



# Formal verification of distributed leader election algorithms with model checking

June 1, 2023

STUDENT

**Kevin Joshua Vinther**  
*kevin20@student.sdu.dk*

SUPERVISOR

**Marco Peressotti**  
*Peressotti@imada.sdu.dk*

Department of Mathematics and Computer Science  
University of Southern Denmark, SDU.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Overview</b>	<b>4</b>
2.1	Leader Election . . . . .	4
2.2	Bully Algorithm . . . . .	4
2.3	Ring Algorithm . . . . .	5
<b>3</b>	<b>Bully Algorithm</b>	<b>7</b>
3.1	Converting to TLA+ . . . . .	7
3.2	Setup . . . . .	7
3.3	Sending Messages . . . . .	8
3.4	Handling Received Messages . . . . .	10
3.5	Delegating Functions with HandleMessages . . . . .	11
3.6	Checking the leader condition . . . . .	11
3.7	Next state relation and killing the leader . . . . .	12
3.8	Invariants and properties . . . . .	13
3.9	Temporal Formula . . . . .	13
<b>4</b>	<b>Ring Algorithm</b>	<b>15</b>
4.1	Similarities to Bully Algorithm . . . . .	15
4.2	Setup . . . . .	15
4.3	Neighbours and Sending Messages . . . . .	15
4.4	HandleMessages . . . . .	18
4.5	Checking The Leader Condition . . . . .	18
4.6	Next state relation and killing the leader . . . . .	18
4.7	Invariants and Properties . . . . .	19
4.8	Temporal Formula . . . . .	19
<b>5</b>	<b>Evaluation</b>	<b>21</b>
5.1	The Model Checker . . . . .	21
5.2	Shortcomings . . . . .	21
5.3	Pseudocode to TLA <sup>+</sup> . . . . .	22
5.4	Potential Future Changes . . . . .	22
<b>6</b>	<b>Conclusion</b>	<b>24</b>
<b>A</b>	<b>Bully Algorithm TLA<sup>+</sup> model</b>	<b>26</b>

<b>B Ring Algorithm TLA<sup>+</sup> model</b>	<b>30</b>
---	-----------

# Chapter 1

## Introduction

Today, more than ever before, distributed systems have a crucial role in computer infrastructure. However, they are also notoriously difficult to implement correctly. Even with only a few interacting processes in a distributed system, it can lead to a tremendous amount of states. Catching each bug and edge case in the many possible states can take an enormous amount of time, and be exceedingly challenging.

Distributed leader election is a distributed algorithm which requires that all processes in a distributed system agree on one specific process as their leader. This is crucial for many modern systems as distributed algorithms are typically not completely symmetrical, and therefore some process has to take the lead in initiating the algorithm. Leader election can find a leader, to become this process which initiates the algorithm [1].

It is a challenge to verify that distributed algorithms work as intended in a distributed system using conventional tools today. Therefore, we use techniques from formal verification methods, because they help us test distributed algorithms in a way that no conventional tool can.

We seek to verify the correctness of two distributed leader election algorithms: the Bully Algorithm and the Ring Algorithm<sup>1</sup>.

We do this using techniques from formal methods to check if there are any bugs, concurrency issues and security issues in the algorithm. Throughout this project the TLA<sup>+</sup> (Temporal Logic of Actions Plus) verification suite is used to model and verify the Ring and Bully Algorithm.

In the TLA<sup>+</sup> verification suite, the TLC model is included. The model checker verifies the correctness of the algorithm by going through each possible state, and making sure that it produces satisfiable behavior, thus verifying if an algorithm works even in the cases that are hard to find using conventional tools [4].

The goal of this project is to have modeled both the Bully Algorithm and the Ring Algorithm in the TLA<sup>+</sup> specification language, and have tested them using the TLC model checker to verify if they are correct.

---

<sup>1</sup>The Ring Algorithm is a familiar name for the Lelang, Chang and Roberts algorithm (also known as LCR). We will continue to use the name “Ring Algorithm”.

# Chapter 2

## Overview

### 2.1 Leader Election

In several algorithms for distributed systems, a process may possess the role of the *leader*. Usually a leader may be required because distributed algorithms typically are not completely symmetrical. Thus, the leader can take the lead in initiating the algorithm.

The aim of an algorithm for the Leader Election problem is to elect a leader from a set of processes in a way that all processes can agree on the same leader. It is crucial that all processes agree on the same leader, as if one process has the wrong leader, it can lead to many issues, such as failures in the system.

This project focuses on the Bully Algorithm and the Ring Algorithm. The Ring Algorithm works on a complete graph, with an undirected ring superimposed. The Bully Algorithm works on a complete undirected graph.

For both the Bully and Ring Algorithm we assume that:

- The network is reliable. Messages do not get lost.
- Nodes may fail at any time, including the leader.
- Fail-stop model. Failed nodes are removed from the system forever.
- Messages are asynchronous.

### 2.2 Bully Algorithm

In the Bully Algorithm, the elected leader will either be the leader with the lowest ID, or the highest ID. In this project, we assume that the leader is the process with the highest id.

In the algorithm, there exists three different types of message to be sent and received. These messages initiate behavior on the receiving process, depending on what the contents of the message is. The messages are:

- *election*: sent to announce an election.
- *alive*: sent to respond to an *election* message from a lower id.
- *victory*: sent by winner to announce victory.

Rather than traditional pseudocode, we base ourself on a series of instructions (which throughout the report will be referred to as pseudocode.)

When a process  $p$  recovers from failure, or the failure detector indicates that the current leader has failed,  $p$  performs the following actions:

1. If  $p$  has the highest process  $id$ , it sends a *victory* message to all other processes and becomes the new leader. Otherwise,  $p$  broadcasts an *election* message to all other processes with higher process  $ids$  than itself.
2. If  $p$  receives no *alive* after sending an *election* message, then it broadcasts a *victory* message to all other processes and becomes the leader.
3. If  $p$  receives an *alive* from a process with a higher  $id$ , it sends no further messages for this election and waits for a *victory* message. (If there is no *victory* message after a period of time, it restarts the process at the beginning.)
4. If  $p$  receives an *election* message from another process with a lower  $id$  it sends an *alive* message back and if it has not already started an election, it starts the election process at the beginning, by sending an *election* message to higher-numbered processes.
5. If  $p$  receives a *victory* message, it treats the sender as the leader.

## 2.3 Ring Algorithm

Unlike the Bully Algorithm, the Ring Algorithm assumes a superimposed undirected ring. This means that we assume the processes to be in a ring, and model the implementation from this assumption.

In the Ring Algorithm, there exists two different types of messages:

- *probe(id)*: sent to search for the leader.
- *selected(id)*: sent to announce the result.

You will notice in the messages, that, unlike in the Bully Algorithm, they have an attached  $id$ . While it may read as if it is a function, it is just an attachment to the message. The  $id$  is used to identify who originally sent the message.

Furthermore, the Ring Algorithm requires a *boolean* variable, called *participate*. The variable indicates whether or not process  $P_i$  participates in the election.

Like in the Bully Algorithm, we base ourself on a set of instructions rather than traditional pseudocode.

When process  $P_i$  wakes up to participate in the election:

1. Send *probe(i)* to right neighbour
2.  $participate \leftarrow \text{TRUE}$

When *probe(k)* message arrives from the left neighbour  $P_j$ :

1. If  $participate = \text{FALSE}$  then  $participate \leftarrow \text{TRUE}$ .
2. if  $i > k$  then discard the probe
3. else if  $i < k$  then forward *probe(k)* to right neighbor

4. else if  $i = k$  then declare  $i$  is the leader; circulate  $selected(i)$  to right neighbour;

When a  $selected(x)$  message arrives from left neighbour:

1. if  $x \neq i$  then note  $x$  as the leader and forward message to right neighbor
2. else do not forward the  $selected$  message.

In this algorithm, like in the Bully Algorithm, we assume that the leader will be the process with the highest id alive.

## Chapter 3

# Bully Algorithm

### 3.1 Converting to TLA+

Since we have based ourself on an instruction set rather than pseudocode for what process  $p$  should do, it has been more of a process of modelling the algorithm correctly, rather than converting it from pseudocode.

The second point in the algorithm described earlier states that, if process  $p$  receives no *alive* message after sending an Election message, it then broadcasts an *victory* message to all other processes. Since TLA<sup>+</sup> does not allow for waiting a specified amount of time, and we assume a *fail-stop* model, instead of waiting to see, it immediately checks if process  $P$  is the process with the highest ID alive, and broadcasts a *victory* message if this is the case. We allow ourself this because the *fail-stop* assumes that all processes will know the failure of other processes, but it is not specified how. Therefore, we can assume that the process will know that it is the highest process. This is also done in the beginning of the algorithm, when a process checks if a leader is dead. However, the rest of the implementation is as true to the set of instructions as possible.

Furthermore, I have made a modification to point 4. However, as this is not done because of limitations of TLA<sup>+</sup>, it is discussed in Section 3.4.

