Algebraic Specification of Dynamic Leader Election Protocols in Broadcast Networks

Jacob Brunekreef
Programming Research Group, University of Amsterdam
Amsterdam, The Netherlands

Joost-Pieter Katoen
Dept. of Computing Science, University of Twente
Enschede, The Netherlands

Ron Koymans
Philips Research Laboratories
Eindhoven, The Netherlands

Sjouke Mauw
Dept. of Math. and Comp. Sc., Eindhoven University of Technology
Eindhoven, The Netherlands

Abstract

The problem of leader election in distributed systems is considered. Components communicate by means of buffered broadcasting as opposed to usual point-to-point communication. In this paper three leader election protocols of increasing maturity are specified. We start with a simple leader election protocol, where an initial leader is present. In the second protocol this assumption is dropped. Eventually a fault-tolerant protocol is constructed, where components may crash and revive spontaneously.

Both the protocols and the required behaviour are formally specified in ACP. Some remarks are made about a formal verification of the protocols.

1 Introduction

In current distributed systems functions (or services) are offered by some dedicated component(s) in the system. Usually many components are capable to offer a certain functionality. However, at any moment only one component is allowed to actually offer the function. Therefore, one component —called the "leader"— must be elected to support that function. Sometimes it suffices to elect an arbitrary component, but for other functions it is important to elect the component which is best suited (according to some appropriate criteria) to perform that function.

The problem of leader election was originally coined by [16] in the late seventies. Various LE protocols have been developed since then, varying in network topology (ring [16, 9, 20], mesh, complete network [15, 2, 22], and so on), communication mechanism (asynchronous, synchronous), available topology information at processes ([17, 3]), and so forth. A few LE protocols are known that tolerate either communication link failures (see e.g. [1, 21]) or process failures ([13, 14, 18, 10]).

Realistic distributed systems are subject to failures. The problem of leader election thus becomes of practical interest when failures are anticipated. In this paper components behave *dynamically*—they may participate at arbitrary moments and stop participating spontaneously without notification to any other component. Crashed components may recover at any time. Thus, a leader has to be elected from a component set whose contents may change continuously. Components communicate with each other by exchanging messages via a *broadcast* network. This network is considered to be fully reliable. A broadcast message is received by all components except the sending component itself. Communication is supposed to be asynchronous and order-preserving.

In this paper we consider a leader election (LE) protocol which elects the most favourable component as leader. We assume a finite number of components. Each component has a fixed unique identity and a total ordering exists on these identities, known to all components. The leader is defined as the component with the largest identity among all participating components. We come to a fault-tolerant protocol in three steps, each step resulting in a LE protocol. We start in section 2 with rather strong —and unrealistic— assumptions about component and system behaviour: components are considered to be perfect and a leader is assumed to be present initially. A component may participate spontaneously, but once it does it remains to do so and does not crash. In section 3 the assumption of an initial leader is dropped. This leads to a fully symmetric protocol which uses a timeout mechanism to detect the absence of a leader. Finally, in section 4 a protocol is designed in which components may crash without giving any notification to other components.

Many communication protocols are *action–oriented*: their aim is to guarantee that each input–action containing a message is followed sooner or later by an output–action containing the same message. On the other hand, LE protocols are first of all *state–oriented*: the aim of these protocols is to reach a state in which the "best" component is elected as the current leader. Therefore, in this paper we have chosen a state–oriented specification style for the various protocols: processes are specified in the UNITY Format as presented in [7]. Furthermore, this choice is motivated by the fact that in this format the normalisation of the parallel composition of a large number of identical processes (the components in the protocol) is rather easy. This plays a role in the *verification* of the protocols, see section 5.

In the UNITY Format a process X is specified as follows:

$$\begin{array}{rll} X(D) = & & a_1 \cdot X(D_1) \lhd c_1 \rhd \delta \\ & + & \ldots \\ & + & a_m \cdot X(D_m) \lhd c_m \rhd \delta \\ & + & a_{m+1} \lhd c_{m+1} \rhd \delta \\ & + & \ldots \\ & + & a_n \lhd c_n \rhd \delta \end{array}$$

D denotes a parameter list, $D_1 \ldots D_m$ denote a parameter list with substitutions for some of the elements of D. $a_1 \ldots a_n$ denote atomic actions. $c_1 \ldots c_n$ denote boolean conditions, based on elements of D. Information about the process state is put in the data parameters. Requirements may be formulated as extra conditions on the data parameters of some or all summands of a process term. This raises the possibility of "state–oriented" protocol verification. In section 5 the informal requirements for the protocols are presented and discussed. The requirements for the first protocol are formally specified in ACP. Some remarks are made about a formal verification of the protocols against these requirements. A "state–oriented" verification is also explored. Finally, in section 6, some concluding remarks are given.

This paper is based on Brunekreef et al. ([8]). In this report the design of the protocols is discussed in depth. Moreover, the protocols are also specified using Extended Finite State Diagrams. Temporal Logic is used for the formalisation of the requirements and for a proof of the correctness of the protocols. In general the efficiency of a Leader Election protocol is considered to be of great importance. Due to the lack of space in this paper the protocol *complexity* will not be discussed. The reader is referred to [8] for a detailed analysis of the worst case message complexity of the protocols presented in this paper.

2 A simple Leader Election protocol

In this first protocol a leader is present initially, it is assumed that all other components have not entered the election yet. On entering the election a component does not know the identity of the current leader, and, consequently it cannot decide whether it will become a leader or not. Once the identity of the leader is known there are two possible outcomes: the component should become (the new) leader or not. From the above we conclude that a component may be in one of the following possible states: *candidate*, when it does not yet know whether it will become a leader or not, *leader* when it actually is a leader, and *failed* when it is defeated. A component that has not entered the election yet is considered to be in the *start* state.

Once a component joins the election, that is, when it becomes a candidate, it transmits its identity my_id by means of an identify message $I(my\_id)$. On receipt of an identity a leader compares this identity with its own. In case the received id is larger than its own id the leader moves to the failed state (there is a 'better' component), and gives the candidate the right of succession by transmitting the candidate's id with an R-message (Response). In the other case, the leader remains leader and transmits its own id using $R(my\_id)$. The actions of a candidate on receipt of a response message follow straightforward—when it receives an R-message with its own id it becomes a leader, when it receives an R-message with a larger id it becomes failed, and otherwise it remains a candidate.

There is however a little flaw in the above informally described protocol: when two (or more) components are in the candidate state and one of them causes the leader to capitulate (i.e., to become failed), the other candidates may not receive a response of the leader, remaining candidate forever. This problem is resolved by letting a candidate (re-)transmit an I-message with its own id on receipt of an $R(id)$ message with $id{<}my\_id$.

Three basic processes are specified: the protocol process and a local buffer process of a single component and the medium process, modelling the broadcast network. We will use the following naming convention for the atomic actions involved in the communication between the various processes. The transmission of a message m by a component i is denoted by $send\_XY(i,m)$. X represents the source and Y represents the destination: P for a protocol process, B for a buffer process, M for the medium process. The parameter i denotes the component identity. In the same way the reception of a message is denoted by $read\_XY(i,m)$ and the resulting communication action by $comm\_XY(i,m)$. Figure 1 shows the various processes and their communications. In the specifications to come ID represents the set of component identities. We consider the size of ID to be fixed and finite. M represents the set of messages. For this protocol $M = \{I(i), R(i) \mid i \in ID\}$.

