ELSEVIER

Theoretical Computer Science 177 (1997) 459–486

Theoretical Computer Science

Formal verification of a leader election protocol in process algebra

Lars-åke Fredlund[a,1], Jan Friso Groote[b], Henri Korver[b,*,2]

[a] *SICS, Box 1263, S-164 28 Kista, Sweden*
[b] *CWI, Kruislaan 413, 1098 SJ Amsterdam, Netherlands*

Abstract

In 1982 Dolev, et al. [10] presented an $O(n \log n)$ unidirectional distributed algorithm for the *circular extrema-finding (or leader-election) problem.* At the same time Peterson came up with a nearly identical solution. In this paper, we bring the correctness of this algorithm to a completely formal level. This relatively small protocol, which can be described on half a page, requires a rather involved proof for guaranteeing that it behaves well in all possible circumstances. To our knowledge, this is one of the more advanced case-studies in formal verification based on process algebra.

1. Introduction

Experience teaches that distributed protocols are hard to define correctly. This is not only due to the inherent complexity of distributed systems, but it is also caused by the lack of adequate techniques to prove the correctness of such protocols. This means that there are no good ways of validating designs for distributed systems. The current approach to proving correctness of distributed systems generally uses stylished forms of hand waving that does not always avoid the intricacies and pitfalls that often appear in distributed systems. We are convinced that more precise proof techniques need to be used, which should allow for computer based proof checking. Concretely this means that a logic based approach should be taken.

The language μCRL (micro Common Representation Language) [12] has been defined as a combination of process algebra and (equational) data types to describe and verify distributed systems. In accordance with the philosophy outlined in the first

* Corresponding author. E-mail: {jfg,henri}@cwi.nl.

[1] Research supported in part by the Swedish Research Council for Engineering Sciences (TFR) (project no. 221-92-722) and the HCM project EXPRESS.

[2] The investigations were (partly) supported by the Foundation for Computer Science in the Netherlands (SION) with financial support from the Netherlands Organization for Scientific Research (NWO).

PII S0304-3975(96)00256-3

paragraph this is a very precisely defined language provided with a logical proof system [13]. It is primarily intended to verify statements of the form

Condition $\rightarrow$ *Specification* = *Implementation.*

This system has been applied to verify a number of data transfer and distributed scheduling protocols of considerable complexity [3, 11, 15, 16]. It incorporates several old and new techniques [4, 3]. Due to the logical nature of the proof system proofs can be verified by computer. Some sizable examples of proofs verified using the proof checker Coq [9] are reported in [13, 17].

If one develops a new technique then it is important that it is validated that the technique meets its purpose. For μCRL this means that it is applied to a wide range of distributed systems. In this paper we show its applicability on Dolev et al.'s *leader election* or *extrema finding* protocol [10] that has been designed for a network with a unidirectional ring topology. At the same time, Peterson published a nearly identical version of this protocol, see [20]. This protocol is efficient, $O(n \log n)$, and highly parallel. As far as we know this is the first leader election protocol verified in a process algebraic style. In [6, 7] a number of leader election protocols for carrier sense networks have been specified and some (informal) proof sketches are given in modal logic.

In Section 2 we specify Dolev et al.'s leader election protocol formally in μCRL. The protocol is proved correct in Section 3 using a detailed argument. Appendix A summarises the proof theory for μCRL, and Appendix B defines the data types used in the specification and proof of the protocol.

2. Specification and correctness of the protocol

We assume n processes in a ring topology, connected by unbounded queues. A process can only send messages in a clockwise manner. Initially, each process has a unique identifier ident (in the following assumed to be a natural number). The task of an algorithm for solving the leader election problem is then to make sure that eventually exactly one process will become the leader.

In Dolev et al.'s algorithm [10] each process in the ring carries out the following task:

Active:
d:= ident
do forever
    send(d)
    receive(e)
    if d=e then **stop** /* *Process is the leader* */
    send(e)
    receive(f)
    if $e > max(d,f)$ **then** d:=e **else goto** *Relay*
end

Relay:
do forever
receive(d)
send(d)
end

The intuition behind the protocol is as follows. In each round the number of electable processes decreases, if there are more than two active processes around. During each round every active process, i.e., a process in state *Active*, receives two different values. If the first value is larger than the second value and its own value, then it stays active. In this case its anti-clockwise neighbour will become a relay process. So, from every set of active neighbours, one will die in every round. Furthermore, the maximal value among the identifiers will never be lost in the ring network, it will traverse the ring in messages, or be stored in a variable in a process, until only one active process remains. If only one active process is left, i.e., not in state *Relay*, then the leader-in-spe sends its own value of d to itself, and then halts.

As the attentive reader may have noticed, there is a simpler way to elect a leader. For example, it would be sufficient for a process to receive just one value, i.e., the value (e) of its direct neighbour. In this case, only two values instead of three values have to be compared ($e > d$ instead of $e > max(d, f)$). However, this approach is not so efficient as one may need $2n^2 + 2n$ actions before a leader is selected. The protocol described earlier is faster. It is bounded by $2n \log n + 2n$ actions because in every round at least one process becomes inactive.[3] For an explanation of these complexity bounds one is referred to [10].

Below we formalise the processes and their configuration in the ring as described above in μCRL.

act $leader$
$r, s : Nat \times Nat$
proc $Active(i{:}Nat, d{:}Nat, n{:}Nat) =$
$s(i,d) \sum_{e:Nat} r(i -_n 1, e)\,(leader\,\delta \lhd eq(d,e) \rhd s(i,e)$
$\sum_{f:Nat}(r(i -_n 1, f)\, Active(i,e,n) \lhd e > max(d,f) \rhd Relay(i,n)))$
$Relay(i{:}Nat, n{:}Nat) = \sum_{d:Nat} r(i -_n 1, d)\, s(i,d)\, Relay(i,n)$

Here a process in the imperative description with value ident for d corresponds to $Active(i, \mathsf{ident}, n)$. Intuitively the μCRL process first sends the value of the variable d to the next process in the ring ($s(i,d)$) via a queue, which is described below. Then it reads a new value e from the queue connected to the preceding process in the ring by an action $r(i -_n 1, e)$. The notation $-_n$ stands for subtraction modulo n, which is defined in Appendix B. Consequently, it executes a then-if-else test denoted by $\_ \lhd \_ \rhd \_$. If the variables d and e are equal, expressed by $eq(d,e)$, then the process declares itself leader by executing the action *leader*. Otherwise the value of e is sent ($s(i,e)$) and a

[3] By $\log n$, we mean $\log_2 n$.

value f is read ($r(i-_n 1, f)$). Now, if e is larger than both d and f the process repeats itself with e replacing d. Otherwise, the process becomes a relay process (denoted by $Relay(i,n)$). The *leader* action has been introduced in the μCRL specification of the protocol for verification purposes; it makes visible the fact that exactly one leader is elected.

The δ process after the leader action in the *Active* process is not essential. We have inserted it for technical reasons and more details on this issue are given at the end of this section.

In order to prove the correctness of the protocol we must be precise about the behaviour of the queues that connect the processes. We assume that the queues have infinite size and deliver data in a strict first in first out fashion without duplication or loss. In the queue process data is stored in a data queue q which is specified in Appendix B. Note that the behaviour of the queue process is straightforward; it reads data via $r(i,d)$ at process i and delivers it via $s(i+_n 1)$ at process $i+_n 1$ ($+_n$ is addition modulo n). Below, $toe(q)$ denotes the first element that was inserted in data queue q.

$$\begin{aligned}\textbf{proc}\quad Q(i{:}Nat, n{:}Nat, q{:}Queue) = {} & \textstyle\sum_{d:Nat} r(i,d)\, Q(i,n,in(d,q)) \\ & + s(i+_n 1, toe(q))\, Q(i,n,untoe(q)) \\ & \lhd \text{not } empty(q) \rhd \delta\end{aligned}$$

It remains to connect all processes together. First we state that send actions s communicate with receive actions r. Then, using the processes $Spec'$ and $Spec$ we combine the processes with the queues, and assign a unique number to them. The process $Spec(n)$ represents a ring network of n processes interconnected by queues. The injective function $id : Nat \to Nat$ maps natural numbers to process identifiers, for convenience also represented as natural numbers. The process identifiers are related by the total ordering $\leqslant$. The abbreviation max will be used to denote the maximal identifier, with respect to the ordering $\leqslant$ and the number of processes n, of the set $\{id(x) : 0 \leqslant x \leqslant n-1\}$.

$$\begin{aligned}&\textbf{func}\quad id : Nat \to Nat\\ &\textbf{act}\quad c : Nat \times Nat\\ &\textbf{comm}\quad r|s = c\\ &\textbf{proc}\quad Spec'(m{:}Nat, n{:}Nat) =\\ &\qquad (Active(m-1, id(m-1), n) \parallel Q(m-1, n, q_0) \parallel Spec'(m-1, n))\\ &\qquad\quad \lhd m > 0 \rhd \delta\\ &\qquad Spec(n{:}Nat) = \tau_{\{c\}}\partial_{\{r,s\}}(Spec'(n,n))\end{aligned}$$

Since the protocol is supposed to select exactly one leader after some internal negotiation we formulate correctness by the following formula, where '=' is to be interpreted as 'behaves the same':

Theorem 2.1. *For all* $n : Nat$

$$n > 0 \to Spec(n) = \tau\, leader\, \delta$$

The theorem says that in a ring with at least one process exactly one leader will be elected after some internal activity.

In the specification of the *Active* process given above, we have inserted a δ process after the leader action. We introduced this δ for technical convenience in our verification. However, omitting δ does not effect the behaviour of the leader protocol, *Spec*, as a whole. In fact, if we leave out this δ the whole system *Spec* still deadlocks after performing a leader action as stated in Theorem 2.1. The reason for this is that *Spec* can only terminate if all processes in the system terminate. In particular, the *Relay* processes cannot terminate and evolve in a deadlock situation when a leader is selected. So, even if the process that performs the leader action terminates successfully (which is not the case here), the full system will still end up in a deadlock.

