Verifying the Ricart-Agrawala Algorithm with mCRL2

M.S.C. Spronck[0000−0003−2909−7515]

Eindhoven University of Technology, Eindhoven, The Netherlands
m.s.c.spronck@student.tue.nl

Abstract. We verify the Ricart-Agrawala distributed mutual exclusion algorithm using the model checker mCRL2. We show that the original presentation of the algorithm contains a potential deadlock, which can be alleviated through more extensive use of the semaphores. Additionally, we identify a mutual exclusion violation when sequence numbers are bounded as suggested in the paper, and provide a simple fix.

Keywords: Mutual exclusion · Formal verification · Distributed algorithm.

1 Introduction

The Ricart-Agrawala algorithm for mutual exclusion in a distributed setting [18] has been around for over 40 years. It is frequently included in overviews of algorithms for distributed systems [7,8,13,17,21], alongside Lamport's algorithm [12], on which it is based. The algorithm contains many moving parts: in its original presentation every node that is competing for access to the critical section runs three processes asynchronously. Additionally, messages between nodes are allowed to arrive out of order, further adding to the difficulty in analyzing the algorithm.

Due to this complexity in analyzing the algorithm, a formal proof of its correctness is desirable. Indeed, several formal verifications of the algorithm have been done already [1,14,19,20]. These are on a version of the algorithm that has unbounded sequence numbers. However, in the original paper it is stated that the sequence numbers may be bounded. We verify the algorithm with bounded sequence numbers using the model checker mCRL2 [3]. We consider networks of up to three nodes.

We identify a race condition in the original presentation of the algorithm that results in a potential deadlock. This race condition can be fixed by calling the semaphores more frequently than is done in the pseudocode given in [18]. While we did not find any reference to this deadlock being observed before, it should be noted that most presentations of the algorithm either give pseudocode that places the semaphore calls correctly or make (implicit) assumptions on how the threads can interleave that avoid this deadlock.

Additionally, we find a mutual exclusion violation in the algorithm when bounded sequence numbers are used. This is caused by a node only keeping track of the highest sequence number it has seen in received REQUEST messages, rather than both received and sent REQUESTs. This does not affect correctness when unbounded sequence numbers are used, only when sequence numbers are bounded. Modifying the algorithm to also take sent messages into account removes the violation.

In Section 2, we will introduce the mutual exclusion problem in the setting of distributed systems, as well as the concept of logical clocks. We will go over the results of previous verifications and introduce the mCRL2 toolset. The Ricart-Agrawala algorithm will be presented in Section 3. The process and results of our verification are presented in Section 4. Since quite a number of changes need to be made to be able to verify the algorithm with three nodes, the verifications for two and three nodes are split into Section 4.1 and Section 4.2 respectively. Finally, we give an overview of our final results in Section 5, and mention avenues for future verification.

2 Background and Preliminaries

The Ricart-Agrawala algorithm is a solution to the mutual exclusion problem in a distributed setting. The mutual exclusion problem, also known as "mutex", was first outlined in [5]. The problem is as follows: there are N nodes which all have a part of their execution we will call the "critical section". We wish to ensure that no two nodes are in their critical section simultaneously. We sometimes refer to a node executing its critical section as the node being in *the* critical section rather than *its* critical section. It is important that if a node wishes to enter its critical section, it eventually gets to do so. If this latter condition is ignored, the problem could be solved trivially by never letting any node enter its critical section, which is not an acceptable solution.

When multiple nodes desire access to their critical sections, they will need to communicate with each other to determine when they each get to enter. Therefore, it is important for a mutual exclusion algorithm to consider how nodes can communicate. Many of the proposed solutions to the mutual exclusion problem, for example those presented in [5], [11] and [16], are designed for a shared memory setting, where the different nodes communicate by reading from and writing to shared registers. Far fewer are designed for a distributed setting, where nodes communicate through the sending and receiving of messages. In the distributed setting, nodes cannot get direct insights into each other's current states: messages take time to transmit and as such the information communicated is frequently outdated, and may even arrive out of order. The Ricart-Agrawala algorithm, first presented in [18], aims to solve mutual exclusion for any number of competing nodes in a distributed setting.

The Ricart-Agrawala algorithm is based on Lamport's algorithm presented in [12]. Lamport's algorithm uses the notion of a "logical clock", also presented in [12]. The idea of a logical clock is that every node N_i has a local function C_i that maps actions taken by that node to timestamps. There is also a function C which represents the entire system of local clocks, satisfying the rule that $C(a) = C_i(a)$ if a is an action by N_i. Additionally, the following two conditions must hold:

1. If N_i does action a before action b, then $C_i(a) < C_i(b)$.
2. If N_i sends a message with timestamp $C_i(s)$ and N_j receives that message with timestamp $C_j(r)$ then $C_i(s) < C_j(r)$.

A logical clock can be implemented by having nodes maintain a counter, which increases whenever they take an action. Nodes should add the current value of their counter to every message they send, and upon receiving a message they must increase their counter to a value strictly greater than the included timestamp. This way, nodes synchronize their clocks as much as possible while respecting the above conditions.

Lamport's algorithm requires $3(N-1)$ messages to be sent per instance of a node gaining access to the critical section. The Ricart-Agrawala algorithm lowers this to $2(N-1)$ messages by reducing the amount of information nodes share with each other. One consequence of less information being communicated is that the algorithm does not contain a true logical clock as described by Lamport. To highlight this distinction and in accordance with the terminology used in [18], we will refer to sequence numbers instead of timestamps when discussing the Ricart-Agrawala algorithm. While the original presentation of Ricart-Agrawala does not have a true logical clock, some presentations of the algorithm, including [1], do use a logical clock. The analysis of the algorithm we do is based on Ricart and Agrawala's original presentation. In particular, our reasoning regarding the mutual exclusion violation we discover would not be valid when a logical clock is used.

As stated in the introduction, Ricart-Agrawala has already been formally verified several times. In each instance that we could find, the proofs are of the variant with unbounded sequence numbers. In [19], STeP [15] is used to verify the algorithm for an arbitrary number nodes and the authors prove both that mutual exclusion is satisfied and that if a node wants access to the critical section, it will eventually gain access (starvation freedom). In [1], backwards reachability analysis is done to prove that the mutual exclusion property is satisfied for an arbitrary number of nodes. In [14] both mutual exclusion and starvation freedom are established for a network with two nodes and unbounded sequence numbers. This was done using the axiomatic system of MSVL [6,22]. Finally, [20] uses Coq [2] to prove that the mutual exclusion property is satisfied. Our verification differs from these previous instances by using a model checker, mCRL2, to verify the version of the algorithm that uses bounded sequence numbers.

The mCRL2 toolset [3] allows users to define behavioral models of software using the mCRL2 language [10] and then analyze these models. An mCRL2 model describes a labeled transition system (LTS) using a process algebra similar to ACP. The toolset contains tools for reducing the resulting LTS modulo several equivalences, including bisimulation and divergence-preserving branching bisimulation. Properties of the software defined in modal μ-calculus can automatically be checked on the resulting model. If a property does not hold, a counterexample can be generated in the form of an LTS showing an execution of the system which violates the property. For more information on mCRL2, we refer to [10] as well as the tool's website[1]

3 The Ricart-Agrawala Algorithm

In this section, we present the Ricart-Agrawala algorithm as originally described in [18]. When executing this algorithm, each node in the network uses three threads[2], see Algorithm 1, Algorithm 2 and Algorithm 3 for the pseudocode of the three threads. The

[1] https://www.mcrl2.org

[2] In [18] these are called "processes". We use the word "thread" to distinguish the different components of the algorithm from the mCRL2 processes. We use "node" for similar reasons, in many papers the nodes are also called processes.

threads run asynchronously and communicate using shared memory. Each node also has a semaphore with which its threads can be serialized when needed.

Each node knows its own unique id, me, and how many nodes are part of the network, N. In addition to these two constants, every node has the following registers[3]:

- flag: a Boolean indicating whether this node is currently requesting access to the critical section, similar to the flag in Peterson's famous algorithm [16].
- $\mathit{deferred}[0..N-1]$: a Boolean array, where $\mathit{deferred}[i]$ means that the reply to node i was deferred.
- $\mathit{awaiting}$: an integer representing how many REPLY messages must still be received before this node is allowed access to the critical section.
- chosen: an integer representing the most recent sequence number that was chosen for a REQUEST message from this node.
- $\mathit{highest}$: an integer representing the highest sequence number seen in any received REQUEST message.
- shared: the binary semaphore used by this node to protect its shared memory.

Finally, the second thread uses a local variable defer to track whether the REQUEST it is currently processing should be deferred or not. All Boolean variables are initialized with **false** and all integers with 0.