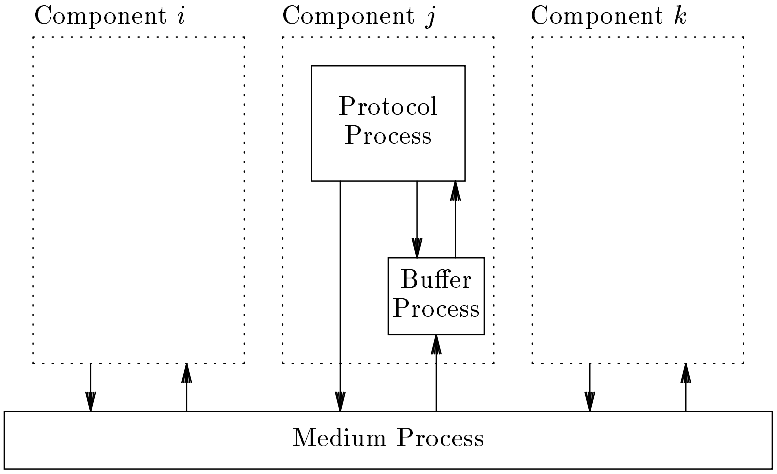

Figure 1: Processes of Protocol 1

The protocol process $P1$ has three parameters: i represents the id of the component the protocol process is part of, ps represents the state of the protocol process, j represents the id contained in a received R-message. In this specification we distinguish seven states:

- S: the start state.
- B: the buffer is reset, no initial I-message has been sent yet.
- C: the candidate state, the initial I-message has been sent.
- T: an R-message is received by a candidate, but has not been processed yet.
- L: the leader state.

- R: an I-message is received by a leader, but has not been processed yet.
- F: the failed state.

Four states (S, C, L and F) have already been introduced in the informal description of the protocol given above. The three other states are added in order to get a neat specification in the UNITY Format. In the specification below the summands are grouped state by state, visualised by a different indentation of the + operator. For a better understanding of the specification we will shortly explain the first three lines. The first line shows that in the state S an incoming message from the buffer is ignored. In the second line, still in the start state, the local buffer is reset (see below for the reason why), after which the state B is entered. The third line shows the transmission of the initial I-message from the B state, after which the candidate state is entered. In the same way each of the following lines shows what action can be performed in what state. The recursive process call shows what state will be entered after the action.

$$
\begin{array}{rll}
P1(i,ps,j) = & & \sum_{m\in M} \mathit{read\_BP}(i,m)\cdot P1(i,ps,j) \lhd ps = S \rhd \delta\\
& & +\ \mathit{send\_PB}(i,\mathit{reset})\cdot P1(i,B,j) \lhd ps = S \rhd \delta\\
& + & \mathit{send\_PM}(i,I(i))\cdot P1(i,C,j) \lhd ps = B \rhd \delta\\
& + & \sum_{k\,\in\,ID} \mathit{read\_BP}(i,I(k))\cdot P1(i,ps,j) \lhd ps = C \rhd \delta\\
& & +\ \sum_{k\,\in\,ID\setminus\{i\}} \mathit{read\_BP}(i,R(k))\cdot P1(i,T,k) \lhd ps = C \rhd \delta\\
& & +\ \mathit{read\_BP}(i,R(i))\cdot P1(i,L,j) \lhd ps = C \rhd \delta\\
& + & \mathit{send\_PM}(i,I(i))\cdot P1(i,C,j) \lhd j < i \ \wedge\ ps = T \rhd \delta\\
& & +\ P1(i,F,j) \lhd j > i \ \wedge\ ps = T \rhd \delta\\
& + & \sum_{k\,\in\,ID} \mathit{read\_BP}(i,I(k))\cdot P1(i,R,k) \lhd ps = L \rhd \delta\\
& + & \mathit{send\_PM}(i,R(i))\cdot P1(i,L,j) \lhd j < i \ \wedge\ ps = R \rhd \delta\\
& & +\ \mathit{send\_PM}(i,R(j))\cdot P1(i,F,j) \lhd j > i \wedge\ ps = R \rhd \delta\\
& + & \sum_{m\in M} \mathit{read\_BP}(i,m)\cdot P1(i,ps,j) \lhd ps = F \rhd \delta
\end{array}
$$

We will illustrate the process algebra specification of the protocol process with an informal drawing (Figure 2). Every state is represented by a circle. An arrow between two states indicates a transition. A transition is labelled with a condition and an action, which are both optional. The interpretation is that the transition takes place by executing the action. This is only allowed if the condition on parameters of the action and parameters of the state yields true.

We will only list the relevant parameters for every state. For example, the index i, which indicates the actual component, is not mentioned explicitly. We will also use a shorthand notation for the atomic actions.

The local buffer process is specified as a queue of unbounded size. The process *BUFFER* has two parameters: the id i of the component the buffer process resides in and a message queue q. The functions *enq* (enqueue, appending a message to the queue), *serve* (delivering the first message in the queue) and *deq* (dequeue, removing the first message from the queue) operate on the message queue. The buffer is reset by the protocol process at the entrance of the election. This reset prevents the handling of messages enqueued before this moment. Although this would not influence the correctness of the protocol, it is regarded as unrealistic.

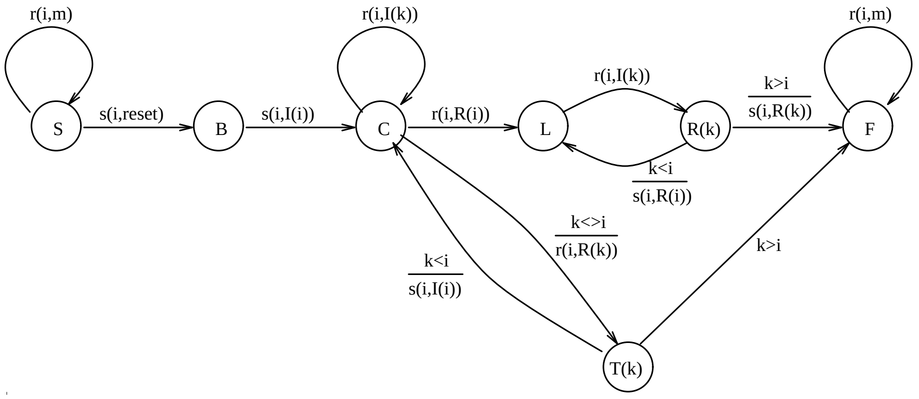

Figure 2: State transitions of protocol 1

$$
\begin{aligned}
&\mathit{BUFFER}(i,q) = \\
&\qquad \textstyle\sum_{m\,\in\, M} \mathit{read\_MB}(i,m)\cdot \mathit{BUFFER}(i,\mathit{enq}(m,q)) \\
&\quad + \mathit{send\_BP}(i,\mathit{serve}(q))\cdot \mathit{BUFFER}(i,\mathit{deq}(q)) \lhd q \neq \mathit{empty\_queue} \rhd \delta \\
&\quad + \mathit{read\_PB}(i,\mathit{reset})\cdot \mathit{BUFFER}(i,\mathit{empty\_queue})
\end{aligned}
$$

