

**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*

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Overview</b>	<b>3</b>
2.1	Leader Election . . . . .	3
2.2	Bully Algorithm . . . . .	3
2.3	Ring Algorithm . . . . .	4
<b>3</b>	<b>Bully Algorithm</b>	<b>6</b>
3.1	Converting to TLA+ . . . . .	6
3.2	Setup . . . . .	6
3.3	Sending Messages . . . . .	7
3.4	Handling Received Messages . . . . .	9
3.5	Delegating Functions with HandleMessages . . . . .	10
3.6	Checking the leader condition . . . . .	11
3.7	Next predicate and killing the leader . . . . .	11
3.8	Invariants and properties . . . . .	12
3.9	Temporal Formula . . . . .	12
<b>4</b>	<b>Ring Algorithm</b>	<b>14</b>
4.1	Similarities to Bully Algorithm . . . . .	14
4.2	Setup . . . . .	14
4.3	Neighbours and Sending Messages . . . . .	14
4.4	HandleMessages . . . . .	17
4.5	Checking The Leader Condition . . . . .	17
4.6	Next predicate and killing the leader . . . . .	17
4.7	Invariants and Properties . . . . .	18
<b>5</b>	<b>Evaluation</b>	<b>20</b>
5.1	The Model Checker . . . . .	20
5.2	Shortcomings . . . . .	23
5.3	Pseudocode to TLA <sup>+</sup> . . . . .	23
<b>6</b>	<b>Conclusion</b>	<b>24</b>
<b>A</b>	<b>Bully Algorithm Implementation</b>	<b>25</b>
<b>B</b>	<b>Ring Algorithm Implementation</b>	<b>26</b>

# Chapter 1

## Introduction



What is the significance of Leader Election? What is the significance of TLA+? What do you solve?

Distributed Systems are more important now than they have ever been before.  
TLA+ checks...  
Leader election is...

# Chapter 2

## Overview

### 2.1 Leader Election

 TODO: Confidence 3/10

In several algorithms for distributed systems, a process may possess the role of the *leader*. Usually a leader may be required because 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. This is crucial in a distributed system, as if one process has the wrong leader, it can lead to failure in the system.

This report focuses on the Bully Algorithm and the Ring Algorithm for Leader Election, both of which solve the problem. Both the Ring Algorithm and Bully Algorithm are algorithms that work on graphs. The Bully Algorithm's overlay is a complete, undirected graph, and the Ring Algorithm's overlay is a superimposed undirected ring, and a logical complete graph. Both algorithms 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

The Bully Algorithm solves the Leader Election problem using a complete, undirected graph as its overlay. The elected leader will either be the leader with the lowest ID, or the highest ID. In this report, we go with the assumption that it is the highest ID.

In the algorithm, three different types of messages can be sent and received. These messages initiate behavior on the receiving process, depending on what the contents of the message is.

- *election*: announce an election
- *alive*: response to *election*

- *victory*: sent by winner to announce victory

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

The Ring Algorithm, like the Bully Algorithm, solves the problem of leader election. However, 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 create the implementation from this assumption.

The Ring Algorithm has two different types of messages, which dictate the behavior of the receiver:

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

You will notice in the messages, that they have attached an  $id$ . The  $id$  is used to identify who originally sent the message.

Furthermore, the Ring Algorithm uses a *boolean* variable, called *participate*. This variable is required for the algorithm, and indicates whether or not process  $P_i$  participates in the election. 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.

([?])

## 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 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, then it broadcasts an *victory* message to all other processes. Since TLA<sup>+</sup> does not allow for waiting a specified amount of time, I have made a modification to the algorithm. 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.

### 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:

- *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*). Due to the assumptions of the algorithm we assume that each process knows when another process is dead.  $Condition \in \{"Dead", "Active"\}$
- *Leader*: The process should know who it's leader is. The leader may be the process itself.  $Leader \in ProcessID$

```

EXTENDS Naturals, FiniteSets, Sequences

CONSTANT N

ASSUME  $N \in \text{Nat}$ 

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

VARIABLES State, MessageBox

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

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

```

Figure 3.1: Setup for the Bully Algorithm

- *Participating*: The process is either participating in the election or not. This information is only used for the process itself.  $\text{Participating} \in \text{BOOLEAN}$

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 later on to a sequence of tuples. 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

Along with the correct modeling of processes, the most important functionality in the algorithm is the sending, receiving and handling of 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  $\text{TLA}^+$  this is done like so:

```

MessageBox'
= [MessageBox EXCEPT ![q] = Append(<<p, 'ELECTION'>>)]

```

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

In 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 the other, except it also removes the newest message in process  $p$ 's inbox. This is done as to not clutter the inbox with irrelevant messages.



