Leadership Election: an Industrial SoS Application of Compositional Deadlock Verification — Extended version

P. R. G. Antonino[1,*], M. V. M. Oliveira[2], A. C. A. Sampaio[1], K. E. Kristensen[3], and J. W. Bryans[4]

[1] Centro de Informática, Universidade Federal de Pernambuco, Brazil
[2] Departamento de Informática e Matemática Aplicada, Universidade Federal do Rio Grande do Norte, Brazil
[3] Bang & Olufsen, Denmark
[4] School of Computing Science, University of Newcastle upon Tyne, UK

Abstract. In distributed computing, the leadership election has been used to distributively designate a node as the central controller (leader) of a network of nodes. The complexity of the algorithm arises due to the unawareness of every node of who the current leader is. After running the algorithm, however, a unique node in the network must be elected as the leader and recognized as so by the remaining nodes. In this paper, using CSP, we formalise the leadership election algorithm used by our industrial partner. Its verification is feasible only due to the use of a pattern based strategy that allows the verification to be carried out in a fully local manner. The pattern used here is novel and a further contribution of the paper. A refinement relation together with predicate abstraction is used to describe pattern conformance. The mechanisation of the behavioural conformance is carried out using FDR.

Keywords: Leadership Election, Local Analysis, Deadlock Freedom.

1 Introduction

The complexity inherent to most distributed algorithms (and systems) can turn their development into a very laborious and error-prone task. The use of formal methods like CSP [11] considerably simplifies this task and provides a better understanding and means for verification of phenomena that are exclusive to the concurrent world, like deadlock and livelock. For CSP, the model checker FDR [5] provides an automatic check of finite state specifications for correctness and properties like deadlock and divergence freedom.

Component-based or Systems of Systems (SoS) development are only feasible in an industrial context of high-quality critical systems if trustworthy architectures are obtained by carefully designing and systematically verifying the constituents integration in a scalable fashion. A naive practice has been to verify

* Email address of the corresponding author: prga2@cin.ufpe.br

and validate them after they have been built [7, 8, 3]. The major issue is the high cost to fix a problem that is found in a late stage of development. Instead of verifying the entire system, more promising approaches focus on iteratively identifying problems in compositions. However, in most approaches the cost of subsequent compositions is not alleviated by the results of the previous ones [1, 2, 4]. Every composition is taken as a monolithic system for verification, and properties of its constituting parts are not considered. Hence, these methods are not compositional and have scalability problems by not considering local analysis.

In [10, 9], we proposed a systematic approach to build trustworthy component-based systems underpinned by CSP [11]. In this approach, constituents may only be composed using composition rules that impose the necessary constraints for a safe interaction among components. Together, the composition rules systematise the approach preserving deadlock-freedom by construction. Although systematic, this approach is not local for cyclic communicating systems, potentially presenting a state explosion in the verification of such systems. This drawback can make our approach inapplicable to complex cyclic systems.

In [14], Roscoe proposed a solution based on architectural patterns that reduce the verification effort, by allowing a local analysis of deadlock, even for cyclic communication topologies.

In this paper, we formalise and analyse the version of the leadership election algorithm used at B&O[5], which is an example of a class of cyclic networks. This algorithm is used in B&O's networks of Audio and Video (AV) systems with up to 32 systems. There are other solutions to this problem [6, 14], whose details are discussed in the conclusions. They make different assumptions on the networks topology and faults. Like [14], we use CSP as our modelling language as it is a well-established notation with industrial strength tools for verification of communicating systems.

The use of standard approaches in the formal verification of our model presented scalability issues. This motivated the use of a local analysis strategy based on architectural patterns. However, none of the existing architectural patterns in the literature (including those in [14]) match directly the structure of our case study. As a further contribution, we formalise a new pattern that allows local deadlock analysis of our example. We use CSP to specify the pattern's behaviour and its stable failures refinement to formalise a conformance relation. We compare the application of our local analysis with a standard global analysis using FDR.

In Section 2, we introduce the CSP notation. Section 3 presents the underpinning model and theory used to analyse systems for deadlocks. In Section 4 we formalise a verification approach that allows local analysis of systems that obey a communication pattern suitable for our case study. The leadership algorithm, its practical use at B&O, and its formalisation in CSP is presented in Section 5. In Section 6, we apply the verification approach of Section 4 to our example. We also present an empirical analysis of the approach and contrast its verifica-

[5] http://www.bang-olufsen.com/

tion effort against the standard approach based on a global analysis. Finally, in Section 7, we present our concluding remarks and future work.

2 CSP

CSP is a process algebra that can be used to describe systems as interacting components, which are independent self-contained processes with interfaces that are used to interact with the environment [11]. Most of the CSP tools, like FDR2 and ProBE, accept a machine-processable CSP, called CSP_M, used in this paper.

The two basic CSP processes are `STOP` and `SKIP`; the former deadlocks, and the latter does nothing and terminates. The prefixing `a -> P` is initially able to perform only the event `a`; afterwards it behaves like process `P`. A boolean guard may be associated with a process: `g & P` behaves like `P` if the predicate `g` is true; it deadlocks otherwise. The alternation `if b then P else Q` is available and has a standard behaviour. The operator `P1;P2` combines `P1` and `P2` in sequence. The external choice `P1[]P2` initially offers events of both processes; the occurrence of the first event or termination resolves the choice in favour of the process that performs either of them. The environment has no control over the internal choice `P1|~|P2`, in which the choice is resolved internally. The synchronised parallel composition `P1[|cs|]P2` synchronises `P1` and `P2` on the events in the set `cs`; events that are not listed occur independently. Processes composed in interleaving `P1|||P2` run independently. The event hiding `P\cs` encapsulates the events that are in `cs`. The renamed process `P[[a<-b]]` behaves like `P` except that all occurrences of `a` in `P` are replaced by `b`. The interrupt operator allows a process `Q` to take over from another process `P`: `P /\ a -> Q` specifies `a` as the signal for Q to start.

CSP also provides replicated versions for most of its compositional operators. For instance, `||| x : S @ P(x)` stands for the interleaving of all `P(x)`, for `x`$\in$ `S`. Local processes are defined using the `let Id = P within Q` construct, which behaves as `Q` and restricts the scope of process `Id` to `Q`.

There are three major semantic models for CSP: *traces*, *stable failures*, and the *failures-divergences* model. In this work we only use the *stable failures* one. In this model processes are described by its traces, a set of finite sequences of events it can perform given by $\mathit{traces}(P)$, and by its set of stable failures, given by the function $\mathit{failures}(P)$. The stable failures set contains all pairs (s, X) where s is a finite trace of P and X is a set of events that P can refuse after performing s. All states in which P may perform an internal action are considered unstable: they are not taken into account. Finally, the function $\mathit{refusals}(P, s)$ gives the set of events the process P can refuse after the trace s. This model also possesses a refinement relation (`[F=`) on processes. The relation `P [F= Q` holds if and only if $\mathit{traces}(P) \supseteq \mathit{traces}(Q) \wedge \mathit{failures}(P) \supseteq \mathit{failures}(Q)$ holds.

