Verifying the Leader Election Algorithm in GROOVE

Amir Hossein Ghamarian and Eduardo Zambon

Formal Methods and Tools Group
Department of Computer Science
University of Twente
{a.h.ghamarian, zambon}@cs.utwente.nl

1 Introduction

In this paper we present how the leader election protocol specified in [1] was modelled with the GROOVE tool set. The goal of this protocol is to find the smallest of a set of uniquely numbered processes arranged in a ring, without having a central controller available. GROOVE is a graph transformation tool, which uses directed simple graphs with labelled edges. In the remainder of this article, first we introduce GROOVE in the next section. Then we define the different variants of the problem in Section 3. We propose three different scenarios, gradually more complex, in Sections 4, 5 and 6. In Section 4 we assume that the processes are synchronised and they all generate their messages in the beginning of the algorithm. In Section 5 this constraint is removed and processes can send their initial message at any time. In Section 6 we present a generalised version of the problem, where processes can dynamically enter and leave the network. We show how we verified these three scenarios in Section 7. The experimental results are given in Section 8. Finally, Section 9 concludes.

2 GROOVE

GROOVE is a freely available graph transformation tool which uses the Single Push-Out (SPO) approach. Given a graph grammar (G, P), with G as a start graph, and P as a set of production rules, GROOVE enables the construction of a labelled transition system (LTS) corresponding to all possible permutations of the rules applications. Furthermore, temporal properties defined using computation tree logic (CTL) can be verified to hold in the LTS.

A graph production rule consists of two partially overlapping graphs, a *left hand side* L and a *right hand side* R, and a set of *negative application conditions* (NACs) N, which are also graphs partially overlapping with L. In the visual presentation of a rule used in this paper (which is taken from GROOVE), we combine all these elements together into one graph, made up of four types of elements:

- *Readers*: elements present in both L and R. They have to be present in the source graph for L to match and are preserved in the target graph;
- *Erasers*: elements present in L but not in R. They are matched in the source graph but are not preserved in the target graph, i.e., they are removed.
- *Creators*: elements absent in L but present in R. They are introduced to the target graph.
- *Embargoes*: elements absent in L but present in one of the NACs in N. The set N is built from embargo elements: each unconnected embargo graph corresponds to a NAC.

(a) *Reader* (b) *Eraser* (c) *Creator* (d) *Embargo*

Figure 1: Graph production rule elements

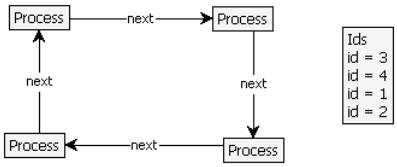


Figure 2: A start state modelling a ring with four processes

To distinguish these four types visually, each element has a distinct colour and form, as shown in Figure 1: readers are black, erasers are dashed blue (darker gray in black and white presentations), creators are bold green (light gray in black and white presentations) and embargoes are bold, dashed red (dark gray in black and white presentations).

A graph production system (GPS) is a set of graph production rules. The effect of applying a GPS (rather than a single rule) is obtained by applying individual rules sequentially, as long as there is an applicable rule; when no more rule can be applied, the transformation terminates and returns the resulting graph.

GROOVE can be downloaded from the website `http://groove.cs.utwente.nl`.

3 Problem Specification

In this paper we consider three different implementations/interpretations of the leader election protocol. Initially, we assume a simple case where all the messages are generated at the same time as the first step of the algorithm. We also address the case where processes are not bound to send their messages at the same time in the beginning. In other words, processes can generate their messages during the execution of the algorithm. Finally, we generalise this setting by allowing processes to enter and leave the network at an arbitrary moment.

We begin by explaining the initial start graphs that are used in the graph grammars. Also, we explain how id's are initially assigned to the processes.

The Start State: Figure 2 shows a start graph that models a ring with four processes. There is an extra dummy node *Ids* containing id's ranging from 1 to 4. This node is used to generate all possible permutations of processes id's in the network. This graph can easily be extended to any arbitrary number of processes. Note that the selection of id's among numbers from 1 to n can be regarded as a canonical representation of any arbitrary set of n natural numbers.