The medium process reads a message from a component and delivers this message to all other components, thus modelling a broadcast communication. The set IDS is a variable set of component identities.

$$
\begin{aligned}
&\mathit{MEDIUM}(IDS,m) = \\
&\qquad \textstyle\sum_{i\,\in\, ID, n\,\in\, M} \mathit{read\_PM}(i,n)\cdot \mathit{MEDIUM}(ID\backslash\{i\},n) \lhd IDS = \emptyset \rhd \delta \\
&\quad + \textstyle\sum_{i\,\in\, IDS} \mathit{send\_MB}(i,m)\cdot \mathit{MEDIUM}(IDS\backslash\{i\},m) \lhd IDS \neq \emptyset \rhd \delta
\end{aligned}
$$

The following communications are defined between the various processes for $i \in ID$, $m \in M$:

$$
\begin{aligned}
&\mathit{send\_BP}(i,m) \mid \mathit{read\_BP}(i,m) = \mathit{comm\_BP}(i,m) \\
&\mathit{send\_PM}(i,m) \mid \mathit{read\_PM}(i,m) = \mathit{comm\_PM}(i,m) \\
&\mathit{send\_MB}(i,m) \mid \mathit{read\_MB}(i,m) = \mathit{comm\_MB}(i,m) \\
&\mathit{send\_PB}(i,\mathit{reset}) \mid \mathit{read\_PB}(i,\mathit{reset}) = \mathit{comm\_PB}(i,\mathit{reset})
\end{aligned}
$$

Next we define the process $S1$, being the encapsulated merge of all protocol processes, local buffers and the medium. PS, J and Q denote sequences of the parameters ps, j and q for all components $i \in ID$.

$$
\begin{aligned}
&S1(PS,J,Q,IDS,m) = \\
&\quad \partial_{H_1}(\|_{i\,\in\, ID}\ (P1(i,ps^i,j^i)\ \|\ \mathit{BUFFER}(i,q^i))\ \|\ \mathit{MEDIUM}(IDS,m)\ )
\end{aligned}
$$

Definition of the encapsulation set H_1:

$$
\begin{aligned}
H_1 = \{&\mathit{read\_BP}(i,m),\ \mathit{send\_BP}(i,m),\ \mathit{read\_PM}(i,m),\ \mathit{send\_PM}(i,m), \\
&\mathit{read\_MB}(i,m),\ \mathit{send\_MB}(i,m),\ \mathit{read\_PB}(i,\mathit{reset}), \\
&\mathit{send\_PB}(i,\mathit{reset}) \mid i\ \in\ ID,\ m\ \in\ M\}
\end{aligned}
$$

Protocol 1 is specified by the following equation:

$$Protocol1 = S1(PS_{init}, J_{dc}, EQ, \emptyset, m_{dc})$$

With PS_{init} a sequence of protocol process states ps^i for which it holds that $ps^i = L$ for the initial leader and $ps^i = S$ for all other components. The subscript dc in J_{dc} and m_{dc} indicates that for these parameters initially "don't care" values may be taken. Finally, EQ denotes a sequence of empty queues.

3 A Leader Election Protocol without initial leader

We now drop the unnatural assumption of a leader being present initially. As in the previous section components are considered to be perfect. In this setting Protocol 1 does not suffice, as no leader will ever be present in case a leader is absent initially. The problem now is what mechanism to use for selecting a new leader in absence of a previous one.

A straightforward approach to detect the absence of a leader is to equip each component with a *timer* process and to detect the absence of a leader by means of a timeout mechanism. A component starts its timer when it becomes a candidate. When receiving a response of the leader on its initial $I(my\_id)$ message the timer plays no role and the component progresses as in the first protocol. In absence of a response of a leader, the candidate goes to the leader state at the occurrence of a timeout. In Protocol 1 two different message types to exchange identities were used. As a timeout guarantees that in absence of a leader a candidate becomes a leader, a leader may go to the failed state without notifying the component that forced it to that state. So, the response message from a leader to a candidate, giving that candidate the right of succession, is no longer needed. Of course a leader still has to defend itself against a candidate with a lower id. For this purpose I-messages can be used as well. This means that all R-messages can be replaced by I-messages. As a consequence, candidates now have to react on I-messages, which means that they can now be forced to become failed by receiving messages from other candidates. In Protocol 1 a candidate only reacts to messages sent by the leader.

A timeout must be disabled in case a leader is present. This might be the leader at the start of the timer, but it might also be a 'fresh' one. Therefore, a timeout may expire only when a component has received and processed all responses to its message sent at starting the timer. Premature timeouts have to be excluded in the specification of the protocol.

Thus we obtain the following formal specification of a protocol process $P2$. This process has three parameters: i represents a component id, ps represents the state of the protocol process, j represents the id of a received I-message. In this specification we distinguish eight states:

- S: the start state.
- B: the buffer is reset, no initial I-message has been sent.

- I : the initial I-message has been sent, the timer has not been started yet.
- C: the candidate state, the timer has been started.
- T: an I-message is received by a candidate, but has not been processed yet.
- L: the leader state.
- R: an I-message is received by a leader, but has not been processed yet.
- F: the failed state.

Compared to the states of $P1$, only the state I is new.

$$\begin{aligned}
P2(i,ps,j) = \quad & \textstyle\sum_{k\in ID} \mathit{read\_BP}(i,I(k)) \cdot P2(i,ps,j) \lhd ps = S \rhd \delta \\
& \quad + \mathit{send\_PB}(i,\mathit{reset}) \cdot P2(i,B,j) \lhd ps = S \rhd \delta \\
+ \quad & \mathit{send\_PM}(i,I(i)) \cdot P2(i,I,j) \lhd ps = B \rhd \delta \\
+ \quad & \mathit{send\_PT}(i,\mathit{start}) \cdot P2(i,C,j) \lhd ps = I \rhd \delta \\
+ \quad & \textstyle\sum_{k\,\in\,ID} \mathit{read\_BP}(i,I(k)) \cdot P2(i,T,k) \lhd ps = C \rhd \delta \\
& \quad + \mathit{read\_TP}(i,\mathit{timeout}) \cdot P2(i,L,j) \lhd ps = C \rhd \delta \\
+ \quad & \mathit{send\_PM}(i,I(i)) \cdot P2(i,C,j) \lhd j < i \ \wedge \ ps = T \rhd \delta \\
& \quad + \mathit{send\_PT}(i,\mathit{stop}) \cdot P2(i,F,j) \lhd j > i \ \wedge \ ps = T \rhd \delta \\
+ \quad & \textstyle\sum_{k\,\in\,ID} \mathit{read\_BP}(i,I(k)) \cdot P2(i,R,k) \lhd ps = L \rhd \delta \\
+ \quad & \mathit{send\_PM}(i,I(i)) \cdot P2(i,L,j) \lhd j < i \ \wedge \ ps = R \rhd \delta \\
& \quad + P2(i,F,j) \lhd j > i \ \wedge \ ps = R \rhd \delta \\
+ \quad & \textstyle\sum_{k\in ID} \mathit{read\_BP}(i,I(k)) \cdot P2(i,ps,j) \lhd ps = F \rhd \delta
\end{aligned}$$

In Figure 3 this specification is illustrated.