### 3.2 Setup

Before modeling the behavior of the specification, we set up the constants, variables and initialize them in the *Init* function. Looking at the specification for the algorithm, we need to model processes and messages.

The algorithm assumes a complete undirected graph. We model this by creating a **State** variable, which contains a record of all *processes* and their fields. We define the **State** variable to be a sequence of records, each of which hold the following metadata in its fields (Figure 3.1):

- *ID*: The process has a unique ID.  $ID \in ProcessID$ , where  $ProcessID$  is the total number of processes, defined as a range from 1 to the natural number  $N$ .
- *Condition*: A process is either dead or alive (called *active*).  $Condition \in \{"Dead", "Active"\}$
- *Leader*: The process should know who its leader is. The leader may be the process itself.  $Leader \in ProcessID$



<i>Setup</i>	
EXTENDS	<i>Naturals, FiniteSets, Sequences</i>
CONSTANT	$N$
ASSUME	$N \in \text{Nat}$
	$\text{ProcessID} \triangleq 1 \dots N$
VARIABLES	$\text{State}, \text{MessageBox}$
	$\text{Message} \triangleq \text{ProcessID} \times \{ \text{"ELECTION"}, \text{"ALIVE"}, \text{"VICTORY"} \}$
	$\text{Init} \triangleq \wedge \text{State} = [p \in \text{ProcessID} \mapsto$ <div style="margin-left: 100px;"> <math>[ID \mapsto p,</math>  <math>\text{Condition} \mapsto \text{"Active"},</math>  <math>\text{Leader} \mapsto N,</math>  <math>\text{Participating} \mapsto \text{FALSE}]</math> </div> $\wedge \text{MessageBox} = [p \in \text{ProcessID} \mapsto \langle \rangle]$

Figure 3.1: Setup for the Bully Algorithm

- *Participating*: The process is either participating in the election or not. *Participating*  $\in$  BOOLEAN

Furthermore, we define a sequence, **MessageBox** which holds every processes received messages. The **MessageBox** maps from each process,  $p$  initially to an empty tuple, and when messages are received, a sequence of messages. These tuples are the **Messages**. We define a **Message** as  $\text{ProcessID} \times \{ \text{"ELECTION"}, \text{"ALIVE"}, \text{"VICTORY"} \}$ , thus an example of an "ELECTION" message from process 1 is:  $\langle 1, \text{"ELECTION"} \rangle$ .

### 3.3 Sending Messages

We earlier defined the **MessageBox** to be a sequence of **Messages**. This makes the handling of messages extremely convenient. One of the core operations we make with messages are sending them. To send a message from process  $p$  to process  $q$  with the contents **ELECTION**, we simply append the tuple  $\langle p, \text{"ELECTION"} \rangle$  to  $\text{MessageBox}[q]$ . In TLA<sup>+</sup> this is done like so:

$\text{MessageBox}'$   
 $= [\text{MessageBox} \text{ EXCEPT } ![q] = \text{Append}(\langle \langle p, \text{"ELECTION"} \rangle \rangle)]$

This defines the **MessageBox** variable to have  $\langle p, \text{"ELECTION"} \rangle$  appended to  $\text{MessageBox}[q]$  in the next state.

In the sender functions (Figure 3.2) you will notice two different types of sending predicates: one which is simply "SendX", and one which is "SendXAndTail". The AndTail variant does the same as SendX, except it also removes the newest message in process  $p$ 's inbox. This is done as to not clutter the inbox with irrelevant messages.

As discussed earlier, there are three different messages in the bully algorithm: *alive*, *victory*, and *election*.



<i>HandleReceivedMessages</i>	
$ReceiveAlive(p, q) \triangleq$	$\wedge State[p].Participating = \text{TRUE}$ Make sure they are already participating $\wedge p > q$ $\wedge State' = [State \text{ EXCEPT } ![p].Participating = \text{FALSE}]$ No longer participate
$ReceiveElection(p, q) \triangleq$	$\vee \wedge p = MaxAliveID$ $\wedge SendVictoryAndTail(p)$ $\wedge State' = [State \text{ EXCEPT } ![p].Participating = \text{FALSE},$ $![p].Leader = p]$ $\vee \wedge p \neq MaxAliveID$ $\wedge \vee \wedge State[p].Participating = \text{TRUE}$ $\wedge SendAliveAndTail(p, q)$ $\wedge \text{UNCHANGED } State$ $\vee \wedge State[p].Participating = \text{FALSE}$ $\wedge SendAliveAndTail(p, q)$ $\wedge State' = [State \text{ EXCEPT } ![p].Participating = \text{TRUE}]$
$ReceiveVictory(p, q) \triangleq$	$State' = [State \text{ EXCEPT } ![p].Leader = q]$ Set the leader of receiver to $p$ sender

Figure 3.5: Message Handlers

This time it's all processes with a higher ID than  $p$ . This is achieved using a helper function *HigherIDs* (see figure 3.3.)

### 3.4 Handling Received Messages

When a message is received, a process will periodically use the *HandleMessages* function. This will then execute the correct functions depending on the message. The functions in Figure 3.5 describe the behavior when receiving a specific message.

Points 3, 4, and 5 from the pseudocode describe what to do when receiving messages:

3. If P receives an alive from a process with a higher ID, it sends no further messages for this election and waits for a Victory message. (If there is no Victory message after a period of time, it restarts the process at the beginning.)
4. If P receives an Election message from another process with a lower ID it sends an alive message back and if it has not already started an election, it starts the election process at the beginning, by sending an Election message to higher-numbered processes.
5. If P receives a victory message, it treats the sender as the leader.

The function *ReceiveAlive* (Figure 3.5) addresses point 3. It checks if the message is from a higher id than itself, and if so, it modifies its participation status to be FALSE.

The function *ReceiveElection* addresses point 4. However, you may notice that it does not correctly follow point 4. Point 4 explicitly states that “...if it has not already started an election, it starts the election process at the beginning, by sending an Election message to higher-numbered processes”. Instead of doing this, it only sends an alive message, and, if its own participation status is FALSE, it sets it to be TRUE instead. This change is done because it will only lead to a number of irrelevant *election* messages. Furthermore, we

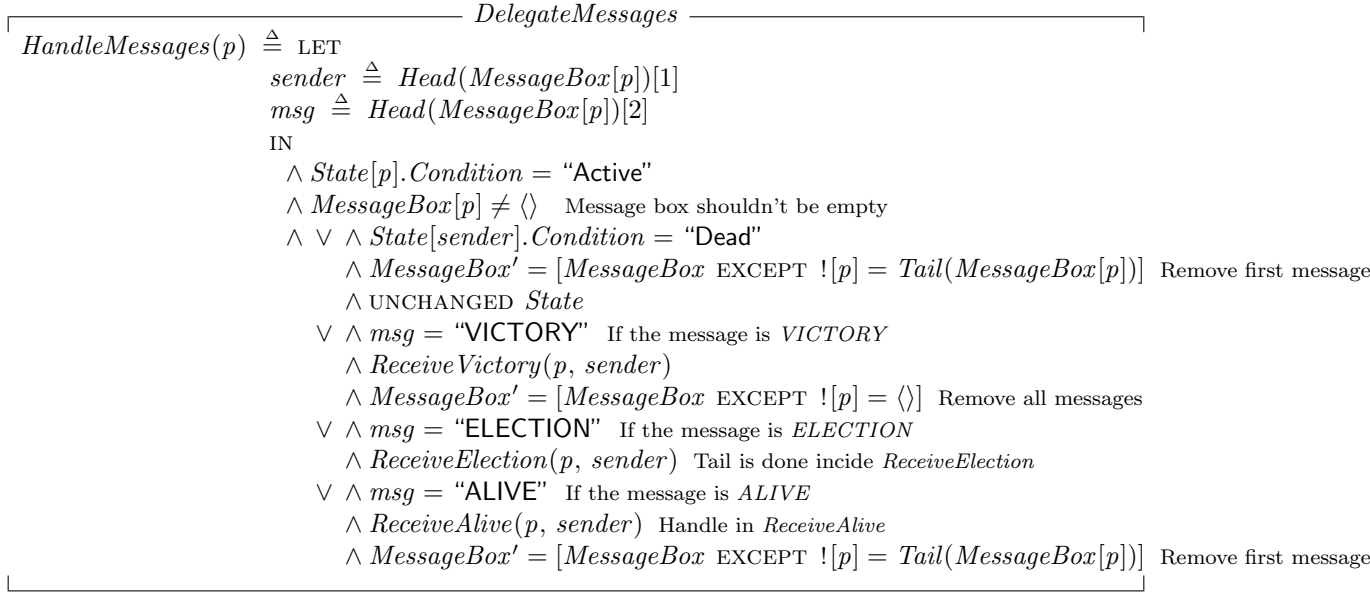


Figure 3.6: Delegating Functions

have deviated from point 4 in the sense that we check if the process has the highest ID alive, and if so, it declares victory. This is done because we assume a fail-stop model.