---

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

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

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

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

---

Figure 3.2: The *sender* functions

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

Figure 3.3: Function that returns the highest ID alive

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

The *alive* message is only sent in a case where a process receives an *election* message from a process with a lower id. Thus, no additional work is needed other than process  $q$  should receive an *alive* message from process  $p$ . This job is done by the *SendAliveAndTail* function. Since it as an “AndTail” function it also removes the newest message by setting the message box of  $p$  to be the tail of the message box.

The *SendVictory* function is similar, only it sends the *victory* message to all processes except the sender. We assume a fail-stop model, meaning that the other processes learn when a process dies, allowing us to send only to the alive processes. This function also has a “AndTail” variant, which is identical except it tails  $p$ ’s message box.

Similar to the *victory* message, the *election* message is sent to a select group of processes. This time it's all processes with a higher ID than  $p$ . This is achieved using a helper function *HigherIDs* (see figure 3.3.)



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

Figure 3.6: Delegating Functions

message, and, if its own participation status is *FALSE*, it sets it to be *TRUE* instead. This change is done because it is a redundancy in the algorithm, and will only lead to a number of irrelevant *election* messages. Furthermore, we 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 (Figure 3.5) addresses point 5 by setting the leader to the sender of the *victory* message.

### 3.5 Delegating Functions with HandleMessages

In the *Next* predicate 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 delete 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 delete the message from the inbox of process  $p$ .

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

Figure 3.7: Checking the condition of the leader

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

Figure 3.8: Next predicate and KillLeader

### 3.6 Checking the leader condition

One of the actions a process can take in the *next* state is to check the condition of the leader. In the pseudocode, it is specified that “When a process P recovers from failure, or the failure detector indicates that the current leader has failed...”. Since we assume that the processes do not recover from failure, and know that a process is dead, 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 predicate and killing the leader

The Next action (Figure 3.8) is the action which 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.

In case the decision to kill the leader is not chosen, a process is chosen to either check it's leader or handle it's messages.

## 3.8 Invariants and properties

### Invariants

In  $TLA^+$ , an *invariant* is a statement that must be true on every state of the program[?]. 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*<sup>1</sup> set
- 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<sup>2</sup>.

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.[?]

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, it's leader is the process with the highest id alive.

The liveness property, *ElectionWillEnd* ensures that an election will eventually end.

## 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 says that if an action is always enabled, it eventually happens.

---

<sup>1</sup>See figure 3.1 for the definition of ProcessID

<sup>2</sup>See figure 3.1 for the defintion of Message

---

*Invariants and Properties*

---

**Invariants**

Each variable should be within the type constraints

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

Every process should have a unique ID

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

If a process is participating in the election, then it should not be the leader

$$Participating \triangleq \forall p \in ProcessID : State[p].Participating = TRUE \Rightarrow State[p].Leader \neq p$$

---

**Properties**

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

The highest alive process should become the leader

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

Eventually, the election will end

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


---

Figure 3.9: Invariants and Properties in the Bully Algorithm

---

*Temporal Formula*

---

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


---

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. One similarity which carries over from our implementation of the Bully Algorithm to the Ring Algorithm is the modeling of the graphs. Since the Ring Algorithm is also an algorithm on graphs, we can reuse a lot of the setup. Furthermore, 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 (See 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. However, there is a flaw in this function, that makes it deviate from the original pseudocode. It assumes that only the leaders die, and no non-leaders will die. This is the only deviation from the original specification.