Of note is that the description of $\mathit{highest}$ we give differs from the description in [18]; there it is described as containing the highest sequence number seen in any REQUEST message "sent or received". However, in the pseudocode presented, both here and in [18], $\mathit{highest}$ is only increased when REQUEST messages are received, not when they are sent. Observe that Algorithm 1 is where REQUESTs are sent, but $\mathit{highest}$ is only increased in Algorithm 2. We change the description to be more accurate to the pseudocode. We are not the first to observe this discrepancy: the description of $\mathit{highest}$ is also changed in [19], and in [17, Section 10.2.2] it is explicitly mentioned that $\mathit{highest}$ is only updated upon receiving REQUESTs and it proven that this does not affect correctness when unbounded sequence numbers are used. However, as we will see in Section 4, when the algorithm is used with bounded sequence numbers a mutual exclusion violation is possible when $\mathit{highest}$ is not updated upon sending a REQUEST. It is therefore possible that the former description was originally intended by Ricart and Agrawala and their pseudocode either contains a typo or the reader was meant to interpret "$\mathit{chosen} \leftarrow \mathit{highest} + 1$" as both incrementing $\mathit{highest}$ and updating chosen.

[3] The register names used here differ slightly from those used in [18], to improve readability.

Algorithm 1 The first thread, *Main*, in the Ricart-Agrawala algorithm. Lines 1 to 9 are referred to as the “entry protocol” and lines 11 to 15 as the “exit protocol”.

```
 1: $\mathbf{P}(shared)$
 2: $flag \leftarrow \mathbf{true}$
 3: $chosen \leftarrow highest + 1$
 4: $\mathbf{V}(shared)$
 5: $awaiting \leftarrow N - 1$
 6: for $0 \leq you < N$ do
 7:     if $you \neq me$ then
 8:         send REQUEST$(chosen, me)$ to $you$
 9: await $awaiting = 0$
10: critical section
11: $flag \leftarrow \mathbf{false}$
12: for $0 \leq you < N$ do
13:     if $deferred[you]$ then
14:         $deferred[you] \leftarrow \mathbf{false}$
15:         send REPLY to $you$
```

Algorithm 2 The second thread, *Receive-Requests*, in the Ricart-Agrawala algorithm.

```
 1: while $\mathbf{true}$ do
 2:     receive REQUEST($k$, $you$)
 3:     $highest \leftarrow \mathbf{max}(highest, k)$
 4:     $\mathbf{P}(shared)$
 5:     $defer \leftarrow (flag \wedge (chosen < k \vee (chosen = k \wedge me < you)))$
 6:     $\mathbf{V}(shared)$
 7:     if $defer$ then
 8:         $deferred[you] \leftarrow \mathbf{true}$
 9:     else
10:         send REPLY to $you$
```

Algorithm 3 The third thread, *Receive-Replies*, in the Ricart-Agrawala algorithm.

```
 1: while $\mathbf{true}$ do
 2:     receive REPLY
 3:     $awaiting \leftarrow awaiting - 1$
```

In addition to the pseudocode, we provide an intuitive explanation of the algorithm. The *Main* thread is awakened when a node wishes to gain access to the critical section, and handles the core of the algorithm. In the entry protocol, this thread sends a REQUEST($chosen, me$) message to every other node in the network, where $chosen$ is a sequence number. To ensure that this sequence number is higher than previously seen sequence numbers, it is set to the highest sequence number seen so far ($highest$) plus one. The *Main* thread then waits until it sees that $awaiting = 0$. This means it has received a REPLY from every node in the network and so has permission from all to enter the critical section, which it does. Once it exits the critical section it starts the exit protocol, in which it sends any REPLY messages that *Receive-Requests* has not sent. Once the exit protocol is complete, the *Main* thread terminates, it will be awakened again when the node once again desires access to the critical section. The two other threads, *Receive-Requests* and *Receive-Replies* are always active to process messages that are received by this node. The *Receive-Requests* thread handles the decision of whether incoming REQUEST messages should be replied to immediately, or if the sending of a REPLY message should be deferred to *Main*'s exit protocol. This choice is made as follows: if this node is not interested in the critical section ($flag = \mathbf{false}$) a REPLY is sent immediately. If this node is interested, then a comparison is made between the sequence number in the received REQUEST and the sequence number of the REQUEST this node sent itself; if the received message has a lower sequence number, the REPLY is sent immediately. If the sequence numbers are equal, the node id is used as a tiebreaker, where the lower id wins. This means that if a node considers its own REQUEST to have a higher priority than the received REQUEST, it will only send the REPLY after it has entered the critical section itself. The *Receive-Requests* thread is also responsible for updating $highest$ when a REQUEST is received. Finally, *Receive-Replies* decrements $awaiting$ every time a REPLY is received.

With this algorithm, there is no bound on the size of sequence numbers; $chosen$ and $highest$ can grow unboundedly large. However, in [18, Section 6.4], Ricart and Agrawala state that the sequence numbers can be stored modulo some $M \geq 2N - 1$. In order to still correctly compare sequence numbers, the following rule is given: if the difference between two sequence numbers is greater than or equal to N, then the smaller of the two numbers should be increased by M before the comparison is done. This means that if $N = 2$ and $M = 3$ and we want to compare 0 and 2, then since the difference between 0 and 2 is exactly N and $0 + 3 > 2$, we say that the sequence number 0 is greater than the sequence number 2. This affects both the $\mathbf{max}$ operation on line 3 and the comparison between $chosen$ and k on line 5 of Algorithm 2. This bounded version of the algorithm is the one we model and analyze. For clarity, in the discussions of specific execution sequences of the algorithm we use $<, \leq, >$ and $\geq$ for normal comparison on natural numbers and $\prec, \preceq, \succ$ and $\succeq$ for the bounded comparison.

For the purposes of modeling the algorithm, we are assuming that there are only two operations on registers: the current value can be read, and a new value can be written. This means that in order to do more complicated operations on registers these need to be divided into multiple smaller steps. For example, to do "$highest \leftarrow \mathbf{max}(highest, k)$" (Line 3 in Algorithm 2) the thread needs to first read $highest$, locally check whether $highest$ or k is the greater number, and then write the result back to $highest$.

4 Verification

We model the algorithm in mCRL2. See Appendix C.1 for the full model with $N = 2$.

We model the threads of a node as separate mCRL2 processes. Because mCRL2 has interleaving semantics, this means the model allows the threads to execute asynchronously. Each shared memory register is also modeled as its own mCRL2 processes. By communicating with the shared memory processes, the thread processes can read and update the stored values. The semaphores are also modeled explicitly through the use of mCRL2 processes: a semaphore process has two operations $lock$ and $unlock$, after it has done a $lock$ it has to do an $unlock$ before the next $lock$ is allowed. We use communication with the semaphore processes to ensure the threads behave as specified in the pseudocode. In general, our model sticks as close as possible to the pseudocode.

The sending and receiving of messages is modeled through channels. The mCRL2 toolset only has primitives for synchronous communication, but by modeling channels as mCRL2 processes we can mimic asynchronous communication. To allow for as much flexibility as possible, we have modeled directed channels between every pair of nodes. Since [18] specifies that messages can arrive out of order, we have modeled the channels as bags of messages, to which new messages can be added and from which arbitrary messages can be removed. We do not allow messages to be dropped or altered in transit, as these faults were not considered in [18].

We check three properties of the algorithm: mutual exclusion, deadlock freedom and starvation freedom. These are the properties claimed to hold for this algorithm [18]. We add two actions to the model to enable us to express these properties in modal μ-calculus: the $crit(i)$ action represents that node i is in its critical section; the $noncrit(i)$ action represents that node i leaves the part of its code in which it does not need the critical section, i.e. that node i now wants access to the critical section. The $crit$ action is inserted at the point where the critical section is accessed, as shown in Algorithm 1, the $noncrit$ action is added to the *Main* thread before the entry protocol is started. We also add the function $valid\_id$ to the model, which checks if a natural number is a valid node id. This is done to avoid summation over the infinite set of natural numbers.