The *ReceiveVictory* function addresses point 5 by setting the leader to the sender of the *victory* message.

### 3.5 Delegating Functions with HandleMessages

In the *Next* state relation of the specification, one of the actions a process can take is to check its messages. This is done via the *HandleMessages* function. If process  $p$  has any messages in their inbox, one of 4 things can happen, depending on the type of message or the condition of the sender.

1. If the sender is dead, we ignore the message and remove it from the inbox of process  $p$ .
2. If the content of the message is *victory*, we delegate work to the *ReceiveVictory* function and remove all messages from process  $p$  because we do not care about other message at this point.
3. If the content of the message is *election*, we delegate the rest of the work to the *ReceiveElection* function.
4. If the content of the message is *alive*, we delegate work to the *ReceiveAlive* function and remove the message from the inbox of process  $p$ .

### 3.6 Checking the leader condition

One of the actions a process can take in the *next* state relation is to check the condition of the leader. In the pseudocode, it is specified that “When a process P recovers from failure, or

$$\begin{aligned}
\text{CheckLeader}(p) \triangleq & \quad \wedge \text{State}[p].\text{Condition} = \text{"Active"} \\
& \wedge \text{State}[\text{State}[p].\text{Leader}].\text{Condition} = \text{"Dead"} \\
& \wedge \vee \wedge p = \text{MaxAliveID} \\
& \quad \wedge \text{SendVictory}(p) \\
& \quad \wedge \text{State}' = [\text{State} \text{ EXCEPT } ![p].\text{Participating} = \text{FALSE}, ![p].\text{Leader} = p] \\
& \vee \wedge p \neq \text{MaxAliveID} \\
& \quad \wedge \text{State}[p].\text{Participating} = \text{FALSE} \\
& \quad \wedge \text{SendElection}(p) \\
& \quad \wedge \text{State}' = [\text{State} \text{ EXCEPT } ![p].\text{Participating} = \text{TRUE}]
\end{aligned}$$

Figure 3.7: Checking the condition of the leader

$$\begin{aligned}
& \text{NextPredicateAndKillLeader} \\
\text{KillLeader} \triangleq & \quad \wedge \text{State}[\text{MaxAliveID}].\text{Leader} = \text{MaxAliveID} \\
& \wedge \text{State}' = [\text{State} \text{ EXCEPT } ![\text{MaxAliveID}].\text{Condition} = \text{"Dead"}, ![\text{MaxAliveID}].\text{Participating} = \text{FALSE}] \\
& \wedge \text{Cardinality}(\{p \in \text{ProcessID} : \text{State}[p].\text{Condition} = \text{"Active"}\}) \geq 2 \\
& \wedge \text{UNCHANGED } \text{MessageBox} \\
\text{Next} \triangleq & \quad \vee \text{KillLeader} \\
& \vee \exists p \in 1 \dots \text{MaxAliveID} : \text{CheckLeader}(p) \vee \text{HandleMessages}(p)
\end{aligned}$$

Figure 3.8: Next state relation and KillLeader

the failure detector indicates that the current leader has failed...". Since we assume that the processes do not recover from failure we use the *CheckLeader* (Figure 3.7) function instead.

The function only runs if the process checking is active, and the leader is dead. If the process is active, the function checks whether process  $p$  is the process with the highest id. If  $p$  has the highest id, it sends a *victory* message, and declares itself the new leader. If  $p$  is not the process with the highest id, we first make sure that it is not currently participating in any election, as to not accidentally start multiple elections in one cycle. Then process  $p$  sends an election message, and sets its own Participation status to TRUE, as it is now in an election.

### 3.7 Next state relation and killing the leader

The Next state relation (Figure 3.8) chooses what should happen in the next state. For this specification there are two options: Either, the leader is killed, or a process handles its messages or checks its leaders status.

In case the choice is made to kill the leader, we make sure that the leader sees itself as the leader, and if they do, we change its condition in the next state to be dead and its participation status to be FALSE, effectively killing it. This is only done if there are 2 or more processes alive, as there is no reason to initiate a leader election when there is only one process.

In case the decision to kill the leader is not chosen, some process is chosen. The process either checks its leader or handles its messages.

## 3.8 Invariants and properties

### Invariants

In  $TLA^+$ , an *invariant* is a statement that must be true on every state of the program. [3] One common type of invariant is the *TypeOK* invariant (Figure 3.9). *TypeOK* makes sure that the types never go out of a specified domain or change to an unwanted value. In the case of this specification this is true for the *State* and *MessageBox* variables. For the *State* variable, it will hold true that:

- A process' *id* is a part of the *ProcessID*
- A process' condition is either *Active* or *Dead*
- A process' *leader* is a part of the *ProcessID* set
- A process' participation status is a BOOLEAN

For the message box it holds that each ID will map to a sequence of messages.

In this specification, we also have an invariant called *UniqueID* which makes sure that no two processes hold the same ID. Furthermore, we also have an invariant called *Participating* which states that, if a process is participating in an election, it cannot be the leader itself, or see itself as leader.

### Properties

There are two kinds of temporal properties: safety properties, and liveness properties. Safety properties make sure that the system doesn't do bad things, and liveness makes sure our system always does a good thing.[3]

We have three properties: two safety properties and one liveness property. The safety properties are *ElectionTerminationImpliesSameLeader*, and *HighestAliveProcessIsLeader*. *ElectionTerminationImpliesSameLeader* ensures that when two processes that have been in an election eventually get out of the election, they will have the same leader. *HighestAliveProcessIsLeader* ensures that if a process is not participating in an election, its leader is the process with the highest id alive.

The liveness property, *ElectionWillEnd* ensures that an election will eventually end, such that it does not just run forever.

## 3.9 Temporal Formula

To run the specification with the model checker, the temporal formula in Figure 3.10 is used. The reason a temporal formula is used rather than just specifying the initial predicate and next-state relation is to use *strong fairness*. Without strong fairness, the model checker might just decide to stop the entire system. That is an intended feature, but it is not what we want. Strong fairness ensures that if an action is always enabled, it eventually happens.[2]

