Formal Verification of Ring-based Leader Election Protocol using Predicate Diagrams

Cecilia E. Nugraheni

Computer Science Dept., Parahyangan Catholic University, Bandung, Indonesia.

Summary
Leader election is an important protocol in distributed computing. The objective of the protocol is to decide which process among all contributing processes in the system should be offer a particular functionality after a system crash long enough. There are two basic properties that the leader election implementation needs to obey: (1) safety: it is never the case that there are two or more leaders at the same time and (2) liveness: in a stable situation (i.e. processes stop dying for a while), a leader will eventually be elected.
In this paper we considered a ring-based leader election protocol proposed by Chang and Roberts. We have proven or verified that this protocol satisfies the both properties. The proof is done by viewing the distributed systems as parameterized systems and using a class diagram called predicate diagrams* to do the verification. We use TLA* for formalization and use TLA+ style for writing specifications.

Key words:
Leader election protocol, distributed systems, verification, TLA TLA+, predicate diagrams*.*

1. Introduction

A distributed system is a collection of independent computers that appears to its users as a single coherent system [1]. The simple examples of distributed systems are World-Wide Web - which is the collection of Web servers that jointly provide the distributed database of hypertext and multimedia documents - and the computers of a local network that provide a uniform view of a distributed file system and the collection of computers on the Internet that implement the Domain Name Service (DNS) [2].

There are a number of economic and technical reasons - including cost, performance, scalability, and reliability - that make distributed systems more attractive than centralized systems. Unfortunately, realistic distributed systems are subject to failures. These failures are usually caused by the problems with the connections (network failures) and mechanical device (drive failures). A distributed system is said to be a self-stabilizing system if it be started in any possible global state after a failure occurs by itself. This property makes this kind of system tolerant to faults which means that it can recover by itself after processors crash long enough.

In distributed systems, it is common that more than one components offer the same functionality. However, whenever a failure happens, only one of them is allow to offer a particular function. As a consequence, a component must be elected. This elected component is called the leader.

The leader election problem is a well-known and extensively studied problem [3]. The objective of the protocol is for the processes among themselves to establish the leader [3]. There are two basic properties that the leader election implementation needs to obey: (1) safety: it is never the case that there are two or more leaders at the same time and (2) liveness: in a stable situation (i.e. processes stop dying for a while), a leader will eventually be elected [3].

Self-stabilizing systems were introduced in the seminal paper of Dijkstra [4]. In that paper, Dijkstra presented three semi-uniform, self-stabilizing, ring-based protocols for mutual exclusion. Following this work, many leader election protocols have been developed. Each of them considered a certain aspect of distributed systems, such as network topology (ring [5,6,7], mesh, complete network [8,9,10], and so on), communication mechanism (asynchronous or synchronous), available topology information at processes [11] and so forth.

In this paper we are interested in verification of a ring-based leader election protocol proposed by Chang and Roberts [6]. We assume a finite number of similar components. Which 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 largest identity among all participating components.

Verification consists of establishing whether a system satisfies some property, that is, whether all possible behaviors of the system are included in the property specified. It is commonly to classify the approach to formal verification into two classes, which are the deductive and the algorithmic approach. The deductive approach which is

Manuscript received August 5, 2009
Manuscript revised August 20, 2009

based on *verification rules* reduces the system validity of a temporal property to the general validity of a set of first-order *verification conditions*. *Model checking* is the most popular algorithmic verification method. This method is fully automatic for finite-state systems. However, it suffers from the so-called *state-explosion* problem. Since the size of the state space is typically exponential in the number of components, the class of systems that can be handled by this method is limited.

The using of diagrams in verifying systems has been proposed, because they can reflect the intuitive understanding of the systems and its specification or the model. Diagram can also be seen as an abstraction of the system, where properties of the diagram are guaranteed to hold for the systems as well. In particular, the use of diagrams in verification of distributed systems can be found, for example in [12]. In [12] the author proposed the use of predicate diagrams (introduced in [13]) for analyzing a self-stabilizing algorithm.

Following [14], in this work we view distributed systems as parameterized systems which are systems that consist of several similar processes whose number is determined by an input parameter. Predicate diagram* are a class of predicate diagrams [13], which are intended as the basis for the verification of parameterized systems. This method integrates deductive verification and algorithmic techniques. The correspondence between the original specification or the model and the diagram is established by non-temporal proof obligations, whereas model checking can be used to verify properties over finite-state abstractions. We use TLA* [15] for formalization and use TLA+ [16] style for writing specifications. This approach has been successfully used in verification of reader writer algorithm [17].