Property 1. **Mutual exclusion** is the primary property of any mutual exclusion algorithm: that no two nodes are in their critical section simultaneously. We express that at no point two different threads can do their $crit$ action simultaneously in modal μ-calculus as follows:

```
[true*] forall i, j: Nat.
    val(valid_id(i) && valid_id(j) && i != j) =>
        !(<crit(i)>true && <crit(j)>true)
```

Property 2. **Deadlock freedom** means that at any point in time, it is possible for at least one requesting node to reach its critical section. This is a stronger property than there not being a deadlock in the model at all: if there exists a cycle where nodes can still take actions but these actions will never let them enter their critical section, this is not acceptable. Since our model only contains the algorithm for competing for the critical section and no other behavior, we can interpret this property as: at any point in time, it is possible to reach a state from which at least one node can do its $crit$ action. We express this as:

```
[true*]<true*><exists i: Nat. crit(i)>true
```

Property 3. **Starvation freedom** is a stronger form of deadlock freedom, where instead of stating that it has to be *possible* for *some* node to reach its critical section, we state that if a node wants access to its critical section, then *this node* will *definitely* eventually gain access. This means that after a node does the $noncrit$ action, it can only finitely often do a non-$crit$ action before it is forced to do a $crit$ action. In modal μ-calculus:

```
[true*] forall i: Nat. val(valid_id(i)) =>
    [noncrit(i)] mu X. [!crit(i)]X
```

Verification of these properties is done using the 202206.1 version of the mCRL2 toolset. The compuser used is an HP ZBOOK Studio G4 with an Intel Core i7-7700HQ CPU running Windows 10.

4.1 Verification for Two Nodes

We first verify the algorithm for two nodes. As we will see, many issues in the algorithm can already be observed even with only two nodes.

Verifying deadlock freedom. We start with deadlock freedom, Property 2. We analyze this property first because the presence of a deadlock could impact our analysis of other properties as well.

Using mCRL2, we get that the property does not hold on our model. The counterexample generated is shown in Figure 1.

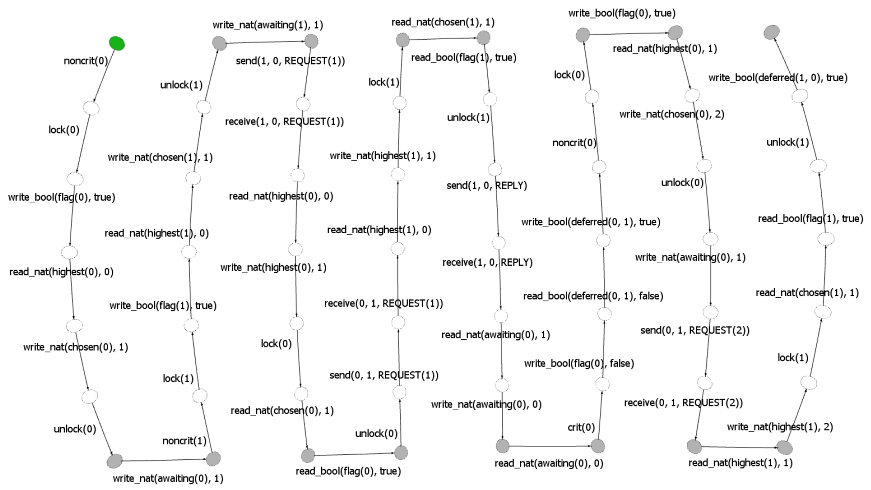

Fig. 1. The counterexample generated by mCRL2 for deadlock freedom on the model of the original Ricart-Agrawala algorithm with 2 competing nodes.

This counterexample can be understood as the following sequence of events. For clarity we write *flag*$[0]$ to indicate the *flag* register belonging to node N_0 and use similar notation for the other registers. For the $deferred$ array, if $deferred[0,1] = \mathbf{true}$ then this means that node N_0 is deferring the reply to node N_1. While in the mCRL2 model we only included the sequence number in REQUEST messages, in the following example we use the notation from the pseudocode: REQUEST(i, j) means a REQUEST with sequence number i from node N_j.

1. N_0 starts the competition for the critical section. The *Main* thread claims the semaphore, sets *flag*$[0]$ to $\mathbf{true}$ and $chosen[0]$ to 1 because $highest[0] = 0$. It then releases the semaphore and sets $awaiting[0]$ to 1. This is lines 1-5 in Algorithm 1.
2. N_1 does the same, also ending up with *flag*$[1] = \mathbf{true}$ and $chosen[1] = 1$.
3. N_1 sends its REQUEST(1, 1) to N_0.
4. The *Receive-Requests* thread of N_0 receives the REQUEST(1, 1). Since $1 \succ 0$, it sets $highest[0]$ to 1. Then it claims the semaphore and starts checking if it needs to send the REPLY now or defer it. Since *flag*$[0] = \mathbf{true}$, $chosen[0] = 1$ and $0 < 1$ (the sequence numbers are equal but N_0 has a lower id), N_0 decides it should defer the reply. It releases the semaphore. This is lines 2-6 of Algorithm 2. Note that the semaphore is released before $deferred[0,1]$ is set.
5. N_0 sends its REQUEST(1, 0) to N_1.
6. The *Receive-Requests* thread of N_1 receives the REQUEST(1, 0). Just like N_0 before it, N_1 updates its $highest[1]$ to be 1. It then claims the semaphore and decides whether to reply immediately. Since $chosen[1] = 1$ and $0 < 1$, N_1 sends the response immediately because N_0 has a lower id and the sequence numbers in their requests are equal. It releases the semaphore and sends the REPLY. This is lines 2-6 and 10 of Algorithm 2.
7. N_0 receives the REPLY, which means that it can now set $awaiting[0]$ to 0. It can enter the critical section. Upon leaving the critical section, it starts the exit protocol. It sets *flag*$[0]$ to $\mathbf{false}$ and then starts to check if it has any deferred REPLY messages to send. When it reads $deferred[0,1]$ it sees $\mathbf{false}$ because the *Receive-Requests* thread has not set it to $\mathbf{true}$ yet. It does not send a REPLY. This is lines 2-3 of Algorithm 3 and lines 9-13 of Algorithm 1. Note that there are no calls to the semaphore in this part of the algorithm.
8. At this point, a deadlock is ensured even if the counterexample in Figure 1 has a few more steps. Node N_0 will never send the REPLY to N_1 anymore: the *Receive-Requests* thread considers the REPLY deferred so will not send it, and in order for the *Main* thread to send the REPLY, N_0 will need to get back to its exit protocol. But in order to reach its exit protocol, N_0 should first regain access to the critical section. Not only can we not take it for granted that N_0 will always try to enter the critical section again, even if it does it will end up sending REQUEST(2, 0) to N_1 since $highest[0] = 1$. When it receives this REQUEST, N_1 will defer sending a REPLY because its own request had a lower sequence number, meaning it will not send the REPLY until it has accessed the critical section itself. So for N_0 to send a REPLY to N_1, N_1 needs to REPLY to N_0 first, but N_1 will not REPLY to N_0 until it has received its own REPLY. We have reached a deadlock.

This counterexample is not unique to the bounded version of the algorithm; the same counterexample is also valid for the unbounded variant. As stated in Section 2, there have already been formal verifications of the unbounded Ricart-Agrawala algorithm that established deadlock freedom does hold. This discrepancy in results can be explained by more extensive use of the semaphores in the variants of the algorithm that have previously been verified. In for example [19], the two receiving threads are given as being entirely protected by the semaphore, and the semaphore is called more frequently in *Main* as well. We adopt this placement of the semaphore calls to get a slightly different version of the algorithm: see Algorithm 4 for the modified *Main* thread pseudocode. The other two algorithms are also modified such that $\mathbf{P}(\mathit{shared})$ is called immediately after receiving the message and that $\mathbf{V}(\mathit{shared})$ is done only after all processing of the message has been done, and no other calls to the semaphores are used. The modified pseudocode for these threads can be seen in Appendix A. We do not adopt the other addition in [19], which is that the two receiving threads are both split into $N - 1$ threads, one for handling messages from each other node. This added parallelism increases the size of the state space when dealing with more than 2 nodes, and since the extra semaphore calls prevent these threads from interfering with each other the splitting of the threads should not result in any new property violations that we could miss by analyzing the version with just three threads per node.

Algorithm 4 The first thread, *Main*, in the Ricart-Agrawala algorithm with additional semaphore calls.

```
 1: $\mathbf{P}(\mathit{shared})$
 2: $\mathit{flag} \leftarrow \mathbf{true}$
 3: $\mathit{chosen} \leftarrow \mathit{highest} + 1$
 4: $\mathit{awaiting} \leftarrow N - 1$
 5: $\mathbf{V}(\mathit{shared})$
 6: for $0 \leq \mathit{you} < N$ do
 7:     if $\mathit{you} \neq \mathit{me}$ then
 8:         send REQUEST$(\mathit{chosen}, \mathit{me})$ to $\mathit{you}$
 9: await $\mathit{awaiting} = 0$
10: critical section
11: $\mathit{flag} \leftarrow$ false
12: for $0 \leq \mathit{you} < N$ do
13:     $\mathbf{P}(\mathit{shared})$
14:     if $\mathit{deferred}[\mathit{you}]$ then
15:         $\mathit{deferred}[\mathit{you}] \leftarrow$ false
16:         send REPLY to $\mathit{you}$
17:     $\mathbf{V}(\mathit{shared})$
```

When we model this version of the algorithm in mCRL2 it reports that for $N = 2$, deadlock freedom indeed holds.

