send their first message or any messages to deliver, or else zero when there are none. As a consequence, the weight function has separate points and positivity properties. Absolute homogeneity and triangle inequality properties are not relevant in our context. Therefore, the weight function is a norm on the states (as introduced in Section 1).

Proposition 1. Given the state of the algorithm, the weight of the state decreases at every step.

Proof. We consider each possible rule in turn:

rule 1: The weight of the node is set to zero and the number of FC messages is increased by 1. Hence Weight is decreased by 2 + depth(node).

rule 2: The weight of the node is set to zero and an Info message is generated for the parent. Hence Weight is decreased by 1.

rule 3: The number of FC messages is decreased by 1, and hence Weight decreases by 1.

rule 6: The number of BC messages is decreased by 1, and hence Weight decreases by 1.

rule 5: The number of AC messages is decreased by 1 and the number of BC messages is increased by 1. Hence Weight decreases by 1.

rule 4a: The Info message is received by a parent with other children, and an AC message is sent to the next sibling. Consequently, Weight is decreased by at least 3, and increased by 2, i.e. it is decreased by at least 1.

rule 4b: The Info message is passed to the parent, and hence the depth of the target is decreased by 1. Hence Weight is decreased by 1.

rule 4c: The Info message received by the root node (with depth 0) is replaced by a BC message. Hence Weight is decreased by at least 3 and increased by 1, i.e. it is decreased by 2.

\qed
Property 5 The algorithm terminates and is self-stabilising.

Proof. Following initialisation, every execution step of the algorithm involves at least one of the above rules. Thus, Weight is strictly monotonic, i.e. it is decreased by at least one while remaining positive. Consequently, the algorithm terminates. Moreover, as stated in section 1, if the norm function Weight is strictly monotonic, then the algorithm is self-stabilising (proof by norm). □

6 Algorithm Correctness

![Successor-based CPN model]

(a) Initialisation phase

(b) Termination phase

(c) Main phase

Fig. 5: Successor-based CPN model

Proposition 2. The algorithm establishes Succ and Pred as mirror images, i.e. Succ = \( \pi_{1,2}(Pred) \).

Proof. This follows directly from properties 2 and 4. □

Proposition 3. The algorithm establishes predecessors of nodes as:

- predecessor(node) = parent of node (case 1: node is the oldest child)
- predecessor(node) = preceding-leaf of node (case 2: node is not the oldest child and not the root)
- predecessor(node) = last-leaf (case 3: node is the root)

Proof. We consider each possible case in turn. They correspond to the coloured arcs in Figure 1(d) (red, green and blue arcs respectively).

case 1—firing sequence T1T3: Every non-leaf node generates an FC message to its oldest child (T1), which sets the parent to be its predecessor (T3), as required.
case 2—firing sequence T2T4b*T4aT5[T6]: Every leaf generates an Info message (T2) which is passed up the tree (T4b) till it finds a sibling. That sibling is sent an AC message (T4a) with the identity of the leaf (from the Info message). The AC message sets the predecessor of the sibling to be the originating leaf (T5), which is the preceding leaf in the tree.

case 3—firing sequence T2T4b*T4c[T6]: Every leaf generates an Info message (T2) which is passed up the tree (T4b) till it reaches the root, which will be the case for the last leaf. In this case, the predecessor of the root is set to the last leaf (T4c).

Thus for each possible firing sequence, the Pred values are set as required, and Proposition 2 tells us that the Succ values are also set as required. ⊓ ⊔

Proposition 4. The algorithm sets the Succ and Pred values so there is one connected component.

Proof. For the purposes of the proof, we consider that a node is connected to the tree until its Succ and Pred values are set. In this way, the algorithm starts with a tree, which is a single connected component. Then we need to show that every time the Succ and Pred values are changed, then the node is still connected to the one component. Thus, we have:

– Every oldest child is connected to the parent (by the Succ and Pred values), which reflects the tree structure, and therefore does not modify the connectedness.
– Every younger sibling is connected to the preceding leaf. Since a leaf has no child, connecting it to another node does not jeopardise the connectivity of the structure. In other words, provided the leaf is connected to the rest of the component, then so is the younger sibling.
– The above items result in a connected structure with the last leaf without a successor. This last leaf is connected to the root.

Thus the algorithm sets the Succ and Pred values to be a single connected component. ⊓ ⊔

Property 6 The algorithm produces a ring topology.