As experience shows the correctness reasoning above is too imprecise to serve as a proof of correctness of the protocol. Many, often rather detailed arguments, are omitted. Actually, the protocol does not have to adhere to the rather synchronous execution suggested by the word 'rounds', but is highly parallel. One can even argue that given the large number of rather 'wild' executions of the protocol, the above description makes little sense. Therefore, we provide in the next sections a completely formalised proof, where we are only interested in establishing correctness of the protocol and not in proving its efficiency.

3. A proof of the protocol

The proof strategy for proving the correctness theorem consists of a number of distinct steps. First in Section 3.1 we define a linear representation of the specification in which the usage of the parallel composition operator in the original specification is replaced by a tabular data structure encoding the states of processes in the network, and actions with guards that check the contents of the data structure. The linearised specification is proven equivalent to the original specification in Lemma 3.3. Then, in Section 3.3, we define a (focus) condition on the tabular data structure such that if the condition holds then no internal computation is any longer possible in the protocol, i.e., no τ-steps can be made [3]. The focus condition is used in Lemma 3.10, in Section 3.6, to separate the proof that the linear specification can be proven equivalent to a simple process into two parts. Lemma 3.10 together with Lemma 3.3 then immediately proves the correctness theorem of the protocol, i.e., Theorem 2.1. The proof of Lemma 3.10 makes use of the Concrete Invariant Corollary (see Appendix A and [4]), i.e., a number of invariance properties are defined (in Section 3.4) on the tabular data structure such that regardless which execution step the linear specification performs, the properties remain true after the step if they were true before the execution of the step. These invariants are used to prove the equality between the linear specification and the simple process in Lemma 3.10. In order to make use of the Concrete Invariant Corollary we have to show that the linear specification can only perform finitely many consecutive τ-steps. This is proven in Section 3.5.

3.1. Linearisation

As a first step the leader election protocol is described as a μCRL process in a state based style, as this is far more convenient for proving purposes. The state based style very much resembles the Unity format [5, 8] or the I/O automata format [5]. Following [5] we call this format the Unity format or a process specification in Unity style. Inspection of the processes *Active* and *Relay* indicates that there are 7 different major states between the actions. The states in *Active* are numbered 0,1,2,3,6 and those in *Relay* get numbers 4 and 5. The processes *Active* and *Relay* can then be restated as follows:

$$
\begin{aligned}
\textbf{proc}\quad & Act(i{:}Nat, d{:}Nat, e{:}Nat, n{:}Nat, s{:}Nat) \\
& = s(i,d)\, Act(i,d,e,n,1) \lhd eq(s,0) \rhd \delta \\
& \quad + \textstyle\sum_{e:Nat} r(i -_n 1, e)\, Act(i,d,e,n,2) \lhd eq(s,1) \rhd \delta \\
& \quad + leader\, Act(i,d,e,n,6) \lhd eq(d,e) \text{ and } eq(s,2) \rhd \delta \\
& \quad + s(i,e)\, Act(i,d,e,n,3) \lhd \text{not } eq(d,e) \text{ and } eq(s,2) \rhd \delta \\
& \quad + \textstyle\sum_{f:Nat} r(i -_n 1, f)\, Act(i,e,e,n,0) \lhd e > max(d,f) \text{ and } eq(s,3) \rhd \delta \\
& \quad + \textstyle\sum_{f:Nat} r(i -_n 1, f)\, Act(i,d,e,n,4) \lhd e < max(d,f) \text{ and } eq(s,3) \rhd \delta \\
& \quad + \textstyle\sum_{d:Nat} r(i -_n 1, d)\, Act(i,d,e,n,5) \lhd eq(s,4) \rhd \delta \\
& \quad + s(i,d)\, Act(i,d,e,n,4) \lhd eq(s,5) \rhd \delta
\end{aligned}
$$

Lemma 3.1. *For all i,d,e,n, we have:*

$$Active(i,d,n) = Act(i,d,e,n,0),$$

$$Relay(i,n) = Act(i,d,e,n,4).$$

Proof. The proof of this lemma is straightforward, using the Recursive Specification Principle (RSP), but note that it uses $a(p \lhd c \rhd q) = a\,p \lhd c \rhd a\,q$ as well as the distributivity of Σ over $+$. □

We now put the processes and queues in parallel. As we work towards the Unity style, we must encode the states of the individual processes in a data structure. For this we take a table (or indexed queue) with an entry for each process i. This entry contains values for the variables d, e, s and the contents of the queue in which process i is putting its data. Furthermore, it contains a variable of type **Bool**, which plays a role in the proof. The data structure has the name *Table* and is defined in Appendix B.

We put the processes and queues together in three stages. First we put all processes together, using Π_{Act} and X_{Act} below. Then we put all queues together, via Π_Q and X_Q. Finally, we combine X_{Act} and X_Q obtaining the process X which is a description in Unity style of the leader election protocol.

$$
\begin{aligned}
\textbf{proc}\quad & Spec(B{:}Table, n{:}Nat) = \tau_{\{c\}}\partial_{\{r,s\}}(\Pi_{Act}(B,n) \parallel \Pi_Q(B,n)) \\
& \Pi_{Act}(B{:}Table, n{:}Nat) = \delta \lhd empty(B) \rhd \\
& \quad (Act(hd_i(B), get_d(hd_i(B),B), get_e(hd_i(B),B), n, get_s(hd_i(B),B)) \parallel \\
& \qquad \Pi_{Act}(tl(B), n))
\end{aligned}
$$

$$\Pi_Q(B{:}Table, n{:}Nat) = \delta \lhd empty(B) \rhd (Q(hd_i(B), n, get_q(hd_i(B), B)) \parallel \Pi_Q(tl(B), n))$$

$$\begin{aligned}
&X_{Act}(B{:}Table, n{:}Nat)\\
&\quad = \sum_{j:Nat} s(j, get_d(j,B))\, X_{Act}(upd_s(1,j,B), n) \lhd eq(get_s(j,B), 0) \text{ and } test(j,B) \rhd \delta\\
&\qquad + \sum_{j:Nat}\sum_{e:Nat} r(j -_n 1, e)\, X_{Act}(upd_e(e, j, upd_s(2,j,B)), n) \lhd eq(get_s(j,B), 1) \text{ and } test(j,B) \rhd \delta\\
&\qquad + \sum_{j:Nat} leader\, X_{Act}(upd_s(6,j,B), n) \lhd eq(get_d(j,B), get_e(j,B)) \text{ and } eq(get_s(j,B), 2) \text{ and } test(j,B) \rhd \delta\\
&\qquad + \sum_{j:Nat} s(j, get_e(j,B))\, X_{Act}(upd_s(3,j,B), n) \lhd \text{not } eq(get_d(j,B), get_e(j,B)) \text{ and } eq(get_s(j,B), 2) \text{ and } test(j,B) \rhd \delta\\
&\qquad + \sum_{f:Nat}\sum_{j:Nat} r(j -_n 1, f)\, X_{Act}(upd_d(get_e(j,B), j, upd_s(0,j,B)), n) \lhd get_e(j,B) > max(get_d(j,B), f) \text{ and } eq(get_s(j,B), 3) \text{ and } test(j,B) \rhd \delta\\
&\qquad + \sum_{f:Nat}\sum_{j:Nat} r(j -_n 1, f)\, X_{Act}(upd_s(4,j,B), n) \lhd get_e(j,B) \leqslant max(get_d(j,B), f) \text{ and } eq(get_s(j,B), 3) \text{ and } test(j,B) \rhd \delta\\
&\qquad + \sum_{d:Nat}\sum_{j:Nat} r(j -_n 1, d)\, X_{Act}(upd_d(d, j, upd_s(5,j,B)), n) \lhd eq(get_s(j,B), 4) \text{ and } test(j,B) \rhd \delta\\
&\qquad + \sum_{j:Nat} s(j, get_d(j,B))\, X_{Act}(upd_s(4,j,B), n) \lhd eq(get_s(j,B), 5) \text{ and } test(j,B) \rhd \delta
\end{aligned}$$

$$\begin{aligned}
&X_Q(B{:}Table, n{:}Nat)\\
&\quad = \sum_{d:Nat}\sum_{j:Nat} r(j,d)\, X_Q(in_q(d,j,B), n) \lhd test(j,B) \rhd \delta\\
&\qquad + \sum_{j:Nat} s(j +_n 1, toe(j,B))\, X_Q(untoe(j,B), n) \lhd \text{not } empty(j,B) \text{ and } test(j,B) \rhd \delta
\end{aligned}$$

The leader election protocol in Unity form is given below and will be the core process of the proof. Note that in many cases verification of a protocol only starts after the process below has been written down. In the description of X most details of the description are directly reflected in corresponding behaviour of the constituents X_{Act} and X_Q. However, there is one difference. It appears that in the protocol two kinds of messages travel around, *active* and *passive* ones. The active messages contain numbers that may replace the current value of the d-variable of its receiver. The passive messages are not essential for the correctness of the protocol, but only used to improve its speed. For the correctness of the protocol it is important to know that the maximum identifier is always somewhere in an active position and that no identifier occurs in more than one active position. In order to distinguish active from passive messages, we have added a boolean b to each message in the queues, where if $b = \mathsf{t}$ the message is active, and if $b = \mathsf{f}$ the message is passive. When processes become *Relay*s then they also act as a queue. Therefore, we have also added a boolean b

to the process parameters, to indicate the status of the message that a process in state 5 is holding. The equation below is referred to by (I) in the remainder of the proof.