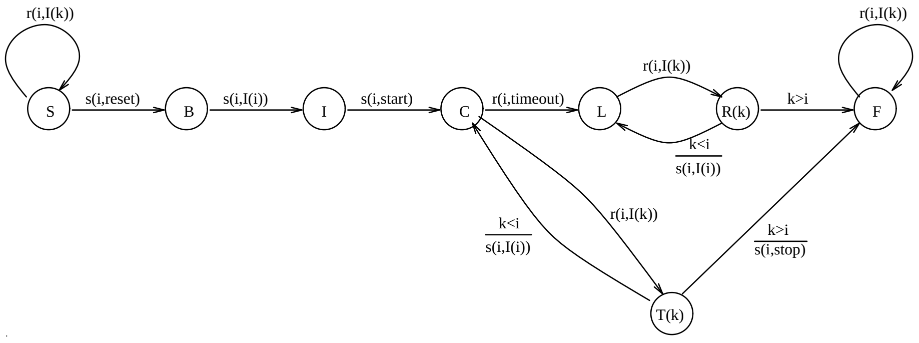

Figure 3: State transitions of protocol 2

The buffer process and the medium process are specified as in Protocol 1. The set of messages now only contains I-messages: $M = \{I(i) \mid i \in ID\}$.

The local timer process is very simple. The timer states are represented by TR (timer running) and TS (timer stopped). We suppose that no start signal is received while the timer is in the state TR.

$$\begin{aligned} \mathit{TIMER}(i,ts) = \quad & \mathit{read\_PT}(i,\mathit{start}) \cdot \mathit{TIMER}(i,TR) \triangleleft ts = TS \triangleright \delta \\ & + \mathit{read\_PT}(i,\mathit{stop}) \cdot \mathit{TIMER}(i,TS) \triangleleft ts = TR \triangleright \delta \\ & + \mathit{send\_TP}(i,\mathit{timeout}) \cdot \mathit{TIMER}(i,TS) \triangleleft ts = TR \triangleright \delta \end{aligned}$$

The communications between a protocol process and its local timer are defined as follows:

$$\begin{aligned} & \mathit{send\_PT}(i,\mathit{start}) \mid \mathit{read\_PT}(i,\mathit{start}) = \mathit{comm\_PT}(i,\mathit{start}) \\ & \mathit{send\_PT}(i,\mathit{stop}) \mid \mathit{read\_PT}(i,\mathit{stop}) = \mathit{comm\_PT}(i,\mathit{stop}) \\ & \mathit{send\_TP}(i,\mathit{timeout}) \mid \mathit{read\_TP}(i,\mathit{timeout}) = \mathit{comm\_TP}(i,\mathit{timeout}) \end{aligned}$$

As mentioned above, this protocol is not robust with respect to premature timeouts: a component has to wait for the reply (if any) of *all* other components participating in the election before a timeout may be enabled. In ACP it is common practise to model such a timeout by means of the priority operator θ. The timeout action gets a lower priority than other actions. As long as one of these actions is enabled the timeout action is prohibited.

Application of the priority operator to our protocol implies the definition of a set of orderings on actions in which a timeout of a component i gets a lower priority than every action that is related to the reply to the initial message from this component. A reply can be made recognisable by labelling the initial I-message with its source and by attaching the same label to all replies to this message. However, in our specification a timeout may be enabled before all replies have been received, due to two causes. First, a message in a buffer queue is only related with the *comm_BP* action if it is at the head of the queue, otherwise no actions are related with this message. This means that a reply that has already been received by a component but is not at the head of the buffer, cannot prevent a timeout. This problem can be solved by a more complex labelling of the messages in the buffer. Second, if the medium is in use (a message has been transmitted to the medium by a component, but has not been buffered by all other components yet), a *comm_PM* action with a reply message to a component i may temporarily be disabled, although the timeout of a component i should still be prohibited by this action.

We solve this problem in a rather crude way by placing more restrictions on a timeout action. The timeout of a component i is given a lower priority than every *comm_MB* action in order to prevent a temporary blockade of a *comm_PM* action. The timeout is also given a lower priority than every *comm_BP* action in order to guarantee that every component has had the possibility to react on a message. Finally, the timeout is given a lower priority than every *comm_PM* action from a component with an id higher than i in order to ensure that every reply is received by component i before its timeout is enabled. This leads to the following ordering relations:

$$\begin{aligned} & \mathit{comm\_TP}(i,\mathit{timeout}) < \mathit{comm\_MB}(k,m), \\ & \mathit{comm\_TP}(i,\mathit{timeout}) < \mathit{comm\_BP}(k,m) \\ & \mathit{comm\_TP}(i,\mathit{timeout}) < \mathit{comm\_PM}(j,m) \end{aligned}$$

with $m \in M$, $i,j,k \in ID, j > i$. Labelling of messages is not useful any more. The whole system is specified with the following equation:

$$
\begin{aligned}
&S2(PS, TS, J, Q, IDS, m) = \\
&\qquad \theta \circ \partial_{H_2}(\|_{i \in ID}\ (P2(i, ps^i, j^i)\ \|\ \mathit{BUFFER}(i, q^i)\ \|\ \mathit{TIMER}(i, ts^i))\ \| \\
&\qquad \mathit{MEDIUM}(IDS, m)\ )
\end{aligned}
$$

Definition of the encapsulation set H_2:

$$
\begin{aligned}
H_2 = H_1 \cup \{&\mathit{read\_PT}(i, \mathit{start}),\ \mathit{send\_PT}(i, \mathit{start}),\ \mathit{read\_PT}(i, \mathit{stop}), \\
&\mathit{send\_PT}(i, \mathit{stop}),\ \mathit{read\_TP}(i, \mathit{timeout}),\ \mathit{send\_TP}(i, \mathit{timeout}) \mid i \in ID\}
\end{aligned}
$$

with H_1 defined in the previous section. Protocol 2 is specified by the following equation:

$$
\mathit{Protocol2} = S2(PS_{init}, TS_{init}, J_{dc}, EQ, \emptyset, m_{dc})
$$

with PS_{init} a sequence of protocol process states ps^i with $ps^i = S$ for all components. TS_{init} denotes a sequence of timer states ts^i with $ts^i = TS$ for all components. As with Protocol 1, the subscript dc in J_{dc} and m_{dc} indicates that for these parameters initially "don't care" values may be taken, EQ denotes a sequence of empty queues.

4 A fault tolerant Leader Election protocol

For the third protocol we drop the assumption of perfect components. Components may cease participating without notifying other components . After halting, a component does not behave maliciously. This kind of failures is known as *crash faults* (see e.g. [11]). Crashed components may recover and (re-)join at any time. It is assumed that recovered components restart in the start state. The number of times a component can crash or recover during an election is unlimited. A component cannot crash during the execution of an atomic event.