3 Networks of processes

The concepts presented in this section are essentially drawn from [12, 13], which present an approach to deadlock analysis of systems described as a network of CSP processes. The most fundamental concept is the one of *atomic tuples*, which represents the most fundamental components of a system. These are triples that contain an identifier for the component, the process describing the behaviour of this component and an alphabet that represents the set of events that this component can perform. A *network* is a finite set of atomic tuples.

Definition 1 (Network) *Let CSP_Processes be the set of all possible CSP processes, Σ the set of CSP events and IdType the set for identifiers of atomic tuples. A network is a finite set V, such that:*

$$V \subset Atomics$$

where: $Atomics \mathrel{\widehat=} IdType \times \mathbb{P}\Sigma \times CSP_Processes$.

The behaviour of a network is given as the alphabetised parallel composition of the behaviour of each component, where processes and alphabets are extracted from atomic tuples. We use an indexed version of the alphabetised parallel operator, which generalises the binary one with processes interacting in the alphabet intersection. The functions $A(id, V)$ and $B(id, V)$ extract the alphabet and the behaviour of an atomic process id from the network V.

Definition 2 (Behaviour of a network) *Let V be a network.*
$B(V) =$ `|| id : dom V @ [A(id,V)] B(id,V)`

By way of illustration, let $V = \{(id_1, B1, A1), (id_2, B2, A2), (id_3, B3, A3)\}$. The behaviour of this network is given by $B(V) =$ `B1 [A1||A2] B2 [A24||A3] B3`.

A *live* network is a structure that satisfies three assumptions. The first one is *busyness*. A busy network is a network whose atomic components are deadlock free. The second assumption is *atomic non-termination*, i.e. no atomic component can terminate. The last assumption concerns interactions. A network is *triple-disjoint* if at most two processes share an event, i.e. if for any three different atomic tuples their alphabet intersection is the empty set.

In a *live* network, a deadlock state can only arise from an improper interaction between processes, since no process can individually deadlock. This particular misinteraction is captured by the concept of *ungranted requests*. The states σ of a network are pairs (s, R), such that s is a trace of the network and R is a vector of refusal sets. The function $R(id)$ returns the refusal set of the process id after $s \upharpoonright A(id)$. The projection $t \upharpoonright s$ takes a trace t and a set of events s as arguments and yields the trace t restricted to s. An ungranted request arises in a state σ when an atom, say id_1, is offering an event to communicate with another atom, say id_2, but id_2 cannot offer any of the events expected by id_1. In addition, both processes must not be able to perform internal actions.

A proper cycle of ungranted requests is an important element of deadlock analysis. It is represented as a sequence of different process identifiers, C, where

each element at the position i, $C(i)$, has an ungranted request to the element at the position $i \oplus 1$, $C(i \oplus 1)$, where $\oplus$ is addition modulo length of the sequence. A *conflict* is a proper cycle of ungranted requests with length 2. After these definitions two fundamental theorems extracted from [12] are introduced.

Theorem 1 *Let V be a live network. Any deadlocked state has a cycle of ungranted requests. If V is conflict-free then a deadlock state has a long cycle.*

The next theorem requires the introduction of three important concepts. A *communication graph* is a representation of the topology of the network where vertexes represent atomic components of the network and edges represent the alphabet intersection between components. A *disconnecting edge* is an edge that, if removed, increases the number of connected components of the graph, i.e., an edge that is not part of a cycle in the communication graph. The components left after the removal of every disconnecting edge are called *essential components*.

Theorem 2 *Let V be a live network with essential components $V_1, \ldots, V_k$ where the pair of processes joined by each disconnecting edge are conflict-free. Then if each V_i of the network is deadlock free, then so is V.*

Theorem 1 allows one to reduce the problem of avoiding deadlock by preventing cycles of ungranted requests. Theorem 2 allows the decomposition of a network in subnetworks called essential components that can be independently verified for deadlock freedom.

With these two results it is already possible to fully verify a tree topology network in a local way, by checking only pairs of processes, due to the fact that only proper cycles of length two can arise in tree networks. Nevertheless, cyclic networks cannot be locally verified by these methods. Moreover, if one tries to verify the freedom of long cycles of ungranted requests, based on Theorem 1, this might be as complex as exploring the whole state space. Therefore, for networks with cycles in their topology a complete and local method for checking deadlock freedom is not generally available.

4 Pattern based approach to cyclic network verification

As a complementary approach to the decomposition strategy presented in the previous section, we consider the adoption of communication patterns in order to support local analysis of cyclic networks. Our approach is based on the design rules described in [11], which proposes resource sharing and client/server design rules, among others. As a novel contribution, we propose a pattern so as to prevent deadlocks by avoiding the emerging of cycles of ungranted requests. The pattern proposed in this section can be used to design and analyse networks that are asynchronous and dynamic (in the sense that nodes might turn on and off) and whose transport layer possesses a mechanism allowing to detect whether a node is on or off.

The pattern can be applied to networks with two different types of nodes: the participants and the transport layer. The participants of the network do not interact directly with each other, but exchange messages via the transport layer. The participants recursively send messages to all its peer participants and receives messages from them. Both, sending and receiving, must follow an order. Furthermore, participants can turn on and off at any time. The transport layer, composed of a set of transport entities, provides communication point-to-point between participants of the network. It has also the ability to identify whether participants are on or off.

The proposed pattern imposes behavioural and structural restrictions on a network as a means to guarantee deadlock freedom. Our approach uses parametrised CSP processes to capture pattern specifications and the stable failures refinement relation to capture a notion of pattern conformance. Structural restrictions are captured as predicates over the network structure and conformance through a predicate satisfaction relation.

A transport entity connects two participants: a sender and a receiver. It is entitled to receive data from its sender participant and to pass this data on to its receiver participant. This communication is unidirectional from the sender to the receiver. It also detects whether its sender is switched on or off.