Verifying mutual exclusion. On the new model, with the additional semaphore calls, we can now check mutual exclusion. We get a counterexample, see Figure 2.

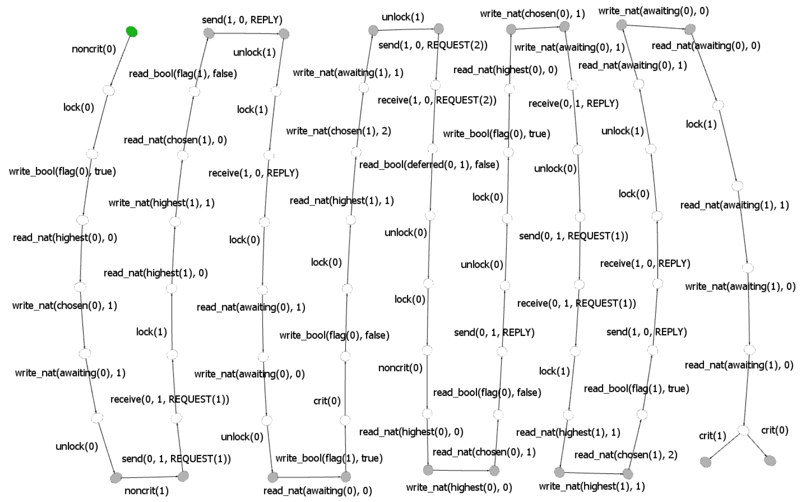

Fig. 2. The counterexample generated by mCRL2 for mutual exclusion on the model of the Ricart-Agrawala algorithm with additional semaphore calls and 2 competing nodes.

We describe this counterexample in more natural language, similarly to the deadlock freedom counterexample above:

1. N_0 starts the competition, it sets its *flag* to $\mathbf{true}$ and sets $chosen[0]$ to 1 since $highest[0] = 0$. Finally, it sets $awaiting[0]$ to 1 and sends the REQUEST(1, 0) message to N_1.
2. N_1 receives the REQUEST(1, 0). Since 1 is greater than $highest[1] = 0$, it updates $highest[1]$ to be 1. Since N_1 is not competing for the critical section ($flag[1] = \mathbf{false}$), it sends the REPLY immediately.
3. N_0 receives the REPLY and therefore sets $awaiting[0]$ to 0. It can then enter the critical section. In the exit protocol, it sets $flag[0]$ to $\mathbf{false}$ again and since no REPLY messages were deferred, it can complete the exit protocol.
4. While N_0 was doing its exit protocol, N_1 starts the competition for the critical section. It sets $flag[1]$ to $\mathbf{true}$ and $chosen[1]$ to 2 because $highest[1] = 1$. It sets $awaiting[1]$ to 1 and sends REQUEST(2, 1).
5. When N_0 receives REQUEST(2, 1), the first thing it needs to do is setting $highest[0]$ to the maximum of 2 and its old value. Since $highest[0]$ has not been updated yet, we still have $highest[0] = 0$. If we were using unbounded sequence numbers, then $\mathbf{max}(0, 2) = 2$ as usual. However, since $2 - 0 = 2 \geq N$, we add $M = 3$ to the smallest of the two numbers when we do the comparison, so $\mathbf{max}(0, 2) = 0$, therefore $highest[0]$ stays 0 even after REQUEST(2, 1) has been received. Since N_0 is not currently competing for the critical section, it sends the REPLY immediately.
6. N_0 starts competing for the critical section again, it sets $flag[0]$ to $\mathbf{true}$ and, since $highest[0] = 0$, it sets $chosen[0]$ to 1. It sets $awaiting[0]$ to 1 and sends REQUEST(1, 0) to N_1.

7. When N_1 receives REQUEST(1, 0), it has $\mathit{highest}[1] = 1$ so the value of $\mathit{highest}[1]$ does not change. N_1 is competing for the critical section, but since $1 \prec 2$, it considers the REQUEST(1, 0) from N_0 to have higher priority than its own REQUEST(2, 1), so it sends the REPLY immediately.
8. At this point, both N_0 and N_1 can receive their respective REPLY messages, decrement their $\mathit{awaiting}$ and enter the critical section, violating mutual exclusion.

As mentioned in point 5, this counterexample relies on the sequence numbers being bounded. This explains why this counterexample did not come up for any of the previous verifications of the Ricart-Agrawala algorithm: this problem is exclusive to the bounded variant. We analyze how this problem can occur. Suppose we have two nodes N_X and N_Y with $\mathit{chosen}[X] = c_X$ and $\mathit{highest}[Y] = h_Y$. If N_X makes a REQUEST(c_X, X) with sequence number c_X such that $h_Y \succeq c_X$, then N_Y can process the REQUEST without changing $\mathit{highest}[Y]$, and if it is not currently competing for the critical section itself it will REPLY immediately. When it later makes its own REQUEST($h_Y + 1$, Y) with sequence number $h_Y + 1$ such that $h_Y + 1 \prec c_X$ (or $h_Y + 1 = c_X$ if $Y < X$) then N_X will give the REQUEST from N_Y priority over its own REQUEST and also send an immediate REPLY. Of course, when the normal ordering on natural numbers is used, there is no assignment for c_X, h_Y such that $h_Y \geq c_X$ and $h_Y + 1 \leq c_X$. With the bounded comparisons however, this is very much possible: if $c_X - h_Y \geq N$ then when comparing c_X and h_Y we will actually compare c_X with $h_Y + M$, and since both c_X and h_Y are natural numbers less than M, we will get $h_Y \succeq c_X$. If we also have that $c_X - (h_Y + 1) = c_X - h_Y - 1 < N$ then when comparing c_X and $h_Y + 1$ we will not add M to $h_Y + 1$, so as long as $c_X > h_Y + 1$ we also have $c_X \succ h_Y + 1$. We can satisfy both $c_X - h_Y \geq N$ and $c_X - h_Y - 1 < N$ exactly when $c_X = h_Y + N$. In other words, this mutual exclusion violation occurs when one node's chosen value and another node's $\mathit{highest}$ value differ by exactly N.

At this point, it is worthwhile to reconsider the discrepancy between what [18] claims $\mathit{highest}$ represents, namely the highest sequence number seen in any received or sent REQUESTs, and what it actually represents in the algorithm: the highest sequence number seen in any received REQUESTs only. Note that in this algorithm, if a node N_i has requested access to the critical section with sequence number s it will no longer send REPLYs to REQUESTs with higher sequence numbers. This means that any node that sends a REQUEST with sequence number greater than s will not gain access to the critical section and get to send another REQUEST until N_i has had access. Therefore once N_i has sent its REQUEST with sequence number s, all nodes will see at most $N - 1$ REQUESTs with a sequence number higher than s until N_i has gained access. Since every REQUEST has a sequence number at most one greater than previously sent REQUESTs, this means the maximum sequence number a node can choose before N_i gains access is $s + N - 1$. Therefore the only way for c_X to be N greater than h_Y is for N_Y to not have updated its $\mathit{highest}$ when it sent out its own REQUEST. If $s = h_Y + 1$, then there is just enough room to reach $c_X = s + N - 1 = h_Y + N$. This means that if N_Y increases its own $\mathit{highest}[Y]$ when it sends out its REQUEST with $\mathit{chosen}[Y] = h_Y + 1$, then the difference between c_X and h_Y is capped at $N - 1$: at most one increment for every node other than N_X and N_Y, and then finally N_X's own increase of the highest sequence number in use when it picks its own sequence number for the competition. We can avoid this mutual exclusion violation if we implement updates to $highest$ as described in the textual description of the Ricart-Agrawala algorithm in [18], rather than the pseudocode given.

Indeed, if we modify the algorithm such that right after "$chosen \leftarrow highest+1$" we do the extra step "$highest \leftarrow highest + 1$", then mCRL2 reports that mutual exclusion is now satisfied. We adopt this change in our further verification. The pseudocode for this final version of the algorithm can be found in Appendix A.

There is an alternative solution to the mutual exclusion violation. Instead of trying to avoid the difference of N, we could accept it and instead require that we only add M to the smaller number when doing a comparison between two numbers with a difference greater than N, rather than at least N. In that case, we will also need to increase the bound from $M \geq N - 1$ to $M \geq N + 1$. We have verified with mCRL2 that for $N = 2$, this approach also works. However, we prefer the solution where $highest$ is increased when sending a REQUEST since a smaller bound results in a smaller state space to analyze, and in real computers it could be the difference between needing an additional bit to store the sequence numbers or not. We therefore stick with adding the extra increase of $highest$ for further verification.