#### 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*.

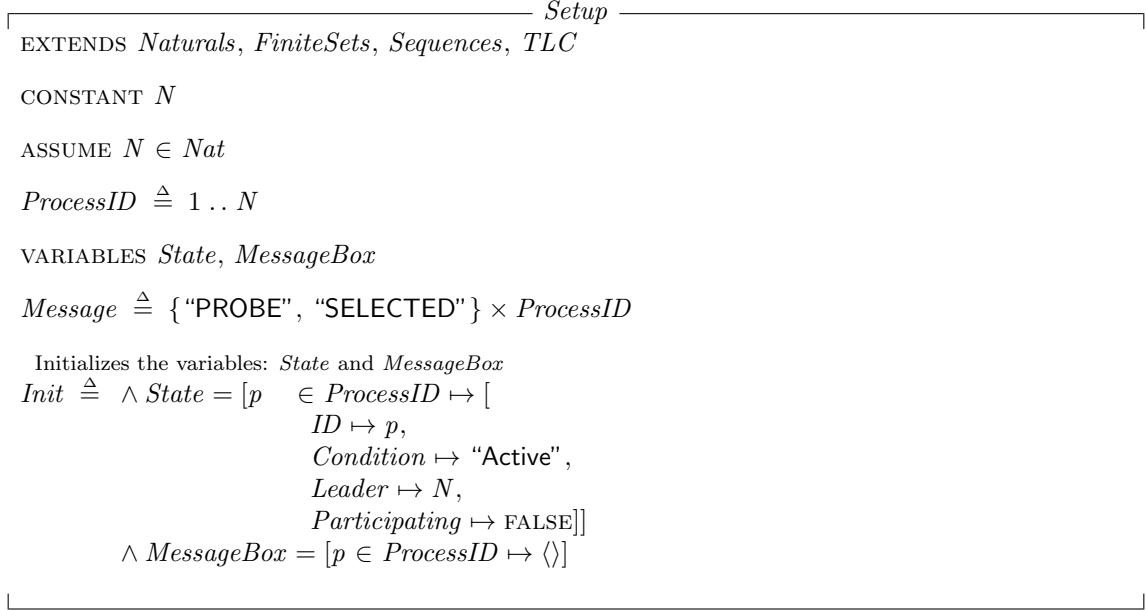


Figure 4.1: The Setup for the Ring Algorithm

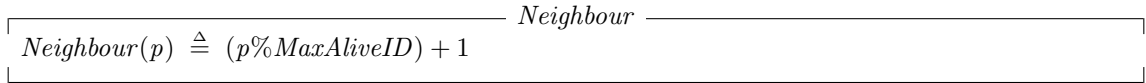


Figure 4.2: Neighbour function

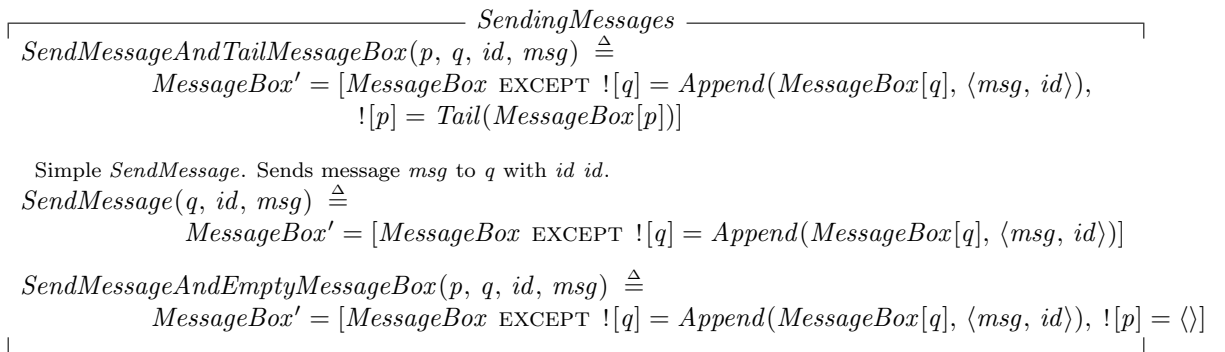


Figure 4.3: Sending Messages



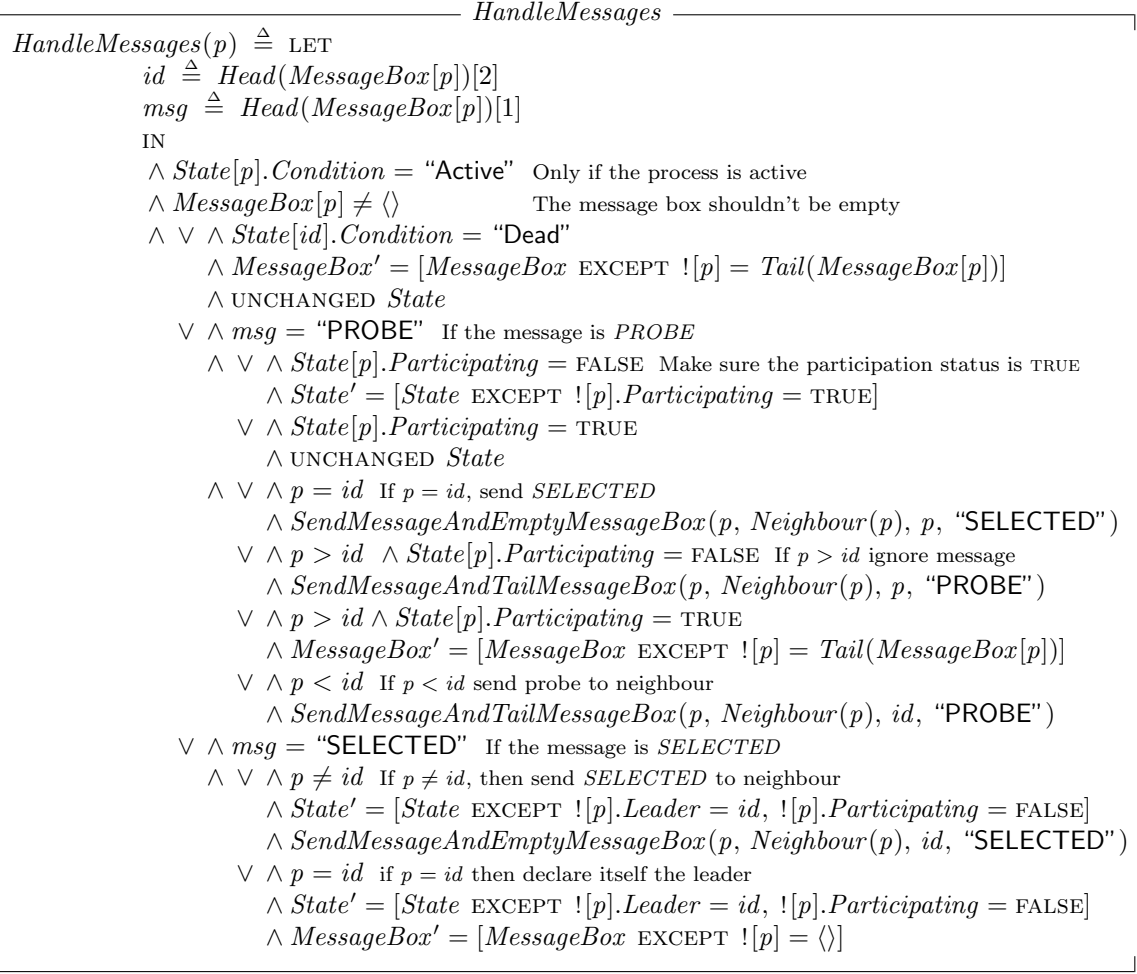


Figure 4.4: HandleMessages

In the Bully Algorithm, a lot had to be done when sending each message. In the 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 (See 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 predicate and killing the leader

KillLeader and the next predicate 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 predicate 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 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*.