proc $X(B{:}Table, n{:}Nat)$

$$
\begin{aligned}
&= \textstyle\sum_{j:Nat} \tau X(upd_s(1,j,in_q(get_d(j,B),\mathsf{t},j,B)),n) \triangleleft eq(get_s(j,B),0) \text{ and } j < n \triangleright \delta \\
&\quad + \textstyle\sum_{j:Nat} \tau X(untoe(j -_n 1, upd_e(toe(j -_n 1,B),j,upd_s(2,j,B))),n) \\
&\qquad \triangleleft eq(get_s(j,B),1) \text{ and not } empty(j -_n 1,B) \text{ and } j < n \triangleright \delta \\
&\quad + \textstyle\sum_{j:Nat} leader\, X(upd_s(6,j,B),n) \\
&\qquad \triangleleft eq(get_s(j,B),2) \text{ and } eq(get_d(j,B),get_e(j,B)) \text{ and } j < n \triangleright \delta \\
&\quad + \textstyle\sum_{j:Nat} \tau X(upd_s(3,j,in_q(get_e(j,B),\mathsf{f},j,B)),n) \\
&\qquad \triangleleft eq(get_s(j,B),2) \text{ and not } eq(get_d(j,B),get_e(j,B)) \text{ and} \\
&\qquad\quad j < n \triangleright \delta \\
&\quad + \textstyle\sum_{j:Nat} \tau X(untoe(j -_n 1, upd_d(get_e(j,B),j,upd_s(0,j,B))),n) \\
&\qquad \triangleleft get_e(j,B) > max(get_d(j,B),toe(j -_n 1,B)) \text{ and} \\
&\qquad eq(get_s(j,B),3) \text{ and not } empty(j -_n 1,B) j < n \triangleright \delta \\
&\quad + \textstyle\sum_{j:Nat} \tau X(untoe(j -_n 1, upd_s(4,j,upd_b(toe_b(j -_n 1,B),j,B))),n) \\
&\qquad \triangleleft get_e(j,B) \leqslant max(get_d(j,B),toe(j -_n 1,B)) \text{ and} \\
&\qquad\quad eq(get_s(j,B),3) \text{ and not } empty(j -_n 1,B) j < n \triangleright \delta \\
&\quad + \textstyle\sum_{j:Nat} \tau X(untoe(j -_n 1, upd_d(toe(j -_n 1,B),j, \\
&\qquad upd_s(5,j,upd_b(toe_b(j -_n 1,B),j,B)))),n) \\
&\qquad\quad \triangleleft eq(get_s(j,B),4) \text{ and not } empty(j -_n 1,B) \text{ and } j < n \triangleright \delta \\
&\quad + \textstyle\sum_{j:Nat} \tau X(in_q(get_d(j,B),get_b(j,B),j,upd_s(4,j,B)),n) \\
&\qquad \triangleleft eq(get_s(j,B),5) \text{ and } j < n \triangleright \delta
\end{aligned}
$$

Definition 3.2. The function $init: Nat \to Table$, which is used for denoting the initial state of the protocol, is defined as follows:

$$init(n) = if(eq(n,0), t_0, in(n-1, id(n-1), 0, 0, \mathsf{f}, q_0, init(n-1))).$$

See also Section B.5.

Lemma 3.3. *For all* $B: Table$ *and* $m,n: Nat$

1. $UniqueIndex(B) \to \Pi_{Act}(B,n) = X_{Act}(B,n)$,
2. $UniqueIndex(B) \to \Pi_{Q}(B,n) = X_{Q}(B,n)$,
3. $UniqueIndex(B) \wedge test(j,B) = j < n \to Spec(B,n) = X(B,n)$,
4. $Spec'(m,n) = \Pi_{Act}(init(m),n) \parallel \Pi_{Q}(init(m),n)$,
5. $Spec(n) = Spec(init(n),n)$,
6. $Spec(n) = X(init(n),n)$.

Proof.

1. A standard expansion using induction on B (cf. [17]).
2. Again a straightforward expansion.
3. $Spec(B,n) = \tau_{\{c\}}\partial_{\{r,s\}}(\Pi_{Act}(B,n) \parallel \Pi_Q(B,n)) = \tau_{\{c\}}\partial_{\{r,s\}}(X_{Act}(B,n) \parallel X_Q(B,n))$. Now expand $X_{Act}(B,n) \parallel X_Q(B,n)$ and apply hiding. The equations obtained in this way match those of $X(B,n)$, except that '$j < n$' is replaced by '$test(j,B)$' or '$test(j,B)$ and $test(j -_n 1,B)$'. As X is convergent (proven in Lemma 3.7) it follows with the Concrete Invariant Corollary [4] that $Spec(B,n)$ and $X(B,n)$ are equal. The invariant '$test(j,B) = j < n$' is used and easy to show true.
4. By induction on m, using associativity and commutativity of the merge.
5. Directly from the previous case, i.e. Lemma 3.3.4.
6. Directly using cases 3 and 5. □

3.2. Notation

In the sequel we will for certain property formulas $\phi(j)$ write

$$\forall_{j<n}\phi(j) \text{ for } \phi'(0,n) \quad \text{and} \quad \forall_{i<j<n}\phi(j) \text{ for } \phi'(i+1,n)$$

and

$$\exists_{j<n}\phi(j) \text{ for } \phi''(0,n) \text{ and } \exists_{i<j<n}\phi(j) \text{ for } \phi''(i+1,n)$$

where $\phi'(j,n)$ and $\phi''(j,n)$ are defined by:

$$\phi'(j,n) = if(j \geqslant n, \mathsf{t}, \phi(j) \text{ and } \phi'(j+1,n)),$$

$$\phi''(j,n) = if(j \geqslant n, \mathsf{f}, \phi(j) \text{ or } \phi''(j+1,n)).$$

Summation over an arithmetic expression $\gamma(j)$ can be written

$$\textstyle\sum_{j<n}\gamma(j) \text{ for } \gamma'(0,n)$$

where

$$\gamma'(j,n) = if(j \geqslant n, 0, \gamma(j) + \gamma'(j+1,n)).$$

Note that if we can prove that

$$(j < n \text{ and } \phi(j)) \rightarrow \psi(j),$$

then we can also show that

$$\forall_{j<n}\phi(j) \rightarrow \forall_{j<n}\psi(j) \text{ and}$$

$$\exists_{j<n}\phi(j) \rightarrow \exists_{j<n}\psi(j).$$

Also note that

$$\text{not } (\forall_{j<n}\phi(j)) = \exists_{j<n} \text{ not } \phi(j) \text{ and}$$

$$\text{not } (\exists_{j<n}\phi(j)) = \forall_{j<n} \text{ not } \phi(j)$$

3.3. Focus condition

The focus condition FC: $Table \times Nat \rightarrow$ **Bool** indicates at which points the leader election protocol cannot do τ-steps. This means it can either do nothing, or do a *leader* action. The focus condition is constructed in a straightforward fashion by collecting the conditions for the τ-steps in process X.

$$
\begin{aligned}
&FC(B,n) = \\
&\forall_{j<n} \text{ not } eq(get_s(j,B),0) \text{ and} \\
&\quad (\text{not } eq(get_s(j,B),1) \text{ or } empty(j -_n 1,B)) \text{ and} \\
&\quad (\text{not } eq(get_s(j,B),2) \text{ or } eq(get_d(j,B),get_e(j,B))) \text{ and} \\
&\quad (\text{not } eq(get_s(j,B),3) \text{ or } empty(j -_n 1,B)) \text{ and} \\
&\quad (\text{not } eq(get_s(j,B),4) \text{ or } empty(j -_n 1,B)) \text{ and} \\
&\quad \text{not } eq(get_s(j,B),5)
\end{aligned}
$$

3.4. Some invariants of X

In this section we state four invariants $(Inv_1,\ldots,Inv_4)$ of the process $X(B,n)$ that are used in Section 3.6 to prove the correctness of the protocol. We prove that the predicates below are indeed invariance properties in a traditional manner. First we show that they hold in the initial state of the protocol, i.e., for invariant Inv_i we show $Inv_i(init(n),n)$. Then for each protocol step (there are eight such steps in the linearised process X) we show that if both the precondition of the step holds and the predicate holds in the state before the protocol step, then the predicate holds also in the state that is the result of performing the step. For example, to prove that Inv_2 is an invariance property we need to establish that the first step in X preserves the property, i.e., that

$$eq(get_s(j,B),0) \text{ and } j < n \text{ and } Inv_2(B,n) \rightarrow Inv_2(upd_s(1,j,in_q(get_d(j,B),\mathsf{t},j,B)),n)$$

where B is a tabular data structure. This entails proving a large number of rather trivial lemmas, such as:

$$qsizes(upd_s(1,j,B),n) = qsizes(B,n)$$

We omit here the rather long and tedious details of these proofs. In order to establish that Inv_3 and Inv_4 are indeed invariants we first have to prove additional statements on the behaviour of the protocol, i.e., $Inv_5, Inv_7, Inv_6, Inv_8$ and Inv_9 in Sections 3.4.5–3.4.9, respectively.

3.4.1. Acceptable states

Each process is in one of the states $0,\ldots,6$:

$$Inv_1(B,n) = \forall_{j<n} 0 \leqslant get_s(j,B) \leqslant 6$$

3.4.2. Bound on the number of messages in queues

Invariant Inv_2 expresses the property that the number of processes in state 1 or 3 is equal to the number of processes in state 5 plus the number of messages in message channels.

$$Inv_2(B,n) = eq(nproc(B,1,n) + nproc(B,3,n), nproc(B,5,n) + qsizes(B,n))$$

where

$$nproc(B,s,n) = \sum_{j<n} if(eq(get_s(j,B),s),1,0)$$

$$qsizes(B,n) = \sum_{j<n} size(get_q(j,B))$$

3.4.3. Termination of one process implies termination of all processes

Invariant Inv_3 expresses that if a process is in state 6, then all processes are either in state 4 or state 6. It is provable using invariant Inv_9.

$$Inv_3(B,n) = (\exists_{j<n} eq(get_s(j,B),6)) \rightarrow \forall_{j<n} eq(get_s(j,B),4) \text{ or } eq(get_s(j,B),6)$$

3.4.4. Max is preserved

In the initial state, $init(n)$, the maximal identifier in the ring is equal to max. Invariant Inv_4 expresses that this value cannot be lost. The invariants Inv_5, Inv_7, Inv_6 are needed to establish Inv_4.

$$Inv_4(B,n) = \exists_{j<n} ActiveNode(\mathsf{max},j,B) \text{ or } ActiveChan(\mathsf{max}, get_q(j,B))$$

where

$$ActiveNode(k,j,B) =$$
$$(eq(get_s(j,B),0) \text{ and } eq(get_d(j,B),k)) \text{ or}$$
$$((eq(get_s(j,B),2) \text{ or } eq(get_s(j,B),3) \text{ or } eq(get_s(j,B),6)) \text{ and}$$
$$eq(get_e(j,B),k)) \text{ or}$$
$$(eq(get_s(j,B),5) \text{ and } get_b(j,B) \text{ and} eq(get_d(j,B),k))$$

$$ActiveChan(k,q) =$$
$$if(empty(q),\mathsf{f},(hd_b(q) \text{ and } eq(k,hd(q))) \text{ or } ActiveChan(k,tl(q))).$$

An identifier has not been lost if it can in the future be received by another process and replace the value of the d variable of that process. Identifiers can be stored either in a variable ($ActiveNode$) or in a channel ($ActiveChan$).