Definition 3 (Transport entity specification) *Let id be an identifier of a transport entity, source*(id) *and target*(id) *the identifiers of the sender and receiver participants associated to the transport entity id, and offCh, sendCh, receiveCh, onCh, timeoutCh functions that, given the identifiers of source and target participants, yield the channels used for detecting that the sender is off, receiving data, sending data, detecting that the sender is on, and signalling a timeout, respectively. The transport entity CSP specification is:*

```
TransportSpec(id) =
let
    idS = source(id)
    idT = target(id)
    On = offCh(idS,idT) -> Off [] sendCh(idS,idT)?data -> OnF(data)
    OnF(d) = offCh(idS,idT) -> Off
        [] sendCh(idS,idT)?data -> OnF(data)
        [] receiveCh(idS,idT)!d -> On
    Off =  onCh(idS,idT) -> On [] timeoutCh(idS,idT) -> Off
within Off
```

The processes `On`, `OnF` and `Off` specify the expected behaviour of a transport entity in a sender detected as on and no data available state, in a sender detected as off and data available state, and in a sender detected as off state, respectively. The transport entity initially behaves as `Off`, in which case it offers two events to its participants: a *timeout* that informs the receiver that the sender is switched off, and a *turn on* that detects when the sender turns on, in which case it behaves as the process `On`. In this state the transport entity is on and empty: it can receive data from the sender participant or detect a switching off event from it. In the case of the latter, the entity behaves as `Off` again. However, if it receives data, the transport entity stores this data and starts behaving as `OnF`. In the `OnF` state, the transport entity can receive new data from its sender

participant, in which case the new data overwrites the data previously stored. However, it can also transmit the data stored to its receiver participant, in which case the transport entity behaves as `On` again. Finally, it can also detect whether its sender participant has turned off, in which case it behaves as the process `Off`.

The participants of the network contain its business logic. They have a dynamic behavioural feature that allows them to turn on and off and a functional behaviour that involves data exchange and any business related function. The behavioural specification of a conform participant is given in the next definition.

Definition 4 (Participant specification) *Let id be the identifier of the participant and sequence(id) a function that yields a sequence of ids representing the order in which this participant interacts with its neighbours. The participant CSP specification is:*

```
ParticipantSpec(id) =
    let s = sequence(id)
        SendReceive(id,s) = Send(id,s);Receive(id,s);SendReceive(id,s)
    within OnDetect(id,s); (SendReceive(id,s) /\ (SKIP |~|STOP));
           OffDetect(id,s); ParticipantSpec(id,s)
```

A participant first behaves as the process `OnDetect`, which sends a signal to inform that it is on to each transport entity to which it acts as a sender. This mechanism abstracts the ability of the transport layer to detect participant status. The `s` parameter gives the order in which the participant interacts with its transport entities. After turning on, it acts recursively, first behaving as a sender (`Send`) and then as a receiver (`Receive`). When behaving as a sender, it sends messages to all transport entities that have this participant as sender, following the order recorded in `s`. When acting as a receiver, in the same way, it interacts with the transport entities that have it as a receiver, also following the order stated in `s`: it accepts both the incoming data and a timeout signal that indicates the sender associated with the transport entity is off.

In order to check whether a concrete model of either a transport entity or a participant conforms to the corresponding abstract behaviour, we use a refinement relation in the stable failures model of CSP. We restrict the behaviour of the processes being tested for conformance to the events that are related interactions, as these are the only events of interest for deadlock analysis. This restriction is given by the `Abs` function. Hence, transport entity and participant conformance is given by the following definition.

Definition 5 (Transport entity and Participant conformance) *Let Spec stand for the specification of either a transport entity or of a participant (as in Definitions 3 and 4). Let id be the identifier of the candidate concrete model. Then id conforms to Spec if, and only if, the following refinement holds:*

```
Spec [F= Abs(id,V)
```

where:

– `Abs(id,V) = B(id,V) \ diff(A(id,V),AVoc(id,V))`

- `AVoc(id,V) = Union({inter(A(id,V),A_(a)) | a <- V, ID_(a) != id})`
- `a <- V` *states that* `a` *is an atomic tuple from* `V`
- `A_(a)` *and* `ID_(a)` *give the alphabet and identifier of* `a`*, respectively.*

In addition to the behavioural restrictions, the pattern also imposes structural restrictions. The first one restricts the alphabet of any two participants or transport entities to be disjoint. This restriction is encoded in the *disjointAlpha* predicate. This ensures that two participants or two transport entities may interact directly. The *controlledAlpha* predicate is satisfied if the alphabet of the interaction between constituents is the set composed of the send, receive, on, off and timeout events. This ensures that the behaviour related to the interaction between constituents of the network is restricted to the controlled behaviour. The sequence of ids used to guide the order of interaction for the participant's behavioural restriction must have only one occurrence of each neighbour of this participant. This restriction is guaranteed by the *validOrder* predicate. Hence, the conformance of a network to this pattern is given by the following predicate, which is a conjunction of the restrictions presented.

Definition 6 (Async Dynamic network) *Let V be a network, participants a set of participants and transport_entities a set of transport entities.*

$$\begin{aligned} & AsyncDynamic(V, participants, transport_entities) \mathrel{\widehat=} \\ & \quad disjointAlpha(participants) \land disjointAlpha(transport_entities) \land \\ & \quad partition(V, participants, transport_entities) \land \\ & \quad \forall\, id : participants \bullet ParticipantBehaviouralRestriction(id) \land \\ & \quad \forall\, id : transport_entities \bullet TransportEntityBehaviouralRestriction(id) \land \\ & \quad \forall\, id : participants \bullet validOrder(id) \land \\ & \quad \forall\, id_1 : participants, id_2 : transport_entities \bullet controlledAlpha(id_1, id_2) \end{aligned}$$

where:

- $disjointAlpha(set) \mathrel{\widehat=} \forall\, id_1, id_2 : set \bullet A(id_1) \cap A(id_2) = \emptyset$
- $partition(v, s_1, s_2) \mathrel{\widehat=} s1 \cap s_2 = \emptyset \land s_1 \cup s_2 = \operatorname{dom} v$
- $ParticipantBehaviouralRestriction(id, V) \mathrel{\widehat=}$ `ParticipantSpec(id,V) [F= Abs(id,V)`
- $TransportEntityBehaviouralRestriction(id, V) \mathrel{\widehat=}$ `TransportSpec(id,V) [F= Abs(id,V)`
- $validOrder(id) \mathrel{\widehat=} neighbours(id) = \operatorname{ran} sequence(id) \land functional(sequence(id))$
- $controlledAlpha(id_1, id_2) \mathrel{\widehat=}$

$$\begin{aligned} & A(id_1) \cap A(id_2) = \{\!|\ sendCh(source(id_2), target(id_2)), \\ & receiveCh(source(id_2), target(id_2)), onCh(source(id_2), target(id_2)), \\ & offCh(source(id_2), target(id_2)), timeoutCh(source(id_2), target(id_2))\ |\!\} \end{aligned}$$

The following theorem ensures the ability of our pattern to prevent deadlock.

Theorem 3 *Let V be a network, and participants and transport_entities two partitions of the domain of this network, then:*

$$AsyncDynamic(V, participants, transport_entities) \Rightarrow V \textit{ is deadlock free}$$

Proof.

Pattern soundness. The soundness of this pattern can be guaranteed by the prevention of cycles of ungranted requests. This can be argued as follows. Let the network in question be conform to the pattern presented. First of all, there cannot be an ungranted request between a pair of participants or a pair of transport entities, since their alphabets are disjoint violating the *request* predicate.