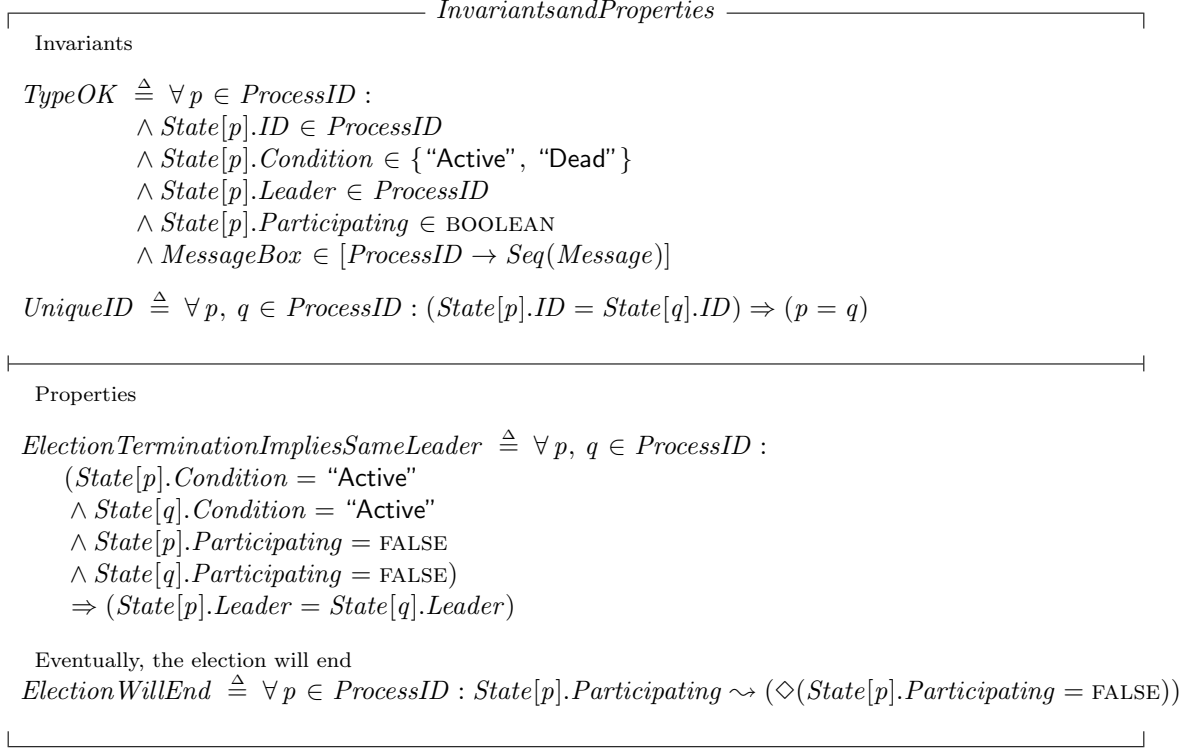


Figure 4.7: Invariants and Properties

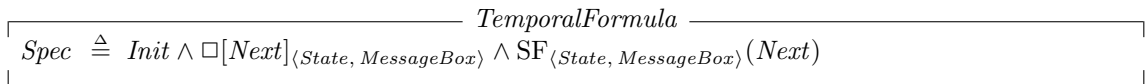


Figure 4.8: Temporal Formula

# 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.

With the current code specification, the temporal property *ElectionWillEnd* (Figure 4.7 and 3.9) is violated. This is not a problem with the *ElectionWillEnd* property, but a problem with the specification. See Figure 5.1 for the values of the variables at the state where the property is violated. As you can see, it looks as if the *selected* message is just about to get to the third process, which would render it the leader. However, this does not happen before the property is violated. You can see a similar result in Figure 5.2, in which the *selected* message never gets to the process which should become the new leader. Instead, the first process is flooded with probe messages, of which only 2 get sent on.

It should be noted that, if we do turn off the *ElectionWillEnd* we get an infinitely running program<sup>1</sup>. However, if we turn on checking for deadlocks, we get states which seemingly work fine, and have attained the goal of electing a leader (See Figure 5.3 for an example state), except for the last process not seeing itself as the leader yet, and still participating in the election. However, this should have been “fixed” in the next state. But, as per how the deadlock detection works in TLC, we cannot generalize this to mean that the entire specification works.

We get similar behavior from checking the model of the Bully Algorithm. The temporal property *ElectionWillEnd* fails. See Figure 5.4 and 5.5. In the figure where  $n = 3$ , we can see that it is almost finished with the election, and should declare itself as the leader in the next state, but the *ElectionWillEnd* property stops this. When  $n = 4$ , we have similar behavior, where only one process is left and needs to declare itself the leader. Note, that this is done in the *CheckLeader* function (Figure 3.7), so it will be done, but the property is violated before it gets the chance. However, as was the case with the Ring Algorithm, if *ElectionWillEnd* is turned off, we get an infinitely running specification.

 Maybe these should not be figures? Or Maybe they should be in the appendix?

---

<sup>1</sup>It might not be running infinitely, but with  $n = 3$  it ran for 1,5 until there was no more space on my hard drive.

```

/\  MessageBox =
<< << <<"PROBE", 3>>,
    <<"PROBE", 3>>,
    <<"PROBE", 3>>,
    <<"PROBE", 3>>,
    <<"PROBE", 3>> >>,
    <<<<"SELECTED", 3>>, <<"PROBE", 3>>, <<"PROBE", 3>>>>,
    <<>>,
    <<>> >>
/\  State =
<< [ID |-> 1, Condition |-> "Active", Leader |-> 3, Participating |-> TRUE],
    [ID |-> 2, Condition |-> "Active", Leader |-> 4, Participating |-> TRUE],
    [ID |-> 3, Condition |-> "Active", Leader |-> 4, Participating |-> TRUE],
    [ID |-> 4, Condition |-> "Dead", Leader |-> 4, Participating |-> FALSE] >>