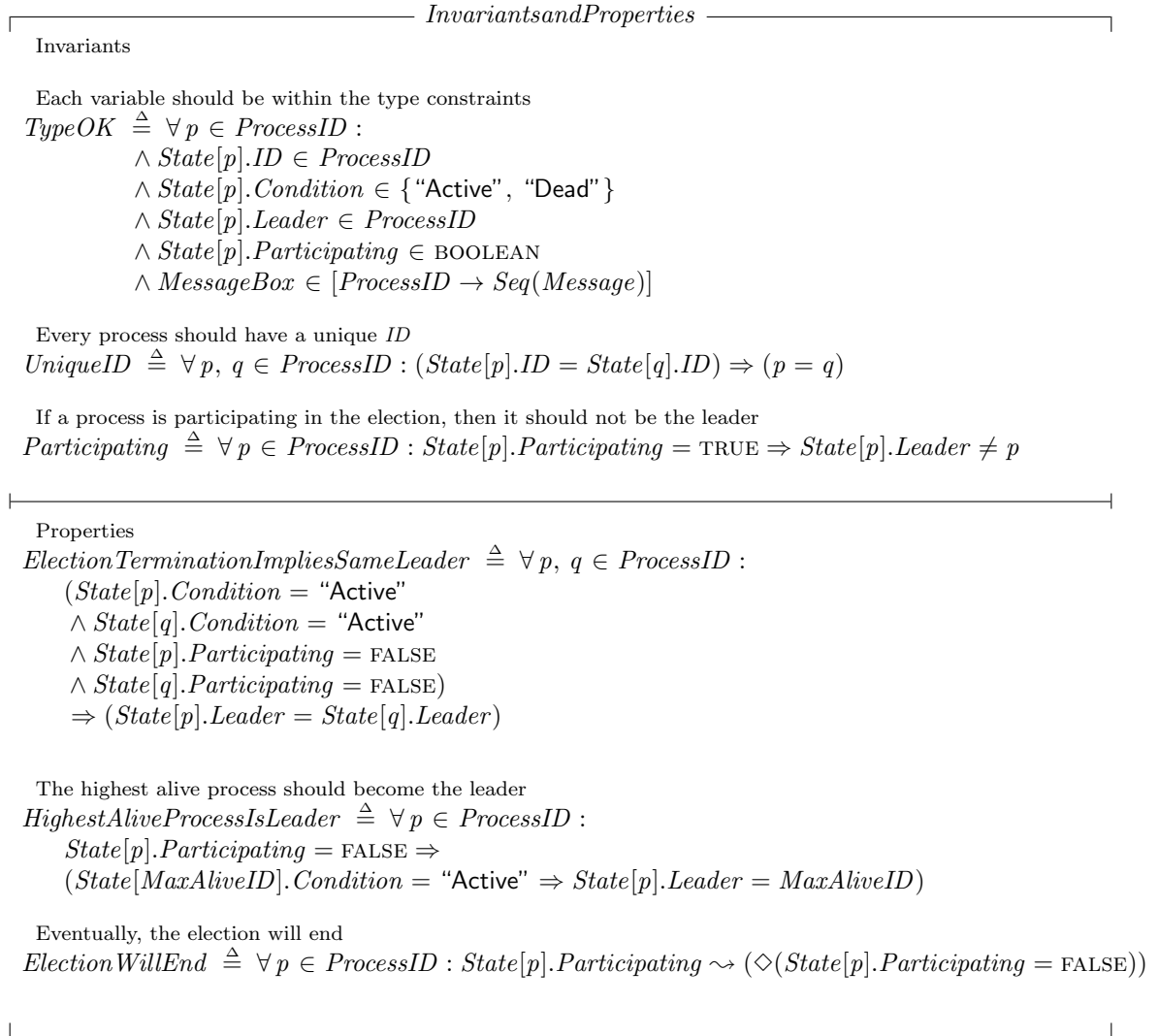


Figure 3.9: Invariants and Properties in the Bully Algorithm

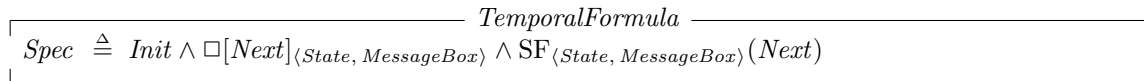


Figure 3.10: Temporal Formula

## Chapter 4

# Ring Algorithm

### 4.1 Similarities to Bully Algorithm

The Ring Algorithm and the Bully Algorithm are both similar in many ways. Since both algorithms are algorithms on graphs, we can reuse a lot of the setup. The Ring Algorithms superimposed overlay is an undirected ring. We will not need to model it is a ring, even though it is a ring. Instead, we assume it to be a ring, and write the specification as if it was a ring.

### 4.2 Setup

The setup (Figure 4.1) for the Ring Algorithm is almost identical to that of the Bully Algorithm. The only difference is how we define the **Messages**. A message in the Ring Algorithm has been changed in two ways. First, we don't send the same messages in the Ring Algorithm. The messages in the Ring Algorithm are *probe* and *selected*. Second, we switch the order of message and process. In the Bully Algorithm it made sense to have the ProcessID as the first part of the message, since it is the sender of the message. In the Ring Algorithm, the process id is the ID of the message, and not the sender. Since both messages in the Ring Algorithm have an ID attached to them, the messages instead have the process on the right side. However, this is purely a syntactic choice.

### 4.3 Neighbours and Sending Messages

#### Neighbour

In a ring topology, each process has a left-side neighbour and a right-side neighbour. For the Ring Algorithm we only care about the right-side neighbour. Thus, the *Neighbour* function (Figure 4.2) returns the left-side neighbour. It does this first by checking if the process asking for its neighbour is the process with the highest id alive, and if so it returns the lowest id alive. Otherwise, it uses CHOOSE to choose the process whose id is higher than p, but lower than every other process whose id is higher than p, i.e. the neighbour.

#### Sending Messages

We deviate a lot from the Bully algorithm in how messages are sent. Instead of having functions for specific messages, we have two functions *SendMessage* and *SendMessageAndTailMessageBox* (Figure 4.3). In the Bully Algorithm, a lot had to be done when sending each message. In the



*Setup*

---

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *TLC*

CONSTANT  $N$

ASSUME  $N \in \text{Nat}$

$\text{ProcessID} \triangleq 1 \dots N$

VARIABLES  $\text{State}$ ,  $\text{MessageBox}$

$\text{Message} \triangleq \{\text{"PROBE"}, \text{"SELECTED"}\} \times \text{ProcessID}$

Initializes the variables:  $\text{State}$  and  $\text{MessageBox}$

$\text{Init} \triangleq \wedge \text{State} = [p \in \text{ProcessID} \mapsto [$   
 $\quad ID \mapsto p,$   
 $\quad \text{Condition} \mapsto \text{"Active"},$   
 $\quad \text{Leader} \mapsto N,$   
 $\quad \text{Participating} \mapsto \text{FALSE}]$   
 $\wedge \text{MessageBox} = [p \in \text{ProcessID} \mapsto \langle \rangle]$

---

Figure 4.1: The Setup for the Ring Algorithm

*Neighbour*

---

$\text{Neighbour}(p) \triangleq$   
 LET  $\text{AliveProcesses} \triangleq \{q \in \text{ProcessID} : \text{State}[q].\text{Condition} = \text{"Active"}\}$   
 IN  
 IF  $p = \text{MaxAliveID}$  THEN CHOOSE  $q \in \text{AliveProcesses} : \text{TRUE}$  ELSE  
 CHOOSE  $\text{neighbour} \in \text{AliveProcesses} :$   
 $\quad \wedge p < \text{neighbour}$   
 $\quad \wedge \forall q \in \text{AliveProcesses} : q \leq p \vee \text{neighbour} \leq q$

---

Figure 4.2: Neighbour function

*SendingMessages*

---

$\text{SendMessageAndTailMessageBox}(p, q, id, msg) \triangleq$   
 $\quad \text{MessageBox}' = [\text{MessageBox} \text{ EXCEPT } ![q] = \text{Append}(\text{MessageBox}[q], \langle msg, id \rangle),$   
 $\quad \quad \quad ![p] = \text{Tail}(\text{MessageBox}[p])]$

Simple  $\text{SendMessage}$ . Sends message  $msg$  to  $q$  with  $id$   $id$ .

$\text{SendMessage}(q, id, msg) \triangleq$   
 $\quad \text{MessageBox}' = [\text{MessageBox} \text{ EXCEPT } ![q] = \text{Append}(\text{MessageBox}[q], \langle msg, id \rangle)]$

$\text{SendMessageAndEmptyMessageBox}(p, q, id, msg) \triangleq$   
 $\quad \text{MessageBox}' = [\text{MessageBox} \text{ EXCEPT } ![q] = \text{Append}(\text{MessageBox}[q], \langle msg, id \rangle), ![p] = \langle \rangle]$

---

Figure 4.3: Sending Messages

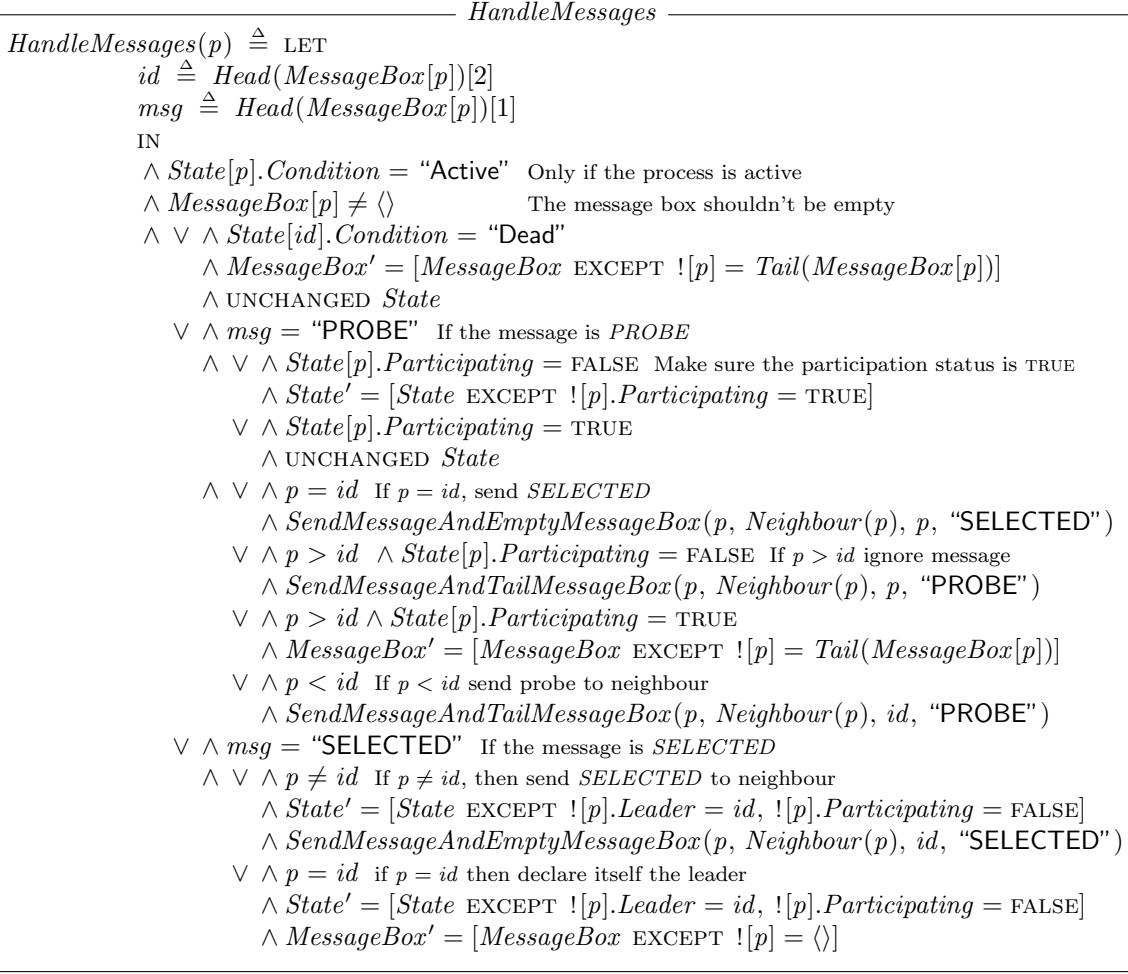


Figure 4.4: HandleMessages

Ring Algorithm, messages are only sent to neighbours, thus requiring less work choosing who to send to. In the Bully Algorithm this work was done in the sender functions, but this work is not needed in the Ring Algorithm.