There cannot be an ungranted request from a participant behaving as a sender to any transport entity. There is a request from a sender participant to one of its transport entity if this participant is offering an on, an off or an send event. When the on event is offered, then the transport entity is mandatorily in an off state, thus accepting the on event which violates the *ungratedness* clause. In the same way, when the off event or the send event are offered, the transport entity is mandatorily on, hence it accepts these two events, violating again the ungrantedness predicate. Therefore, there cannot be a cycle C in which atom $C(i)$ is behaving as a sender participant to the transport entity atom $C(i \oplus 1)$.

Therefore, there can only be cycle of alternate participants behaving as receiver and transport entities. If a transport entity is off or full, then it must offer either the timeout event or the transmit event, this two events are accepted by a participant acting as a receiver, what violates the ungratedness condition. Hence, the ungranted request from a $C(i)$ participant acting as a receiver to a $C(i \oplus 1)$ transport entity can only occur when the corresponding transport entity is on and empty.

If the transport entity is on and empty, the receiving participant has acquired some data from the bus cell, it has proceeded to its sending behaviour and it finally behaves again as a receiver waiting for data from the bus cell. Meanwhile, the sender participant to this same transport entity has not progressed in its behaviour, being stuck waiting for data from another bus cell, otherwise it would have sent some data to the transport entity in question that would not be empty, preventing the ungranted request. Therefore, in this case the receiver participant has performed a complete sending behaviour more recently than the sender participant.

In a more formal definition, if we consider an inequality predicate that compares which process completed a behaviour more recently, i.e. given two processes trace s_1 and s_2; $s_1 > s_2$ if and only if s_1 has completed a behaviour after the last shared behaviour and s_2 has not. For instance, if we consider traces $s_1 = \langle Send.0, Receive.0, Send.0\rangle$ and $s_2 = \langle Send.1, Receive.0\rangle$, where $Send.0$ represents the trace corresponding to a completed sending behaviour and $Receive.0$ the same for the receiving behaviour. Hence, $s_1 > s_2$ since after $Receive.0$, the shared behaviour, s_1 completed its send behaviour given by $Send.0$ and s_2 did not complete any behaviour. In this case, the ungranted request arises from a process that has completed a behaviour more recently than the process target of the ungranted request. For instance, if we consider the ungranted request shown in Figure 1, between Participant 0 and Transport Entity, Participant 0 has completed a behaviour, the *Send.0*, after the *Receive.0* behaviour, the last behaviour completed by the Transport Entity, which is also the last occurence of a shared

behaviour, since Participant 0 receives data from this bus cell. This means that, in a cycle of ungranted request after trace s, if $C(i)$ has an ungranted request to $C(i \oplus 1)$ and the inequality operator described given by $>$, then $s_i > s_{i\oplus 1}$ for any i. Hence, there cannot be a cycle of ungranted requests, since given that our network is finite, if a cycle is present this would imply that $s_i > s_i$, as our relation is transitive, a contradiction which proves our point. Hence, we argued for each possible combination of ungranted requests, the conditions why a cycle cannot occur. Therefore, by the Theorem 1 our pattern prevents a deadlock state of occurring.

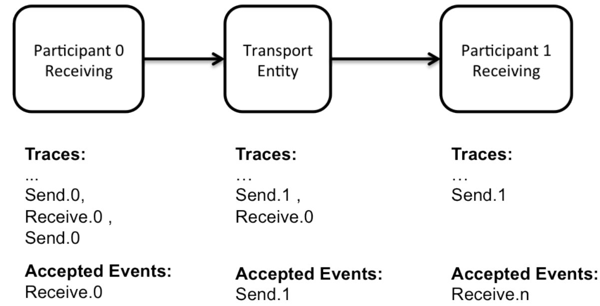


Fig. 1: Example of ungrated requests

5 Industrial Case Study: the leadership election at B&O

A critical concept in B&O product networks is that of the *dynamic global system configuration*, which describes the current combined configuration of all the products in the network. For example, the currently active user experiences (such as current song, planned playlist, volume) are stored in the system configuration, enabling the B&O system to allow the experiences to be reproduced as the user moves around the home, thus giving the impression that the experiences follow the user.

The requirements for availability and consistency of the system configuration must be realised by the communications architecture, which is based on a publisher-subscriber pattern. To enable this communication pattern the underlying network must always be able to identify a leader (the publisher). Conceptually the architecture of a B&O product network contains two global states:

- The publisher-subscriber state: a single publisher (the leader) is present and the product network can guarantee availability and consistency of user experience. All other connected products are subscribers (followers), and newly joined products are undecided, until they learn the identity of the leader.
- The election state: no publisher is present and the user experiences are inconsistent or unavailable. In this state all connected products are undecided.

In the election state a leadership protocol is executed by the products in the network. During this state each product reacts to a set of local transition rules that will guarantee the desired emergent property. A B&O network is inherently asynchronous, and therefore the algorithm must tolerate the following cases:

- Products may join or leave the network at any point during or after an election. Products may enter a power-saving state, restart because of defects, or be turned on or off by their users. As a consequence, the algorithm must handle the disappearance of leaders and the appearance of new contenders for leadership.
- Communication is asynchronous, with some latency in the network. There is, therefore, no coordination of when an election is started, and so any product can initiate an election independently. The likelihood of simultaneously initiated elections is increased by network latency.

One of the risks that such a fluid environment increases is that the protocol might reach a deadlocked state. To mitigate this, we develop a formal model of the B&O leadership election protocol and show that it is deadlock-free. B&O invests in a formal analysis of this kind because of its desire to develop and analyse models in the early design stages, before expensive implementation commitments are made.

Our leadership election model is composed of distributed nodes that store internal data in a set of memory cells; this data storage is managed by a memory controller. A node communicates with another node through a bus cell, which provides a point-to-point unidirectional communication. To illustrate these connections, a 2-node configuration architecture is given in Figure 2(a).

The nodes are distinguished by their `id` parameter, which is drawn from the set {`0..N`}. The processes prefixed with `BroadCast` specify the order in which messages are sent and received by nodes: the process `BroadCastData(id,data)` is used by node `id` to broadcast `data` to all other processes, while the process `BroadCastControl(id,..)` distributes status messages throughout the network.