```

Figure 5.1: Ring Algorithm last state with violated property. N = 4. State num = 16

```

/\  MessageBox = << << <<"SELECTED", 2>>,
    <<"PROBE", 2>>,
    <<"PROBE", 2>>,
    <<"SELECTED", 2>>,
    <<"PROBE", 2>>,
    <<"PROBE", 2>>,
    <<"PROBE", 2>>,
    <<"PROBE", 2>>,
    <<"PROBE", 2>>,
    <<"PROBE", 2>>,
    <<"PROBE", 2>>,
    <<"PROBE", 2>> >>,
    <<>>,
    <<>> >>
/\  State = << [ID |-> 1, Condition |-> "Active", Leader |-> 3, Participating |-> TRUE],
    [ID |-> 2, Condition |-> "Active", Leader |-> 3, Participating |-> TRUE],
    [ID |-> 3, Condition |-> "Dead", Leader |-> 3, Participating |-> FALSE] >>

```

Figure 5.2: Ring Algorithm last state with violated property. N = 3. State num = 19

```

/\  MessageBox = <<<<>>, <<>>, <<>>>>
/\  State = <<
    [ID |-> 1, Condition |-> "Active", Leader |-> 1, Participating |-> FALSE],
    [ID |-> 2, Condition |-> "Dead", Leader |-> 2, Participating |-> FALSE],
    [ID |-> 3, Condition |-> "Dead", Leader |-> 3, Participating |-> FALSE] >>

```

Figure 5.3: Ring algorithm last state with deadlock detection. N = 3. State num = 9

```

/\  MessageBox = <<<<<>>, <<<>>, <<>>>>
/\  State =
<< [ID |-> 1, Condition |-> "Active", Leader |-> 3, Participating |-> TRUE],
    [ID |-> 2, Condition |-> "Dead", Leader |-> 2, Participating |-> FALSE],
    [ID |-> 3, Condition |-> "Dead", Leader |-> 3, Participating |-> FALSE] >>

```

Figure 5.4: Bully Algorithm with last state with violated property. N = 3. State num = 6

```

/\  MessageBox = <<<<<>>, <<<<<1, "ELECTION">>>, <<<3, "VICTORY">>>>, <<<>>, <<>>>>
/\  State =
<< [ID |-> 1, Condition |-> "Active", Leader |-> 2, Participating |-> TRUE],
    [ID |-> 2, Condition |-> "Dead", Leader |-> 2, Participating |-> FALSE],
    [ID |-> 3, Condition |-> "Dead", Leader |-> 3, Participating |-> FALSE],
    [ID |-> 4, Condition |-> "Dead", Leader |-> 4, Participating |-> FALSE] >>

```

Figure 5.5: Bully Algorithm with last state with violated property. N = 4. State num = 9

```

/\  MessageBox = <<<<<>>, <<<<<3, "VICTORY">>>>, <<<>>, <<>>>>
/\  State =
<< [ID |-> 1, Condition |-> "Active", Leader |-> 1, Participating |-> FALSE],
    [ID |-> 2, Condition |-> "Dead", Leader |-> 2, Participating |-> FALSE],
    [ID |-> 3, Condition |-> "Dead", Leader |-> 3, Participating |-> FALSE],
    [ID |-> 4, Condition |-> "Dead", Leader |-> 4, Participating |-> FALSE] >>

```

Figure 5.6: Bully Algorithm last state with deadlock detection. N = 4. State num = 8

## 5.2 Shortcomings

The biggest shortcoming, which has an actual effect on the algorithm is the Neighbour function (Figure 4.2), which does not accurately model a neighbour in a ring structure, as it assumes that only the leader can die. I decided against modeling the function in a more realistic way, as to focus on attempting to find out why the model would not run as expected.

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

Instead of converting from 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* (Figure 3.7.)

However, the approach of using a set of instructions like this, rather than a specific pseudocode 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 set of instructions (to be called pseudocode from here on), it is not specified when each process should toggle their *participation* status. 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. Although it seems to me that the implementations of both algorithms should work, and follow the pseudocode in a way that makes sense, it does not work, and this might be the fault of too much interpretation work. It is specifically important that everything is modelled in TLA<sup>+</sup> as it explores every possibility, thus, if some edge case is not described in the pseudocode, and therefore hasn't been implemented, it might be the fault of the model checker not leading to a successful termination state.



## Chapter 6

# Conclusion

 Missing

In this project we have implemented the Bully Algorithm and the Ring Algorithm for leader election. The algorithms have been implemented in the language  $TLA^+$ , which is a formal specification language used for designing, modelling, documentation and verification of programs.

## Appendix A

# Bully Algorithm Implementation

## Appendix B

# Ring Algorithm Implementation