This paper is structured as follows. Section 2 describes briefly the specification used for parameterized systems in TLA*. The definition of predicate diagrams* will be given in Section 3. Section 4 describes how to verify the leader election protocol using predicate diagrams*. Section 5 concludes this paper.

2. Specification of parameterized systems

In this work, we restrict on the parameterized systems which are *interleaving* and consist of finitely, but arbitrarily, *discrete* components. Let M denotes a finite and non-empty set of processes running in the system being considered. A parameterized system can be described as a formula of the form:

$$parSpec \equiv Init \wedge \Box\, [\exists k \in M: Next(k)]_v \qquad (1)$$

$$\wedge\ \forall\ k \in M : L(k)$$

where

- *Init* is a state predicate that describes the global initial condition,
- *Next*(k) is an action that characterizes the next-state relation of a process k,
- v is a state function representing the variables of the system and
- $L(k)$ is a formula stating the liveness conditions expected from the process k.

Formulas such as *Next*(k) and $L(k)$ are called parameterized actions.

3. Predicate diagrams*

Now we present a class of diagrams that can be used for the verification of parameterized systems. The underlying assertion language, by assumption, contains a finite set **O** of binary relation symbols $\prec$ that are interpreted by well-founded orderings. For $\prec \in \mathbf{O}$, its reflexive closure is denoted by $\preceq$. We write $\mathbf{O}^{=}$ to denote the set of relation symbols $\prec$ and $\preceq$ for $\prec \in \mathbf{O}$.

3.1 Definition of predicate diagrams*

A predicate diagram* is a finite graph whose nodes are labeled with sets of (possibly negated) predicates, and whose edges are labeled with parameterized actions as well as optional annotations that assert certain expressions to decrease with respect to an ordering in $\mathbf{O}^{=}$. Intuitively, a node of a predicate diagram* represents the set of system states that satisfy the formulas contained in the node. An edge (n,m) is labeled with a parameterized action $A(k)$ if $A(k)$ can cause a transition from a state represented by n to a state represented by m. A parameterized action $A(k)$ may have an associated fairness condition; fairness conditions apply to all transitions labeled by the action rather than to individual edges.

Formally, the definition of predicate diagrams* is relative to two finite sets **P** and **A** that contain the state predicates and the parameterized actions of interest; we will later use τ to denote a special stuttering action. We write $Cl(\mathbf{P})$ to denote the set of literals formed by the predicates in **P**, that is, the union of **P** and the negations of the predicates in **P**. Assume given two finite sets **P** and **A** of state predicates and parameterized actions. A predicate diagram* $G = (N, I, \delta, o, \zeta)$ over **P** and **A** consists of:

- a finite set $N \subseteq 2^{Cl(\mathbf{P})}$ of nodes,
- a finite set $I \subseteq N$ of initial nodes,
- a family of $\delta = (\delta_{A(k)})_{A(k)\in\mathbf{A}}$ of relations $\delta_{A(k)} \subseteq N \times N$,

- an edge labeling o that associates a finite set $\{(t_1, \prec_1), \ldots, (t_d, \prec_d)\}$, of terms t_i paired with a relation $\prec_i \in \mathbf{O}^{=}$ with every edge $(n,m) \in \delta$, and
- a mapping $\zeta : \mathbf{A} \rightarrow \{NF,WF,SF\}$ that associates a fairness condition with every parameterized action in **A**; the possible values represent no fairness, weak fairness, and strong fairness.

We say that the parameterized action $A(k)$ can be taken at node $n \in N$ iff $(n,m) \in A$ holds for some $m \in N$, and denote by $En(A(k)) \subseteq N$ the set of nodes where $A(k)$ can be taken. We say that the parameterized action $A(k)$ can be taken along an edge (n,m) iff $(n,m) \in \delta_{A(k)}$.

We now define runs and traces through a diagram as the set of those behaviors that correspond to fair runs satisfying the node and edge labels. To evaluate the fairness conditions we identify the enabling condition of a parameterized action $A(k)$ with the existence of $A(k)$-labeled edges at a given node. We use the symbol $\aleph$ to denote the natural numbers.