Id Distribution: As seen in the start graph, processes initially do not have an id. Id's are assigned to the processes using the *pick-id* rule (shown in Figure 3), which has the highest priority among all the rules. This rule is applied until all processes have an id. Due to the negative application condition in the rule, *pick-id* prevents processes from receiving more than one id. By exploring all possible sequences of application of this rule, we generate all possible permutations of id's in a ring of size n. The number of these permutations is $(n-1)!$.

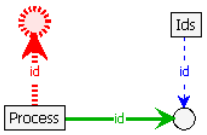


Figure 3: *pick-id* rule

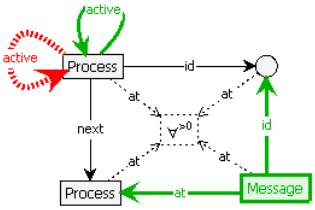
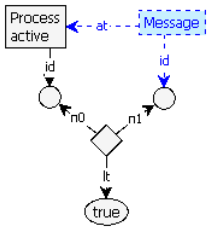
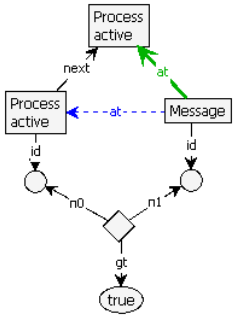


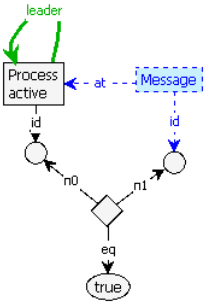
Figure 4: *create-msgs* rule



(a) *drop-msg* rule



(b) *propagate-msg* rule



(c) *new-leader* rule

Figure 5: Rules for the synchronous protocol

4 Scenario 1: A Synchronous Protocol

The following rules define the behaviour of the protocol for the case where the generation of messages is synchronous and happens at the beginning of the algorithm.

With the quantified rule *create-msgs* we make all the processes generate their messages at the same time. This rule, shown in Figure 4, has the second highest priority after *pick-id*. Therefore, every process sends a message containing its id to the next process, before any other rule becomes applicable. Also, every process marks itself using an active self-loop after it generates its message. An embargo edge prevents the assignment of more than one active self-loop to a process. Consequently, it prevents processes from sending multiple messages. The use of an universal quantifier in this rule implies synchronicity (common clock) among all the processes. We will remove this constraining assumption in the next scenario.

Here, we explain all the rules that are required for the implementation of the synchronous protocol, after the construction of the initial messages. Some additional explanation on the visual notation of the rules is needed. A node drawn as an ellipsis (circle) is an attribute and diamond nodes represent attribute operations. Diamond nodes perform an operation on their input attributes (identified by the π_i edges) and compare the result to an attribute node. The operation to be performed is defined by the edge label (e.g., *lt* is the $<$ comparison). Thus, a rule with a diamond node is only applicable if the operation described can be performed.

- *drop-msg*: This rule, shown in Figure 5(a), discards messages whose id is higher than the id of the receiving process.
- *propagate-msg*: This rule, shown in Figure 5(b), relays a message to the next process if the id of the message is smaller than the one of the receiving process.
- *new-leader*: In this rule, shown in Figure 5(c), the process whose id equals the id of its received message declares itself as the leader.

5 Scenario 2: An Asynchronous Protocol

In this section we describe a second variant of the protocol, where not all messages are necessarily generated in the beginning of the algorithm.

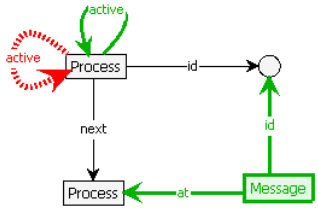


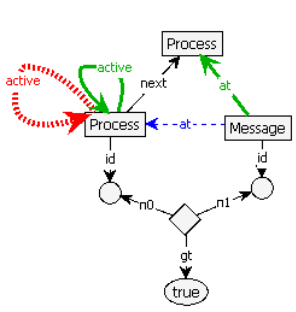
Figure 6: *create-msg* rule (asynchronous)

The *create-msg* rule is the same as the one explained in Section 4 with the quantifier removed. The rule is shown in Figure 6. Also the three rules *drop-msg*, *propagate-msg*, and *new-leader* are the same as those described in Section 4. We have two extra rules in this variant. When a process has not yet generated its message but has already received a message, depending on the relative values of the message id and the process id, one of the following rules become applicable.