3.4.5. Trivial facts

Inv_5 formulates two trivial protocol properties, that all identifiers are less than n (less than or equal to the maximal identifier max), and that the values of variables d and e differ when a process is in state 3.

$$Inv_5(B,n) =$$
$$\forall_{j<n} Bounded_q(get_q(j,B),n) \text{ and}$$

$get_d(j,B) < n$ and $get_e(j,B) < n$ and
$if(eq(get_s(j,B),3), \text{not } eq(get_d(j,B), get_e(j,B)), \mathsf{t})$

where

$$Bounded_q(q,n) = if(empty(q), \mathsf{t}, hd(q) < n \text{ and } Bounded_q(tl(q),n)).$$

3.4.6. Active and passive messages

The invariant Inv_6 characterises the relation between neighbour processes and channel contents.

$$Inv_6(B,n) = \forall_{j<n} Alt(j,B,n)$$

where

$$\begin{aligned}
&Alt(j,B,n) = \\
&\quad if(eq(get_s(j,B),0) \text{ or} \\
&\qquad eq(get_s(j,B),3) \text{ or} \\
&\qquad (eq(get_s(j,B),4) \text{ and not } get_b(j,B)) \text{ or} \\
&\qquad (eq(get_s(j,B),5) \text{ and } get_b(j,B)), \\
&\qquad secondary(get_q(j,B),j,B,n), \\
&\qquad primary(get_q(j,B),j,B,n))
\end{aligned}$$

$$\begin{aligned}
&primary(q,j,B,n) = \\
&\quad if(empty(q), \\
&\qquad eq(get_s(j +_n 1,B),2) \text{ or } eq(get_s(j +_n 1,B),3) \text{ or } eq(get_s(j +_n 1,B),6) \text{ or} \\
&\qquad ((eq(get_s(j +_n 1,B),4) \text{ or } eq(get_s(j +_n 1,B),5)) \text{ and } get_b(j +_n 1,B)), \\
&\qquad hd_b(q) \text{ and } secondary(tl(q),j,B,n))
\end{aligned}$$

$$\begin{aligned}
&secondary(q,j,B,n) = \\
&\quad if(empty(q), \\
&\qquad eq(get_s(j +_n 1,B),0) \text{ or } eq(get_s(j +_n 1,B),1) \text{ or} \\
&\qquad ((eq(get_s(j +_n 1,B),4) \text{ or } eq(get_s(j +_n 1,B),5)) \text{ and not } get_b(j +_n 1,B)), \\
&\qquad \text{not } hd_b(q) \text{ and } primary(tl(q),j,B,n)).
\end{aligned}$$

This rather complex looking invariant captures the protocol property that there are two kinds of messages sent: *active messages* which are received by the following process as values on the e variable and which can subsequently replace the d value of the process. The *passive messages* are received as values on the f variables (state 3) and will not replace the original d value of the process.

The Alt property guarantees that an active message can never be received as a passive message (or vice versa), i.e., neighbour processes and channels are always kept synchronised by the protocol. Inv_6 is needed to establish the invariants Inv_8 and Inv_4, to guarantee that identifiers are neither duplicated nor lost.

In order to prove Inv_6 in particular the following two lemmas are useful. $Lemma_{61}$ allows to conveniently prove that $secondary(get_q(j,B),j,B)$ implies $secondary(get_q(j,B'),j,B)$, assuming that the channels $get_q(j,B)$ and $get_q(j,B')$ are identical.

$Lemma_{61}(B,B',n) =$
$\forall_{j<n} eq(get_q(j,B),get_q(j,B')) \rightarrow$
$(secondary(get_q(j,B),j,B,n) \rightarrow secondary(get_q(j,B'),j,B',n)) \leftrightarrow$
$(even(size(get_q(j,B))) \rightarrow (secondary(q_0,j,B,n) \rightarrow secondary(q_0,j,B',n)))$ and
$(\text{not}(even(size(get_q(j,B))))) \rightarrow (primary(q_0,j,B,n) \rightarrow primary(q_0,j,B',n))))$.

Similarly, $Lemma_{62}$ is convenient for proving that the transition from state 3 to state 4 preserves the invariant:

$Lemma_{62}(B,n) =$
$\forall_{j<n}(Alt(j,B,n)$ and not $empty(j,B)$ and $secondary(get_q(j,B),j,B,n)$ and
$eq(get_s(j +_n 1,B),3) \rightarrow$ not $toe_b(j,B))$.

3.4.7. Consecutive identifiers are distinct

Inv_7 guarantees that when an identifier in an active position follows an identifier in a passive position, the identifiers are distinct. This invariant depends on Inv_5 and Inv_6.

$Inv_7(B,n) = \forall_{j<n} Cons(j,B,n)$

where

$Cons(j,B,n) =$
$Cons_q(get_q(j,B),j,B,n)$ and
$if(eq(get_s(j,B),5)$ and not $get_b(j,B)$,
$Neq_q(get_d(j,B),get_q(j,B),j,B,n)$,
$if(eq(get_s(j,B),1)$ or $eq(get_s(j,B),2),Eq_q(get_d(j,B),get_q(j,B),j,B,n),\mathsf{t}))$
$Cons_q(q,j,B,n) =$
$if(empty(q),\mathsf{t},Cons_q(tl(q),j,B,n)$ and $if(hd_b(q),\mathsf{t},Neq_q(hd(q),tl(q),j,B,n)))$

$Neq(k,j,B,n) =$
$((eq(get_s(j,B),2)$ or $eq(get_s(j,B),3))$ and not $eq(get_e(j,B),k))$ or
$(eq(get_s(j,B),4)$ and $Neq_q(k,get_q(j,B,n),j,B,n))$ or
$(eq(get_s(j,B),5)$ and $get_b(j,B)$ and not $eq(get_d(j,B),k))$
$Neq_q(k,q,j,B,n) = if(empty(q),Neq(k,j +_n 1,B,n),hd_b(q)$ and not $eq(hd(q),k))$

$Eq(k,j,B,n) =$
$((eq(get_s(j,B),2)$ or $eq(get_s(j,B),3))$ and $eq(get_e(j,B),k))$ or
$(eq(get_s(j,B),4)$ and $Eq_q(k,get_q(j,B,n),j,B,n))$ or
$(eq(get_s(j,B),5)$ and $get_b(j,B)$ and $eq(get_d(j,B),k))$
$Eq_q(k,q,j,B,n) = if(empty(q),Eq(k,j +_n 1,B,n),hd_b(q)$ and $eq(hd(q),k))$

3.4.8. Uniqueness of identifiers

Inv_8 expresses the fact that identifiers can occur in at most one active position in the ring of processes. It is provable with the help of Inv_6.

$Inv_8(B,n) = \forall_{k<n} Count(B,k,n) \leqslant 1$

where

$$Count(B,k,n) = \sum_{j<n} if(ActiveNode(k,j,B),1,0) + \sum_{j<n} ActiveChanOcc(k,get_q(j,B))$$

$$ActiveChanOcc(k,q) = if(empty(q),0,if(hd_b(q) \text{ and } eq(k,hd(q)),1,0) + ActiveChanOcc(k,tl(q)))$$

Intuitively, the definition of *Count* counts the number of times an identifier occurs in an active position, i.e., in a position such that the identifier can be transmitted and received by another process and later replace the d value of that process. An identifier in an active position can either occur in a variable (*ActiveNode*) or in a channel (*ActiveChanOcc*).

3.4.9. *Identifier travel creates relay processes*

Inv_9 points out that if two processes contain the same identifier (k) then the processes in between are guaranteed to be in state 4 and the connecting channels all empty. It is provable using Inv_8.

$$\begin{aligned}
&Inv_9(B,n) = \\
&\forall_{k<n}\forall_{i<n}(eq(get_s(i,B),1) \text{ or } eq(get_s(i,B),2)) \text{ and } eq(get_d(i,B),k) \rightarrow \\
&\quad (\forall_{j<n} eq(get_s(j,B),0) \rightarrow \text{not } eq(get_d(j,B),k)) \text{ and} \\
&\quad (\forall_{j<n} ActiveNode(k,j,B) \rightarrow empty(i,B) \text{ and } EmptyNodes(i,j,n,B)) \text{ and} \\
&\quad (\forall_{j<n} ActiveChan(k,get_q(j,B)) \rightarrow \\
&\qquad eq(hd(j,B),k) \text{ and } hd_b(j,B) \text{ and} \\
&\qquad if(eq(i,j),\mathrm{t},eq(get_s(j,B),4) \text{ and } empty(i,B) \text{ and } EmptyNodes(i,j,n,B)))
\end{aligned}$$

where

$$\begin{aligned}
&EmptyNode(j,B) = eq(get_s(j,B),4) \text{ and } empty(j,B) \\
&EmptyNodes(i,j,n,B) = \\
&\quad if(i<j,\forall_{i<l<j} EmptyNode(l,B),(\forall_{l<j} EmptyNode(l,B)) \text{ and} \\
&\quad (\forall_{i<l<n} EmptyNode(l,B)))
\end{aligned}$$

3.5. *Convergence of the protocol*

In this section we prove that the linear process X is convergent, i.e., that we can find a decreasing measure on the data parameter over the τ-steps in the X process operator. This result implies that all sequences of τ-steps are finite, which is a necessary condition for applying the Concrete Invariant Corollary. We prove that the function *Meas* defined below is a decreasing measure, and thus proving convergence.

$$\begin{aligned}
&Meas(B,n) \\
&\quad = \sum_{j<n}[if(eq(get_s(j,B),0),(n-get_d(j,B)+2)\,6\,n^3, \\
&\qquad if(eq(get_s(j,B),1 \text{ or } eq(get_s(j,B),2),(1+n-get_d(j,B))\,6\,n^3+3\,n^3, \\
&\qquad if(eq(get_s(j,B),3),(1+n-get_d(j,B))\,6\,n^3,0))))]
\end{aligned}$$

$$+\sum_{j<n}\sum_{k<size(get_q(j,B))} Term(j,B,k)$$
$$+\sum_{j<n} if(eq(get_s(j,B),5),1+Term(j+_n 1,B,size(get_q(j,B))),0).$$

$Term(j,B,st)$
$$+if([eq(st,1) \text{ and } (eq(get_s(j,B),0) \text{ or } eq(get_s(j,B),3),1))] \text{ or } [eq(st,0) \text{ and } get_s(j,B) \leqslant 3],1,$$
$$2+Term(j+_n 1,B,size(get_q(j,B)))+st$$
$$+if(eq(get_s(j,B),5),1,0)-if(eq(get_s(j,B),1,3),1,0)).$$

We have a sequence of theorems that are useful to show that $Meas(B,n)$ shows that all τ-sequences in X are finite.