Let $G = (N, I, \delta, o, \zeta)$ be a predicate diagram* over sets **P** and **A**. A run of G is an ω-sequence $\sigma = (s_0,n_0,A_0)(s_1,n_1,A_1) \ldots$ of triples where s_i is a state, $n_i \in N$ is a node and A_i is a parameterized action such that all of the following conditions hold:

- $n_0 \in I$ is an initial node.
- $s_i|[n_i]|$ holds for all $i \in N$.
- For all $i \in I$, either $A_i = \tau$ and $n_i = n_{i+1}$ or $A_i \in \mathbf{A}$ and $(n_i,n_{i+1}) \in \delta_{Ai}$.
- If $A_i \in \mathbf{A}$ and $(t, \prec) \in o(n_i,n_{i+1})$, then $s_{i+1}|[t]| \prec s_i|[t]|$.
- If $A_i = \tau$ then $s_{i+1}|[t]| \prec s_i|[t]|$ holds whenever $(t, \prec) \in o(n_i,m)$ for some $m \in N$.
- For every parameterized action $A(k)$ such that $\zeta(A(k)) = WF$ there are infinitely many $i \in \aleph$ such that either $A_i = A(k)$ or $n_i \notin En(A(k))$.
- For every parameterized action $A(k)$ such that $\zeta(A(k)) = SF$, either $A_i = A(k)$ holds for infinitely many $i \in \aleph$ or $n_i \notin En(A(k))$ holds for only finitely many $i \in \aleph$.

We write $runs(G)$ to denote the set of runs of G. The set $tr(G)$ of traces through G consists of all behaviors $\sigma = s_0s_1 \ldots$ such that there exists a run $\rho = (s_0,n_0,A_0)\ (s_1,n_1,A_1) \ldots$ of G based on the states in σ.

Informally, $\sigma = s_0s_1 \ldots$ is a trace through the predicate diagram* G if we can find a sequence of nodes n_i whose associated formulas are true at s_i and that are related by transitions whose edge labels, including the ordering annotations, are satisfied by consecutive states. In addition to the transitions that are explicitly represented by edges of the diagram, we allow stuttering transitions that remain in the source node.

Fairness conditions are used to prevent infinite stuttering. Their interpretation is standard, based on the intuition that the enabledness of actions with non-trivial fairness requirements is reflected in the diagram.

3.2 Verification using predicate diagrams*

The verification process using predicate diagrams is done in two steps [14]. The first step is to find a predicate diagram that can be proven to be the correct representation of the system to be verified, i.e. the diagram conforms to the system specification. For proving whether a diagram conforms to a specification or not, the so-called conformance theorem is used. Thus the first step is done deductively.

With the current setting, i.e. the using of parameterized actions, some modifications should be done on the conformance theorem. In particular, the conditions related to the fairness conditions should be treated slightly differently from non-parameterized ones. We need to address one important issue that will be used later, which is the issue about fairness. Note that in the specification the fairness condition is represented as a conjunction of formulas of the forms $\forall\ k \in M$: $WF_v(A(k))$ and/or $\forall\ k \in M$: $SF_v(A(k))$, i.e. for every process k in M and for some parameterized action $A(k)$, we associate weak and strong fairness, respectively, with $A(k)$. Let's turn to the definition of predicate diagrams, in particular the definition of ζ. In the context of parameterized systems, ζ: $\mathbf{A} \rightarrow \{NF,WF,SF\}$ is now a mapping that associates a fairness condition with every parameterized action $A(k)$ in **A**. For example, for some parameterized action $A(k)$, if $\zeta(A(k)) = WF$ then we mean $\zeta(\exists k \in M : A(k)) = WF$.

We say that a predicate diagram* G *conforms* to a parameterized program *parSpec* if every behavior that satisfies *parSpec* is a trace through G.

Theorem 1. Let $G = (N, I, \delta, o, \zeta)$ be a predicate diagram* over **P** and **A** and let $parSpec \equiv Init \wedge \Box[\exists k \in M: Next(k)]_v \wedge \forall k \in M: L(k)$ be a parameterized system. If all the following conditions hold then G conforms to *parSpec*:

1. $\models Init \Rightarrow \bigvee_{n \in I} n$.
2. $\approx n \wedge [\exists k \in M: Next(k)]_v \rightarrow n' \vee \bigvee_{(m,A(k)):(n,m) \in \delta_{A(k)}} \langle A \rangle_v \wedge m'$.
3. For $n,m \in N$ and all $(t, \prec) \in o(n,m)$:
   a. $\approx n \wedge m' \wedge \bigvee_{A:(n,m) \in \delta_{A(k)}} \langle \exists k \in M: A(k) \rangle_v \rightarrow t' \prec t$.
   b. $\approx n \wedge [\exists k \in M: Next(k)]_v \wedge n' \Rightarrow t' \preceq t$.
4. For every action $A(k) \in \mathbf{A}$ such that $\zeta(A(k)) \neq NF$:

a. If $\zeta(A(k))$ = WF then $\models$ *parSpec* $\rightarrow$ $\mathrm{WF}_v(\exists k \in M{:}Next(k))$.
b. If $\zeta(A(k))$ = SF then $\models$ *parSpec* $\rightarrow$ $\mathrm{SF}_v(\exists k \in M{:}Next(k))$.
c. $\models n \rightarrow \exists k \in M{:}\mathrm{ENABLED}\langle A(k)\rangle_v$ holds whenever $n \in En(A(k))$.
d. $\models n \wedge \langle A(k)\rangle_v \rightarrow \neg m$' holds for all $n,m \in N$ such that $(n,m) \notin \delta_{A(k)}$.

Condition 1 asserts that every initial state of the system must be covered by some initial node. This ensures that every run of the system can start at some initial node of the diagram. Condition 2 asserts that from every node, every transition, if it is enabled then it must have a place to go, i.e., there is a successor node which represents the successor state of the transition. It proves that every run of the system can stay in the diagram. Condition 3 is related to the ordering annotations and Condition 4 is related to the fairness conditions.

The second verification step is to prove that all traces through a predicate diagram satisfy some property *F*. On this case, we view the diagram as a finite transition system that is amenable to model checking. All predicates and actions that appear as labels of nodes or edges are then viewed as atomic propositions.

Regarding predicate diagrams* as finite labeled transition systems, their runs can be encoded in the input language of standard model checkers such as SPIN [18]. Two variables indicate the current node and the last action taken. The predicates in **P** are represented by boolean variables, which are updated according to the label of the current node, non-deterministically, if that label contains neither **P** nor $\neg$**P**. We also add variables $b_{(t,\prec)}$, for every term *t* and relation $\prec \in \mathbf{O}$ such that $(t,\prec)$ appears in some ordering annotation $o(n,m)$. These variables are set to 2 if the last transition taken is labeled by $(t,\prec)$, to 1 if it is labeled by $(t,\preceq)$ or is stuttering transition and to 0 otherwise. Whereas the fairness conditions associated with the actions of a diagram are easily expressed as LTL (Linear Temporal Logic) assumptions for SPIN.

4. Chang and Roberts' protocol

We now consider the leader election protocol proposed by Chang and Roberts. Chang and Roberts is a ring-based leader election algorithm used to find a process with the largest identification. It is a useful method of election in decentralized distributed computing. We take the interleaving version of this protocol, which means every time only one process is active. The informal description of the protocol is given as follows.

The algorithm assumes there exist a set *M* of processes running in the system and that each process has a Universal Identification (UID) and also that the processes can arrange themselves in a unidirectional ring with a communication channel going to the clockwise (successor) and anticlockwise (previous) neighbor. The 2 part algorithm can be described as follows:

1 Initially each process in the ring is marked as non-participant.
2 A process that notices a lack of leader starts an election. It marks itself as participant and creates an election message containing its UID. It then sends this message clockwise to its neighbor.
3 When a process receives an election message it compares the UID with its own, if the current process has a larger UID it replaces the one in the election message with its UID. The process the marks itself as participant and again forwards the election message in a clockwise direction.
4 If the process was already marked as participant when it receives an election message the procedure is different. In this case it will compare the UID as before but only forward the election message if it has needed to replace the UID.

The algorithm finishes when a process receives an election message containing its own UID. Then the second stage of the algorithm takes place

1 This process marks itself a non-participant and sends an elected message to its neighbor announcing its election and UID.
2 When a process receives an elected message it marks itself as non-participant records the elected UID and again forward the elected message.
3 When the elected message reaches the newly elected process the election is over.

Assuming there are no failures this algorithm will finish. We may also notice that the participation and non-participation states are used so that when 2 or more processes start an election at roughly the same time only a single winner will be announced.

The formal specification of the protocol is given in Figure 1. This protocol is modeled in terms of four sets *np*, *p*, ∂ and *f* that contains (the identification of) non-participant, participant, leader and failed processes, respectively. Besides the four sets, we also use two arrays namely *Msg* and *tMsg*. The elements of arrays *Msg* and *tMsg* are consist of two parts. The first part is an integer which represents a UID and the second part is an indicator whether the message is an election message when the value is FALSE and an elected message when the value is TRUE. Each *k*-th element of *Msg* and *tMsg* represents the

newest and the *last* message received by process *k* that comes from the previous neighbor, respectively.