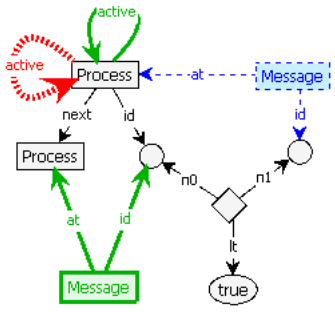
- *activate-and-propagate*: When the received message has a smaller id than the process, this rule (shown in 7(a)) becomes applicable and prevents the process from generating a message. The process also relays the received message to the next process.
- *activate-and-create*: As opposed to the previous case, this rule (shown in Figure 7(b)) applies when the message id is larger than the process id. In this case the message is discarded and a new message containing the process id is sent to the next process.

6 Scenario 3: A Dynamic Protocol

In the previously discussed synchronous and asynchronous protocols, it was assumed that the processes are fixed and active in the network all the time, and neither they could leave the network nor new processes could enter. Here, we explain a more general protocol, where we remove these constraints, allowing processes to enter and leave the network. However, to preserve the required properties of the protocol, we imposed the following assumptions.



(a) *activate-and-propagate* rule



(b) *activate-and-create* rule

Figure 7: Rules for the asynchronous protocol

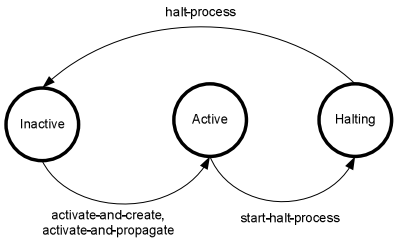


Figure 8: The Process Modes

- We assume that the ring has a maximum size.
- We assume that a process can have at most one message in a given moment.
- When an active process decides to leave the network it should first enter an intermediate *halting* state. Only from the halting state it can be deactivated. The relation between different states of a process is shown in Figure 8, in terms of a finite state machine labelled with the applicable rules on each state.

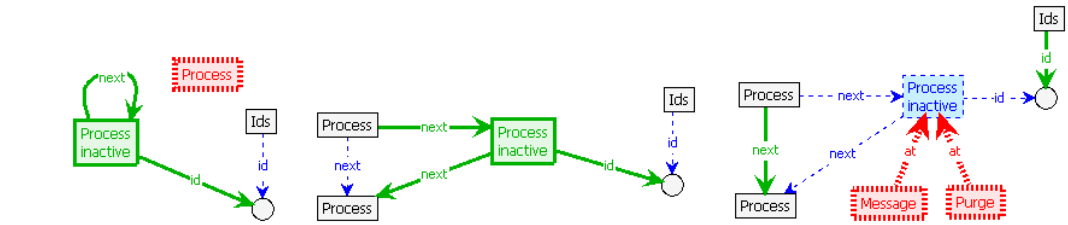
Process Creation Rules: First we explain the set of rules that create new processes. Note that all the newly created processes are in an inactive state. These rules are illustrated in Figures 9(a) and 9(b). It is important to note that, as in the the previous scenarios, a newly created process picks an id from the id pool, which guarantees that there is no id duplication. When a process is deleted it returns its id to the id pool.

- *create-first-process*: creates the first process, and assigns itself as the next process in the ring.
- *create-inactive-process*: inserts a new process in the network.

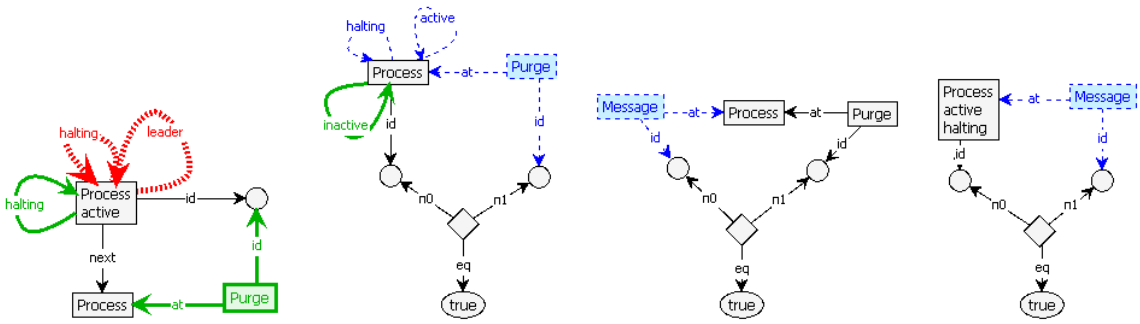
Process Deletion Rules: An inactive process can remove itself from the network at any time. An active process, however, needs to deactivate itself before leaving the network, since removing an active process with the minimum id after its message has been sent results in an ever circulating message. Therefore, a process first goes to an intermediate halting state while generating a purge message with its id. A process which receives a purge message, will drop its message if its id is equal to that of its purge message. Consequently, the previously sent message of the process will eventually be dropped. The process can go to the inactive state after receiving its own purge message. The process deletion rules are shown in Figures 9(c)-9(g).