Lemma 3.4. *If* $n>0$, $0\leqslant j,k<n$ *and*

$$st<\sum_{i=j}^{k}[if(eq(get_s(i,B),1) \text{ or } eq(get_s(i,B),3),1,0)-if(eq(get_s(i,B),5),1,0)$$
$$-size(get_q(i-_n 1,B))]+size(get_q(j-_n 1,B)).$$

then

1. $Term(j,B,st)\leqslant 2(k-_n j)+1$.
2. *If* $get_s(j,B)=5$ *and* $B'=in_q(d,b,j,upd_s(4,j,B))$ *then* $Term(i,B',st)=Term(i,B,st)$.
3. *If* $get_s(j,B)=4$, $B'=untoe(j-_n 1,upd_s(5,j,B))$ *then* $Term(i,B',st)\leqslant Term(i,B,st+if(eq(i,j),1,0))$.

Proof. All statements are proven by induction on $(k-_n j)$. □

Corollary 3.5.

1. *For* $n>0$ *and* $0\leqslant k<n$ *we find* $Term(k,B,st)<2n$ *provided* $st<size(get_q(k,B))$.
2. *If* $get_s(j,B)=5$, $B'=in_q(d,b,j,upd_s(4,j,B))$ *and* $st<size(get_q(i,B))$ *then* $Term(i,B',st)\leqslant Term(i,B,st)$.
3. *If* $get_s(j,B)=4$, $st<size(get_q(i,B))$, $i\neq j$, $B'=untoe(j-_n 1,upd_s(5,j,B))$ *then* $Term(i,B',st)\leqslant Term(i,B,st)$.

Proof. Respectively, instantiate case 1 of Lemma 3.4 with $j=k+_n 1$; case 1 with $k=l-_n 1$ and $j=l+_n 1$; case 2 with $j=k+_n 1$ and at last case 3 with $j=k+_n 1$. □

Lemma 3.6.

$$get_s(j,B)=0\rightarrow Meas(upd_s(1,j,B),n)+3n^3\leqslant Meas(B,n),$$
$$get_s(j,B)=1\rightarrow Meas(upd_s(2,j,B),n)=Meas(B,n),$$
$$get_s(j,B)=2\rightarrow Meas(upd_s(3,j,B),n)+3n^3\leqslant Meas(B,n),$$
$$get_s(j,B)=3\rightarrow Meas(upd_s(0,j,B),n)\leqslant Meas(B,n),$$

$$get_s(j,B) = 3 \rightarrow Meas(upd_s(4,j,B),n) < Meas(B,n),$$

$$get_s(j,B) = 0 \rightarrow Meas(in_q(get_d(j,B),b,j,B),n) < Meas(T,n) + 3n^3,$$

$$get_s(j,B) = 2 \rightarrow Meas(in_q(get_e(j,B),b,j,B),n) < Meas(B,n) + 3n^3,$$

$$get_s(j,T) = 1 \rightarrow Meas(untoe(j -_n 1,B),n) < Meas(B,n),$$

$$get_s(j,B) = 3 \rightarrow Meas(untoe(j -_n 1,B),n) < Meas(B,n),$$

$$get_s(j,B) = 4 \rightarrow Meas(upd_s(5,j,B),n) < Meas(B,n),$$

$$get_s(j,B) = 5 \rightarrow Meas(upd_s(4,j,B),n) < Meas(B,n),$$

$$get_s(j,B) = 4 \rightarrow Meas(in_q(get_d(j,B),b,j,B),n) < Meas(B,n),$$

$$get_s(j,B) = 5 \rightarrow Meas(untoe(j -_n 1,B),n) < Meas(B,n).$$

Theorem 3.7. *X is convergent.*

Proof. This follows as with the help of Lemma 3.6 it is straightforward to see that $Meas(B,n)$ is a decreasing measure. □

Remark 3.8. The measure *Meas* is certainly not optimal. It suggest that the algorithm requires about $6n^4(n+2)$ actions to select a leader. This is a very rough measure; looking at the far sharper bound in [10] suggests that the bound can actually be improved to $4n \log_2 n + 2n$ actions. However, we did not try this yet.

3.6. Final calculations

We now prove the following crucial lemma that links the *leader* action to X. But first we provide an auxiliary function that expresses that no process $j < n$ is in state 6.

Definition 3.9.

$$nonsix(B,n) = \forall_{j<n} \text{not } eq(get_s(j,B),6).$$

Lemma 3.10. *The invariants $Inv_1(B,n),\ldots,Inv_4(B,n)$ imply*:

$$X(B,n) = (leader\,\delta \triangleleft nonsix(B,n) \triangleright \delta) \triangleleft FC(B,n) \triangleright \tau(leader\,\delta \triangleleft nonsix(B,n) \triangleright \delta).$$

Proof. We show assuming the invariants $Inv_1(B,n),\ldots,Inv_4(B,n)$ that

$$\lambda B : Table, n : Nat.(leader\,\delta \triangleleft nonsix(B,n) \triangleright \delta) \triangleleft FC(B,n)$$

$$\triangleright \tau(leader\,\delta \triangleleft nonsix(B,n) \triangleright \delta)$$

is a solution for X in (I). As (I) is convergent, the lemma follows from the Concrete Invariant Corollary (see [4]). First suppose $FC(B,n)$ holds. This means that we must show that

$$\begin{aligned} &leader\ \delta \triangleleft nonsix(B,n) \triangleright \delta \\ &= \sum_{j:Nat} leader\,(leader\ \delta \triangleleft nonsix(upd_s(6,j,B)) \triangleright \delta) \\ &\quad \triangleleft eq(get_s(j,B),2) \text{ and } eq(get_d(j,B),get_e(j,B)) \text{ and } j < n \triangleright \delta. \end{aligned} \tag{1}$$

Note that it follows from $FC(B,n)$ that the other summands of (I) may be omitted. As $nonsix(mathitu\,pd_s(6,j,B)) = \mathbf{f}$, Eq. (1) reduces to:

$$\begin{aligned} &leader\ \delta \triangleleft nonsix(B,n) \triangleright \delta \\ &= \sum_{j:Nat} leader\ \delta \triangleleft eq(get_s(j,B),2) \text{ and } eq(get_d(j,B),get_e(j,B)) \text{ and } j < n \triangleright \delta. \end{aligned} \tag{2}$$

Now assume $nonsix(B,n)$. From $FC(B,n)$ and $Inv_1(B,n)$ is follows that

$$\forall_{j<n} 1 \leqslant get_s(j,B) \leqslant 4. \tag{3}$$

First we show that $\exists_{j<n} eq(get_s(j,B),2)$ and $eq(get_d(j,B),get_e(j,B))$. Now suppose

$$\exists_{j<n} eq(get_s(j,B),1) \text{ or } eq(get_s(j,B),3).$$

Hence, using $Inv_2(B,n)$ and $nproc(B,1,n)+nproc(B,3,n) > 0$ and (3), it follows that $qsizes(B,n) > 0$. Hence, $\exists_{j<n} size(j -_n 1,B) > 0$. Hence, using the focus condition and $Inv_1(B,n)$:

$$\exists_{j<n} eq(get_s(j,B),2) \text{ and } eq(get_d(j,B),get_e(j,B)).$$

Now suppose

$$\text{not } \exists_{j<n} eq(get_s(j,B),1) \text{ or } eq(get_s(j,B),3).$$

Hence, using (3) it follows that

$$\forall_{j<n} eq(get_s(j,B),2) \text{ or } eq(get_s(j,B),4). \tag{4}$$

Now assume

$$\forall_{j<n} eq(get_s(j,B),4).$$

But this contradicts $Inv_4(B,n)$ in conjunction with $Inv_2(B,n)$. Hence, using (4) it follows that

$$\exists_{j<n} eq(get_s(j,B),2).$$

From this and $FC(B,n)$ it follows that

$$\exists_{j<n} eq(get_s(j,B),2) \text{ and } eq(get_d(j,B),get_e(j,B)).$$

Hence, using SUM3 (see appendix) the right-hand side of (2) has a summand

$$leader\,\delta. \tag{5}$$

But using some straightforward calculations (5) has the right-hand side of (2) as a summand. Hence, if $nonsix(B,n)$ then (2) is equivalent to

$$leader\,\delta = leader\,\delta$$

which is clearly a tautology. Now assume not $nonsix(B,n)$. Hence, $\exists_{j\leqslant n} eq(get_s(j,B),6)$. Using $Inv_3(B,n)$ it follows that

$$\forall_{j<n} eq(get_s(j,B),4) \text{ or } eq(get_s(j,B),6).$$

Hence (2) reduces to

$$\delta = \delta$$

which is clearly true. Now suppose the focus condition does not hold, i.e., not $FC(B,n)$. We find (where we use that $n > 0$ and Milner's second τ-law (T2)):

$$\begin{aligned}
&\tau(leader\,\delta \lhd nonsix(B,n) \rhd \delta) = \\
&\sum_{j:Nat} \tau(leader\,\delta \lhd nonsix(B,n) \rhd \delta) \lhd j < n \rhd \delta + \\
&\sum_{j:Nat} leader\,\delta \\
&\quad \lhd nonsix(B,n) \text{ and } eq(get_s(j,B),2) \text{ and } eq(get_d(j,B),get_e(j,B)) \text{ and } j < n \rhd \delta
\end{aligned} \tag{6}$$