Proof. It suffices to have a single connected component where all nodes have only one predecessor and one successor. The connectedness is stated by Proposition 4. In the terminal state, there is no message left and InitP is empty. Thus, from Property 1, we deduce $\pi_1(Succ) = Proc \setminus \{fake\}$. Similar to the cases in the proof of Proposition 4, the different possibilities entail that $\pi_1(Pred) = Proc \setminus \{fake\}$. ⊓ ⊔

7 Slight Simplifications of the Algorithm

From the properties proved in the previous sections, we can again simplify the model and reflect these simplifications in the algorithm.
7.1 New Models

The topology obtained by the predecessor and successor information is a ring, and these are just mirror images of one another. It is thus not necessary to keep them both. So, let us remove place \texttt{Pred} from the model of Figures 4(c) and 4(b).

In the resulting net, transition \texttt{T3} only discards FC messages. Since no other transition handles these messages, they are also unnecessary. Therefore, we remove the arc producing FC messages from the net in Figure 4(a).

The resulting net is depicted in Figures 5(a), 5(c) and 5(b). The figures are meant to show the modifications on the structure of the net, and the inscriptions are not altered.

Additional simplifications are not possible: even though we could be tempted to get rid of AC messages that are immediately transformed into BC messages by transition \texttt{T5}. This modification would not be sound. Indeed, the successor information for a process \texttt{p} must be updated by process \texttt{p} itself. This is obviously the case for transition \texttt{T1}. The same holds for transition \texttt{T6} where process \texttt{r} receives a BC message and updates its successor information. However, if transition \texttt{T4a} were to send immediately a BC message, it should be \((BC, q, q, I)\) (to be the same as the one generated by \texttt{T4aT5}). But then transition \texttt{T4a} would handle reception of an \texttt{Info} message by process \texttt{r}, as well as the sending of a message by its child \texttt{q}, hence not the same process and thus not consistent.

We could equally well decide to remove place \texttt{Succ} and keep place \texttt{Pred}. In this case, FC messages remain while BC messages are no more necessary. Hence transition \texttt{T6} is also deleted. The resulting net is depicted in Figures 6(a), 6(c) and 6(b).

7.2 New Algorithms

Algorithm 2 shows the corresponding simplified algorithm, where variable \texttt{Pred} has been removed, as well as FC messages. Note that rule 3 is not necessary anymore, and that some parts of the algorithm are more balanced.

Indeed, the initialisation part is such that leaf processes send \texttt{Info} messages while the others only update their successor value (rules 1 and 2), while in the end (rules 5 and 6) leaf processes update their successor value and the others send a message to a leaf. The main part (rule 4) also features a balanced treatment of \texttt{Info} messages: all types of process send a single message only.

Algorithm 7 is similar, for the case where variable \texttt{Succ} has been removed, as well as BC messages.

7.3 The algorithms comparison

Both new algorithms send less messages than the original since, in each case, there is one type of message which is no longer used. Let \(m_k\) be the number of messages sent by algorithm \(k\), \(n_l\) the number of leaf nodes and \(n\) the total number of nodes in the tree. We have: \(m_2 = m_1 - 2(n - n_l)\) and \(m_3 = m_1 - 2n_l\).
Algorithm 2: Successor-based algorithm

**Constants:**
- **Parent**: ID
- **Children**: List(ID)
- **Id**: ID

**Output:**
- **Succ**: ID

**Initialisation:**
1. If **Children** ≠ ∅
   - **Succ** = First(Children)
2. If **Children** = ∅
   - Send (Info, Id) to Parent

**Run:**
4. Receive (Info, I) from p →
   - If **p** ∈ **Children**
     - Let **q** = next(p, Children)
     - If **q** ≠ ⊥ then
       - Send (Ask_Connect, I) to **q**
       - Else
         - If **Parent** ≠ ⊥
           - Send (Info, I) to **Parent**
         - Else
           - Send (B_Connect, Id) to **I**
         - End if
     - End if
   - Else
     - Send (B_Connect, Id) to **I**
5. Receive (Ask_Connect, I) from **p** →
6. Receive (B_Connect, Id) from **p** →

Therefore, the algorithm to apply depends on the structure of the tree: if there are more leaf nodes than other nodes, Algorithm 7 is preferred; otherwise, Algorithm 2.
Fig. 7: Predecessor-based algorithm