The crucial point for a protocol in this setting is that after a crash of the leader component a failed component might be a valid successor. To involve failed components in the election we consider two cases. First, to avoid a candidate to become a leader in case a leader crashed and a better failed component is present, failed processes become a candidate again on receipt of an I-message with a smaller id than their own id—thus joining the competition about the leadership. Other I-messages are still ignored when being failed. On becoming a candidate, an I-message with *my_id* is broadcasted and the local timer is started. This does not suffice in case a leader crashes, at least one failed component is present (that will never crash), and no candidate will ever appear. In this scenario no leader will ever be elected, although there is some component that will never crash. Therefore, we should have a mechanism via which failed components will rejoin the election in absence of a leader. Several techniques can be applied to accomplish this[1]. Here we abstract from a specific technique and model this by adding an unconditional non–deterministic choice of a tran-

[1] For instance, a leader may transmit on a regular basis "I am here" messages and in absence of such messages a timeout could expire in a failed component, thus forcing it to become starting (or candidate). Another possibility would be to let a failed component regularly check whether a leader is present (see e.g. [13]).

sition from the failed state to the candidate state, such that a failed component may (re-)join the election spontaneously by identifying itself and starting its timer. It should be noticed that we now have two transitions from the failed state to the candidate state with equivalent actions, one after the reception of an I-message with a lower id, the other without a preceeding action.

We now turn to the formal specification of this protocol. A component crash has consequences not only for the protocol process, but also for the local buffer process and the local timer process. Therefore all component processes need to be reconsidered. In the specification below we will use a simple model of a component crash:

- A "dead state" is added to the other states (start, candidate, leader, failed).
- Only the protocol process has the possibility to crash. The buffer process and the timer process will simply continue (as far as possible) after a crash of the protocol process.
- The "revival" of a component is modelled by the revival of the protocol process. At its revival this process resets the local timer. The local buffer is reset in the start state, which is entered after the revival.
- In the specification of the protocol process a transition from a state to the dead state is modelled by the atomic action *crash*. The transition from the dead state to the start state is modelled by the atomic action *revive*. These actions do not communicate with any action from any other process.

Remark: in the Finite State Diagram specification of this protocol in [8] the transition to the dead state is modelled with a *may* transition, which may be ignored indefinitely. This opposed to a *must* transition, which has to be chosen sooner or later when it is continuously enabled. This distinction cannot be modelled in ACP: under the usual fairness assumptions each component *will* crash (and revive) at some moment in the future.

The protocol process $P3$ has the same parameters as $P2$: i (the component id), ps (the protocol state) and j (the id of a received I-message). In the specification we distinguish eleven states. The first eight states, from S to F, are identical to the states of $P2$. Three states are new:

- X: an I-message is received by a failed process, but has not been processed yet.
- D: the dead state.
- A: the component becomes alive again (the revive action has been executed), the timer has not been reset yet.

In the specification below the transition to the dead state is not added to the process term for each separate state $S \ldots X$. Instead, a single summand with the action *crash* is added with the condition $ps \neq D$).

$$
\begin{array}{rll}
P3(i,ps,j) = & & \sum_{k\,\in\,ID} \mathit{read\_BP}(i,I(k)) \cdot P3(i,ps,j) \lhd ps = S \rhd \delta \\
& + & \mathit{send\_PB}(i,\mathit{reset}) \cdot P3(i,B,j) \lhd ps = S \rhd \delta \\
+ & & \mathit{send\_PM}(i,I(i)) \cdot P3(i,I,j) \lhd ps = B \rhd \delta \\
+ & & \mathit{send\_PT}(i,\mathit{start}) \cdot P3(i,C,j) \lhd ps = I \rhd \delta \\
+ & & \sum_{k\,\in\,ID} \mathit{read\_BP}(i,I(k)) \cdot P3(i,T,k) \lhd ps = C \rhd \delta \\
& + & \mathit{read\_TP}(i,\mathit{timeout}) \cdot P3(i,L,j) \lhd ps = C \rhd \delta \\
+ & & \mathit{send\_PM}(i,I(i)) \cdot P3(i,C,j) \lhd j < i \ \wedge \ ps = T \rhd \delta \\
& + & \mathit{send\_PT}(i,\mathit{stop}) \cdot P3(i,F,j) \lhd j > i \ \wedge \ ps = T \rhd \delta \\
+ & & \sum_{k\,\in\,ID} \mathit{read\_BP}(i,I(k)) \cdot P3(i,R,k) \lhd ps = L \rhd \delta \\
+ & & \mathit{send\_PM}(i,I(i)) \cdot P3(i,L,j) \lhd j < i \ \wedge \ ps = R \rhd \delta \\
& + & P3(i,F,j) \lhd j > i \ \wedge \ ps = R \rhd \delta \\
+ & & \sum_{k\,\in\,ID} \mathit{read\_BP}(i,I(k)) \cdot P3(i,X,k) \lhd p = F \rhd \delta \\
& + & P3(i,B,j) \lhd ps = F \rhd \delta \\
+ & & P3(i,B,j) \lhd j < i \ \wedge \ ps = X \rhd \delta \\
& + & P3(i,F,j) \lhd j > i \ \wedge \ ps = X \rhd \delta \\
+ & & \mathit{crash}(i) \cdot P3(i,D,j) \lhd ps \neq D \rhd \delta \\
+ & & \mathit{revive}(i) \cdot P3(i,A,j) \lhd ps = D \rhd \delta \\
+ & & \mathit{send\_PT}(i,\mathit{stop}) \cdot P3(i,S,j) \lhd ps = A \rhd \delta
\end{array}
$$

In Figure 4 the specification is illustrated. We did not draw all *crash* actions from every state to the D state.

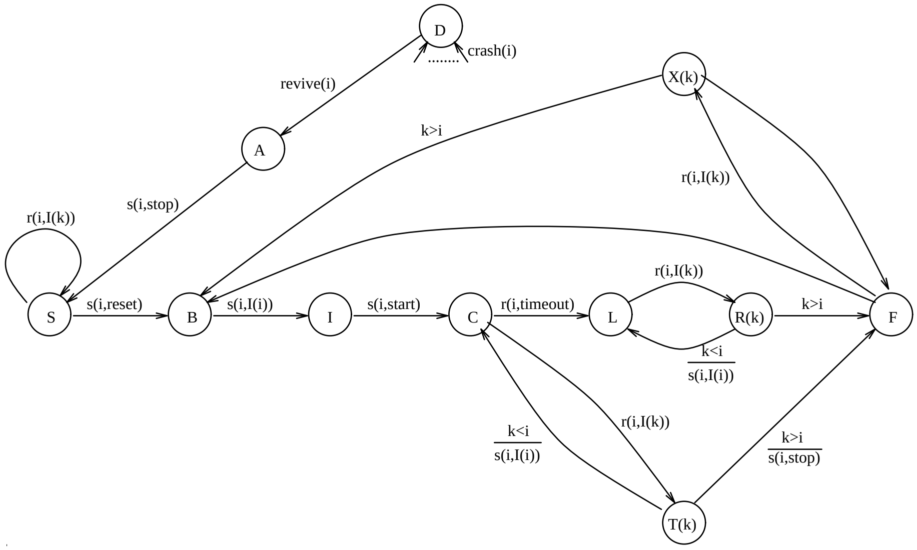

Figure 4: State transitions of protocol 3

The buffer process, the timer process and the medium process are the same as in the previous sections. The process $S3$ has the same data parameters as

$S2$. So we get

$$\begin{array}{l} S3(PS,TS,J,MS,IDS,m) = \\ \qquad \theta \circ \partial_{H_2}(\|_{i \in ID}\ (P3(i,ps^i,j^i)\ \|\ \mathit{BUFFER}(i,q^i)\ \|\ \mathit{TIMER}(i,ts^i))\ \| \\ \qquad \mathit{MEDIUM}(IDS,m)) \end{array}$$

with H_2 as defined before. Protocol 3 is specified by the following equation:

$$\mathit{Protocol}3 = S3(PS_{init},TS_{init},J_{dc},EQ,\emptyset,m_{dc})$$

The initial value of the parameters is the same as with Protocol 2 in the previous section.