Now note that it follows from $Inv_3(B,n)$ that if $\exists_{j<n} eq(get_s(j,B),2)$, then $nonsix(B,n)$. So, (6) reduces to:

$$\begin{aligned}
&\textstyle\sum_{j:Nat} \tau(leader\,\delta \lhd nonsix(B,n) \text{ and } j < n \rhd \delta) + \\
&\textstyle\sum_{j:Nat} leader\,\delta \lhd eq(get_s(j,B),2) \text{ and } eq(get_d(j,B),get_e(j,B)) \text{ and } j < n \rhd \delta = \\
&(\textstyle\sum_{j:Nat} \tau(leader\,\delta \lhd nonsix(B,n) \text{ and } j < n \rhd \delta) \lhd \text{not } FC(B,n) \rhd \delta) + \\
&\textstyle\sum_{j:Nat} leader\,\delta \lhd eq(get_s(j,B),2) \text{ and } eq(get_d(j,B),get_e(j,B)) \text{ and } j < n \rhd \delta = \\
&\textstyle\sum_{j:Nat} \tau(leader\,\delta \\
&\quad \lhd nonsix(upd_s(1,j,in_q(get_d(j,B),j,B)),n) \rhd \delta) \lhd eq(get_s(j,B),0) \text{ and} \\
&\qquad j < n \rhd \delta + \\
&\textstyle\sum_{j:Nat} \tau(leader\,\delta \lhd nonsix(untoe(j -_n 1, upd_e(toe(j -_n 1,B),j,upd_s(2,j,B))),n) \rhd \delta) \\
&\quad \lhd eq(get_s(j,B),1) \text{ and not } empty(j -_n 1,B) \text{ and } j < n \rhd \delta + \\
&\textstyle\sum_{j:Nat} leader(leader\,\delta \lhd nonsix(upd_s(6,j,B),n) \rhd \delta) \\
&\quad \lhd eq(get_s(j,B),2) \text{ and } eq(get_d(j,B),get_e(j,B)) \text{ and } j < n \rhd \delta + \\
&\textstyle\sum_{j:Nat} \tau(leader\,\delta \lhd nonsix(upd_s(3,j,in(get_e(j,B),j,B)),n) \rhd \delta) \\
&\quad \lhd eq(get_s(j,B),2) \text{ and not } eq(get_d(j,B),get_e(j,B)) \text{ and } j < n \rhd \delta + \\
&\textstyle\sum_{j:Nat} (\tau(leader\,\delta \lhd nonsix(untoe(j -_n 1, upd_d(get_e(j,B),j,upd_s(0,j,B))),n) \rhd \delta) \\
&\qquad \lhd get_e(j,B) > max(get_d(j,B),toe(j -_n 1,B)) \rhd \\
&\qquad \tau(leader\,\delta \lhd nonsix(untoe(j -_n 1, upd_s(4,j,B)),n) \rhd \delta)) \\
&\qquad \lhd eq(get_s(j,B),3) \text{ and not } empty(j -_n 1,B) \text{ and } j < n \rhd \delta +
\end{aligned}$$

$\sum_{j:Nat} \tau(leader\,\delta \lhd nonsix(untoe(j -_n 1, mathitu\,pd_d(toe(j -_n 1, B), j,$
$upd_s(5, j, B))), n) \rhd \delta)$
$\lhd eq(get_s(j, B), 4)$ and not $empty(j -_n 1, B)$ and $j < n \rhd \delta +$
$\sum_{j:Nat} \tau(leader\,\delta \lhd nonsix(in_q(get_d(j, B), j, upd_s(4, j, B)), n) \rhd \delta)$
$\lhd eq(get_s(j, B), 5)$ and $j < n \rhd \delta$

Because $FC(B, n) = \mathsf{f}$, nearly all the summands given above are equal to δ. □

3.6.1. Proving Theorem 2.1

Finally we are ready to prove that the main theorem of the paper holds, i.e.,

$$n > 0 \rightarrow Spec(n) = \tau\,leader\,\delta$$

Proof. Using Lemma 3.3 we know

$$Spec(n) = X(init(n), n).$$

From Lemma 3.10 it then follows that

$Spec(n) =$
$(leader\,\delta \lhd nonsix(init(n), n) \rhd \delta) \lhd FC(init(n), n)$
$\rhd \tau(leader\,\delta \lhd nonsix(init(n), n) \rhd \delta).$

However, $FC(init(n), n)$ is not true if $n > 0$ while $nonsix(init(n), n)$ is true. Therefore

$$n > 0 \rightarrow Spec(n) = \tau\,leader\,\delta$$

is true. □

4. Conclusion

We have outlined a formal proof of the correctness of the leader election or extrema finding protocol of Dolev et al. in μCRL. The proof is now ready to be proof checked conform [2, 15, 17, 21].

It is shown that process algebra, in particular μCRL, is suited to prove correctness of non-trivial protocols. A drawback of the current verification is that it is rather complex and lengthy. A possible lead towards improvement is given by Frits Vaandrager in [22], where by using the notion of confluency (see e.g. [19]) one only needs to consider one trace to establish correctness. Currently we are formalising this notion in [14]. We expect that using this idea our proof can be simplified significantly.

Appendix A. An overview of the proof theory for μCRL

We provide here a very short account of the axioms that have been used. We also give the Concrete Invariant Corollary for referencing purposes.

Table 1
The axioms of ACP in μCRL

A1	$x+y=y+x$	CF	$n(\bar{t})\mid m(\bar{t})$
A2	$A2x+(y+z)=(x+y)+z$		$=\begin{cases}\gamma(n,m)(\bar{t}) & \text{if } \gamma(n,m)\downarrow\\ \delta & \text{otherwise}\end{cases}$
A3	$x+x=x$		
A4	$(x+y)\cdot z=x\cdot z+y\cdot z$		
A5	$(x\cdot y)\cdot z=x\cdot(y\cdot z)$		
A6	$x+\delta=x$		
A7	$\delta\cdot x=\delta$	CD1	$\delta\mid x=\delta$
		CD2	$x\mid\delta=\delta$
CM1	$x\parallel y=x\between y+y\between x+x\mid y$	CT1	$\tau\mid x=\delta$
CM2	$a\between x=a\cdot x$	CT2	$x\mid\tau=\delta$
CM3	$a\cdot x\between y=a\cdot(x\parallel y)$		
CM4	$(x+y)\between z=x\between z+y\between z$	DD	$\partial_H(\delta)=\delta$
CM5	$a\cdot x\mid b=(a\mid b)\cdot x$	DT	$\partial_H(\tau)=\tau$
CM6	$a\mid b\cdot x=(a\mid b)\cdot x$	D1	$\partial_H(n(\bar{t}))=n(\bar{t})$ if $n\notin H$
CM7	$a\cdot x\mid b\cdot y=(a\mid b)\cdot(x\parallel y)$	D2	$\partial_H(n(\bar{t}))=\delta$ if $n\in H$
CM8	$(x+y)\mid z=x\mid z+y\mid z$	D3	$\partial_H(x+y)=\partial_H(x)+\partial_H(y)$
CM9	$x\mid(y+z)=x\mid y+x\mid z$	D4	$\partial_H(x\cdot y)=\partial_H(x)\cdot\partial_H(y)$

Table 2
Axioms of standard concurrency (SC)

$(x\between y)\between z=x\between(y\parallel z)$	$(x\mid y)\mid z=x\mid(y\mid z)$	
$x\parallel\delta=x\delta$	$x\mid(ay\between z)=(x\mid ay)\between z$	
$x\mid y=y\mid x$	$x\mid(y\mid z)=\delta$	Handshaking

All the process algebra axioms used to prove the leader election protocol can be found in Tables 1–6. We do not explain the axioms (see [1, 4, 13]) but only include them to give an exact and complete overview of the axioms that we used. Actually, the renaming axioms are superfluous, but have been included for completeness.

Besides the axioms we have used the Concrete Invariant Corollary [4] that says that if two processes p and q can be shown a solution of a well founded recursive specification using an invariant, then p and q are equal, for all starting states where the invariant holds. It is convenient to use linear process operators, which are functions that transform a parameterised process into another parameterised process. If such an operator is well founded, it has a unique solution, and henceforth defines a process. Note that if a linear process operator is applied to a process name, it becomes a process in Unity format.

Definition A.1. A linear process operator Ψ is an expression of the form

$$\lambda p:D\to\mathbb{P}.\lambda d:D.\sum_{i\in I}\sum_{e_i:D_i}c_i(f_i(d,e_i))\cdot p(g_i(d,e_i))\lhd b_i(d,e_i)\rhd\delta+\sum_{i\in I'}\sum_{e_i:D'_i}c'_i(f'_i(d,e_i))\lhd b'_i(d,e_i)\rhd\delta$$

Table 3
Axioms for abstraction

TID	$\tau_I(\delta) = \delta$	
TIT	$\tau_I(\tau) = \tau$	
TI1	$\tau_I(n(\bar{t})) = n(\bar{t})$	if $n \notin I$
TI2	$\tau_I(n(\bar{t})) = \tau$	if $n \in I$
TI3	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$	
TI4	$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$	

Table 4
Axioms for summation.