A node that is off is modelled by the process `OffNode`:

```
OffNode(id, priority) = switchOn.id ->
        BroadCastControl(id,onSource,OnNode(id, max(LOWER_LIMIT_PET,priority-1)))
```

When a node is turned on it broadcasts that fact, then behaves as a switched on node. The priority is decremented as a heuristic strategy to elect a stable leader, *i.e.* the one that has the least occurrences of leaving the network. Following an initialisation, the process `Node` repeatedly checks for updates in the network configuration. Using the CSP interruption (`/\`), we specify that this behaviour can be interrupted at any time via a `switchOff` event. When switched off, the node first informs all other nodes that it has been turned off (by broadcasting the message `offSource`). This behaviour, of broadcasting messages after having been turned off, abstracts the behaviour of the B&O protocol, in which any node can always detect when another node is off.

```
OnNode(id, priority) = Node(id, <id..N>, priority, undecided)
    /\ ((switchOff.id -> BroadCastControl(id,offSource,OffNode(id, priority))) |~| STOP)
```

The main behaviour of a node, given by process `Node`, regulates the status exchange cycle between nodes, controlled by the list `<id..N>`, as well as the election process. The process `Node` either broadcasts its local state or receives status updates from other nodes. The list of nodes is re-initialised to `<id..N>` when it is empty. The local state is given by the *priority* of a node and its *claim*: its current state in the election process – `undecided`, `leader` or `follower`. After this initial broadcast it waits for the local state of each of its neighbours in turn. The node receives either the current state of the neighbour (through channel `cp_pack.in.a.id`) or a timeout event (through channel `timeout.in.a.id`) if the corresponding neighbour is turned off.

```
Node(id, <a>^list, mypriority, myclaim) =
    if a == id then BroadCastData(id, myclaim.mypriority);
        Node(id, list, mypriority, myclaim)
    else ((cp_pack.in.a.id?valC?valP -> setPack.id.a!valC!valP -> SKIP)
           []
          (commTimeout.in.id.a -> setPack.id.a!off!0 -> SKIP));
          Choice(id, <a>^list, mypriority, myclaim)
```

Each incoming message is stored, and the node reassesses its own local state in `Choice`.

```
Choice(id, <a>^list, mypriority, myclaim) =
    if myclaim == undecided then Undecided(id, <a>^list, mypriority)
    else if myclaim == leader then Leader(id, <a>^list, mypriority)
         else Follower(id, list, mypriority)
```

A `Leader` begins by retrieving the number of other nodes that are also claiming to be leaders. If this is not zero, the node becomes `undecided`; otherwise it remains a leader. The priority of a leader node is incremented (up to an upper limit) when it has completed a full cycle of status exchanges. This ensures that stable nodes are more likely to become leaders.

```
Leader(id, <a>^list, mypriority) =
    nleaders.id?valLeaders ->
      if valLeaders > 0 then Node(id, list, mypriority, undecided)
      else if id == next(a) then Node(id, list, min(UP_LMT,mypriority+1),
      leader) else Node(id, list, mypriority, leader)
```

A `Follower` remains so if there exists a leader; it becomes undecided, otherwise.

```
Follower(id, list, mypriority) =
    nleaders.id?valLeaders ->
      if valLeaders == 0 then Node(id, list, mypriority, undecided)
      else Node(id, list, mypriority, follower)
```

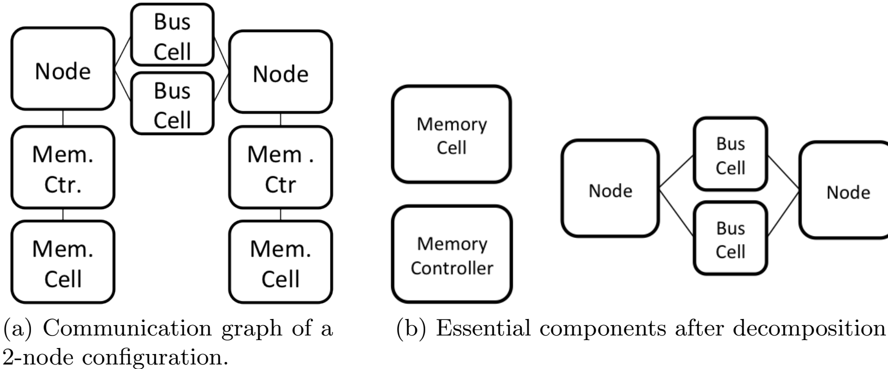
An `Undecided` node decides to lead or follow by first retrieving the number of competing leaders, the value of the highest priority among these, and the value of the largest identity among the highest priority nodes. The node follows a leader if it finds one. Otherwise, it remains undecided until the end of the status exchange cycle (`id = next(a)`). It then becomes a leader if its priority is higher than all other nodes. If multiple nodes have the same priority, the node becomes a leader if it has the highest `id`.

```
Undecided(id, <a>^list, mypriority) =
    nleaders.id?valLeaders -> hpetition.id?highest -> hpetitionid.id?highestid ->
    (let myclaim =
          if valLeaders > 0 then  follower
          else if id == next(a) then
                 if highest == mypriority and highestid < id
                      or highest < mypriority then leader
                 else follower
               else undecided
        within Node(id, list, mypriority, myclaim))
```

Communication between nodes takes place over a `Bus` that provides bidirectional communication between every pair of nodes. The `Bus` is composed of various `BusCells`, each of which provides an unidirectional channel between a source and a target node.

```
BusCell(idSource,idTarget) =
    let On(data) = cp_pack.out.idSource.idTarget?val -> On(val)
            [] data != -1 & cp_pack.in.idSource.idTarget!data -> On(-1)
            [] offSource.idSource.idTarget -> Idle
        Idle = timeout.idSource.idTarget -> Idle
            [] onSource.idSource.idTarget -> On(-1)
    within Idle
```

We create our fully connected model using the alphabetised parallel operator to connect bus cells and nodes.



(a) Communication graph of a 2-node configuration.

(b) Essential components after decomposition.

Fig. 2: Views of the system.

6 A local strategy for deadlock analysis of the leadership election and experimental results

Although, in principle, our CSP model can be fully analysed by tools like FDR, this approach to analyse the complete model for deadlock freedom is not local and incurs in an exponential growth in the number of states to be analysed, becoming infeasible at early stages. Our alternative to this problem is to use a strategy that combines the theory for deadlock analysis presented in Section 3 together with the pattern based approach that we proposed in Section 4.

The model presented for the leadership election can be decomposed based on Theorem 2. This decomposition gives the memory cells, the memory controllers, and the subnetwork as essential components. The latter is composed of interconnected nodes and bus cells as depicted in Figure 2(b).

This decomposition alone enables the local verification for deadlock freedom of both memory cells and memory controllers. Nevertheless, based on the results of [13, 12] summarised in Section 3, the verification of the subnetwork of bus cells and nodes is still left to be verified as a single component, which also leads to an exponential analysis in the number of nodes, as shown later in this section. As a major contribution of this paper, we show that, using the pattern proposed in Section 4, we need to verify only local behavioural conditions for guaranteeing deadlock freedom. As an example of how the conformance notions are encoded as assertions that can be automatically verified by FDR, we present the following two assertions, which verify the conformance of atom `BUS_CELL.0.1` to the transport entity specification and the conformance of atom `NODE.0` to the participant specification, respectively.

```
assert TransportSpec(BUS_CELL.0.1,LENetwork) [F= Abs(BUS_CELL.0.1,LENetwork)
assert ParticipantSpec(NODE.0,LENetwork) [F= Abs(NODE.0,LENetwork)
```

As expected, by conducting a full local analysis using FDR, we verified that all the restrictions imposed by the pattern are satisfied. This guarantees that that our example is indeed deadlock free.