Verifying starvation freedom. We verify starvation freedom on the variant that adds the extra increase of $highest$, rather than the variant that uses $M \geq 2N + 1$. If we place the *noncrit* action before the node sets its *flag* to $\mathbf{true}$, as was described at the start of Section 4, we get a violation of the starvation freedom property. The counterexample shows that this is a rather trivial violation: if a node N_i decides it wants access to the critical section but never sets its *flag* to $\mathbf{true}$, then its *Receiving-Requests* thread will always send REPLY messages to the other node(s) immediately, so there exists an infinite loop in the model where N_i wants access to the critical section but never gets it.

It is not necessarily realistic that a node would never get to set its *flag* to $\mathbf{true}$, so we want to see if starvation freedom is satisfied if we avoid this case. This could for example be done by using a starvation freedom formula that incorporates a form of fairness or justness [9], but in the case of this model it is enough to make the $noncrit$ action simultaneous with the action of setting the *flag* to $\mathbf{true}$. The $noncrit$ action then represents the node making it known it wants access to the critical section, rather than merely deciding it. Since the model does not contain any other loops where one node never gets to take an action, we do not need to modify the starvation freedom formula.

After moving the $noncrit$ action, starvation freedom holds. How the extra increase of $highest$ and the moved $noncrit$ action affect the model is described in Appendix C.2.

4.2 Verifying for Three Nodes

Since we use model checking for our verification, we cannot check the properties for an arbitrary number of nodes. Still, to get more confidence that our fixed algorithm also works for more than two nodes we check it with three nodes as well. The final model we ended up with in Section 4.1 can easily be extended to three nodes, in fact it already contains registers for the third node. However, without taking steps to reduce the state space the resulting model is too big to be analyzed on our computer. The model in Appendix C.2 consists of 6 799 states and 14 231 transitions. At this size we can check properties in seconds and generate counterexamples in just a few minutes. If we add the third node and its channels without further changes, the resulting model has 47 685 971 states and 178 546 668 transitions. On our computer this is too big to be able to do property checking or reductions.

In order to reduce the state space, we change the sending of the REQUEST messages from sending each message individually to a single broadcast that sends the message to all other nodes simultaneously, see Appendix C.3 for how this is done. This reduces the number of actions it takes to run through the algorithm. Receiving the message from the channel is still a single action that the other nodes do separately, which is why we believe no property violations have been removed that are present in the original model, even though the new one deviates slightly from the pseudocode. In [18, Section 6.2] it is mentioned that this initial REQUEST message can be done as a broadcast. We get a model with 34 670 731 states and 122 337 318 transitions, this is still too big for us to verify, but we can now apply reductions. By hiding all actions except $crit$ and $noncrit$ and reducing modulo divergence-preserving branching bisimulation, we obtain a model with just 1 324 states and 4 113 transitions. This reduced model is much smaller and consequently is missing a lot of information. However, we can still check the three properties we are concerned with and get meaningful results. This is because we have not hidden the actions that are referenced in the μ-calculus formulae and divergence-preserving branching bisimulation preserves both divergences and the relative ordering of non-hidden actions. In our formulae, we are only checking for how the non-hidden actions can be reached and how they are ordered with respect to each other, we do not care for the exact number of steps between the actions. The aspects we care about are preserved in our reduced model, so the verification is meaningful.

On this reduced model mutual exclusion, starvation freedom and deadlock freedom all hold.

Out of curiosity, we also analyzed the model with the extra increase of $highest$ removed again, to see if there were any additional effects beyond the presence of a mutual exclusion violation. The resulting model had 39 127 901 states and 152 012 135 transitions, which was once again past the boundary of what our computer could work with. We further reduced the size of the state space through a number of modifications, described in detail in Appendix C.4. The resulting model has 27 903 701 states and 106 078 644 transitions, small enough for us to reduce modulo divergence-preserving branching bisimulation. We find that as expected, deadlock freedom still holds but there now is a mutual exclusion violation. Since we hid most actions and reduced the model, we can no longer get a useful counterexample. However, based on our counterexample in Section 4.1 and the reasoning given there, we can come up with a counterexample ourselves. See Appendix B for such a counterexample, which may help one understand why the additional increase of $highest$ is necessary.

5 Conclusion & Future Work

We have modeled the Ricart-Agrawala algorithm with bounded sequence numbers in mCRL2 and checked mutual exclusion, deadlock freedom and starvation freedom for two and three competing nodes. We demonstrated that with the semaphore calls placed as shown in the original pseudocode presented in [18], the algorithm is not deadlock free. This counterexample is valid for both the version with bounded sequence numbers and the version with unbounded sequence numbers. After placing the semaphore calls as shown in [19], deadlock freedom was shown to hold.

We also showed that when using bounded sequence numbers, a mutual exclusion violation can occur that is not present when using unbounded sequence numbers. This mutex violation can be fixed by increasing $highest$ whenever a node chooses a sequence number for its own REQUEST message. In other words, by ensuring that $highest$ stores the highest sequence number encountered in any sent or received REQUEST message, rather than only those in received messages. With this change implemented, mutual exclusion, deadlock freedom and starvation freedom hold.

The Ricart-Agrawala algorithm was presented in 1981 and has been frequently discussed in literature on distributed algorithms. However, to our knowledge neither property violation demonstrated in this paper has been published before. Most presentations of the Ricart-Agrawala algorithm we could find do not have the deadlock, but the mutual exclusion violation is still present and seems to have flown under the radar until now. This demonstrates the importance of formal verification as a technique for establishing the correctness of algorithms: it forces us to make assumptions explicit and so avoids mistakes through interpretations that are too strong; if we, for example, assume that *Receive-Requests* processes REQUESTs without interleaving with the *Main* thread the deadlock is absent. However, this assumption is not given in [18] and seems unlikely to have been intended as there are semaphore calls placed within *Receive-Requests*. Additionally, formal verification lets us identify edge cases we could easily miss in more informal reasoning, in this case that when the difference between one node's $chosen$ value and another node's $highest$ value is exactly N, a mutual exclusion violation is possible in the bounded version of the algorithm.

We have shown that the version of the Ricart-Agrawala algorithm given in Appendix A satisfies mutual exclusion, deadlock freedom and starvation freedom for two and three nodes. But there are still open questions, for example how the bound on the number of times a node is overtaken is affected by the use of bounded sequence numbers. It would also be interesting to use verification techniques that can handle unbounded variables to verify the version of Ricart-Agrawala with bounded sequence numbers for an arbitrary number of nodes. Finally, there are many variants on the Ricart-Agrawala algorithm that could be checked using the same techniques demonstrated here. In particular, in [18] a number of variants are given, including how to use the algorithm to solve the Readers-Writers problem, how to let nodes leave and enter the network, and how to handle node failures. A further optimization of the Ricart-Agrawala algorithm is presented in [4], which is a prime candidate for further verification.