The *SendMessage* function updates the *MessageBox* of a process with a *Message*. The *SendMessageAndTailMessageBox* is similar, except it also removes the first message in the *MessageBox* of *p*. This is done because the *HandleMessages* function requires both sending messages and deleting in the same state. However, in  $\text{TLA}^+$  you cannot update a variable twice in the same state, thus requiring a function like *SendMessageAndTailMessageBox* to update both message boxes at the same time. *SendMessageAndEmptyMessageBox* is similar to *SendMessageAndTailMessageBox*, but instead of deleting just the first message, it deletes all messages.

## 4.4 HandleMessages

Instead of having separate functions for handling the different messages, all of this is done in the *HandleMessages* function (Figure 4.4).

The function checks if there are any messages for process  $p$ , and then proceeds to check what kind of message the first message is.

### Receiving a *probe* message

In case the message received is *probe*, we check first if process  $p$  is participating. If it is not, we change the *participating* field to be true.

Afterwards, we compare  $p$  with the *id* from the message. If  $p = id$ , then the probe has circled around all processes, and we send the *selected* message to the neighbour. The message box of  $p$  is then emptied. We empty the message box to ensure that there aren't any excess messages that will harm the election.

If  $p > id$ , we deviate from the original pseudocode. The original pseudocode states that: “if  $p > id$ , discard the probe.” However, this leads to cases with infinite loops of *probe* messages. Thus, instead what we do is, if  $p > id$  and  $p$  is not participating in the election, we send a new *probe* message with  $p$  as the *id*, so that it can become the leader. If  $p$  however is participating, we just discard the message.

If  $p < id$  we send the probe to the right neighbour, and delete the message.

### Receiving a *selected* message

In case the msg is *selected* we do not need to check whether or not the process is participating in the election, as they will have participated no matter what, as the *probe* message has been all around the ring. If the  $id \neq p$  then we note *id* to be the leader and set our participation status to be FALSE. We also send the *selected* message to our right neighbour, so that everyone knows who the new leader is. However, if  $p = id$  then  $p$  is the leader, and he is noted as such by himself. There is a slight deviation from the pseudocode here. In the pseudocode it is stated that  $p$  is noted by itself to be the leader, when it first sends the *selected* message to its right neighbour. However, due to the limitations of TLA<sup>+</sup>, I have decided against doing it there, as it would require a check of participation in each equality check with the *id* when it receives the *probe* message. This solution is simply easier to look at. The desired outcome will be the same, i.e. every process having the same leader and not being participating.

## 4.5 Checking The Leader Condition

When we check if the leader is dead in the Ring Algorithm (Figure 4.5), we do it a bit differently than in the Bully Algorithm. Here, the only check we, other than to see if the leader is dead, do is to see if there is only one process remaining. If this is the case, we set that one process as the leader, and its participation status to be FALSE ending the election (once and for all). However, if this is not the case, we send a *probe* message to the right neighbour and set *ps* participation status to TRUE.

## 4.6 Next state relation and killing the leader

KillLeader and the next state relation (Figure 4.6) are identical to the Bully Algorithm.

$$\begin{array}{c}
\text{CheckLeader}(p) \triangleq \begin{array}{l}
\wedge \text{State}[p].\text{Condition} = \text{"Active"} \\
\wedge \text{State}[\text{State}[p].\text{Leader}].\text{Condition} = \text{"Dead"} \\
\wedge \vee \wedge \text{MaxAliveID} = 1 \\
\quad \wedge \text{State}' = [\text{State} \text{ EXCEPT } ![p].\text{Leader} = p, ![p].\text{Participating} = \text{FALSE}] \\
\quad \wedge \text{UNCHANGED } \text{MessageBox} \\
\vee \wedge \text{MaxAliveID} \neq 1 \\
\quad \wedge \text{SendMessage}(\text{Neighbour}(p), p, \text{"PROBE"}) \\
\quad \wedge \text{State}' = [\text{State} \text{ EXCEPT } ![p].\text{Participating} = \text{TRUE}]
\end{array}
\end{array}$$

Figure 4.5: CheckLeader

$$\begin{array}{c}
\text{KillLeader} \triangleq \begin{array}{l}
\wedge \text{State}[\text{MaxAliveID}].\text{Leader} = \text{MaxAliveID} \\
\wedge \text{State}' = [\text{State} \text{ EXCEPT } ![\text{MaxAliveID}].\text{Condition} = \text{"Dead"}] \\
\wedge \text{Cardinality}(\{p \in \text{ProcessID} : \text{State}[p].\text{Condition} = \text{"Active"}\}) \geq 2 \\
\wedge \text{UNCHANGED } \text{MessageBox}
\end{array} \\
\\
\text{Next} \triangleq \begin{array}{l}
\vee \text{KillLeader} \\
\vee \exists p \in 1 \dots \text{MaxAliveID} : \text{HandleMessages}(p) \vee \text{CheckLeader}(p)
\end{array}
\end{array}$$

Figure 4.6: Next state relation and KillLeader

## 4.7 Invariants and Properties

The Ring Algorithm does not introduce any new invariants or properties (Figure 4.7) that the Bully Algorithm does not already have. However, the invariant *Participating* (Figure 3.9) is not present. The Participating invariant does not work in the Ring Algorithm, since it checks whether or not a leader is participating, however, a leader will participate in the election if it receives a *probe* message. However, all of the other invariants and properties are identical to those of the Bully Algorithm.

## 4.8 Temporal Formula

Like in the Bully Algorithm, the Ring Algorithm also uses a temporal formula (Figure 4.8) rather than stating the initial predicate and the next-state relation in the model checker. Like in the Bully Algorithm, this is done to use *strong fairness*.

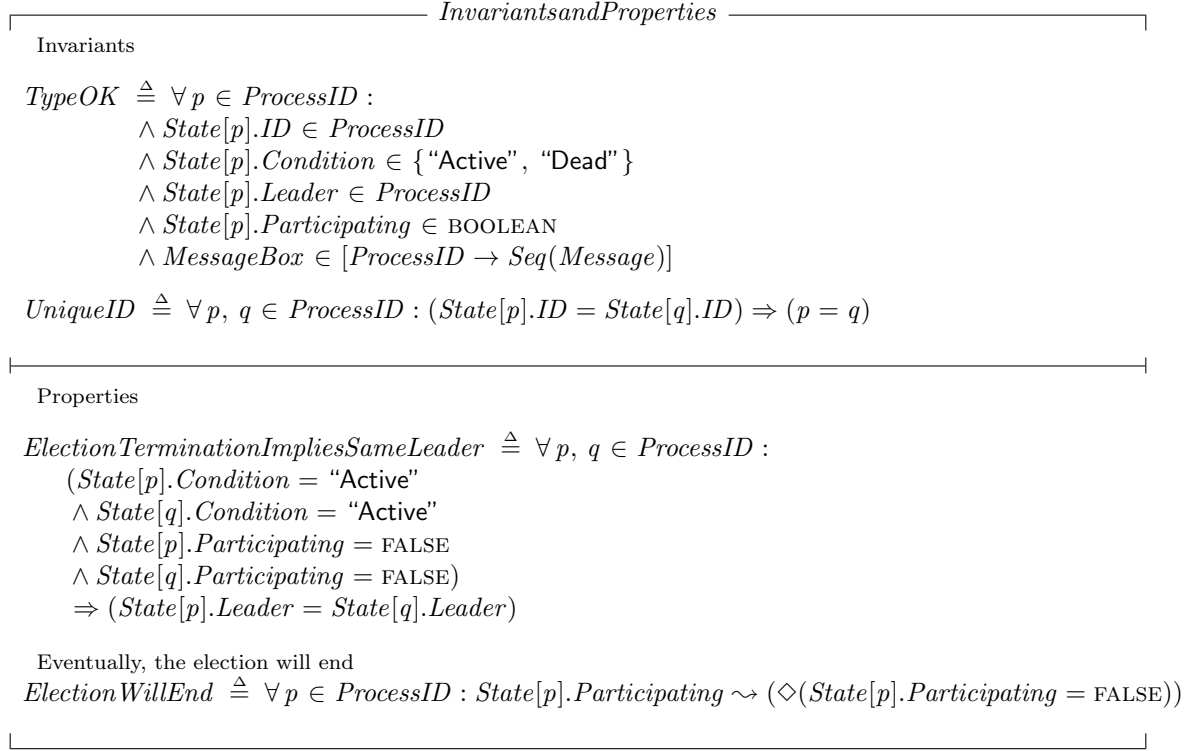


Figure 4.7: Invariants and Properties

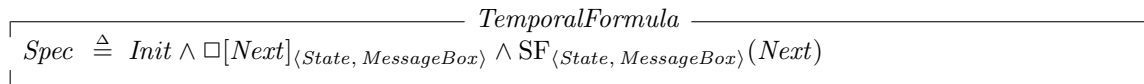


Figure 4.8: Temporal Formula

# Chapter 5

## Evaluation

### 5.1 The Model Checker

Both the Ring Algorithm and the Bully Algorithm have been thoroughly checked with the TLC Model Checker, which comes with the TLA<sup>+</sup> toolbox. The Ring Algorithm has been checked with  $n = 1..10$ , where  $n$  is the number of processes. The Bully Algorithm has been checked with  $n = 1..5$ . Both algorithms pass the checker without any errors, invariant violations or temporal property violations. As it is the case that there have been no violations from both algorithms with  $n = 1..10$  and  $n = 1..5$ , we can assume it to be true that the models and algorithms are verified to be correct.