In order to demonstrate that our local analysis avoids combinatorial explosion, we conducted a comparative analysis of three verification approaches, all using FDR: (i) analysis of the complete model; (ii) local analysis based on the decomposition supported by [13, 12], as presented in Section 3 and Figure 1b; (iii) the decomposition considered in (ii) in addition to the pattern based approach proposed in Section 4. For the analysis of our strategy (iii), we only evaluate state, transitions and time for behavioural restriction as this is the most complex task in checking pattern adherence, the predicate satisfaction time being insignificant in comparison to that time.

Our goal was to analyse a model with 32 nodes, which is the maximal number of nodes of a B&O network of devices. For this reason, we conducted the analysis for 2, 3, 4, 5, 10, 20 and 32 node instances of this model. The results are presented in Table 1, in which we provide the number of states analysed, the number of transitions, the number of processes in the network and the amount of time spent in the verification. The number of states and transitions are the ones of the Labelled Transitions System generated and analysed by FDR. The time is measured in seconds, and transitions and states in thousands. We used a dedicated server with an 8 core Intel(R) Xeon(R) 2.67GHz and 16 GB of RAM in an Ubuntu 4.4.3 system.

As expected, the exponential explosion quickly makes the leadership election model intractable by the strategy (i). The verification time for deadlock freedom for a 2-node configuration is 11 seconds, but the 3-node instance needs more than 16 GB of memory, which is beyond the configuration of the server used. Even if the decomposition strategy is applied, as described in (ii), FDR is not capable of

		(iii) Proposed strategy			(ii) Decomposed model			(i) Complete model		
Nodes	#Procs	States	Trans	Time	States	Trans	Time	States	Trans	Time
2	8	0.4	1.5	0.5	17.5	81.5	0.3	1,695	7,663	11.27
3	18	1.7	6.7	1.7	242,626	1,886,533	3,115	*	*	*
4	32	4.7	19.9	3.86	*	*	*	*	*	*
5	50	10	47	7	*	*	*	*	*	*
10	200	156	740	46	*	*	*	*	*	*
20	800	2,490	12,149	659	*	*	*	*	*	*
32	2,048	16,414	80,939	5,161	*	*	*	*	*	*

* FDR exceeds the machine's memory available

Table 1: Practical comparison

analysing further than the 3-node configuration. Also, the state space explosion in this case is very clear as the number of states leaps from about 17,500 in the 2-node configuration to 242,626,600 in the 3-node configuration. Our strategy (iii) is by far the only viable option, being able to analyse the 2,048 processes of the 32-node configuration, in 1.43 hours. Note that, in addition to the state explosion, the processes being analysed also grow in complexity as the number of nodes in the configuration increases because every node needs to communicate with more nodes, making the analysis of this example even more expensive.

7 Conclusion and Related work

In this paper, we proposed a pattern that prevents deadlocks and the formalisation of a notion of pattern conformance using first order logic and refinement expressions. We also presented a formal specification of the leadership election algorithm. This algorithm is used by one of our industrial partners, B&O, to define the *publisher* of their *publisher-subscriber* protocol, in which one of the products (the publisher) is the leader of the other products (the subscribers). We applied the proposed pattern to this industrial case study and compared the efficiency of our verification approach to a global approach.

As demonstrated by the analysis in Section 6, the verification of a complex algorithm using a global approach rapidly becomes infeasible. Our pattern based approach is a valid and promising alternative to verifying complex systems for deadlock freedom. By verifying adherence to a pattern that requires only local analysis, we were able to guarantee that a complex distributed algorithm used in industry is deadlock free. In the case of B&O, we were able to guarantee the deadlock freedom of their distributed algorithm with up to 32-nodes (the maximum number of nodes in a B&O network), involving 2,048 processes. Moreover, during the development and verification of this model several issues were identified and the real C++ implementation was modified as a result.

A CSP specification of the leadership election "Bully algorithm" of [6] is given in [14]. The assumptions on processes are similar to the ones we make: processes

may fail and revive at any time, and contain some stable storage. Although both models assume that messages will not be duplicated or lost, their network assumptions differ. The Bully algorithm does not allow a failed communication between two live processes. We model this possibility using a `timeout`. Furthermore, in the Bully algorithm, messages may not overtake each other: they must be processed in the order in which they are sent. In our model, message may overtake each other. Finally, in the Bully algorithm communication is synchronous, whilst we model asynchronous communications.

In terms of algorithm design, a leadership competition in the Bully algorithm is resolved by the node with the highest identifier "bullying" the other nodes into accepting its claim. In the case of competing claims in the algorithm we present, the primary decision mechanism is the value of `priority`, although the node `id` may be used as a last resource.

In the future, we plan to increase the range of systems to which our approach is applicable through the development and verification of new architectural patterns. Furthermore, we also intend to extend the application of our approach to other properties such as livelock-freedom. Finally, we plan to apply the pattern-based strategy to a wide spectrum of real systems.