References

1. Abdulla, P.A., Delzanno, G., Rezine, A.: Parameterized verification of infinite-state processes with global conditions. In: International Conference on Computer Aided Verification. pp. 145–157. Springer (2007)
2. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer Science & Business Media (2013)
3. Bunte, O., Groote, J.F., Keiren, J.J.A., Laveaux, M., Neele, T., Vink, E.P.d., Wesselink, W., Wijs, A., Willemse, T.A.C.: The mCRL2 toolset for analysing concurrent systems. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 21–39. Springer (2019)
4. Carvalho, O.S.: An mutual exclusion in computer networks. Commun ACM **26**, 146–147 (1983)
5. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Communications of the ACM **8**(9), 569 (1965)
6. Duan, Z., Tian, C.: A unified model checking approach with projection temporal logic. In: International Conference on Formal Engineering Methods. pp. 167–186. Springer (2008)
7. Fokkink, W.: Distributed algorithms: an intuitive approach. Mit Press (2018)
8. Garg, V.K.: Elements of distributed computing. John Wiley & Sons (2002)
9. van Glabbeek, R.J., Höfner, P.: Progress, justness, and fairness. ACM Computing Surveys (CSUR) **52**(4), 1–38 (2019)
10. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press Ltd. (2014)
11. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Communications of the ACM **17**(8), 453–455 (1974)
12. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications (1978)
13. Lynch, N.A.: Distributed algorithms. Elsevier (1996)
14. Ma, Q., Duan, Z., Zhang, N., Wang, X.: Verification of distributed systems with the axiomatic system of MSVL. Formal Aspects of Computing **27**(1), 103–131 (2015)
15. Manna, Z., Bjørner, N., Browne, A., Chang, E., Colón, M., Alfaro, L.d., Devarajan, H., Kapur, A., Lee, J., Sipma, H., et al.: STeP: The stanford temporal prover. In: Colloquium on Trees in Algebra and Programming. pp. 793–794. Springer (1995)
16. Peterson, G.L.: Myths about the mutual exclusion problem. Information Processing Letters **12**(3), 115–116 (1981)
17. Raynal, M.: Distributed algorithms for message-passing systems, vol. 500. Springer (2013)
18. Ricart, G., Agrawala, A.K.: An optimal algorithm for mutual exclusion in computer networks. Communications of the ACM **24**(1), 9–17 (1981)
19. Sedletsky, E., Pnueli, A., Ben-Ari, M.: Formal verification of the Ricart-Agrawala Algorithm. In: Kapoor, S., Prasad, S. (eds.) FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science. pp. 325–335. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
20. Shishkin, E.: Construction and formal verification of a fault-tolerant distributed mutual exclusion algorithm. In: Proceedings of the 16th ACM SIGPLAN International Workshop on Erlang. pp. 1–12 (2017)
21. Wu, J.: Distributed system design. CRC press (2017)
22. Yang, X., Duan, Z., Ma, Q.: Axiomatic semantics of projection temporal logic programs. Mathematical Structures in Computer Science **20**(5), 865–914 (2010)

A Final Pseudocode

See Algorithm 5, Algorithm 6 and Algorithm 7 for the pseudocode of the final version of the algorithm. On this version all three properties hold for two and three nodes.

Algorithm 5 The first thread, *Main*, in the final version of the Ricart-Agrawala algorithm.

```
$\mathbf{P}(shared)$
$flag \leftarrow \mathbf{true}$
$chosen \leftarrow highest + 1$
$highest \leftarrow highest + 1$
$awaiting \leftarrow N - 1$
$\mathbf{V}(shared)$
for $0 \leq you < N$ do
    if $you \neq me$ then
        send REQUEST$(chosen, me)$ to $you$
await $awaiting = 0$
critical section
$flag \leftarrow$ false
for $0 \leq you < N$ do
    $\mathbf{P}(shared)$
    if $deferred[you]$ then
        $deferred[you] \leftarrow$ false
        send REPLY to $you$
    $\mathbf{V}(shared)$
```

Algorithm 6 The second thread, *Receive-Requests*, in the final version of the Ricart-Agrawala algorithm.

```
while $\mathbf{true}$ do
    receive REQUEST($k$, $you$)
    $\mathbf{P}(shared)$
    $highest \leftarrow \mathbf{max}(highest, k)$
    $defer \leftarrow (flag \land (chosen < k \lor (chosen = k \land me < you)))$
    if $defer$ then
        $deferred[you] \leftarrow \mathbf{true}$
    else
        send REPLY to $you$
    $\mathbf{V}(shared)$
```

Algorithm 7 The third thread, *Receive-Replies*, in the final version of the Ricart-Agrawala algorithm.

```
while $\texttt{true}$ do
    receive REPLY
    $\mathbf{P}(shared)$
    $awaiting \leftarrow awaiting - 1$
    $\mathbf{V}(shared)$
```

B Mutual Exclusion Violation with N = 3

To help the reader understand why the mutual exclusion violation can take place when $highest$ is not increased when $chosen$ is set, we have included here a description of a mutual exclusion violation with three nodes. This counterexample was not generated by mCRL2, since we could only check mutual exclusion after reducing the model modulo divergence-preserving branching bisimulation, which meant no useful counterexample was produced. However, we can extrapolate how such a violation could take place from the counterexample produced for the case of two nodes and our reasoning given in Section 4.1.

We use $N = 3$ and $M = 5$.

1. N_0 starts the competition, since $highest[0] = 0$, it sets $chosen[0]$ to 1 and sends REQUEST(1, 0) to both N_1 and N_2. It is awaiting two replies. Note that we still have $highest[0] = 0$.
2. N_1 and N_2 both receive the REQUEST(1, 0) message from N_0, neither is currently competing so they both send a REPLY immediately. They update their $highest$ value, so now we have $highest[1] = highest[2] = 1$.
3. N_0 enters the critical section and then exits again, it has no messages to send in the exit protocol.
4. N_2 starts the competition, since $highest[2] = 1$ it picks 2 for $chosen[2]$. It sends REQUEST(2, 1) to both N_0 and N_1. It is awaiting two responses.
5. N_1 receives the REQUEST(2, 1) message, it can send a REPLY immediately because it is not competing. It updates $highest[1]$ to be 2.
6. N_1 starts competing, it sends REQUEST(3, 1) to N_0 and N_2.
7. N_0 upon receiving REQUEST(3, 1) will REPLY immediately because it is not competing, but it will do so without updating $highest[0]$ since $0 \succ 3$.
8. Next, N_0 receives REQUEST(2, 2), it again sends a REPLY immediately because it is not competing. This time, it does update $highest[0]$ to be 2.
9. N_2 has received both REPLY messages and can now enter and exit the critical section. It has no messages to send in its exit protocol.
10. N_0 starts competing and sends REQUEST(3, 0) to N_1 and N_2.
11. When N_1 receives the REQUEST(3, 0) message, it sees that their sequence numbers are the same but $0 < 1$ so it sends the REPLY immediately.
12. N_2 will send a REPLY to both REQUEST(3, 0) and REQUEST(3, 1), since it is not competing.

13. Both N_0 and N_1 have received two REPLY messages to their respective REQUESTs, so they can both enter the critical section.

Note that this counterexample specifically uses the tiebreaker of comparing the node ids, which was not needed in the counterexample mCRL2 generated for two nodes. We did not manage to come up with a counterexample for three nodes that does not use the tiebreaker, but that does not mean such a counterexample does not exist.

Note that this counterexample is only possible because N_0 can REPLY to REQUEST(3, 1) without updating its $highest$ value: if this were not possible it would end up sending a REQUEST(4, 0) later rather than a REQUEST(3, 0), and N_1 would not REPLY to REQUEST(4, 0) immediately. Hence, the central issue is that it is possible for one node's $chosen$ value to be exactly N greater than another node's $highest$ value, as discussed in Section 4.1.