You can see the state space progress, with the diameter, number of states found and distinct states for the Ring Algorithm in Table 5.1 and for the Bully Algorithm in Table 5.2.

#### **Ring Algorithm**

As you can see in table 5.1, the number of states increase exponentially, but in a way that makes sense. I believe that, although there has been some modifications, the algorithm runs as it should.

#### **Bully Algorithm**

As you can see in table 5.2, the number of states, like the Ring Algorithm rise exponentially, although much faster here. The reason for only going to  $n = 5$ , rather than  $n = 10$  as with the Ring Algorithm is the amount of time it takes for the model checker to be done. With  $n = 5$  in the Bully Algorithm it took 8 minutes and 23 seconds, to contrast, it only took 2 seconds with  $n = 4$ .

### 5.2 Shortcomings

The only shortcoming is that only the leader gets killed, and not any of the other processes, in spite of the assumption that any process including (but not limited to) the leader can die. This has been a deliberate choice, as it is enough to show that when a leader dies, the rest of the processes will find a new leader that they can all agree on together.

$n$	Diameter	States Found	Distinct States
1	1	1	1
2	3	3	3
3	9	17	13
4	19	66	38
5	27	232	101
6	40	773	262
7	53	2478	676
8	69	7710	1760
9	87	23434	4584
10	107	69923	11967

Table 5.1: Number of states found with the model checker in the Ring Algorithm

$n$	Diameter	States Found	Distinct States	Queue Size
1	1	1	1	0
2	3	3	3	0
3	7	50	28	0
4	14	7235	2628	0
5	29	7315267	2090268	0

Table 5.2: Number of states found with the model checker in the Bully Algorithm

### 5.3 Pseudocode to TLA<sup>+</sup>

Instead of converting from what we traditionally think of as pseudocode to TLA<sup>+</sup>, I used a set of instructions for both algorithms. Both of these specified the behavior for when a process  $p$  wakes up or detects a leader to be dead. As we have assumed that processes that die are removed, there has been no need to model the waking up of a process, thus leaving us only with an option to create a function like *CheckLeader*. This is also the reason that the implementation of the Ring Algorithm is so different when the process id is higher than the id in the message. If it was only when the process checked when the leader died, a lot of messages would float around that were of no use.

The approach of using a set of instructions like this, rather than a specific pseudo code has left a lot of ambiguity and things to be interpreted. While this may not seem the case at first glance, a few things aren't specified. Such as in the Ring Algorithm, in the pseudocode it is not specified when each process should toggle their *participation* status back to FALSE. Neither is it clear to see what the purpose of the *participation* variable is in general. The instruction set does not say at any point that something isn't allowed if a process isn't participating. This along with smaller things both from the Ring Algorithm and the Bully Algorithm has left a lot of interpretation work to do. There may be some unseen mistakes that do hinder the algorithm in some way.

### 5.4 Potential Future Changes

In the Bully Algorithm, we get an enormous amount of states found compared to the Ring Algorithm. A potential reason for this is the way the messages are checked. One of the options when it checks messages is to throw out the message if the sender is dead, rather than just

ignoring it. If we instead just checked if the process is dead in each case of messages, it would likely reduce the amount of states drastically. However, there might be some potential issues here, such as deadlocked states where nothing happens because a message will be stuck. But a potential “fix” to this, might result in getting a more realistic number of states. Intuitively, the Ring Algorithm does not need to check if a message sender is dead as it comes from the neighbour, and will count no matter what.

The other reason might be something that I have not yet found, but has been a product of the interpretation work. I might have done too much interpretation, and really changed something did I did not think would change anything, or I did not realize to be a change.



## Chapter 6

# Conclusion

In this project we have implemented the Bully Algorithm and the Ring Algorithm for leader election. The algorithms have been modeled in the TLA<sup>+</sup> specification language and thoroughly checked with the TLC model checker.

Throughout the implementation of the algorithms, the pseudocode has been followed, but also modified to work correctly with TLA<sup>+</sup> and our assumption of the *fail-stop* model. As the pseudocode format was a list of instructions rather than traditional pseudocode for both algorithms, some interpretation has been done on both algorithms, which may have caused minor problems, such as an excess of states in the Bully Algorithm.

We have assumed a *fail-stop* model, the network to be reliable, and that no nodes can come back to life. This means that any other cases have not been considered such as messages getting lost, processes not learning of other processes deaths, or nodes becoming alive again after being dead.

Both the Ring Algorithm and the Bully Algorithm have been proven to be verifiably correct with the TLC model checker. The model checker for the Bully Algorithm may have found an unreasonable amount of states, but this doesn't affect the correctness of the algorithm. I would have liked to have had more time to find the issue of the number states found, but ultimately the goal of the project has been reached, and I am satisfied with the work done.

# Bibliography

- [1] A. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2011.
- [2] H. Wayne. Weak and Strong Fairness, July 2020.
- [3] H. Wayne. Learn TLA+ — Learn TLA+, July 2022.
- [4] Y. Yu, P. Manolios, and L. Lamport. Model checking tla+ specifications. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods*, pages 54–66, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

# Appendix A

## Bully Algorithm TLA<sup>+</sup> model

---

MODULE *BullyAlgorithm*

---

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *TLC*

CONSTANT *N*

ASSUME  $N \in \text{Nat}$

$\text{ProcessID} \triangleq 1 \dots N$

VARIABLES *State*, *MessageBox*

$\text{Message} \triangleq \text{ProcessID} \times \{ \text{"ELECTION"}, \text{"ALIVE"}, \text{"VICTORY"} \}$

$\text{Init} \triangleq \wedge \text{State} = [p \in \text{ProcessID} \mapsto$   
 $\quad [ID \mapsto p,$   
 $\quad \text{Condition} \mapsto \text{"Active"},$   
 $\quad \text{Leader} \mapsto N,$   
 $\quad \text{Participating} \mapsto \text{FALSE}]]$   
 $\wedge \text{MessageBox} = [p \in \text{ProcessID} \mapsto \langle \rangle]$

Finds the maximum *ID* which is not dead

$\text{MaxAliveID} \triangleq \text{CHOOSE } id \in \text{ProcessID} :$   
 $\quad \forall p \in \text{ProcessID} : \text{State}[p].\text{Condition} = \text{"Dead"}$   
 $\quad \vee id \geq \text{State}[p].ID$

$\text{HigherIDs}(p) \triangleq \{ q \in 1 \dots \text{MaxAliveID} : q > p \}$

$\text{SendAliveAndTail}(p, q) \triangleq$   
 $\quad \text{MessageBox}' = [\text{MessageBox} \text{ EXCEPT } ![q] = \text{Append}(\text{MessageBox}[q], \langle p, \text{"ALIVE"} \rangle),$   
 $\quad \quad \quad ![p] = \text{Tail}(\text{MessageBox}[p])]$

$\text{SendVictory}(p) \triangleq$   
 $\quad \text{MessageBox}' = [q \in \text{ProcessID} \mapsto$   
 $\quad \quad \text{IF } q \neq p \wedge \text{State}[q].\text{Condition} = \text{"Active"}$   
 $\quad \quad \text{THEN } \text{Append}(\text{MessageBox}[q], \langle p, \text{"VICTORY"} \rangle)$   
 $\quad \quad \text{ELSE } \text{MessageBox}[q]]$

---

$SendVictoryAndTail(p) \triangleq$   
 $MessageBox' = [q \in ProcessID \mapsto$   
 $IF State[q].Condition = "Active"$   
 $THEN IF q = p THEN Tail(MessageBox[q]) ELSE Append(MessageBox[q], \langle p, "VICTORY" \rangle)$   
 $ELSE MessageBox[q]]$