A CSPM models

A.1 Auxiliary network definitions

Auxiliary file containing some network definitions.

```
include "../LeaderElection.csp"
-- Network model
----------------------
-- Abstract type for the atoms of the network
----------------------
datatype atomIds = NODE.NODES_IDS | MEMORY_CELL.NODES_IDS.NODES_IDS |
                        MEMORY_CONTROLLER.NODES_IDS | BUS_CELL.NODES_IDS.NODES_IDS

-----------------
-- Subnetworks of the different components of the systems
-----------------
NodesNetwork =
    { (NODE.id,OffNode(id, LOWER_LIMIT_PET),alphaNode(NODE.id))| id<-NODES_IDS}

BusCellsNetwork =
    { (BUS_CELL.idS.idR,BusCell(idS,idR),alphaBusCell(BUS_CELL.idS.idR))
    | idS<-NODES_IDS, idR <- NODES_IDS, idS != idR}

MemoryCellsNetwork =
    { (MEMORY_CELL.id.idN,CellPack(id,idN, off.0),alphaMemoryCell(MEMORY_CELL.id.idN))
    | id <- NODES_IDS, idN <- NODES_IDS, idN != id}
```

```
MemoryControllersNetwork =
    { (MEMORY_CONTROLLER.id, ControllerPack(id, 0, 0, 0),
    alphaMemoryController(MEMORY_CONTROLLER.id)) | id <- NODES_IDS}

------------------
-- Leader Election network
------------------
LeaderElectionNetwork =
    Union({NodesNetwork,BusCellsNetwork,
    MemoryCellsNetwork,MemoryControllersNetwork})

------------------
-- Subnetwork Bus cell + Nodes
------------------
SubnetworkNodesBusNetwork = Union({NodesNetwork,BusCellsNetwork})

-----------------
-- Auxiliary definition for the network model
-----------------

-- Functions to recover the ID, Behavior and Alphabet given a atomic tuple.
ID_((x,y,z)) = x
B_((x,y,z)) = y
A_((x,y,z)) = z

-- Functions to recover the Alphabet and Behaviour of an atom
-- given an Id and a Network containing this id
A(id,V) = A_(getElement(id,V))
B(id,V) = B_(getElement(id,V))

-- Specific functions to find the alphabet and behaviour of tuple identified by its id
-- in the leader election network.
A_LE(id) = A(id,LeaderElectionNetwork)
B_LE(id) = B(id,LeaderElectionNetwork)

-- Auxiliary functional definitions
pick({x}) = x
getElement(id,V) = pick({ a | a <- V, ID_(a) == id})

-- Function to recover the vocabulary of the network V
Voc(V) = Union({ inter(A_(a1),A_(a2)) | a1 <- V, a2 <- V, NEQ(a1,a2)})

-- Function to recover the alphabet of the network V
AlphaNetwork(V) = Union({ A_(a)| a <- V})
```

```
-- Function to recover the union of every alphabetical triple joint
-- Alphabetical triple joint is given by Inter({A_(a1),A_(a2),A_(a3)}) where
-- a1,a2 and a3 are three different triples
UnionTripleJoints(V) =
   Union({ Inter({A_(a1),A_(a2),A_(a3)}) |
   a1 <- V, a2 <- V, a3 <- V,NEQ(a1,a2),NEQ(a3,a2),NEQ(a1,a3)})

-- Function that gives the behaviour of a network V
Behaviour(V) = || a : V @ [A_(a)] B_(a)

-- Constant that yields the behaviour of the leader election network
BehaviourLeaderElectionNetwork = Behaviour(LeaderElectionNetwork)

-- Constant that yields the behaviour of the leader election network
BehaviourNodesBusSubNetwork = Behaviour(SubnetworkNodesBusNetwork)

-- Auxiliary definition of not equal tuples
NEQ(A1,A2) = ID_(A1) != ID_(A2)

-- The definition of commA given to the nodes and bus cells of the network
commA(NODE.id) = {|sendCh(id,n),receiveCh(n,id),onCh(id,n),
    offCh(id,n),timeoutCh(n,id)| n <- NODES_IDS|}
commA(BUS_CELL.idS.idR) = {|sendCh(idS,idR),receiveCh(idS,idR),
    onCh(idS,idR),offCh(idS,idR),timeoutCh(idS,idR)|}

print commA(BUS_CELL.0.1)

----------------------------
-- Function giving the alphabets of the atoms
----------------------------

alphaNode(NODE.id) =
    {| cp_pack.in.a.id, cp_pack.out.id.a, timeout.a.id, offSource.id.a,
    onSource.id.a, switchOn.id, switchOff.id, setPack.id.a, nleaders.id,
    hpetition.id, hpetitionid.id  | a <- NODES_IDS, a != id |}

alphaBusCell(BUS_CELL.idS.idR) =
    {|cp_pack.in.idS.idR, cp_pack.out.idS.idR,
    timeout.idS.idR, offSource.idS.idR, onSource.idS.idR|}

alphaMemoryCell(MEMORY_CELL.id.idN) =
    {| setCellPack.id.idN, getCellPack.id.idN |}

alphaMemoryController(MEMORY_CONTROLLER.id) =
```

```
    {| setCellPack.id.a, getCellPack.id.a, setPack.id.a,
    nleaders.id, hpetition.id, hpetitionid.id | a <- NODES_IDS, a != id |}
```

A.2 Bus model

CSP_M file modelling the bus.

```
-- Bus
include "../RAM.csp"
datatype IO = in | out
channel cp_pack : IO.NODES_IDS.NODES_IDS.CLAIM.DIST
channel cp_pack_bus : IO.NODES_IDS.NODES_IDS.CLAIM.DIST
channel timeout : NODES_IDS.NODES_IDS
channel onSource , offSource : NODES_IDS.NODES_IDS

-- Unitary bus cell
BusCell(idSource,idTarget) =
    let
        On(data) =
            cp_pack.out.idSource.idTarget?val -> On(val)
            [] data != -1 & cp_pack.in.idSource.idTarget!data -> On(-1)
            [] offSource.idSource.idTarget -> Idle
        Idle =
            timeout.idSource.idTarget -> Idle
            [] onSource.idSource.idTarget -> On(-1)
    within
        Idle
```

A.3 Memory model

CSP_M file modeling the memory.

```
-- Global variables
N = NODES-1
UPPER_LIMIT_PET = 1
LOWER_LIMIT_PET = 1

datatype CLAIM = leader | follower | undecided | off

nametype NODES_IDS = {0..N}
nametype DIST = {0..UPPER_LIMIT_PET}
nametype DIST_VALID = {LOWER_LIMIT_PET..UPPER_LIMIT_PET}

channel setPack : NODES_IDS.NODES_IDS.CLAIM.DIST
channel setCellPack, getCellPack :  NODES_IDS.NODES_IDS.CLAIM.DIST
```

```
channel nleaders : NODES_IDS.{0..NODES}
channel hpetition : NODES_IDS.DIST
channel hpetitionid : NODES_IDS.NODES_IDS

--------------------------
-- Cell of memory
--------------------------
Cell(chSet, chGet, idd, vald) =
    let
        CellI(value) =
            chSet.idd?val -> CellI(val)
            []
            chGet.idd.value -> CellI(value)
    within
        CellI(vald)
--------------------------
-- Memory of Petitions and Claims
--------------------------

-- A memory cell
CellPack(id,idN, vald) =
    Cell(setCellPack.id, getCellPack.id, idN, vald)

-- All Cell Memories
StatePack(id) =
    ||| idN : diff(NODES_IDS,{id}) @ CellPack(id,idN, off.0)

-- Controller
-- valOld == 0 means that the petititon has been reset

ControllerPack(id, highest_petition, highest_petition_id, leaders) =
    let
        InnerControllerPack(highest_petition, highest_petition_id, leaders) =
        leaders < NODES and leaders >= 0 &
            (
            setPack.id?idN:diff(NODES_IDS,{id})?valNewClaim?valNewPet ->
            getCellPack.id.idN?valOldClaim?valOldPet ->
                setCellPack.id.idN.valNewClaim.valNewPet ->
                    (let
                        new_highest_petition =
                            if valNewPet > highest_petition then
                                valNewPet
                            else
                                highest_petition
                        new_highest_petition_id =
```

```
                                if valNewPet > highest_petition then
                                    idN
                                else
                                    highest_petition_id
                            new_leaders =
                                if valNewClaim != valOldClaim and
                                  valNewClaim == leader then
                                    leaders+1
                                else if valNewClaim != valOldClaim and
                                  valOldClaim == leader then
                                    leaders-1
                                else
                                    leaders
                        within
                            InnerControllerPack(new_highest_petition,
                                new_highest_petition_id, new_leaders)
                        ))
        [] leaders < NODES and leaders >= 0 &
            nleaders.id!leaders ->
            InnerControllerPack(highest_petition, highest_petition_id, leaders)
        []
            hpetition.id!highest_petition ->
            InnerControllerPack(highest_petition, highest_petition_id, leaders)
        []
            hpetitionid.id!highest_petition_id ->
            InnerControllerPack(highest_petition, highest_petition_id, leaders)
    within
        InnerControllerPack(highest_petition, highest_petition_id, leaders)

-- The overall Claims memory
MemePack(id) =
    (ControllerPack(id, 0, 0, 0)
   [|MemeCellSyncSet(id)|]
    StatePack(id))
    \ MemeCellSyncSet(id)

MemeCellSyncSet(id) = {|getCellPack.id, setCellPack.id|}
--------------------------
-- Overall Memory
--------------------------
RAM(id) = MemePack(id,diff(NODES_IDS, {id}))
```