Constants:
- Parent : ID
- Children : List(ID)
- Id : ID

Output:
- Pred : ID

Initialisation:
1. Children ≠ ∅ →
   - Send (F.getExternalConnection, Id) to First(Children)

2. Children = ∅ →
   - Send (Info, Id) to Parent

Run:
3. Recv (F.getExternalConnection, I) from p →
   - if p = Parent then Pred = I

4. Recv (Info, I) from p →
   - if p ∈ Children then
     - let q = next(p, Children)
     - if q ≠ ⊥ then
       - Send (Ask.getExternalConnection, I) to q
     - else
       - if Parent ≠ ⊥ then
         - Send (Info, I) to Parent
       - else
         - Pred = I
  - end if
  - end if

5. Recv (Ask.getExternalConnection, I) from p →
   - Pred = I

8 Experimental confirmation of the Algorithm

While the formal results presented above in the paper can stand on their own, we confirmed the results experimentally. We built a graphical model of the Petri net of Figures 4(a), 4(b) and 4(c) in the CosyVerif tool [EA13] and proceeded to analyse the state space using the prod reachability analyser as a backend tool [KV05].

The example of Figure 1(a) was entered as the initial marking and prod reported that the state space consisted of 1,275,750 nodes, 9,470,925 arcs with one terminal node. The terminal node was manually examined to confirm that it represented the appropriate ring structure.

Further validation is considered in the following subsections.

8.1 Exploring different topologies with a pre-initialisation phase

As described in Section 4, the algorithm was modelled as a Petri net with an Initialisation, Main and Termination phases. Rather than just consider one topology, we introduced a pre-initialisation phase to generate an arbitrary topology from a given number of nodes. The Petri net segment for this is given in Figure 8.
The pre-initialisation phase was designed so that each topology would be generated with a unique labelling of nodes. Further, the labels would be such that a depth-first traversal of the tree would result in a ring with node labels in increasing numeric order. To achieve this, the last node added to the tree is stored in place parents together with its parents and their associated depths in the tree. The depth of the last node added is given in place depth. In order to add another node — with the numeric label given in place nextNode — there are two possible options: either the new node is added as a child of the last node added (with the transition addNode) or the last node can be discarded (with transition dropParent) and its immediate parent becomes a possible candidate for the parent of the new node. Transition dropParent can drop all but the first node, in which case the next node to be added will be a child of the root. Transition addNode can fire for all nodes up to a given node number. When that node number is reached, transition nodesDone can fire and add a token to place configOK. This place then becomes a side condition of the initialisation transitions $T1$ and $T2$ in Figure 4(a).

With the above pre-initialisation phase, we obtain the state space results of Table 1. The first column indicates the number of nodes in the tree to be processed. The column labelled Topologies indicates the number of tree topologies which can be generated with that number of nodes (as given by the pre-initialisation phase above). The column labelled Pre gives the time taken to execute the pre-initialisation phase for that number of nodes. The next three columns give the total number of nodes and arcs in the state space for all those topologies (combined) as well as the time to process those topologies in seconds.
Table 1: State space results for self stabilisation algorithm.

| Nodes | Topologies | Initial state space | Reduced state space |
|-------|------------|----------------------|---------------------|
|       |            | Pre Nodes | Arcs | Sec | Pre Nodes | Arcs | Sec |
| 2     | 1          | 0.001     | 11   | 17  | 0.001     | 5     | 5    | 0.263 |
| 3     | 2          | 0.001     | 103  | 229 | 0.004     | 0.014 | 17   | 17   | 1.078 |
| 4     | 5          | 0.001     | 1,123 | 3,314 | 0.057  | 0.038 | 60   | 60   | 6.210 |
| 5     | 14         | 0.002     | 13,474 | 49,807 | 0.983  | 0.176 | 217  | 217  | 48.607 |
| 6     | 42         | 0.006     | 172,248 | 766,076 | 17.835 | 0.976 | 798  | 798  | 381.137 |
| 7     | 132        | 0.010     | 2,301,624 | 11,968,306 | 328.869 | 6.220 | >1800 |
| 8     | 429        | 0.063     | >10M  | >50M | >1800   | 39.042 |     |      |
| 9     | 1430       | 0.230     |       |     |         | 232.136 |     |      |
| 10    | 4862       | 0.902     |       |     |         | 1802.274 |     |      |

With the pre-initialisation phase, each run of the Petri net will consider a number of topologies and that will be the number of terminal states. In order to ensure that the only difference between the terminal states is due to the different starting topologies, we can add a post-termination phase to remove the topology from the state, i.e. empty the tokens out of place Topology. It is a simple change to the net, and its addition does confirm that despite processing a number of topologies, there is only one terminal node with the correct ring structure.