$SendElection(p) \triangleq$   
 $MessageBox' = [q \in ProcessID \mapsto$   
 $IF q \in HigherIDs(p) \wedge State[q].Condition = "Active"$   
 $THEN Append(MessageBox[q], \langle p, "ELECTION" \rangle)$   
 $ELSE MessageBox[q]]$

$ReceiveAlive(p, q) \triangleq$   $\wedge State[p].Participating = TRUE$  Make sure they are already participating  
 $\wedge p > q$   
 $\wedge State' = [State \text{ EXCEPT } ![p].Participating = FALSE]$  No longer participate

$ReceiveElection(p, q) \triangleq$   $\vee \wedge p = MaxAliveID$   
 $\wedge SendVictoryAndTail(p)$   
 $\wedge State' = [State \text{ EXCEPT } ![p].Participating = FALSE,$   
 $![p].Leader = p]$   
 $\vee \wedge p \neq MaxAliveID$   
 $\wedge \vee \wedge State[p].Participating = TRUE$   
 $\wedge SendAliveAndTail(p, q)$   
 $\wedge UNCHANGED State$   
 $\vee \wedge State[p].Participating = FALSE$   
 $\wedge SendAliveAndTail(p, q)$   
 $\wedge State' = [State \text{ EXCEPT } ![p].Participating = TRUE]$

$ReceiveVictory(p, q) \triangleq State' = [State \text{ EXCEPT } ![p].Leader = q]$  Set the leader of receiver to  $p$  sender

Handle messages  
 $HandleMessages(p) \triangleq$  LET  
 $sender \triangleq Head(MessageBox[p])[1]$   
 $msg \triangleq Head(MessageBox[p])[2]$   
 IN  
 $\wedge State[p].Condition = "Active"$   
 $\wedge MessageBox[p] \neq \langle \rangle$  Message box shouldn't be empty  
 $\wedge \vee \wedge State[sender].Condition = "Dead"$   
 $\wedge MessageBox' = [MessageBox \text{ EXCEPT } ![p] = Tail(MessageBox[p])]$  Remove first message  
 $\wedge UNCHANGED State$   
 $\vee \wedge msg = "VICTORY"$  If the message is *VICTORY*  
 $\wedge ReceiveVictory(p, sender)$   
 $\wedge MessageBox' = [MessageBox \text{ EXCEPT } ![p] = \langle \rangle]$  Remove all messages, so that the election d  
 $\vee \wedge msg = "ELECTION"$  If the message is *ELECTION*  
 $\wedge ReceiveElection(p, sender)$  Tail is done incide *ReceiveElection*  
 $\vee \wedge msg = "ALIVE"$  If the message is *ALIVE*  
 $\wedge ReceiveAlive(p, sender)$  Handle in *ReceiveAlive*  
 $\wedge MessageBox' = [MessageBox \text{ EXCEPT } ![p] = Tail(MessageBox[p])]$  Remove first message

If the leader is found to be dead  
 $CheckLeader(p) \triangleq \wedge State[p].Condition = "Active"$

$ \begin{aligned} & \wedge \text{State}[\text{State}[p].\text{Leader}].\text{Condition} = \text{"Dead"} \\ & \wedge \vee \wedge p = \text{MaxAliveID} \\ & \quad \wedge \text{SendVictory}(p) \\ & \quad \wedge \text{State}' = [\text{State} \text{ EXCEPT } ![p].\text{Participating} = \text{FALSE}, ![p].\text{Leader} = p] \\ & \vee \wedge p \neq \text{MaxAliveID} \\ & \quad \wedge \text{State}[p].\text{Participating} = \text{FALSE} \\ & \quad \wedge \text{SendElection}(p) \\ & \quad \wedge \text{State}' = [\text{State} \text{ EXCEPT } ![p].\text{Participating} = \text{TRUE}] \end{aligned} $
$ \begin{aligned} \text{KillLeader} & \triangleq \\ & \wedge \text{State}[\text{MaxAliveID}].\text{Leader} = \text{MaxAliveID} \\ & \wedge \text{State}' = [\text{State} \text{ EXCEPT } ![\text{MaxAliveID}].\text{Condition} = \text{"Dead"}, ![\text{MaxAliveID}].\text{Participating} = \text{FALSE}] \\ & \wedge \text{Cardinality}(\{p \in \text{ProcessID} : \text{State}[p].\text{Condition} = \text{"Active"}\}) \geq 2 \\ & \wedge \text{UNCHANGED } \text{MessageBox} \end{aligned} $
$ \begin{aligned} \text{Next} & \triangleq \vee \text{KillLeader} \\ & \vee \exists p \in 1 \dots \text{MaxAliveID} : \text{CheckLeader}(p) \vee \text{HandleMessages}(p) \end{aligned} $
$ \text{Spec} \triangleq \text{Init} \wedge \Box[\text{Next}]_{\langle \text{State}, \text{MessageBox} \rangle} \wedge \text{SF}_{\langle \text{State}, \text{MessageBox} \rangle}(\text{Next}) $
<p>Invariants</p> <p>Each variable should be within the type constraints</p> $ \begin{aligned} \text{TypeOK} & \triangleq \forall p \in \text{ProcessID} : \\ & \wedge \text{State}[p].\text{ID} \in \text{ProcessID} \\ & \wedge \text{State}[p].\text{Condition} \in \{\text{"Active"}, \text{"Dead"}\} \\ & \wedge \text{State}[p].\text{Leader} \in \text{ProcessID} \\ & \wedge \text{State}[p].\text{Participating} \in \text{BOOLEAN} \\ & \wedge \text{MessageBox} \in [\text{ProcessID} \rightarrow \text{Seq}(\text{Message})] \end{aligned} $ <p>Every process should have a unique ID</p> $ \text{UniqueID} \triangleq \forall p, q \in \text{ProcessID} : (\text{State}[p].\text{ID} = \text{State}[q].\text{ID}) \Rightarrow (p = q) $ <p>If a process is participating in the election, then it should not be the leader</p> $ \text{Participating} \triangleq \forall p \in \text{ProcessID} : \text{State}[p].\text{Participating} = \text{TRUE} \Rightarrow \text{State}[p].\text{Leader} \neq p $
<p>Properties</p> $ \begin{aligned} \text{ElectionTerminationImpliesSameLeader} & \triangleq \forall p, q \in \text{ProcessID} : \\ & (\text{State}[p].\text{Condition} = \text{"Active"} \\ & \wedge \text{State}[q].\text{Condition} = \text{"Active"} \\ & \wedge \text{State}[p].\text{Participating} = \text{FALSE} \\ & \wedge \text{State}[q].\text{Participating} = \text{FALSE}) \\ & \Rightarrow (\text{State}[p].\text{Leader} = \text{State}[q].\text{Leader}) \end{aligned} $ <p>The highest alive process should become the leader</p> $ \begin{aligned} \text{HighestAliveProcessIsLeader} & \triangleq \forall p \in \text{ProcessID} : \\ & \text{State}[p].\text{Participating} = \text{FALSE} \Rightarrow \\ & (\text{State}[\text{MaxAliveID}].\text{Condition} = \text{"Active"} \Rightarrow \text{State}[p].\text{Leader} = \text{MaxAliveID}) \end{aligned} $ <p>Eventually, the election will end</p> $ \text{ElectionWillEnd} \triangleq \forall p \in \text{ProcessID} : \text{State}[p].\text{Participating} \rightsquigarrow \text{State}[p].\text{Participating} = \text{FALSE} $

[

## Appendix B

# Ring Algorithm TLA<sup>+</sup> model

---

MODULE *RingAlgorithm*

---

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *TLC*

CONSTANT *N*

ASSUME  $N \in \text{Nat}$

$\text{ProcessID} \triangleq 1 \dots N$

VARIABLES *State*, *MessageBox*

$\text{Message} \triangleq \{\text{"PROBE"}, \text{"SELECTED"}\} \times \text{ProcessID}$

Initializes the variables: *State* and *MessageBox*

$\text{Init} \triangleq \wedge \text{State} = [p \in \text{ProcessID} \mapsto [$   
     $\text{ID} \mapsto p,$   
     $\text{Condition} \mapsto \text{"Active"},$   
     $\text{Leader} \mapsto N,$   
     $\text{Participating} \mapsto \text{FALSE}]$   
 $\wedge \text{MessageBox} = [p \in \text{ProcessID} \mapsto \langle \rangle]$

Finds the highest *ID* that is not dead.