A.4 Leadership election model

CSP_M model of the leader election.

```
include "../BUS.csp"

------------------------------------------------
-- A Turned Off Node Might Initialise
------------------------------------------------
channel switchOff, switchOn : NODES_IDS

min(x,y) = if x < y then x else y
max(x,y) = if x > y then x else y

SEQ_NEIGHBOURS = <0..N>

OffNode(id, prior) =
    switchOn.id -> BroadCastControl(id,onSource,OnNode(id, max(LOWER_LIMIT_PET, prior-1)))

OnNode(id, prior) =
    Node(id, <id..N>, prior, undecided)
    /\
    ((switchOff.id -> BroadCastControl(id,offSource,OffNode(id, prior)))|~| STOP)

------------------------------------------------
-- Node Initial Behaviour After Initialisation
------------------------------------------------
Node(id, <>, mypetition, myclaim) =
    Node(id, SEQ_NEIGHBOURS, mypetition, myclaim)

Node(id, <a>^list, mypetition, myclaim) =
    if a == id then
        BroadCastData(id, myclaim.mypetition);
        Node(id, list, mypetition, myclaim)
    else
        (cp_pack.in.a.id?valC?valP ->
            setPack.id.a!valC!valP ->
                Choice(id, <a>^list, mypetition, myclaim))

        []
        (timeout.a.id ->
            setPack.id.a!off!0 ->
                Choice(id, <a>^list, mypetition, myclaim))

------------------------------------------------
-- Broadcast
------------------------------------------------
-- Broadcast can either communicate, or if a timeout is reached
```

```
-- it can only pass and try to send a message to the next peer.

BroadCastData(id, pack) =
    let
        BroadCast_Aux(<>) = SKIP
        BroadCast_Aux(<a>^list) =
            if a == id then
                BroadCast_Aux(list)
            else
                cp_pack.out.id.a!pack -> BroadCast_Aux(list)
    within
        BroadCast_Aux(SEQ_NEIGHBOURS)

BroadCastControl(id, ch,P) =
    let
        BroadCast(<>) = P
        BroadCast(<a>^list) =
            if a == id then
                BroadCast(list)
            else
                ch.id.a ->
                    BroadCast(list)
    within
        BroadCast(SEQ_NEIGHBOURS)
------------------------------------------------
-- Choice
------------------------------------------------
-- Choice and Undecided both have a new argument called timeout,
-- this argument is set to true if an election timeout occurs.

Choice(id, <a>^list, mypetition, myclaim) =
    if myclaim == undecided then
        Undecided(id, <a>^list, mypetition)
    else
        if myclaim == leader then
            Leader(id, <a>^list, mypetition)
        else
            if myclaim == follower then
                Follower(id, list, mypetition)
            else
                STOP

------------------------------------------------
-- Follower
------------------------------------------------
```

```
Follower(id, list, mypetition) =
    nleaders.id?valLeaders ->
        if valLeaders == 0 then
            Node(id, list, mypetition, undecided)
        else
            Node(id, list, mypetition, follower)

------------------------------------------------
-- Leader
------------------------------------------------

Leader(id, <a>^list, mypetition) =
    nleaders.id?valLeaders ->
        if valLeaders > 0 then
            Node(id, list, mypetition, undecided)
        else
            if id == next(a) then
                Node(id, list, min(UPPER_LIMIT_PET,mypetition+1), leader)
            else
                Node(id, list, mypetition, leader)
------------------------------------------------
-- Undecided
------------------------------------------------

Undecided(id, <a>^list, mypetition) =
        nleaders.id?valLeaders ->
        hpetition.id?highest ->
        hpetitionid.id?highestid ->
        (
            let myclaim =
                if valLeaders > 0 then
                    follower
                else if id == next(a) then
                        if highest == mypetition and highestid < id
                         or highest < mypetition then
                            leader
                        else
                            follower
                else undecided
            within
                Node(id, list,mypetition, myclaim)
        )

next(a) = (a + 1) % NODES
```

Acknowledgments

The EU Framework 7 Integrated Project COMPASS (Grant Agreement 287829) financed most of the work presented here. INES and CNPq supports the work of Marcel Oliveira: grants 573964/2008-4, 560014/2010-4 and 483329/2012-6.

References

1. R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Conference on Fundamental Approaches to Software Engineering (FASE)*, Lisbon, Portugal, March 1998.
2. M. Bernardo, P. Ciancarini, and L. Donatiello. Architecting families of software systems with process algebras. *ACM Transactions on Software Engineering and Methodology*, 11(4):386–426, 2002.
3. E. Cheung, X. Chen, H. Hsieh, A. Davare, A. Sangiovanni-Vincentelli, and Y. Watanabe. Runtime deadlock analysis for system level design. *Design Automation for Embedded Systems*, 13(4):287–310, 2009.
4. S. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 5(4):334–377, 1996.
5. Formal Systems Ltd. *FDR: User Manual and Tutorial, version 2.82*, 2005.
6. H. Garcia-Molina. Elections in a distributed computing system. *Computers, IEEE Transactions on*, C-31(1):48–59, 1982.
7. J. He, X. Li, and Z. Liu. A theory of reactive components. *Electronic Notes in Theoretical Computer Science*, 160:173–195, 2006.
8. F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.
9. R. T. Ramos, A. C. A. Sampaio, and A. C. Mota. Systematic development of trustworthy component systems. In *2nd World Congress on Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 140–156. Springer, 2009.
10. R. T. Ramos, A. C. A. Sampaio, and A. C. Mota. Conformance notions for the coordination of interaction components. *Science of Computer Programming*, 75(5):350–373, 2010.
11. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
12. A. W. Roscoe and S.D. Brookes. Deadlock analysis in networks of communicating processes. *Distributed Computing*, (4):209–230, 1991.
13. A. W. Roscoe and Naiem Dathi. The pursuit of deadlock freedom. *Information and Computation*, 75(3):289–327, December 1987.
14. A.W. Roscoe. *Understanding Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.