C Models

C.1 Original Ricart-Agrawala

```
map
    M: Pos;
    valid_id: Nat -> Bool;
    valid_sn: Nat -> Bool;
    next_sn: Nat -> Nat;
    greater_sn: Nat # Nat -> Bool;
    max_sn: Nat # Nat -> Nat;
    equal_sn: Nat # Nat -> Bool;
    others: Nat # Pos -> List(Nat);
var
    n, n': Nat;
    l: List(Nat);
eqn
    M = Int2Pos(2* N - 1 );
    valid_id(n) = n < N;
    valid_sn(n) = n < M;
    next_sn(n) = (n + 1) mod M;
    (n < n' && abs(n - n') >= N) -> greater_sn(n, n') = (n + M) > n';
    (n' < n && abs(n - n') >= N) -> greater_sn(n, n') = n > (n' + M);
    (abs(n - n') < N) ->            greater_sn(n, n') = n > n';
    (greater_sn(n, n')) ->          max_sn(n, n') = n;
    (!greater_sn(n, n')) ->         max_sn(n, n') = n';
    equal_sn(n, n') = n == n';
    others(0, 3) = [1, 2];
    others(1, 3) = [0, 2];
    others(2, 3) = [0, 1];
    others(0, 2) = [1];
    others(1, 2) = [0];

sort
```

```
    Message = struct REQUEST(seqnr: Nat) | REPLY;

sort
    Register = struct chosen(pid: Nat) | highest(pid: Nat)
                    | awaiting(pid: Nat) | flag(pid: Nat)
                    | deferred(pid: Nat, pid': Nat);

act
    crit, noncrit: Nat;
    read_nat_s, read_nat_r, read_nat,
        write_nat_s, write_nat_r, write_nat: Register # Nat;
    read_bool_s, read_bool_r, read_bool,
        write_bool_s, write_bool_r, write_bool: Register # Bool;
    send_s, send_r, send,
        receive_s, receive_r, receive: Nat # Nat # Message;
    lock_s, unlock_s,
        lock_r, unlock_r,
        lock, unlock: Nat;

proc

%% The algorithm
% A node has three processes that run asynchronously
Node(ME: Nat) =
    Main(ME) || Requests(ME) || Replies(ME) ||
    Reg_Nat(chosen(ME), 0) ||
    Reg_Nat(highest(ME), 0) ||
    Reg_Nat(awaiting(ME), 0) ||
    Reg_Bool(flag(ME), false) ||
    Reg_Bool(deferred(ME, 0), false) ||
    Reg_Bool(deferred(ME, 1), false) ||
    Reg_Bool(deferred(ME, 2), false)||
    Semaphore(ME);

% The Main thread
Main(ME: Nat) =
    noncrit(ME).
    Main_2(ME);

Main_2(ME: Nat) =
    lock_s(ME).
    write_bool_s(flag(ME), true).
    sum h: Nat.
    read_nat_r(highest(ME), h).
    write_nat_s(chosen(ME), next_sn(h)).
    unlock_s(ME).
    write_nat_s(awaiting(ME), Int2Nat(N - 1)).
    Main_3(ME, others(ME, N), next_sn(h));

Main_3(ME: Nat, to_send: List(Nat), sn: Nat) =
    (#to_send > 0) -> (
        send_s(ME, head(to_send), REQUEST(sn)).
        Main_3(ME, tail(to_send), sn)
    ) <> (
        Main_4(ME)
    );

Main_4(ME: Nat) =
    read_nat_r(awaiting(ME), 0).
    Main_5(ME);

Main_5(ME: Nat) =
    crit(ME).
    Main_6(ME);

Main_6(ME: Nat) =
    write_bool_s(flag(ME), false).
    Main_7(ME, others(ME, N));

Main_7(ME: Nat, to_check: List(Nat)) =
    (#to_check > 0) -> (
        sum d: Bool.
        read_bool_r(deferred(ME, head(to_check)), d).
        (d) -> (
            write_bool_s(deferred(ME, head(to_check)), false).
            send_s(ME, head(to_check), REPLY).
            Main_7(ME, tail(to_check))
        ) <> (
            Main_7(ME, tail(to_check))
        )
    ) <> (
        Main(ME)
    );

% The second process receives and processes request messages
Requests(ME: Nat) =
    sum YOU: Nat. (valid_id(YOU) && YOU != ME) -> (
        sum sn: Nat. valid_sn(sn) -> (
            receive_r(YOU, ME, REQUEST(sn)).
            sum h: Nat.
            read_nat_r(highest(ME), h).
            write_nat_s(highest(ME), max_sn(sn, h)).
            lock_s(ME).
            sum c: Nat.
            read_nat_r(chosen(ME), c).
            sum f: Bool.
            read_bool_r(flag(ME), f).
            unlock_s(ME).
            ((greater_sn(c, sn)) || (equal_sn(c, sn) && ME > YOU) || (!f) ) -> (
                send_s(ME, YOU, REPLY).
                Requests()
            ) <> (
                write_bool_s(deferred(ME, YOU), true).
                Requests()
            )
        )
    );

% The third process receives and processes reply messages
Replies(ME: Nat) =
    sum YOU: Nat. (valid_id(YOU) && YOU != ME) -> (
        receive_r(YOU, ME, REPLY).
        sum n: Nat.
        read_nat_r(awaiting(ME), n).
        write_nat_s(awaiting(ME), Int2Nat(n - 1)).
        Replies()
    );


%% The registers
Reg_Nat(ME: Register, n: Nat) =
    read_nat_s(ME, n). Reg_Nat() +
    sum n': Nat. (valid_sn(n')) -> (
        write_nat_r(ME, n').
        Reg_Nat(ME, n')
    );

Reg_Bool(ME: Register, b: Bool) =
    read_bool_s(ME, b). Reg_Bool() +
    sum b': Bool. write_bool_r(ME, b').
    Reg_Bool(ME, b');

%% The channels
Channel(from: Nat, to: Nat, messages: Set(Message)) =
    sum m: Message. send_r(from, to, m).
    Channel(messages = messages + {m}) +
    sum m: Message. (m in messages) -> (
        receive_s(from, to, m).
        Channel(messages = messages - {m})
    );

%% Semaphores
Semaphore(ME: Nat) =
    lock_r(ME).unlock_r(ME).Semaphore();

%% initialization
init
    allow({
        crit, noncrit,
        read_nat, write_nat,
        read_bool, write_bool,
        send, receive,
        lock, unlock
        },
    comm({
        read_nat_s | read_nat_r -> read_nat,
        read_bool_s | read_bool_r -> read_bool,
        write_nat_s | write_nat_r -> write_nat,
        write_bool_s | write_bool_r -> write_bool,
        send_s | send_r -> send,
        receive_s | receive_r -> receive,
        lock_s | lock_r -> lock,
        unlock_s | unlock_r -> unlock
        },
    Node(0) || Node(1) || Channel(0, 1, {}) || Channel(1, 0, {})
    ));

map
    N : Pos;
eqn
    N = 2;
```

C.2 Final Model for $N = 2$

The model is identical to the model in Appendix C.1, save for the following changes.

For the *Main* thread, the $Main$, $Main\_2$ and $Main\_7$ processes have been altered to incorporate the new placement of $noncrit$ and the new semaphore calls, as well as the additional increase of $highest$. They are replaced by the following:

```
Main(ME: Nat) =
    Main_2(ME);

Main_2(ME: Nat) =
    lock_s(ME).
    write_bool_s(flag(ME), true)|noncrit(ME).
    sum h: Nat.
    read_nat_r(highest(ME), h).
    write_nat_s(chosen(ME), next_sn(h)).
    write_nat_s(highest(ME), next_sn(h)).
    write_nat_s(awaiting(ME), Int2Nat(N - 1)).
    unlock_s(ME).
    Main_3(ME, others(ME, N), next_sn(h));

Main_7(ME: Nat, to_check: List(Nat)) =
    (#to_check > 0) -> (
        lock_s(ME).
        sum d: Bool.
        read_bool_r(deferred(ME, head(to_check)), d).
        (d) -> (
            write_bool_s(deferred(ME, head(to_check)), false).
            send_s(ME, head(to_check), REPLY).
            unlock_s(ME).
            Main_7(ME, tail(to_check))
        ) <> (
            unlock_s(ME).
            Main_7(ME, tail(to_check))
        )
    ) <> (
        Main(ME)
    );
```

The *Recieves-Requests* and *Receive-Replies* threads have altered placement of the semaphore calls as well, so the *Requests* and *Replies* process are replaced by:

```
Requests(ME: Nat) =
    sum YOU: Nat. (valid_id(YOU) && YOU != ME) -> (
        sum sn: Nat. valid_sn(sn) -> (
            receive_r(YOU, ME, REQUEST(sn)).
            lock_s(ME).
            sum h: Nat.
            read_nat_r(highest(ME), h).
            write_nat_s(highest(ME), max_sn(sn, h)).
            sum c: Nat.
            read_nat_r(chosen(ME), c).
            sum f: Bool.
            read_bool_r(flag(ME), f).
            ((greater_sn(c, sn)) || (equal_sn(c, sn) && ME > YOU) || (!f) ) -> (
                send_s(ME, YOU, REPLY).
                unlock_s(ME).
                Requests()
            ) <> (
                write_bool_s(deferred(ME, YOU), true).
                unlock_s(ME).
                Requests()
            )
    ));

Replies(ME: Nat) =
    sum YOU: Nat. (valid_id(YOU) && YOU != ME) -> (
        receive_r(YOU, ME, REPLY).
        lock_s(ME).
        sum n: Nat.
        read_nat_r(awaiting(ME), n).
        write_nat_s(awaiting(ME), Int2Nat(n - 1)).
        unlock_s(ME).
        Replies()
    );
```

We also alter the initialization to accommodate that we moved *noncrit*. In Appendix C.1, we did not hide any actions. However, since the starvation freedom property references $noncrit$, not $write\_bool|noncrit$, we need to ensure that at the very least the $write\_bool$ action is hidden. We also need to allow the $write\_bool|noncrit$ multi-action to take place. Our new initialization is as follows:

```
init
    hide({
        read_nat, write_nat,
        read_bool, write_bool,
        send, receive,
        lock, unlock
    },
    allow({
        crit, write_bool|noncrit,
        read_nat, write_nat,
        read_bool, write_bool,
        send, receive,
        lock, unlock
        },
    comm({
        read_nat_s | read_nat_r -> read_nat,
        read_bool_s | read_bool_r -> read_bool,
        write_nat_s | write_nat_r -> write_nat,
        write_bool_s | write_bool_r -> write_bool,
        send_s | send_r -> send,
        receive_s | receive_r -> receive,
        lock_s | lock_r -> lock,
        unlock_s | unlock_r -> unlock
        },
    Node(0) || Node(1) || Channel(0, 1, {}) || Channel(1, 0, {})
    )));
```

C.3 Model with Broadcast

To make the sending of a REQUEST message a single broadcast rather than multiple messages being send separately, we made to following changes to the model in Appendix C.2.