$\text{MaxAliveID} \triangleq \text{CHOOSE } id \in \text{ProcessID} :$   
     $\forall p \in \text{ProcessID} :$   
         $\text{State}[p].\text{Condition} = \text{"Dead"} \vee id \geq p$

Get the next alive process

$\text{Neighbour}(p) \triangleq$   
    LET  $\text{AliveProcesses} \triangleq \{q \in \text{ProcessID} : \text{State}[q].\text{Condition} = \text{"Active"}\}$   
    IN  
    IF  $p = \text{MaxAliveID}$  THEN CHOOSE  $q \in \text{AliveProcesses} : \text{TRUE}$  ELSE  
        CHOOSE  $\text{neighbour} \in \text{AliveProcesses} :$   
             $\wedge p < \text{neighbour}$   
             $\wedge \forall q \in \text{AliveProcesses} : q \leq p \vee \text{neighbour} \leq q$

Sends a message and removes the head from the inbox. Sends message *msg* to *q* with *id id*, and removes the first message from *p*.

$$\begin{aligned}
&SendMessageAndTailMessageBox(p, q, id, msg) \triangleq \\
&\quad MessageBox' = [MessageBox \text{ EXCEPT } ![q] = Append(MessageBox[q], \langle msg, id \rangle), \\
&\quad \quad \quad ![p] = Tail(MessageBox[p])]
\end{aligned}$$

Simple *SendMessage*. Sends message *msg* to *q* with *id* *id*.

$$\begin{aligned}
&SendMessage(q, id, msg) \triangleq \\
&\quad MessageBox' = [MessageBox \text{ EXCEPT } ![q] = Append(MessageBox[q], \langle msg, id \rangle)]
\end{aligned}$$

Send message to *q* and empty message box of *p*

$$\begin{aligned}
&SendMessageAndEmptyMessageBox(p, q, id, msg) \triangleq \\
&\quad MessageBox' = [MessageBox \text{ EXCEPT } ![q] = Append(MessageBox[q], \langle msg, id \rangle), ![p] = \langle \rangle]
\end{aligned}$$

Handle Messages

$$\begin{aligned}
&HandleMessages(p) \triangleq \text{LET} \\
&\quad id \triangleq Head(MessageBox[p])[2] \\
&\quad msg \triangleq Head(MessageBox[p])[1] \\
&\text{IN} \\
&\quad \wedge State[p].Condition = \text{"Active"} \quad \text{Only if the process is active} \\
&\quad \wedge MessageBox[p] \neq \langle \rangle \quad \text{The message box shouldn't be empty} \\
&\quad \wedge \vee \wedge State[id].Condition = \text{"Dead"} \\
&\quad \quad \wedge MessageBox' = [MessageBox \text{ EXCEPT } ![p] = Tail(MessageBox[p])] \\
&\quad \quad \wedge \text{UNCHANGED } State \\
&\quad \vee \wedge msg = \text{"PROBE"} \quad \text{If the message is PROBE} \\
&\quad \quad \wedge \vee \wedge State[p].Participating = \text{FALSE} \quad \text{Make sure the participation status is TRUE} \\
&\quad \quad \quad \wedge State' = [State \text{ EXCEPT } ![p].Participating = \text{TRUE}] \\
&\quad \quad \vee \wedge State[p].Participating = \text{TRUE} \\
&\quad \quad \quad \wedge \text{UNCHANGED } State \\
&\quad \vee \wedge \wedge p = id \quad \text{If } p = id, \text{ send SELECTED} \\
&\quad \quad \wedge SendMessageAndEmptyMessageBox(p, Neighbour(p), p, \text{"SELECTED"}) \\
&\quad \quad \vee \wedge p > id \wedge State[p].Participating = \text{FALSE} \quad \text{If } p > id \text{ ignore message} \\
&\quad \quad \quad \wedge SendMessageAndTailMessageBox(p, Neighbour(p), p, \text{"PROBE"}) \\
&\quad \quad \vee \wedge p > id \wedge State[p].Participating = \text{TRUE} \\
&\quad \quad \quad \wedge MessageBox' = [MessageBox \text{ EXCEPT } ![p] = Tail(MessageBox[p])] \\
&\quad \quad \vee \wedge p < id \quad \text{If } p < id \text{ send probe to neighbour} \\
&\quad \quad \quad \wedge SendMessageAndTailMessageBox(p, Neighbour(p), id, \text{"PROBE"}) \\
&\quad \vee \wedge msg = \text{"SELECTED"} \quad \text{If the message is SELECTED} \\
&\quad \quad \wedge \vee \wedge p \neq id \quad \text{If } p \neq id, \text{ then send SELECTED to neighbour} \\
&\quad \quad \quad \wedge State' = [State \text{ EXCEPT } ![p].Leader = id, ![p].Participating = \text{FALSE}] \\
&\quad \quad \quad \wedge SendMessageAndEmptyMessageBox(p, Neighbour(p), id, \text{"SELECTED"}) \\
&\quad \quad \vee \wedge p = id \quad \text{if } p = id \text{ then declare itself the leader} \\
&\quad \quad \quad \wedge State' = [State \text{ EXCEPT } ![p].Leader = id, ![p].Participating = \text{FALSE}] \\
&\quad \quad \quad \wedge MessageBox' = [MessageBox \text{ EXCEPT } ![p] = \langle \rangle]
\end{aligned}$$

If the leader is dead, send a *PROBE* message, if the only alive process is the first one, then make *p* the process, end the election and go into deadlock

$$\begin{aligned}
&CheckLeader(p) \triangleq \wedge State[p].Condition = \text{"Active"} \\
&\quad \wedge State[State[p].Leader].Condition = \text{"Dead"} \\
&\quad \wedge State[p].Participating = \text{FALSE} \\
&\quad \wedge \vee \wedge MaxAliveID = 1 \\
&\quad \quad \wedge State' = [State \text{ EXCEPT } ![p].Leader = p, ![p].Participating = \text{FALSE}]
\end{aligned}$$



$$\begin{aligned}
& \wedge \text{UNCHANGED } \text{MessageBox} \\
& \vee \wedge \text{MaxAliveID} \neq 1 \\
& \wedge \text{SendMessage}(\text{Neighbour}(p), p, \text{"PROBE"}) \\
& \wedge \text{State}' = [\text{State} \text{ EXCEPT } ![p].\text{Participating} = \text{TRUE}]
\end{aligned}$$

Makes sure the algorithm runs correctly

$$\begin{aligned}
\text{KillLeader} & \triangleq \\
& \wedge \text{State}[\text{MaxAliveID}].\text{Leader} = \text{MaxAliveID} \\
& \wedge \text{State}' = [\text{State} \text{ EXCEPT } ![\text{MaxAliveID}].\text{Condition} = \text{"Dead"}] \\
& \wedge \text{Cardinality}(\{p \in \text{ProcessID} : \text{State}[p].\text{Condition} = \text{"Active"}\}) \geq 2 \\
& \wedge \text{UNCHANGED } \text{MessageBox}
\end{aligned}$$

$$\begin{aligned}
\text{Next} & \triangleq \vee \text{KillLeader} \\
& \vee \exists p \in 1 \dots \text{MaxAliveID} : \text{HandleMessages}(p) \vee \text{CheckLeader}(p)
\end{aligned}$$

$$\text{Spec} \triangleq \text{Init} \wedge \Box[\text{Next}]_{\langle \text{State}, \text{MessageBox} \rangle} \wedge \text{SF}_{\langle \text{State}, \text{MessageBox} \rangle}(\text{Next})$$

---

Invariants

$$\begin{aligned}
\text{TypeOK} & \triangleq \forall p \in \text{ProcessID} : \\
& \wedge \text{State}[p].\text{ID} \in \text{ProcessID} \\
& \wedge \text{State}[p].\text{Condition} \in \{\text{"Active"}, \text{"Dead"}\} \\
& \wedge \text{State}[p].\text{Leader} \in \text{ProcessID} \\
& \wedge \text{State}[p].\text{Participating} \in \text{BOOLEAN} \\
& \wedge \text{MessageBox} \in [\text{ProcessID} \rightarrow \text{Seq}(\text{Message})]
\end{aligned}$$

$$\text{UniqueID} \triangleq \forall p, q \in \text{ProcessID} : (\text{State}[p].\text{ID} = \text{State}[q].\text{ID}) \Rightarrow (p = q)$$


---

Properties

$$\begin{aligned}
\text{ElectionTerminationImpliesSameLeader} & \triangleq \forall p, q \in \text{ProcessID} : \\
& (\text{State}[p].\text{Condition} = \text{"Active"} \\
& \wedge \text{State}[q].\text{Condition} = \text{"Active"} \\
& \wedge \text{State}[p].\text{Participating} = \text{FALSE} \\
& \wedge \text{State}[q].\text{Participating} = \text{FALSE}) \\
& \Rightarrow (\text{State}[p].\text{Leader} = \text{State}[q].\text{Leader})
\end{aligned}$$

Eventually, the election will end

$$\text{ElectionWillEnd} \triangleq \forall p \in \text{ProcessID} : \text{State}[p].\text{Participating} \leadsto (\Diamond(\text{State}[p].\text{Participating} = \text{FALSE}))$$

$$\begin{aligned}
\text{HighestAliveProcessIsLeader} & \triangleq \forall p \in \text{ProcessID} : \\
& \text{State}[p].\text{Participating} = \text{FALSE} \Rightarrow \\
& (\text{State}[\text{MaxAliveID}].\text{Condition} = \text{"Active"} \Rightarrow \text{State}[p].\text{Leader} = \text{MaxAliveID})
\end{aligned}$$


---