- *remove-inactive-process*: removes an inactive process when it does not have any message.
- *start-halt-process*: starts the halting mode for an active, non-leader process by adding a halting label and sending a purge message containing the id of the process.
- *halt-process*: changes the state of a halting process to inactive when the process receives a purge message whose id equals its own.
- *purge-msg*: forces a process to drop its message if the id of the message and the purge message are equal.
- *halting-leader*: makes the halting process drop the message if it receives a message with an id equal to its own. In other words, it does not claim itself a leader, and wait for its purge message.

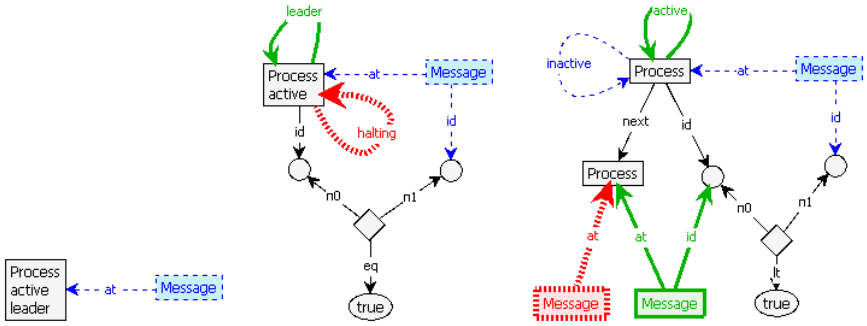
Leader Announcement Rules: When a process becomes the leader, it discards all the messages that it receives. Figures 9(h) and 9(i) illustrate these rules.



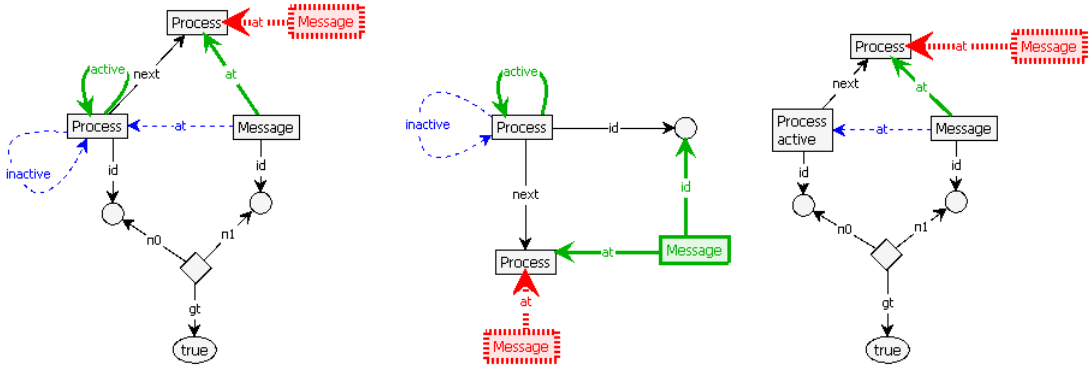
(a) *create-first-process* rule (b) *create-inactive-process* rule (c) *remove-inactive-process* rule



(d) *start-halt-process* rule (e) *halt-process* rule (f) *purge-msg* rule (g) *halting-leader* rule



(h) *leader-filter* rule (i) *new-leader* rule (j) *activate-and-create* rule



(k) *activate-and-propagate* rule (l) *create-msg* rule (m) *propagate-msg* rule

Figure 9: Rules for the dynamic protocol

(a) The *multiple-leaders* rule

(b) The *exists-leader* rule

Figure 10: Verification Rules

- *leader-filter*: A leader discards all messages that it receives.
- *new-leader*: If the process is not in the halting state and receives a message whose id is equal to its own, it proclaims itself to be the leader.