SUM1	$\sum_{d:D}(p) = p$	if d not free in p
SUM2	$\sum_{d:D}(p) = \sum_{e:D}(p[e/d])$	if e not free in p
SUM3	$\sum_{d:D}(p) = \sum_{d:D}(p) + p$	
SUM4	$\sum_{d:D}(p_1 + p_2) = \sum_{d:D}(p_1) + \sum_{d:D}(p_2)$	
SUM5	$\sum_{d:D}(p_1 \cdot p_2) = \sum_{d:D}(p_1) \cdot p_2$	if d not free in p_2
SUM6	$\sum_{d:D}(p_1 \Vert p_2) = \sum_{d:D}(p_1) \Vert p_2$	if d not free in p_2
SUM7	$\sum_{d:D}(p_1 \mid p_2) = \sum_{d:D}(p_1) \mid p_2$	if d not free in p_2
SUM8	$\sum_{d:D}(\partial_H(p)) = \partial_H(\sum_{d:D}(p))$	
SUM9	$\sum_{d:D}(\tau_I(p)) = \tau_I(\sum_{d:D}(p))$	
SUM11	$\dfrac{p_1 \overset{\mathscr{D}}{=} p_2}{\sum_{d:D}(p_1) = \sum_{d:D}(p_2)}$	provided d not free in the assumptions of $\mathscr{D}$

Table 5
Axioms for the conditional construct and **Bool**

COND1	$x \lhd \mathsf{t} \rhd y = x$
COND2	$x \lhd \mathsf{f} \rhd y = y$
BOOL1	$\neg(\mathsf{t} = \mathsf{f})$
BOOL2	$\neg(b = \mathsf{t}) \rightarrow b = \mathsf{f}$

for some finite index sets I, I', actions c_i, c_i', data types D_i, D_i', D_{c_i} and $D_{c_i'}$, functions $f_i : D \rightarrow D_i \rightarrow D_{c_i}$, $g_i : D \rightarrow D_i \rightarrow D$, $b_i : D \rightarrow D_i' \rightarrow \mathbf{Bool}$, $f_i' : D \rightarrow D_i' \rightarrow D_{c_i'}$, $b_i' : D \rightarrow D_i' \rightarrow \mathbf{Bool}$.

Definition A.2. A linear process operator (LPO) Ψ written in the form above is called *convergent* iff there is a well-founded ordering $<$ on D such that $g_i(d, e_i) < d$ for all $d \in D$, $i \in I$ and $e_i \in D_i$ with $c_i = \tau$ and $b_i(d, e_i)$.

Table 6
Some τ-laws

B1	$x\tau = x$
B2	$\tau x = \tau x + x$

Corollary A.3 (Concrete invariant corollary). *Assume*

$$\Phi = \lambda p{:}D \to \mathbb{P}.\lambda d{:}D.\sum_{j\in J}\sum_{e_j:D_j} c_j(f_j(d,e_j)) \cdot p(g_j(d,e_j)) \lhd b_j(d,e_j) \rhd \delta + \sum_{j\in J'}\sum_{e_j:D'_j} c'_j(f'_j(d,e_j)) \lhd b'_j(d,e_j) \rhd \delta$$

is a LPO. If for some predicate $I : D \to \mathbf{Bool}$

$\lambda pd.\Phi pd \lhd I(d) \rhd \delta$ is convergent, and

$I(d) \wedge b_j(d,e_j) \to I(g_j(d,e_j))$ for all $j \in J$, $d \in D$ and $e_j \in D_j$,

i.e. I is an invariant of Φ, *and for some* $q : D \to \mathbb{P}$, $q' : D \to \mathbb{P}$ *we have*

$I(d) \to q(d) = \Phi qd$,

$I(d) \to q'(d) = \Phi q'd$,

then

$I(d) \to q(d) = q'(d)$.

Appendix B. Data types

B.1. Booleans

sort	**Bool**
cons	$\mathsf{t}, \mathsf{f} :\to$ **Bool**
func	not : **Bool** $\to$ **Bool**
	and, or, eq : **Bool** $\times$ **Bool** $\to$ **Bool**
	if : **Bool** $\times$ **Bool** $\times$ **Bool** $\to$ **Bool**
var	b, b' : **Bool**
rew	not $\mathsf{t} = \mathsf{f}$
	not $\mathsf{f} = \mathsf{t}$
	t and $b = b$
	f and $b = \mathsf{f}$
	t or $b = \mathsf{t}$
	f or $b = b$
	$eq(\mathsf{t},\mathsf{t}) = \mathsf{t}$
	$eq(\mathsf{f},\mathsf{f}) = \mathsf{t}$
	$eq(\mathsf{t},\mathsf{f}) = \mathsf{f}$

$eq(\mathsf{f},\mathsf{t}) = \mathsf{f}$
$if(\mathsf{t},b,b') = b$
$if(\mathsf{f},b,b') = b'$

B.2. Natural numbers

sort Nat
cons $0 :\rightarrow Nat$
$S : Nat \rightarrow Nat$
func $1,2,3,4,5,6 :\rightarrow Nat$
$P : Nat \rightarrow Nat$
$even : Nat \rightarrow \mathbf{Bool}$
$+,-,*,max : Nat \times Nat \rightarrow Nat$
$eq,\ \geqslant,\ \leqslant,\ <,\ > \ : Nat \times Nat \rightarrow \mathbf{Bool}$
$if : \mathbf{Bool} \times Nat \times Nat \rightarrow Nat$
var $n,m : Nat$
rew $1 = S(0)$
$2 = S(1)$
$3 = S(2)$
$4 = S(3)$
$5 = S(4)$
$6 = S(5)$
$P(0) = 0$
$P(S(n)) = n$
$even(0) = \mathsf{t}$
$even(S(0)) = \mathsf{f}$
$even(S(S(n))) = even(n)$
$n + 0 = n$
$n + S(m) = S(n + m)$
$n - 0 = n$
$n - S(m) = P(n - m)$
$n * 0 = 0$
$n * S(m) = n + n * m$
$max(n,m) = if(n \geqslant m,n,m)$
$eq(0,0) = \mathsf{t}$
$eq(0,S(n)) = \mathsf{f}$
$eq(S(n),0) = \mathsf{f}$
$eq(S(n),S(m)) = eq(n,m)$
$n \geqslant 0 = \mathsf{t}$
$0 \geqslant S(n) = \mathsf{f}$
$n \geqslant S(m) = n \geqslant m$
$n \leqslant m = m \geqslant n$
$n > m = n \geqslant S(m)$

$n < m = S(n) \leqslant m$
$if(\mathrm{t}, n, m) = n$
$if(\mathrm{f}, n, m) = m$

B.3. Modulo arithmetic

func $mod : Nat \times Nat \rightarrow Nat$
$+, - : Nat \times Nat \times Nat \rightarrow Nat$
var $k, m, n : Nat$
rew $m \, mod \, 0 = m$
$m \, mod \, S(n) = if(m \geqslant S(n), m - S(n) \, mod \, S(n), m)$
$k +_n m = k + m \, mod \, n$
$k -_n m = if(k \, mod \, n \geqslant m \, mod \, n, k \, mod \, n - m \, mod \, n, n - m \, mod \, n - k \, mod \, n)$

B.4. Queues

We use two kind of queues which are subtlely different. The first is of sort *Queue* with the usual operations. The second is of sort $Queue_b$ which is similar to *Queue* except that a boolean is added for technical purposes. The specification of $Queue_b$ is given below. We do not present the data type *Queue* here because it can be considered as a simple instance of $Queue_b$ as follows: omit the functions hd_b, toe_b and remove all boolean arguments. For example, $in : Nat \times \mathbf{Bool} \times Queue_b \rightarrow Queue_b$ corresponds with $in : Nat \times Queue \rightarrow Queue$.

sort $Queue_b$
cons $q_0 :\rightarrow Queue_b$
$in : Nat \times \mathbf{Bool} \times Queue_b \rightarrow Queue_b$
func $rem : Nat \times Queue_b \rightarrow Queue_b$
$tl, untoe : Queue_b \rightarrow Queue_b$
$con : Queue_b \times Queue_b \rightarrow Queue_b$
$hd, toe : Queue_b \rightarrow Nat$
$hd_b : Queue_b \rightarrow \mathbf{Bool}$
$toe_b : Queue_b \rightarrow \mathbf{Bool}$
$eq : Queue_b \times Queue_b \rightarrow \mathbf{Bool}$
$empty : Queue_b \rightarrow \mathbf{Bool}$
$test : Nat \times Queue_b \rightarrow \mathbf{Bool}$
$size : Queue_b \rightarrow Nat$
$if : \mathbf{Bool} \times Queue_b \times Queue_b \rightarrow Queue_b$
var $d, e : Nat$
$b, c : \mathbf{Bool}$
$q, r : Queue_b$
rew $rem(d, q_0) = q_0$
$rem(d, in(e, b, q)) = if(eq(d, e), q, in(e, b, rem(d, q)))$
$tl(q_0) = q_0$

$tl(in(d,b,q)) = q$
$untoe(q_0) = q_0$
$untoe(in(d,b,q_0)) = q_0$
$untoe(in(d,b,in(e,c,q))) = in(d,b,untoe(in(e,c,q)))$
$con(q_0,q) = q$
$con(in(d,b,q),r) = in(d,b,con(q,r))$
$hd(q_0) = 0$
$hd(in(d,b,q)) = d$
$hd_b(q_0) = \mathsf{f}$
$hd_b(in(d,b,q)) = b$
$toe(q_0) = 0$
$toe(in(d,b,q_0)) = d$
$toe(in(d,b,in(e,c,q))) = toe(in(e,c,q))$
$toe_b(q_0) = \mathsf{f}$
$toe_b(in(d,b,q_0)) = b$
$toe_b(in(d,b,in(e,c,q))) = toe_b(in(e,c,q))$
$eq(q_0,q_0) = \mathsf{t}$
$eq(q_0,in(d,b,q)) = \mathsf{f}$
$eq(in(d,b,q),q_0) = \mathsf{f}$
$eq(in(d,b,q),in(e,c,r)) = eq(d,e)$ and $eq(b,c)$ and $eq(q,r)$
$empty(q) = eq(size(q),0)$
$test(d,q_0) = \mathsf{f}$
$test(d,in(e,b,q)) = eq(d,e)$ or $test(d,q)$
$size(q_0) = 0$
$size(in(d,b,q)) = S(size(q))$
$if(\mathsf{t},q,r) = q$
$if(\mathsf{f},q,r) = r$

B.5. Protocol states

sort $Table$

cons $t_0 :\rightarrow Table$
$in : Nat \times Nat \times Nat \times Nat \times \mathbf{Bool} \times Queue_b \times Table \rightarrow Table$

func $init : Nat \rightarrow Table$
$get_d, get_e, get_s : Nat \times Table \rightarrow Nat$
$get_b : Nat \times Table \rightarrow \mathbf{Bool}$
$get_q : Nat \times Table \rightarrow Queue_b$
$upd_d, upd_e, upd_s : Nat \times Nat \times Table \rightarrow Table$
$upd_b : \mathbf{Bool} \times Nat \times Table \rightarrow Table$
$upd_q : Queue_b \times Nat \times Table \rightarrow Table$
$test : Nat \times Table \rightarrow \mathbf{Bool}$
$in_q : Nat \times \mathbf{Bool} \times Nat \times Table \rightarrow Table$
$hd : Nat \times Table \rightarrow Nat$