5 Towards the verification of the protocols

In this section a set of requirements for the protocols is given. As an example the requirements for Protocol 1 are formalised. Some remarks are made about proving the correctness of the protocol with respect to these requirements. The formalisation of the requirements of the other protocols as well as actual correctness proofs are left for future research. We start with a set of four requirements for the fault–free protocols, Protocol 1 and Protocol 2.

R1. *A leader will be elected: at the termination of the protocol the component with the highest identity is elected as the leader.*

R2. *At each moment at most one component is the current leader. At no moment during or after the election more than one leader is allowed.*

R3. *The capitulation of a leader is caused by an active component with a higher identity than the capitulated leader.*

R4. *A new leader will have a higher identity than the one that just has capitulated.*

The first requirement is obvious. The second requirement states that during the election there may always be at most one leader (since a change of leadership may take some time there can be temporarily no leader at all). The last two requirements make sense of the ordering of component identities. Components with a higher identity have priority in being elected as leader over components with a lower identity. The third requirement states that a leader capitulates only in the presence of a component with a higher identity, which is participating in the election. We do not state anything about the possible future leadership of this ‘better’ component. The last requirement states that the next leader will be an improvement over the previous one (i.e., will have a higher identity). The last two requirements impose constraints on the capitulation of a leader and the ordering of its successor. Note that R4 implies that a component that capitulates once, will not become a leader any more.

For a fault–tolerant protocol (Protocol 3) these requirements are too strong. R1 is already problematic: as components may crash and come up again, a fault tolerant LE protocol will never terminate. So it cannot be required that finally a leader will be elected. Under a certain notion of fairness we may expect that some component becomes a leader for some time, just as we may expect that it will crash at some other moment. Of course requirement R2 still holds during the election. A component crash may cause the capitulation of a leader. Therefore R3 has to be restricted to the capitulation of a leader that remains alive after its capitulation. In that case there has to be a cause as stated in R3. In case of a leader crash nothing can be required about the identity of the new leader compared to the identity of the previous one. So R4 also has to be restricted to the capitulation of a leader that stays alive until the new leader is present. This leads to the following set of requirements for Protocol 3:

R1'. *At any moment it holds that at some moment in the future a leader will be elected.*

R2'. *At each moment at most one component is the current leader. At no moment during the election more than one leader is allowed.*

R3'. *The capitulation of a leader is caused by an active component with a higher identity or by a crash of the leader.*

R4'. *Under the condition that the old leader has not crashed, a new leader will have a higher identity than the one that just has capitulated.*

In ACP traditionally requirements are formalised by the specification of the desired behaviour of a process. The verification of a protocol then consists of an algebraic proof of the equivalence of the requirement specification and the protocol specification with abstraction from certain actions.

Another approach to verification is based on the information contained in the data parameters of the process equations. In particular when the UNITY Format is used, much of the information about the process state is put in these parameters. Requirements can be translated to extra conditions on some or all summands of a process term. These conditions have to be invariantly true. As ACP has no formal syntax and semantics of data types, a data–oriented proof will always be "pseudo formal". In case a strictly formal treatment of data is required the related formalism μCRL ([12]) has to be used. In [5] and [6] correctness proofs in μCRL are given in which formal reasoning about data plays an important role.

In this case study we will give a formalisation of the requirements for Protocol 1 in ACP in terms of desired actions. Next some remarks are made about a data–oriented formalisation of each requirement. Finally, a formalisation of the requirements for the other protocols is discussed briefly.

Before we turn to the requirements we first will derive an equation for the parallel composition of Protocol 1 in the UNITY Format. The expansion of the process term for $S1$ from section 2 leads to the following equation. In this equation the substitution of a new value x for the old value of a parameter y in a sequence S is denoted by $S[x/y]$.

Lemma 5.1

$$
\begin{array}{l}
S1(PS, J, Q, IDS, m) = \\
\qquad \sum_{i \in ID}(\sum_{n \in M} \mathit{comm\_BP}(i, n) \cdot S1(PS, J, Q[\mathit{deq}(q^i)/q^i], IDS, m) \\
\qquad\qquad\qquad \triangleleft\, \mathit{serve}(q^i) = n \;\wedge\; ps^i = S \triangleright \delta \\
\qquad\qquad + \mathit{comm\_PB}(i, \mathit{reset}) \cdot S1(PS[B/ps^i], J, Q[\mathit{empty\_queue}/q^i], IDS, m) \\
\qquad\qquad\qquad \triangleleft\, ps^i = S \triangleright \delta) \\
\quad + \sum_{i \in ID}(\mathit{comm\_PM}(i, I(i)) \cdot S1(PS[C/ps^i], J, Q, ID\backslash\{i\}, I(i)) \\
\qquad\qquad\qquad \triangleleft\, IDS = \emptyset \wedge ps^i = B \triangleright \delta) \\
\quad + \sum_{i \in ID}(\sum_{j \in ID} \mathit{comm\_BP}(i, I(j)) \cdot S1(PS, J, Q[\mathit{deq}(q^i)/q^i], IDS, m) \\
\qquad\qquad\qquad \triangleleft\, \mathit{serve}(q^i) = I(j) \;\wedge\; ps^i = C \triangleright \delta \\
\qquad\qquad + \sum_{j \in ID\backslash\{i\}} \mathit{comm\_BP}(i, R(j)) \cdot \\
\qquad\qquad\qquad S1(PS[T/ps^i], J[j/j^i], Q[\mathit{deq}(q^i)/q^i], IDS, m) \\
\qquad\qquad\qquad \triangleleft\, \mathit{serve}(q^i) = R(j) \;\wedge\; ps^i = C \triangleright \delta \\
\qquad\qquad + \mathit{comm\_BP}(i, R(i)) \cdot S1(PS[L/ps^i], J, Q[\mathit{deq}(q^i)/q^i], IDS, m) \\
\qquad\qquad\qquad \triangleleft\, \mathit{serve}(q^i) = R(i) \;\wedge\; ps^i = C \triangleright \delta) \\
\quad + \sum_{i \in ID}(\mathit{comm\_PM}(i, I(i)) \cdot S1(PS[C/ps^i], J, Q, ID\backslash\{i\}, I(i)) \\
\qquad\qquad\qquad \triangleleft\, IDS = \emptyset \;\wedge\; j^i < i \;\wedge\; ps^i = T \triangleright \delta \\
\qquad\qquad + S1(PS[F/ps^i], J, Q, IDS, m) \triangleleft j^i > i \;\wedge\; ps^i = T \triangleright \delta) \\
\quad + \sum_{i \in ID}(\sum_{j \in ID} \mathit{comm\_BP}(i, I(j)) \cdot \\
\qquad\qquad\qquad S1(PS[R/ps^i], J[j/j^i], Q[\mathit{deq}(q^i)/q^i], IDS, m) \\
\qquad\qquad\qquad \triangleleft\, \mathit{serve}(q^i) = I(j) \wedge ps^i = L \triangleright \delta) \\
\quad + \sum_{i \in ID}(\mathit{comm\_PM}(i, R(i)) \cdot S1(PS[L/ps^i], J, Q, ID\backslash\{i\}, R(i)) \\
\qquad\qquad\qquad \triangleleft\, IDS = \emptyset \;\wedge\; j^i < i \;\wedge\; ps^i = R \triangleright \delta \\
\qquad\qquad + \mathit{comm\_PM}(i, R(j^i)) \cdot S1(PS[F/ps^i], J, Q, ID\backslash\{i\}, R(j^i)) \\
\qquad\qquad\qquad \triangleleft\, IDS = \emptyset \;\wedge\; j^i > i \;\wedge\; ps^i = R \triangleright \delta) \\
\quad + \sum_{i \in ID}(\sum_{n \in M} \mathit{comm\_BP}(i, n) \cdot S1(PS, J, Q[\mathit{deq}(q^i)/q^i], IDS, m) \\
\qquad\qquad\qquad \triangleleft\, \mathit{serve}(q^i) = n \;\wedge\; ps^i = F \triangleright \delta \\
\quad + \sum_{i \in IDS}(\mathit{comm\_MB}(i, m) \cdot S1(PS, J, Q[\mathit{enq}(m, q^i)/q^i], IDS\backslash\{i\}, m) \\
\qquad\qquad\qquad \triangleleft\, IDS \neq \emptyset \triangleright \delta)
\end{array}
$$