Initially every process is in *np* so that *np* is equal to *M*, and the three other sets are empty. The elements of array *Msg* and *tMsg* are set to ⟨0, FALSE⟩.

Action *Start*(*k*) can be taken on two conditions. The first condition is that the *k* is in *np* which means that the process is a non-participant process. The second one is that the first part of the *k*-th element of *Msg* is 0. This represents the situation in where a process has not received any messages yet.

The election process is modeled by two separated actions: *Election1*(*k*) and *Election2*(*k*). *Election1*(*k*) is active whenever a process *k* is a non-participant process but has already received an election message from its previous neighbor. Whereas *Election2*(*k*) is active whenever the process *k* is already a participant and it received an election message with larger UID than its own.

The *Elected*(*k*) can be taken only by process which will be the leader. The *Failed*(*k*) is similar to *Election2*(*k*), only now the process *k* received an elected message rather that an election message. As consequence process *k* is failed to be the leader.

Notice that in this version we don't take into account the communication process between two processes, in particular the sending message. We prevent a process from sending the same messages over and over to its successor neighbor by using *tMsg*. Some parameterized actions, for example *Election1*(*k*), are active only if the content of *k*-th element of *Msg*[*k*] and *tMsg*[*k*] are different. This guarantees that *Election1*(*k*) is active only if process *k* received a new message from its previous process.

We will verify two basic properties that the leader election implementation obey the : (1) safety: it is never the case that there are two or more leaders at the same time; (2) – liveness: in a stable situation (i.e. processes stop dying for a whie), a leader will eventually be elected. The two properties can be expressed as formulas:

$$LdrElct \rightarrow \Box(\forall\, i,j \in M : i \in \partial \wedge j \in \partial \rightarrow i = j) \qquad (2)$$

$$LdrElct \rightarrow \Box\,((\partial = \{\} \wedge \mathrm{p} \neq \{\}) \rightarrow \Diamond\,(\partial \neq \{\})) \qquad (3)$$

Figure 2 depicts the suitable predicate diagram* for this protocol. The number outside each node is not the part of the diagram. We use this numbering only for explanation purpose. For the sake of clearness, we put the parameterized action label on the left side of the nodes and the ordering annotations on the right side of the nodes.

On node 2, 3, 4 and 5 we put some ordering annotations for avoiding infinite loops on those nodes. On node 2 we put four annotations. The first annotation, (|{*j*:*Msg*[*j*][1]=0}|,<) guarantee that eventually the action *Start*(*k*) can not be taken since the set {*j*:*Msg*[*j*][1]=0} is eventually empty. The second annotation (|*np*|,<) is used to prevent the action *Election1*(*k*) to be active forever. This is because *np* is finite and eventually is empty. Two other annotations are used to guarantee that the action *Election2*(*k*) eventually cannot be taken. The explanation of the ordering annotations on nodes 3, 4 and 5 are quite similar.

Using Theorem 1 we can prove that the diagram conforms to the specification in Figure 1. From the diagram in Figure 2 we can produce 21 verification conditions. Some of those conditions are:

- $Init \rightarrow |np| > 0 \wedge |p| = 0 \wedge |\partial| = 0 \wedge |f| = 0$
- $(|np| > 0 \wedge |p| = 0 \wedge |\partial| = 0 \wedge |f| = 0) \wedge [\exists k \in M : Next(k)]_v \rightarrow (|np'| > 0 \wedge |p'| = 0 \wedge |\partial'| = 0 \wedge |f'| = 0) \vee \langle \exists k \in M : Start(k) \rangle_v \wedge \quad (|np'| > 0 \wedge |p'| > 0 \wedge |\partial'| = 0 \wedge |f'| = 0)$
- $(|np| > 0 \wedge |p| > 0 \wedge |\partial| = 0 \wedge |f| = 0) \wedge [\exists k \in M : Next(k)]_v \rightarrow (|np'| > 0 \wedge |p'| > 0 \wedge |\partial'| = 0 \wedge |f'| = 0) \vee \langle \exists k \in M : Election1(k) \rangle_v \wedge (|np'| = 0 \wedge |p'| > 0 \wedge |\partial'| = 0 \wedge |f'| = 0)$
- $(|np| > 0 \wedge |p| > 0 \wedge |\partial| = 0 \wedge |f| = 0) \wedge \quad (|np'| > 0 \wedge |p'| > 0 \wedge |\partial'| = 0 \wedge |f'| = 0) \wedge \langle \exists k \in M : Election1(k) \rangle_v \rightarrow |\{j:Msg[j][1]=0\}'| < |\{j:Msg[j][1] = 0\}|$