$hd_b : Nat \times Table \to \mathbf{Bool}$
$hd_i : Table \to Nat$
$toe : Nat \times Table \to Nat$
$toe_b : Nat \times Table \to \mathbf{Bool}$
$untoe : Nat \times Table \to Table$
$empty : Nat \times Table \to \mathbf{Bool}$
$tl : Table \to Table$
$rem : Nat \times Table \to Table$
$UniqueIndex : Table \to \mathbf{Bool}$
$empty : Table \to \mathbf{Bool}$
$if : \mathbf{Bool} \times Table \times Table \to Table$

var $d, e, s, v, i, j, n : Nat$
$B, B' : Table$
$b, b' : \mathbf{Bool}$
$q, q' : Queue_b$

rew $init(n) = if(eq(n,0), t_0, in(n-1, id(n-1), 0, 0, \mathsf{f}, q_0, init(n-1)))$
$get_d(i, t_0) = 0$
$get_d(i, in(j,d,e,s,b,q,B)) = if(eq(i,j), d, get_d(i,B))$
$get_e(i, t_0) = 0$
$get_e(i, in(j,d,e,s,b,q,B)) = if(eq(i,j), e, get_e(i,B))$
$get_s(i, t_0) = 0$
$get_s(i, in(j,d,e,s,b,q,B)) = if(eq(i,j), s, get_s(i,B))$
$get_b(i, t_0) = \mathsf{f}$
$get_b(i, in(j,d,e,s,b,q,B)) = if(eq(i,j), b, get_b(i,B))$
$get_q(i, t_0) = q_0$
$get_q(i, in(j,d,e,s,b,q,B)) = if(eq(i,j), q, get_q(i,B))$
$upd_d(v,i,t_0) = in(i,v,0,0,\mathsf{f},q_0,t_0)$
$upd_d(v,i,in(j,d,e,s,b,q,B)) = if(eq(i,j),$
$\quad in(j,v,e,s,b,q,B), in(j,d,e,s,b,q,upd_d(v,i,B)))$
$upd_e(v,i,t_0) = in(i,0,v,0,\mathsf{f},q_0,t_0)$
$upd_e(v,i,in(j,d,e,s,b,q,B)) = if(eq(i,j),$
$\quad in(j,d,v,s,b,q,B), in(j,d,e,s,b,q,upd_e(v,i,B)))$
$upd_s(s,i,t_0) = in(i,0,0,s,\mathsf{f},q_0,t_0)$
$upd_s(v,i,in(j,d,e,s,b,q,B)) = if(eq(i,j),$
$\quad in(j,d,e,v,b,q,B), in(j,d,e,s,b,q,upd_s(v,i,B)))$
$upd_b(b',i,t_0) = in(i,0,0,0,b',q_0,t_0)$
$upd_b(b',i,in(j,d,e,s,b,q,B)) = if(eq(i,j),$
$\quad in(j,d,e,s,b',q,B), in(j,d,e,s,b,q,upd_b(b',i,B)))$
$upd_q(q',i,t_0) = in(i,0,0,0,\mathsf{f},q',t_0)$
$upd_q(q',i,in(j,d,e,s,b,q,B)) = if(eq(i,j),$
$\quad in(j,d,e,s,b,q',B), in(j,d,e,s,b,q,upd_q(q',i,B)))$
$test(i,t_0) = \mathsf{f}$
$test(i,in(j,d,e,s,b,q,B)) = eq(i,j)$ or $test(i,B)$

$$untoe(i,B) = upd_q(untoe(get_q(i,B)),i,B)$$
$$hd(i,B) = hd(get_q(i,B))$$
$$hd_b(i,B) = hd_b(get_q(i,B))$$
$$hd_i(t_0) = 0$$
$$hd_i(in(j,d,e,s,b,q,B)) = j$$
$$toe(i,B) = toe(get_q(i,B))$$
$$toe_b(i,B) = toe_b(get_q(i,B))$$
$$untoe(i,B) = upd_q(untoe(get_q(i,B)),i,B)$$
$$empty(i,B) = empty(get_q(i,B))$$
$$tl(t_0) = t_0$$
$$tl(in(j,d,e,s,b,q,B)) = B$$
$$rem(i,t_0) = t_0$$
$$rem(i,in(j,d,e,s,b,q,B)) = if(eq(i,j),B,in(j,d,e,s,b,q,rem(i,B)))$$
$$UniqueIndex(t_0) = \mathsf{t}$$
$$UniqueIndex(in(j,d,e,s,b,q,B)) = \text{not } test(j,B) \text{ and } UniqueIndex(B)$$
$$empty(t_0) = \mathsf{t}$$
$$empty(in(j,d,e,s,b,q,B)) = \mathsf{f}$$
$$if(\mathsf{t},B,B') = B$$
$$if(\mathsf{f},B,B') = B'$$

Acknowledgements

We thank Frits Vaandrager for pointing out this protocol to us. Also Marco Pouw is thanked for his suggestions for improvement.

References

[1] J.C.M. Baeten and W.P. Weijland, *Process Algebra*, Cambridge Tracts in Theoretical Computer Science, Vol. 18 (Cambridge Univ. Press, Cambridge, 1990).

[2] M.A. Bezem, R. Bol and J.F. Groote, A formal verification of the alternating bit protocol in the calculus of constructions (revised version). Original version appeared as M.A. Bezem and J.F. Groote, A formal verification of the alternating bit protocol in the calculus of constructions, Tech. Report 88, Logic Group Preprint Series, Utrecht University, March 1993.

[3] M.A. Bezem and J.F. Groote, A correctness proof of a one bit sliding window protocol in μCRL, *Computer J.* **37** (4) (1994) 289–307.

[4] M.A. Bezem and J.F. Groote, Invariants in process algebra with data, in: *Proc. CONCUR '94 Conf. on Concurrency Theory*, Lecture Notes in Computer Science, Vol. 836 (Springer, Berlin, 1994) 401–416.

[5] J.J. Brunekreef, Process specification in a UNITY format, in: A. Ponse, C. Verhoef and S.F.M. van Vlijmen, eds., *Proc. Workshop on Algebra of Communicating Processes ACP94*, Workshops in Computing (Springer, Berlin, 1995) 319–337.

[6] J.J. Brunekreef, J.-P. Katoen, R.L.C. Koymans and S. Mauw, Algebraic specification of dynamic leader election protocols in broadcast networks, in: A. Ponse, C. Verhoef and S.F.M. van Vlijmen, eds., *Proc. Workshop on Algebra of Communicating Processes ACP'94*, Workshops in Computing (Springer, Berlin, 1995) 338–358.

[7] J.J. Brunekreef, J.-P. Katoen, R.L.C. Koymans and S. Mauw, Design and analysis of dynamic leader election protocols in broadcast networks, *Distributed Comput.* **9** (4) (1996) 157–171.

[8] K.M. Chandy and J. Misra, *Parallel Program Design. A Foundation* (Addison-Wesley, Reading, MA, 1988).

[9] T. Coquand and G. Huet, The calculus of constructions, *Inform. and Control* (76) (1988).

[10] D. Dolev, M. Klawe and M. Rodeh, An O(n log n) unidirectional distributed algorithm for extrema finding in a circle, *J. Algorithms* (3): (1982) 245–260.

[11] J.F. Groote and H. Korver, A correctness proof of the bakery protocol in μCRL, in: A. Ponse, C. Verhoef and S.F.M. van Vlijmen, eds., *Proc. workshop on Algebra of Communicating Processes ACP94*, Workshops in Computing (Springer, Berlin, 1995) 63–105.

[12] J.F. Groote and A. Ponse, The syntax and semantics of μCRL, in: A. Ponse, C. Verhoef and S.F.M. van Vlijmen, eds., *Proc. workshop on Algebra of Communicating Processes ACP94*, Workshops in Computing (Springer, Berlin, 1995) 26–62. Tech. Report CS-R9076, CWI, Amsterdam, 1990.

[13] J.F. Groote and A. Ponse, Proof theory for μCRL: a language for processes with data, in: D.J. Andrews, J.F. Groote and C.A. Middelburg, eds., *Proc. Internat. Workshop on Semantics of Specification Languages*, Workshops in Computing (Springer, Berlin, 1994) 231–250.

[14] J.F. Groote and M.P.A. Sellink, Confluency for process verification, in: *Proc. CONCUR '95 Conf. on Concurrency Theory*, Lecture Notes in Computer Science, Vol. 962 (Springer, Berlin, 1995) 204–218.

[15] J.F. Groote and J. van de Pol, A bounded retransmission protocol for large data packets. A case study in computer checked algebraic verification, Tech. Report 100, Department of Philosophy, Utrecht University, 1993.

[16] H. Korver, *Protocol Verification in* μCRL, Ph.D. Thesis, University of Amsterdam, 1994.

[17] H. Korver and J. Springintveld, A computer-checked verification of Milner's scheduler, in: *Proc. 2nd Internat. Symp. on Theoretical Aspects of Computer Software, Sendai, Japan*, Lecture Notes in Computer Science, Vol. 769 (Springer, Berlin, 1994) 161–178.

[18] N.A. Lynch and M.R. Tuttle, An introduction to input/output automata, *CWI Quart.*, 2(3) (1989) 219–246.

[19] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Vol. 92 (Springer, Berlin, 1980).

[20] G.L. Peterson, An O($n \log n$) unidirectional algorithm for the circular extrema problem, *ACM Trans. Programming Languages Systems* **4** (4): (October 1982) 758–762.

[21] M.P.A. Sellink, Verifying process algebra proofs in type theory, in: D.J. Andrews, J.F. Groote, and C.A. Middelburg, eds. *Proc. Internat. Workshop on Semantics of Specification Languages*, Workshops in Computing (Springer, Berlin, 1994) 314–338.

[22] F.W. Vaandrager, Uitwerking take-home tentamen protocolverificatie, Unpublished manuscript, in Dutch, 1993.