Proof: straightforward, see [8].

With this equation as a starting point, we will investigate how the requirements R1–R4 can be formalised.

R1. *The protocol will terminate with the component with the highest identity being elected as the leader.*
A fault–free leader election protocol terminates when the component with the highest identity is elected as the leader and all other components have lost the election (are in the failed state). In the ACP specification of Protocol 1 this represents a deadlock: no action is enabled. However, this deadlock can be avoided in the following way: we add to the equation for the protocol process in lemma 5.1 an extra summand $\sum_{i \in ID} \mathit{exit}(i) \triangleleft ps^i = L \triangleright \delta$ without a recursive call. By giving the *exit* action a lower priority than every other action in the protocol, it will only be enabled if no other action is. We now require that, with abstraction from all other actions, only the action $\mathit{exit}(id_{max})$ will

be observed before successful termination of the protocol. Define the process $Req1$ as follows:

$$Req1 = \tau \cdot exit(id_{max})$$

For Protocol 1 R1 is fulfilled if the following identity is proved:

$$\tau_{I_1} \circ \theta(Protocol1) = Req1 \qquad \text{(R1.ACP)}$$

With Protocol1 defined in section 2. The abstraction set I_1 contains all actions except the $exit$ action.

R2. *At each moment at most one component is the current leader.*
This means that the action on which a component i becomes a leader has to be preceeded by an action connected with the capitulation of the previous leader k . In Protocol 1 the action $comm\_BP(i, R(i))$ reflects the transition to the leader state, the action $comm\_PM(k, R(i))$, $k < i$, reflects the capitulation of a leader.

The specification of the process $Req2(k, IDS)$ also contains a summand with the $exit$ action, indicating the termination of the protocol. As before, IDS denotes a variable set of component ids. The components in this set have not yet become (or will not inevitably be soon) failed or leader.

$$\begin{aligned} Req2(k, IDS) = \tau \cdot (&\textstyle\sum_{i \in IDS} \tau \cdot comm\_PM(k, R(i)) \cdot \\ &(Req2a(i, IDS) \lhd i > k \rhd Req2(k, IDS \backslash \{i\})) \\ &+ exit(id_{max})) \end{aligned}$$

$$Req2a(i, IDS) = \tau \cdot comm\_BP(i, R(i)) \cdot Req2(i, IDS \backslash \{i\})$$

For Protocol 1 R2 is fulfilled if the following identity is proved (id_{il} denotes the identity of the initial leader):

$$\tau_{I_2} \circ \theta(Protocol1) = Req2(id_{il}, ID \backslash \{id_{il}\}) \qquad \text{(R2.ACP)}$$

The set I_2 contains all actions except the actions showed in the specification of $Req2$.

R3. *The capitulation of a leader is caused by an active component with a higher identity than the capitulated leader.*
This requirement is formalised in a weaker version: the following equation for $Req3$ states that the capitulation of a leader k is preceeded by the transmission of one or more I-messages from components with a higher identity. A strict causal relation is not specified. As with R1 and R2 the termination of the protocol is specified with the $exit$ action in the first equation. IDS contains the ids of the components who have not yet become (or will not inevitably be soon) failed or leader.

$$\begin{aligned} Req3(k, IDS) = \tau \cdot (&\textstyle\sum_{i \in ID} \tau \cdot comm\_PM(i, I(i)) \cdot \\ &(Req3(k, i, IDS \backslash \{i\}) \lhd i > k \rhd Req3(k, k, IDS \backslash \{i\})) \\ &+ exit(id_{max})) \end{aligned}$$

$$\begin{aligned}&Req3(k,i,IDS) = \tau \cdot (\textstyle\sum_{j \in ID} \tau \cdot comm\_PM(j, I(j)) \cdot \\ &\quad (Req3(k,i,IDS) \lhd j > i \rhd Req3(k,i,IDS \backslash \{j\})) \\ &\quad\; + \tau \cdot comm\_PM(k, R(i)) \cdot Req3(i, IDS)\ )\end{aligned}$$

R3 is fulfilled if the following identity is proved:

$$\tau_{I_3} \circ \theta(Protocol1) = Req3(id_{il}, ID/\{id_{il}\}) \qquad \text{(R3.ACP)}$$

The set I_3 contains all actions except the actions used in the specification of $Req3$.

R4. *A new leader will have a higher identity than the one that just has capitulated.*
For Protocol 1 this requirement is formalised by requiring a sequence of zero or more $comm\_BP(i, R(i))$ actions with i greater than the identity of the last leader (k). This action marks the transition from the candidate state to the leader state. We get the following equation:

$$Req4(k) = \tau \cdot (\textstyle\sum_{i \in ID} \tau \cdot comm\_BP(i, R(i)) \cdot (Req4(i) \lhd i > k \rhd \delta) + exit(id_{max}))$$

R4 is fulfilled if the following identity is proved:

$$\tau_{I_4} \circ \theta(Protocol1) = Req4(id_{il}) \qquad \text{(R4.ACP)}$$

The set I_4 contains all actions except the actions used in the specification of $Req4$.

Next we will shortly discuss a data–oriented formalisation of the requirements. With R1 we will also need the *exit* action for the successful termination of the protocol. However, without the detour of the priority operator, needed in R1.ACP, we now may directly specify the condition under which this action is enabled. It is clear that the *exit* action is enabled if for the state of the protocol processes the following holds: $ps^{id_{max}} = L$ and, for all other components $j \in ID$, $ps^j = F$. This can directly be checked by investigating the sequence of protocol states PS. Remark: In the UNITY Format the priority operator may be "translated" to extra conditions on several summands, see [7]. This will lead to the same condition for the *exit* action as stated above.