8.2 State space reduction

In the algorithm we note that there are a number of messages exchanged between a node and what will eventually be its successor and predecessor as well as the intermediate nodes. We hypothesise that this set of messages is independent of the sets of messages exchanged between other pairs of nodes. In other words, the complexity of the algorithm is largely due to the arbitrary interleaving of the message transmission.

Accordingly, we considered reducing the state space by the stubborn set technique [Val88] which eliminates much of the interleaving while maintaining the terminal states. This form of reduction can be activated by running prod with option -s.

It turns out that there are a number of complexities in applying this technique. Firstly, the prod tool applies the technique to the unfolded net — it unfolds the Coloured Petri Net into a Place Transition Net. The size of the unfolded net is determined not just by the transitions which can fire in the coloured state space but by the possible range of values for the tokens.

Consequently, our initial result was that none of our test cases — even the pre-initialisation phase — reached their terminal states in less than 30 minutes! (We had allowed for up to 12 nodes and up to 14 children per node.) Therefore, the nets were modified to reduce the ranges of values for node labels and for child indices to be slightly larger than required for the example under consideration. With these modifications, we were able to reach terminal states for some of our test cases.
The second complexity of the stubborn set method is the computational penalty for computing the stubborn sets. This means that this reduction technique may or may not be effective in reducing the state space.

In Table 1, the last four columns give similar results to the preceding four columns, but this time using the stubborn set technique to reduce the state space. The results clearly show that the size of the state space can be reduced but that the computational penalty can be overwhelming. Even the pre-initialisation phase can be very expensive.

9 Conclusions

This paper has demonstrated the benefits of using formal techniques in the analysis of a non-trivial distributed algorithm for converting a tree structure of processes (or processors) into a ring structure.

In such an exercise, the choice of formalism is significant. The Petri Net formalism has proved to be ideal because of its support for concurrency and the variability of sequencing and timing of concurrent processes.

We built a model of the distributed algorithm and then validated it, i.e. ensured that it accurately reflected the modelled system. In our case, it was important to ensure that the model faithfully reflected the distributed nature of the algorithm. Thus, we examined each transition to ensure that it only accessed information local to a given process.

Having modelled and validated the system, we observed that without adding any new information, the making explicit of the source and target of each message facilitated the identification of some invariant and liveness properties. These were then utilised to prove termination, correctness of the algorithm, and that it was self-stabilising and silent. These properties could easily be exhibited on the model, but they are far from obvious when considering the algorithm itself.

Further, the above properties helped us to identify non-essential information which then allowed us to simplify the algorithm, leading to a more efficient one.

We also employed automated tools to explore the state space of the system. This validated our earlier results and confirmed that the complexity of the algorithm was due to the level of concurrency, which was reflected in the large state space. While this state space could be significantly reduced using the stubborn set technique, the cost of doing so quickly became prohibitive.

The approach adopted in this paper presents several advantages: first, proving invariant properties induces that the algorithm is correct whatever the initial tree topology; second, the encoding of the network topology is crucial, and the approach can be generalised to other algorithms provided a suitable encoding of the topology they address.

References

[ABD07] Thara Angskun, George Bosilca, and Jack Dongarra. Binomial graph: A scalable and fault-tolerant logical network topology. In Ivan Stojmenovic,
[AFB+06] Thara Angskun, Graham Fagg, George Bosilca, Jelena Pjesivac-Grbovic, and Jack Dongarra. Scalable fault tolerant protocol for parallel runtime environments. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI Users’ Group Meeting (EuroPVM/MPI’06), pages 141–149. Springer, 2006.

[AM] Dorian C. Arnold and Barton P. Miller. A scalable failure recovery model for tree-based overlay networks. Technical Report TR1626, University of Wisconsin, Computer Science Department.