The $Main\_3$ process is replaced by a process that sends all REQUEST messages in a multi-action. Since our model only needs to accommodate two or three nodes, we can handle both cases explicitly as follows:

```
Main_3(ME: Nat, to_send: List(Nat), sn: Nat) =
    (#to_send == 1) -> (
        send_s(ME, head(to_send), REQUEST(sn)).
        Main_4(ME)
    ) +
    (#to_send == 2) -> (
        send_s(ME, to_send.0, REQUEST(sn))|send_s(ME, to_send.1, REQUEST(sn)).
        Main_4(ME)
    );
```

We also need to update the initialization to allow the multi-action of two *send* actions. Additionally, to model three nodes we need to add the third node and its channels, as well as updating N. The initialization becomes:

```
 init
    hide({
        read_nat, write_nat,
        read_bool, write_bool,
        send, receive,
        lock, unlock
    },
    allow({
        crit, write_bool|noncrit,
        read_nat, write_nat,
        read_bool, write_bool,
        send, receive,
        send|send,
        lock, unlock
        },
    comm({
        read_nat_s | read_nat_r -> read_nat,
        read_bool_s | read_bool_r -> read_bool,
        write_nat_s | write_nat_r -> write_nat,
        write_bool_s | write_bool_r -> write_bool,
        send_s | send_r -> send,
        receive_s | receive_r -> receive,
        lock_s | lock_r -> lock,
        unlock_s | unlock_r -> unlock
        },
    Node(0) ||
        Node(1) ||
        Node(2) ||
        Channel(0, 1, {}) ||
        Channel(1, 0, {}) ||
        Channel(0, 2, {}) ||
        Channel(1, 2, {}) ||
        Channel(2, 0, {}) ||
        Channel(2, 1, {})
    )));

map
    N : Pos;
eqn
    N = 3;
```

C.4 The Reduced Model for $N = 3$

We had to make significant modifications to the previous model to make the state space small enough to be able to verify it with three nodes when the extra increase of *highest* is removed. These were as follows:

- Releasing the semaphores before REPLY messages are sent rather than afterwards. The semaphore is not needed to ensure the message is sent since the sending thread will always do that as its next action. The additional flexibility reduces the state space.
- If *Receive-Requests* notices that its *flag* is **false**, it will always send the REPLY immediately, it does not need to read $chosen$ anymore. We model this explicitly to reduce the number of actions the thread needs to take.
- We add *set_max* and *decrement* operations to the integer registers: *set_max* takes a value x and updates the register's stored value only if x is greater than its current value; *decrement* reduces the stored value by one. We stated in Section 3 that we assumed operations like this had to be broken up into multiple smaller steps, but since we now protect all these sequences of steps with semaphores we can model them as a single step without compromising our verification. Once again, this reduces the number of actions a thread needs to take.
- We removed the semaphore calls from *Receive-Replies*. This thread only affects $awaiting$. When *Main* writes to $awaiting$ this is protected by the semaphore, but this is not really needed: a node will not receive replies before it has sent its REQUEST, so there will be no interference between *Receive-Replies* and *Main*'s writing of $awaiting$. As for when *Main* reads $awaiting$, this is already not protected by a semaphore. So the semaphore in *Receive-Replies* is unnecessary and can be removed to reduce the number of steps taken.

We believe that these modifications do not compromise the validity of our verification, but they are rather significant. Note that for the verification with three nodes *with* the extra increase of $highest$, we did not use any of these modifications and only did a reduction by making the sending of REQUEST messages a broadcast.

The following parts of the model are changed with respect to Appendix C.3.

We added new actions, so act is extended as follows:

```
act
   crit, noncrit: Nat;
   read_nat_s, read_nat_r, read_nat,
       write_nat_s, write_nat_r, write_nat: Register # Nat;
   read_bool_s, read_bool_r, read_bool,
       write_bool_s, write_bool_r, write_bool: Register # Bool;
   send_s, send_r, send,
       receive_s, receive_r, receive: Nat # Nat # Message;
   lock_s, unlock_s,
       lock_r, unlock_r,
       lock, unlock: Nat;
   set_max_s, set_max_r, set_max: Register # Nat;
   decrement_s, decrement_r, decrement: Register;
```

For the *Main* thread, we removed the increase to highest so that line is taken out of $Main\_2$. We release the semaphore before sending messages now, which requires changing $Main\_7$ as well. Those processes are replaced by:

```
Main_2(ME: Nat) =
    lock_s(ME).
    write_bool_s(flag(ME), true)|noncrit(ME).
    sum h: Nat.
    read_nat_r(highest(ME), h).
    write_nat_s(chosen(ME), next_sn(h)).
    % write_nat_s(highest(ME), next_sn(h)).
    write_nat_s(awaiting(ME), Int2Nat(N - 1)).
    unlock_s(ME).
    Main_3(ME, others(ME, N), next_sn(h));

Main_7(ME: Nat, to_check: List(Nat)) =
    (#to_check > 0) -> (
        lock_s(ME).
        sum d: Bool.
        read_bool_r(deferred(ME, head(to_check)), d).
        (d) -> (
            write_bool_s(deferred(ME, head(to_check)), false).
            unlock_s(ME).
            send_s(ME, head(to_check), REPLY).
            Main_7(ME, tail(to_check))
        ) <> (
            unlock_s(ME).
            Main_7(ME, tail(to_check))
        )
    ) <> (
        Main(ME)
    );
```

We updated the model of *Receive-Requests* to not read *chosen* when it has already determined that its *flag* is **false**. We also added the set_max action. The $Requests$ process is altered accordingly:

```
 Requests(ME: Nat) =
    sum YOU: Nat. (valid_id(YOU) && YOU != ME) -> (
        sum sn: Nat. valid_sn(sn) -> (
            receive_r(YOU, ME, REQUEST(sn)).
            lock_s(ME).
            set_max_s(highest(ME), sn).
            sum f: Bool.
            read_bool_r(flag(ME), f).
            (!f) -> (
                unlock_s(ME).
                send_s(ME, YOU, REPLY).
                Requests()
            ) <> (
                sum c: Nat.
                read_nat_r(chosen(ME), c).
                (greater_sn(c, sn) || (equal_sn(c, sn) && ME > YOU)) -> (
                    unlock_s(ME).
                    send_s(ME, YOU, REPLY).
                    Requests()
                ) <> (
                    write_bool_s(deferred(ME, YOU), true).
                    unlock_s(ME).
                    Requests()
                )
            )
        )
    );
```

As for the *Receive-Replies* thread, we now have the *decrement* action, and we removed the semaphore calls:

```
 Replies(ME: Nat) =
    sum YOU: Nat. (valid_id(YOU) && YOU != ME) -> (
        receive_r(YOU, ME, REPLY).
        decrement_s(awaiting(ME)).
        Replies()
    );
```

The process that models integer registers has to be updated to allow the *set_max* and *decrement* operations:

```
Reg_Nat(ME: Register, n: Nat) =
    read_nat_s(ME, n). Reg_Nat() +
    sum n': Nat. (valid_sn(n')) -> (
        write_nat_r(ME, n'). Reg_Nat(ME, n') +
        set_max_r(ME, n'). Reg_Nat(ME, max_sn(n, n'))
    ) +
    decrement_r(ME). Reg_Nat(ME, Int2Nat(n - 1));
```

Finally, we need to update the initialization so that the new actions are hidden, allowed and correctly handled with regards to communication:

```
init
    hide({
        read_nat, write_nat,
        read_bool, write_bool,
        send, receive,
        lock, unlock,
        decrement,
        set_max
    },
    allow({
        crit, write_bool|noncrit,
        read_nat, write_nat,
        read_bool, write_bool,
        decrement,
        set_max,
        send, receive,
        send|send,
        lock, unlock
        },
    comm({
        read_nat_s | read_nat_r -> read_nat,
        read_bool_s | read_bool_r -> read_bool,
        write_nat_s | write_nat_r -> write_nat,
        write_bool_s | write_bool_r -> write_bool,
        send_s | send_r -> send,
        receive_s | receive_r -> receive,
        lock_s | lock_r -> lock,
        unlock_s | unlock_r -> unlock,
        decrement_s | decrement_r -> decrement,
        set_max_s | set_max_r -> set_max
        },
    Node(0)
    || Node(1)
    || Node(2)
    || Channel(0, 1, {})
    || Channel(1, 0, {})
    || Channel(0, 2, {})
    || Channel(1, 2, {})
    || Channel(2, 0, {})
    || Channel(2, 1, {})
    )));
```