The next step is to encode the predicate diagram* in Promela, the input language of SPIN. To do this, six variables are used which are *action*, *node*, *np*, *p*, *ldr*, and *f*. *action* and *node* are used to indicate the last action taken and the current node; whereas *np*, *p*, *ldr*, and *f* are used to represent the predicate that hold on every node, for example, if *np* = 0 then the predicate |*np*| = 0 holds and if *np* = 1 the the predicate |*np*| > 0 holds. *action* = 1 if *Start*(*k*) is taken, action = 2 if *Election1*(*k*) is taken and so on.

The properties to be verified are now can be written as $\Box(ldr = 0 \vee ldr = 1)$ and $\Box\,((ldr = 0 \wedge p = 1) \rightarrow \Diamond\,(ldr = 1))$. Last, by using SPIN we model-check the diagram. As result, we concluded that the protocol satisfies the two properties.

5. Conclusion and future work

We have shown that the leader election protocol proposed by Chang and Roberts satisfy the safety and liveness properties as required. We have viewed the distributed

systems as parameterized systems. The verification is then done by using predicate diagrams*.

There are many work that are devoted to the formal specification and verification of distributed systems, in particular leader-election protocol. Some of them are [3, 5, 12, 14]. This work is very closed to [12, 14]. The similarity between this work and [12] is that we use the diagram-based approach to do the verification. We also use TLA to formalize our approach. However, [12] did not treat the distributed systems as parameterized systems. Following [14], in this work we view distributed systems as parameterized systems. In [14] the verification of parameterized systems is done pure deductively. Whereas the difference between [14] and this work is we proposed the use of predicate diagram* for the verification process, rather than do the verification pure deductively.

In the context of parameterized systems, there are two classes of properties may be considered, namely the properties related to the whole processes and the ones related to a single process in the system. The latter class is sometimes called the universal property. In this work we only consider the properties which are related to the whole processes. It is planned to investigate the universal properties of the protocol, such as once a process becomes a participant then eventually it will be the leader or not. In this case we will use a variant of predicate diagram* which is parameterized predicate diagrams [19, 20].