[APM06] Dorian C. Arnold, G. D. Pack, and Barton P. Miller. Tree-based overlay networks for scalable applications. In Proceedings of the 21st International Parallel & Distributed Processing Symposium (IPDPS’06). IEEE, 2006.

[BBC+02] George Bosilca, Aurélien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fédak, Cécile Germain, Thomas Hérault, Pierre Lemonier, Oleg Lodygensky, Frédéric Magniette, Vincent Néri, and Anton Selikhov. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In High Performance Networking and Computing (SC2002), Baltimore USA, November 2002. IEEE/ACM.

[BBF99] Joffroy Beauquier, Béatrice Bérard, and Laurent Fribourg. A new rewrite method for convergence of self-stabilizing systems. In Prasad Jayanti, editor, Distributed Computing, 13th International Symposium, Bratislava, Slovak Republic, September 27-29, 1999, Proceedings, volume 1693 of Lecture Notes in Computer Science, pages 240–253. Springer, 1999.

[BCH+09] George Bosilca, Camille Coti, Thomas Hérault, Pierre Lemonier, and Jack Dongarra. Constructing resilient communication infrastructure for runtime environments. In International Conference in Parallel Computing (ParCo2009), Lyon, France, September 2009.

[BDDL09] George Bosilca, Remi Delmas, Jack Dongarra, and Julien Langou. Algorithm-based fault tolerance applied to high performance computing. J. Parallel Distrib. Comput., 69(4):410–416, 2009.

[BGL00] Ralph M. Butler, William D. Gropp, and Ewing L. Lusk. A scalable process-management environment for parallel programs. In Jack Dongarra, Péter Kacsuk, and Norbert Podhorszki, editors, Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users’ Group Meeting (EuroPVM/MPI’02), volume 1908, pages 168–175. Springer, 2000.

[BHK+97] Joshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. IEEE Transactions on Parallel and Distributed Systems, 8(11):1143–1156, November 1997.

[Cou02] Pierre Courtiex. Proving self-stabilization with a proof assistant. In 16th International Parallel and Distributed Processing Symposium (IPDPS 2002), 15-19 April 2002, Fort Lauderdale, FL, USA, CD-ROM/Abstracts Proceedings. IEEE Computer Society, 2002.

[DGS99] Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization. Acta Informatica, 36(6):447–462, 1999.
[EA13] L. Petrucci F. Hulin-Hubard A. Linard L.-M. Hillah F. Kordon E. Andre, Y. Lembachar. Cosyverif: an open source extensible verification environment. In Proceedings of 18th IEEE International Conference on Engineering of Complex Computer Systems - ICECCS, pages 33–36. IEEE, 2013.

[FD02] Graham E. Fagg and Jack J. Dongarra. HARNESS fault tolerant MPI design, usage and performance issues. Future Generation Computer Systems, 18(8):1127–1142, October 2002.

[For94] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, Department of Computer Science, University of Tennessee, April 1994. Tue, 22 May 101 17:44:55 GMT.

[GL04] William D. Gropp and Ewing L. Lusk. Fault tolerance in MPI programs. Technical Report ANL/MCS-P1154-0404, Argonne National Laboratory, Mathematics and Computer Science Division, April 2004.

[Gou95] Mohamed G Gouda. The triumph and tribulation of system stabilization. In Distributed Algorithms, pages 1–18. Springer, 1995.

[Jen92] Kurt Jensen. Coloured Petri Nets, Volume 1: Basic Concepts. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.

[Jen94] K. Jensen. Coloured Petri Nets: Basic concepts, analysis methods and practical use. Volume 2: analysis methods. Monographs in Theoretical Computer Science. 1994.

[KV05] Johan Liljus Kimmo Varpaaniemi, Keijo Heljanko. Prod 3.2 an advanced tool for efficient reachability analysis. In Proceedings of Computer Aided Verification, volume 1254 of Lecture Notes in Computer Science, pages 472–475. Springer, 2005.

[Mas77] Gerald M. Masson. Binomial switching networks for concentration and distribution. COM-25(9), Sept. 1977.

[PLP98] James S. Plank, Kai Li, and Michael A. Puening. Diskless checkpointing. IEEE Transactions on Parallel and Distributed Systems, 9(10):972–986, October 1998.

[Val88] Antti Valmari. Error detection by reduced reachability graph generation. In Proceedings of European Workshop on Application and Theory of Petri Nets, pages 95–112, 1988.