In a data–oriented formalisation of R2 we have to count the number of protocol processes in the sequence PS for which $ps^i = L$ or $ps^i = R$ holds. This number may never be greater than one. This condition may be imposed on every summand of the equation in lemma 5.1. However, it seems sufficient to impose the condition only on the summand with the action after which a component enters the leader state ($comm\_BP(i, R(i))$).

Requirement R3 can be formalised by adding an extra boolean parameter to the parameter list of $S1$. Every time an I-message with an id greater than the id of the current leader (or last leader) is sent, this parameter is set to $true$. The capitulation of a leader (the action $send\_PM(k, R(i))$ with $i > k$) is only permitted under the extra condition that the boolean parameter has been set to $true$. Execution of this action resets the parameter to $false$.

In a data–oriented formalisation of R4 an extra parameter may be added to the parameter list of $S1$, containing the identity of the current or last leader, say k. Now, according to R4, for a component to become a leader it must hold that its identity is greater than k. So the summand with the action $comm\_BP(i, R(i))$ gets an extra condition: $i > k$.

Along the same lines the requirements for Protocol 2 and Protocol 3 can be formalised. With these protocols we have the problem that no atomic action is directly connected with the capitulation of a leader. This means that for a formalisation of R2 and R3 an extra action, say $capitulate(i)$, has to be introduced in order to resolve this problem. With Protocol 3 the *crash* actions make a formalisation of the requirements R1'–R4' fairly complicated. The crash of a leader has to be observed separately from the crash of other components.

In this paper we will not give a formalisation of the requirements for Protocol 2 or Protocol 3 or a formal proof of the correctness of Protocol 1 with regard to the requirements stated above. This is left for future research.

6 Conclusions

In this paper we have specified a series of dynamic leader election protocols in broadcast networks. The protocols are presented in a stepwise fashion. The stepwise approach aids not only in the clarity and conciseness of the protocols, but also —and more importantly— in reasoning about them ('separation of concerns').

The specification of a protocol in ACP contains a complete formal description, not only of the various processes but also of the complete distributed behaviour of the protocol. The UNITY Format for process specification, used in this paper, provides specifications that are well-readable and that can serve as a solid base for algebraic manipulations like normalising the parallel composition of several processes or proving the correctness of a complete system. While specifying the protocols, problems were encountered in modelling the desired timeout semantics (which appeared to be impossible due to the scope of the priority operator) and in modelling the crash of a component (which appeared to be counter–intuitive somehow, due to the usual fairness assumptions in ACP).

The timeout semantics problem may be overcome by specifying the protocols in a formalism with real–time features included, e.g. real–time ACP ([4]). This is left for future research.

The requirements for Protocol 1 are formalised *after* the formal specification of the protocol. The atomic actions from the protocol specification needed to be known before the required behaviour could be specified. It is an interesting question whether this kind of "reverse software development" is intrinsic to a formalism like ACP or whether it is due to the specific character of the protocols studied in this paper.

The requirements can be formalised in two different ways: process–oriented and data–oriented. At this moment it is not clear which approach is best-suited for a formal verification of the protocol. The protocols in this paper are too large for manual algebraic verification. Probably automated verification in a related formalism as μCRL ([12]) is possible.

The specifications from this paper have been translated into the executable formalism PSF ([19]). Simulation runs of these specifications appeared to be very helpful during the various stages of the protocol development.

Acknowledgements: The authors gratefully acknowledge Jan Bergstra (Univ. of Amsterdam & Univ. of Utrecht) for initiating and stimulating our fruitful cooperation. We are also grateful to Jan Friso Groote (Univ. of Utrecht) for his assistance during the beginning of our work.

References

[1] H.H. Abu-Amara. Fault-tolerant distributed algorithm for election in complete networks. *IEEE Transactions on Computers*, 37(4):449–453, 1988.

[2] Y. Afek and E. Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. *SIAM Journal on Computing*, 20(2):376–394, 1991.

[3] H. Attiya, J. van Leeuwen, N. Santoro, and S. Zaks. Efficient elections in chordal ring networks. *Algorithmica*, 4(3):437–446, 1989.

[4] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[5] M.A. Bezem and J.F. Groote. A correctness proof of a One-bit Sliding Window Protocol in μCRL. Technical Report Logic Group Preprint Series nr. 99, Department of Philosophy, Utrecht University, 1993.

[6] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. Technical Report Logic Group Preprint Series nr. 98, Department of Philosophy, Utrecht University, 1993.

[7] J.J. Brunekreef. Process Specification in a UNITY Format. Technical Report P9329, Programming Research Group, University of Amsterdam, 1993. An exented abstract is published in this volume.

[8] J.J. Brunekreef, J.P. Katoen, R.L.C. Koymans, and S. Mauw. Design and analysis of dynamic leader election protocols in broadcast networks. Technical Report P9324, Programming Research Group, University of Amsterdam, 1993.

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

[10] S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election part 1: Complete graph protocols. (Preliminary version appeared in *Proc. 6th Int. Workshop on Distributed Algorithms*, (S. Toueg et. al., eds.), LNCS 579, 167–180, 1992), 1993.

[11] M.J. Fischer. A theoretican's view of fault tolerant distributed computing. In *Fault Tolerant Distributed Computing*, number 448 in LNCS, pages 1–9. Springer Verlag, 1991.

[12] J.F. Groote and A. Ponse. μCRL: A base for analyzing processes with data. In E. Best and G. Rozenberg, editors, *Proceedings of the 3^{rd} Workshop on Concurrency and Compositionality*, pages 125–130. Universität Hildesheim, 1991.

[13] R. Gusella and S. Zatti. An election algorithm for a distributed clock synchronization program. In *Proc. 6th IEEE Int. Conf. on Distributed Computing Systems*, pages 364–371, 1986.

[14] A. Itai, S. Kutten, Y. Wolfstahl, and S. Zaks. Optimal distributed t-resilient election in complete networks. *IEEE Transactions on Software Engineering*, 16(4):415–420, 1990.

[15] E. Korach, S. Moran, and S. Zaks. Tight lower and upper bounds for some distributed algorithms for a complete network of processors. In *Proc. 3rd Annual ACM Symp. on Principles of Distributed Computing*, pages 199–207. ACM, 1984.

[16] G. LeLann. Distributed systems—towards a formal approach. In B. Gilchrist, editor, *Information Processing (vol. 77) (IFIP)*, pages 155–160. North-Holland, Amsterdam, 1977.

[17] M.C. Loui, T.A. Matsushita, and D.B. West. Election in a complete network with a sense of direction. *Information Processing Letters*, 22:185–187, 1986. (see also *Inf. Proc. Letters*, 28:327, 1988).

[18] T. Masuzawa, N. Nishikawa, K. Hagihara, and N. Tokura. Optimal fault-tolerant distributed algorithms for election in complete networks with a global sense of direction. In J.-C. Bermond and M. Raynal, editors, *Distributed Algorithms*, LNCS 392, pages 171–182. Springer-Verlag, 1989.

[19] S. Mauw and G.J. Veltink. A Process Specification Formalism. *Fund. Inf.*, XII:85–139, 1990.

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

[21] L. Shrira and O. Goldreich. Electing a leader in a ring with link failures. *Acta Informatica*, 24:79–91, 1987.

[22] G. Singh. Efficient distributed algorithms for leader election in complete networks. In *Proc. 11th IEEE Int. Conf. on Distributed Computing Systems*, pages 472–479, 1991.