$$
\begin{aligned}
\mathit{Init} &\equiv \land \forall\, k \in M : \mathit{Msg}[k] = \langle 0, \text{FALSE}\rangle \land \mathit{tMsg}[k] = \langle 0, \text{FALSE}\rangle \\
&\quad \land \mathit{np} = M \land p = \{\} \land \partial = \{\} \land f = \{\} \\
\mathit{Start}(k) &\equiv \land k \in \mathit{np} \land \mathit{Msg}[k][1] = 0 \\
&\quad \land \mathit{np}' = \mathit{np} \setminus \{k\} \land p' = p \cup \{k\} \land \partial' = \partial \land f' = f \\
&\quad \land \mathit{Msg}' = [\mathit{Msg} \text{ EXCEPT!} \mathit{succ}(k) = \langle k, \text{FALSE}\rangle \land \mathit{tMsg}' = \mathit{tMsg} \\
\mathit{Election1}(k) &\equiv \land \mathit{Msg}[k] \neq \mathit{tMsg}[k] \land k \in \mathit{np} \land \mathit{Msg}[k][1] \neq 0 \\
&\quad \land \mathit{np}' = \mathit{np} \setminus \{k\} \land p' = p \cup \{k\} \land \partial' = \partial \land f' = f \\
&\quad \land \lor \mathit{Msg}[k][1] > k \land \mathit{Msg}' = [\mathit{Msg} \text{ EXCEPT!} \mathit{succ}\{k\} = \langle \mathit{Msg}[k][1], \\
&\quad \text{FALSE}\rangle] \\
&\quad\quad \lor \mathit{Msg}[k][1] < k \land \mathit{Msg}' = [\mathit{Msg} \text{ EXCEPT!} \mathit{succ}\{k\} = \langle k, \text{FALSE}\rangle] \\
&\quad \land \mathit{tMsg}' = [\mathit{tMsg} \text{ EXCEPT!} k = \mathit{Msg}[k]] \\
\mathit{Election2}(k) &\equiv \land \mathit{Msg}[k] \neq \mathit{tMsg}[k] \land k \in p \land \mathit{Msg}[k][1] > k \\
&\quad \land \mathit{np}' = \mathit{np} \quad \land p' = p \land \partial' = \partial \land f' = f \\
&\quad \land \mathit{Msg}' = [\mathit{Msg} \text{ EXCEPT!} \mathit{succ}(k) = \mathit{Msg}[k]] \\
&\quad \land \mathit{tMsg}' = [\mathit{tMsg} \text{ EXCEPT!} k = \mathit{Msg}[k]] \\
\mathit{Elected}(k) &\equiv \land \mathit{Msg}[k] \neq \mathit{tMsg}[k] \land k \in p \land \mathit{Msg}[k][1] = k \\
&\quad \land \mathit{np}' = \mathit{np} \land p' = p \setminus \{k\} \land \partial' = \partial \cup \{k\} \land f' = f \\
&\quad \land \mathit{Msg}' = [\mathit{Msg} \text{ EXCEPT!} \mathit{succ}(k) = \langle k, \text{TRUE}\rangle] \\
&\quad \land \mathit{tMsg}' = [\mathit{tMsg} \text{ EXCEPT!} k = \mathit{Msg}[k]] \\
\mathit{Failed}(k) &\equiv \land \mathit{Msg}[k] \neq \mathit{tMsg}[k] \land k \in p \land \mathit{Msg}[k][1] \neq k \land \mathit{Msg}[k][2] = \text{TRUE} \\
&\quad \land \mathit{np}' = \mathit{np} \quad \land p' = p \setminus \{k\} \land \partial' = \partial \land f' = f \cup \{k\} \\
&\quad \land \mathit{Msg}' = [\mathit{Msg} \text{ EXCEPT!} \mathit{succ}(k) = \langle k, \text{TRUE}\rangle] \\
&\quad \land \mathit{tMsg}' = [\mathit{tMsg} \text{ EXCEPT!} k = \mathit{Msg}[k]] \\
v &\equiv \langle \mathit{np}, p, \partial, f, \mathit{Msg}, \mathit{tMsg}\rangle \\
\mathit{Next}(k) &\equiv \mathit{Start}(k) \lor \mathit{Election1}(k) \lor \mathit{Election2}(k) \lor \mathit{Elected}(k) \lor \mathit{Failed}(k) \\
L(k) &\equiv \land \mathrm{WF}_v(\mathit{Start}(k)) \land \mathrm{WF}_v(\mathit{Election1}(k)) \land \mathrm{WF}_v(\mathit{Election2}(k)) \land \\
&\quad \mathrm{WF}_v(\mathit{Elected}(k)) \\
&\quad \land \mathrm{WF}_v(\mathit{Failed}(k)) \\
\mathit{LdrElct} &\equiv \mathit{Init} \land \Box[\exists\, k \in M : \mathit{Next}(k)]v \land \forall k \in M : L(k)
\end{aligned}
$$

Figure 1. Formal specification of Chang and Roberts' Leader Election protocol.