The Message Propagation Rules: The *drop-msg* rule is the same as the one explained in the previous scenarios. The rules *activate-and-create*, *activate-and-propagate*, *create-msg* and *propagate-msg* are similar to the ones explained in the asynchronous case, except from the fact that they assume a size one for the message buffers. These rules are shown in Figures 9(j)-9(m).

7 Verification

GROOVE allows us to verify if CTL specified properties hold in the generated LTS. To verify the leader election protocol, we added two non-modifying rules to assist us with the model checking part. Two different properties need to be checked. The first property (safety) is that at most one leader should exist in any state of the LTS. This property is only true if the rule *multiple-leaders* (Figure 10(a)) is not applicable in any state. The rule has an special embargo node labeled =, which is used to indicate that the processes should be different. This NAC is necessary because normally the matching of rules in GROOVE is non-injective. The second property (liveness) to be checked is that the algorithm should always declare a leader after a finite number of steps of the protocol. This property holds in a state if the rule *exists-leader* (shown in Figure 10(b)) is applicable.

Verifying our rule sets now can be done by checking CTL formulae on the generated LTS. The safety property can be verified if there is no counter example to the CTL formula $AG(!$*multiple-leaders*$)$, which means that there should not be two different leaders in any state. The liveness property is preserved if we have no counter example to $AF($*exists-leader*$)$, meaning that all paths in the the LTS eventually lead to the choice of a leader.

8 Experimental Results

We verified our rule sets on rings with three to eight processes. For our tests we used a Dual Quad Core Xeon with 2.66 GHz clock frequency and 16 GB of memory. We conducted our experiments for all three cases of synchronous, asynchronous and dynamic protocols. The results are shown in Table 1. The first two columns, which are common among all cases, show the number of processes (size of the ring) and the number of different configurations for the ring of the given size, respectively. The results of the experiments for each case are given in three columns. The first two columns denote the number of states and transitions in the generated transition system and the third column shows the amount of time necessary for the verification.

After generating the complete LTS for a given scenario and a ring of size n, we checked the validity of the formulae explained in Section 7. In scenarios 1 and 2 (synchronous and asynchronous cases) both the safety and liveness properties hold, at least for rings of size between 3 and 7. For $n \geq 8$ it was not possible to generate the complete state space and hence we could not check the CTL formulae. In scenario 3 (dynamic case), the safety property also

Table 1: Experimental results

		Synchronous			Asynchronous			Dynamic		
n	# rings	states	trans.	time	states	trans.	time	states	trans.	time
3	2	52	113	<1 s	73	174	<1 s	1443	4246	2.4 s
4	6	473	1304	<1 s	741	2238	<1 s	122018	432379	1.91 min
5	24	6358	21621	4 s	10703	39582	10 s	out of memory		
6	120	113102	459834	57 s	199988	877590	3.63 min	out of memory		
7	720	2492888	11806644	67.64 min	4747432	23914934	19.62 h	out of memory		
8	5040	out of memory			out of memory			out of memory		

hold. However, the liveness does not hold. One reason for this is that, in this scenario, processes can keep entering and leaving the network without electing a leader. In the LTS this translates to cycles where the *exists-leader* rule is never active and thus the formula $AF(\textit{exists-leader})$ does not hold. In this particular case, we verified a weaker property, specified by the formula $AG(EF(\textit{exists-leader}))$, checking that for any state there exists a path which eventually leads to the election of a leader.

9 Conclusion

We have defined, implemented and verified three different scenarios: synchronous, asynchronous, and dynamic versions of the leader election protocol. In order to verify our rule sets, we explored all possible permutations of networks with size n. For the first two scenarios no assumptions were imposed on the message relaying (except that each process has a buffer of size n). Hence, we have verified that the algorithm works regardless of the buffer policy adopted (e.g., FIFO, LIFO, etc). This is a very interesting general result.

All three different variants were very straightforward to model with GROOVE and fit very well within its scope. The specification of the properties to verify was also a very simple task. Nevertheless, as is usually the case with model checking techniques, the verification was computationally expensive. Another important observation is that GROOVE can be used as fast prototyping tool. This is more visible in the third scenario, where the counter-examples provided by the tool can be used to refine the rules of the protocol until a satisfiable solution is found.

References

[1] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, 1979.