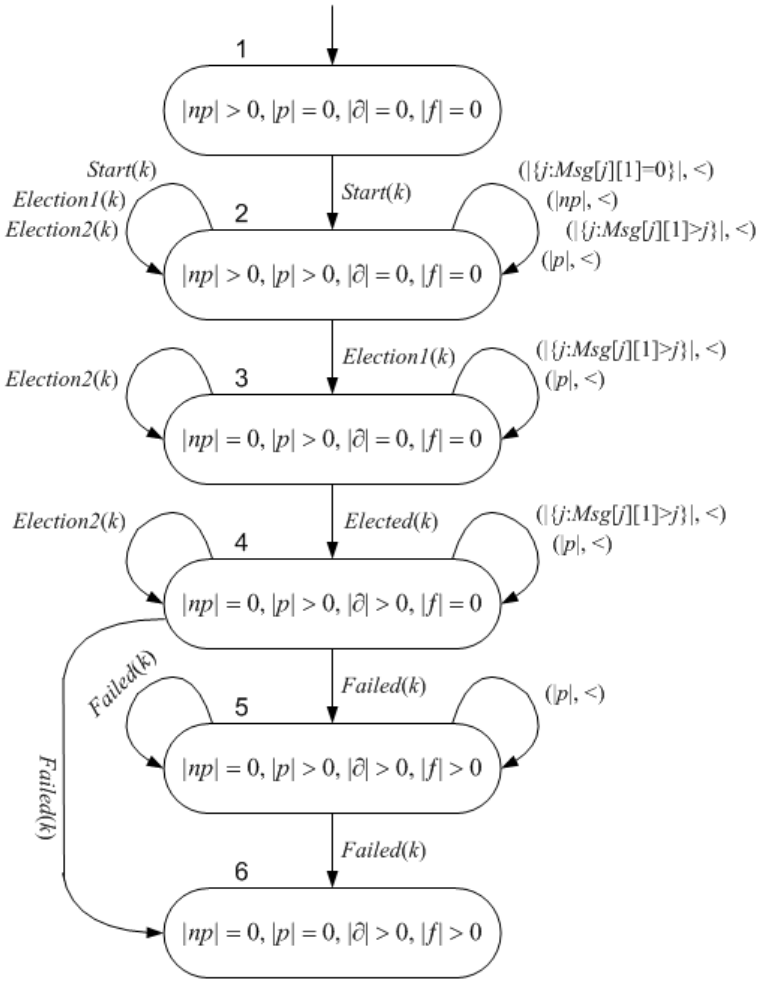

Figure 2. Predicate diagram* for Chang and Roberts' Leader Election protocol.

References

[1] Andrew S. Tanenbaum and Maarten van Steen. Distributed Systems: Principles and Paradigms. 2nd ed. Prentice Haill. 2007. ISBN-13: 9780132392273.

[2] Ihor Kuz, Manuel M.T. Chakravarty, and Gernot Heiser. A Distributed Systems. School of Computer Science and Engineering, the University of New South Wales. http://gernot-heiser.org/~cs9243/lectures/intro-notes.pdf.

[3] Thomas Arts, Koen Claessen, and Hans svensson. Semi-formal development of a fault-tolerant leader election protocol in Erlang. FATES 2004: 140-154.

[4] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. Comm. of the ACM (ACM) 17 (11): 643-644, 1974.

[5] G. Lelann. Distributed systems towards a formal approach. In B. Gilchrist, ed., Information Processing vol. 77, pp. 155-160. 1977.

[6] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. Comm. of the ACM (ACM) 22(5):281-283, 1979.

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

[8] 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, pp. 199-207. ACM, 1984.

[9] G. Singh. Efficient distributed algorithms for leader election in complete networks. In Proc 11th IEEE Intl. Conf. on Distributed Computing Systems, pp. 472-479, 1991.

[10] 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.

[11] 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.

[12] D. Cansell, D. Méry and S. Merz Formal analysis of a self-stabilizing algorithm using predicate diagrams. GI Jahrestagung (1) 2001: 628-634.

[13] D. Cansell, D. Méry and S. Merz, , "Predicate diagrams for the verification of reactive systems", Intl. Conf. on

Integrated Formal Methods (IFM 2000), vol 1945 of Lecture Notes in Computer Science, Springer-Verlag, 2000.
[14] Nikolaj S. Bjørner, Uri Lerner, and Zohar Manna. Deductive verification of parameterized fault-tolerant systems: a case study.
[15] Stephan Merz. Logic-based analysis of reactive systems: hiding, composition and abstraction. Habilitationsschrift. Institut für Informatik. Ludwig-Maximilians Universität, Munich, Germany, 2002.
[16] Leslie Lamport. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, 2002.
[17] Cecilia E. Nugraheni. Diagram-based verification of parameterized systems. JCMCC 65 (2008), pp. 91-102.
[18] G. Holzmann. The SPIN Model checker. IEEE Trans. Of software engineering, 16(5):1512-1542, 1997.
[19] Cecilia E. Nugraheni. Predicate diagrams as basis for the verification of reactive systems. PhD Thesis. Institut für Informatik. Ludwig-Maximilians Universität, Munich, Germany, 2004.
[20] Cecilia E. Nugraheni. Universal properties verification of parameterized parallel systems. Lecture Notes in Computer Science Vol. 3482. Springer. 2005.

Cecilia E. Nugraheni received the B.S. and M.S. degrees in Informatics Engineering from Bandung Institute of Technology in 1993 and 1997, respectively. In 2004 she got her PhD degree from Institut für Informatik, Ludwig-Maximilians Universität, Munich, Germany. She is now an academic staff at Computer Science Dept., Parahyangan Catholic University, Bandung, Indonesia.