# Modeling and Verifying Leader Election Algorithm in CSP

Yucheng Fang Huibiao Zhu* Huiwen Wang
Shanghai Key Laboratory of Trustworthy Computing,
School of Computer Science and Software Engineering, East China Normal University, China

***Abstract*—Leader election is a fundamental problem in distributed systems and has a variety of applications in wireless networks, such as key distribution, routing coordination, and general control. The main statement of the leader election problem is to eventually elect a unique leader from a fixed set of nodes. As the wireless network is becoming more and more important in daily life, leader election algorithm plays a vital important role in wireless network, which makes the correctness and robustness of such algorithms become evermore important and challenging to establish. In this paper, firstly, we study an election algorithm LE for MANETs (Mobile Ad Hoc Network) designed by Vasudevan et al. Then we present a formal model for LE based on process algebra CSP (Communicating Sequential Process). Modeling algorithm like LE sometimes pose non-trivial challenges, time, geometry, communication delays and failures, mobility and bi-directionality can interact in unforeseen ways that are hard to model and analyze by automatic formal methods, but we will take on these challenges. On that basis, we use the model checker FDR (Failures Divergence Refinement) to automatically simulate the developed model and verify whether the model is consistent with the specification and exhibits relevant secure properties. Our results show the correctness and safety of LE in this respect.**

***Index Terms*—leader election, formal methods, CSP, FDR**

## I. Introduction

In a network, leader election algorithm is to select a unique leader of each node in a network. It is a fundamental control problem in distributed systems and has a variety of applications in wireless networks, such as key distribution [7], routing coordination [2] and sensor coordination [8].

The purpose of electing a leader in an interconnected network is to permit the control of the network by a unique node in order to perform a specific action or activity with the other members of the network. Several algorithms have been proposed to solve this problem such as [5], [6]. Only a few of the proposed algorithms can be applied to MANETs. In this paper, we focus on the algorithm, which is called LE, designed by Vasudevan et al. in [11]. It aims at electing the most-valued node according to some measure, e.g., the amount of remaining battery life in a network. In LE, several spanning trees were established. Then, these spanning trees were reduced to a unique spanning tree and the root to decide which is the leader in the network. In this paper, we mainly model LE in a context of static topology, under the assumption that nodes with its neighbors are fixed the nodes number of the network would not increase when LE is running. We give a detailed explanation in next section. Our intention is to prove that an abstraction of the LE works correctly.

* Corresponding Author. Email: hbzhu@sei.ecnu.edu.cn

There actually exist many research on leader election algorithm [1], [13]. Most of them use model checking based on a specific network such as ring [13] but seldom of them consider to verify LE. In this paper we formalize LE based on process algebra CSP and we concentrate on the status changing of a node, modeling the operations of each node. Our work build a baseline for verifying LE in CSP way. Based on the formalized model, we use FDR to automatically simulate the achieved model and verify whether it caters for some significant properties such as deadlock freedom, divergence-free and unique leader scheme.

CSP is a formal language for describing patterns of interaction in concurrent systems [9] and it has been practically applied in industry as a tool for specifying and verifying the concurrent aspects of a variety of different systems, such as [12]. There are many model checkers for CSP, such as FDR [3], Process Analysis Toolkit (PAT) [10] and so on. FDR has the best performance among them because it includes a parallel refinement-checking engine that achieves a linear speed-up as the number of cores increase. It is able to check processes with billions of states, and is able to make efficient use of on-disk storage to complement memory.

The remainder of this paper is organized as follows. In Section 2, we give a brief introduction to LE, and the process algebra CSP. In Section 3, we model LE in CSP and In Section 4, we give three basic properties and verify that the LE model respects them. In Section 5, we conclude and outline the future work.

## II. Background

In this section, we give a brief introduction to LE algorithm and give an example to illustrate. Further, we also present the relevant introduction on CSP.

### A. Brief Introduction to LE

When an election is triggered at a node, the node broadcasts an *election* message to its immediate neighbors (one hop neighbors). A node that receives an *election* message for the first time, records the sender of the message as its *parent* in the spanning tree under construction, and multicasts an *election* message to its other immediate neighbors. When a node receives an *election* message from a node that is not its parent, it immediately responds with an *ack* message. When

DOI reference number: 10.18293/SEKE2018-067

a node has received *ack* messages from all of its children, it sends an *ack* message to its parent. Each such *ack* message to a parent includes the identity and value of the most-valued node in the subtree rooted at the sender. Therefore, when the source node has received an *ack* message from all of its children, it can determine the most-valued node in the entire spanning tree. The source node then broadcasts a *leader* message to all of its immediate neighbors to announce a new leader. When a node receives a *leader* message, it updates its own leader and broadcasts it to its immediate neighbors.

Fig. 1 shows a run of LE under a static topology of five nodes, with node 1 being the source and node 5 being the most-valued node. In this figure, thin arrows indicate the direction of flow of messages and thick arrows indicate parent pointers. These parent pointers together represent the constructed spanning tree. Node 1 starts its diffusing computation by sending out *election* messages to its immediate neighbors 2 and 3, shown in Fig. 1(a). As indicated in Fig. 1(b), nodes 2 and 3 set its parent pointer to point node 1 and in turn propagate an *election* message to all their neighbors except their parent nodes. Hence 2 and 3 send *election* to each other, as we explain before, they will send *ack* to each other immediately but not taken the other as its parent. In Fig. 1(c), a complete spanning tree is built. In Fig. 1(d), nodes 4 and 5 send its value to its parent nodes, since they are the leaves of the tree. Eventually, the source 1 hears pending acknowledgments from both 2 and 3 in Fig. 1(e) and then broadcasts the identity of the leader, 5, via *leader* message shown in Fig. 1(f).

Multiple nodes can concurrently initiate multiple elections; in this case, only one election should "survive". This is done by associating to each election a *priority*, so that a node already in an election ignores incoming elections with lower priority, but participates in an election with higher priority. In some cases, a node maybe *fail* and will not send *ack* to its parent. To handle the case like this, every node sets a expire time $T$, when the time exceeds $T$, the node will be removed from its parent's waiting list. The report [11] gives a detailed pseudo-code specification of LE.

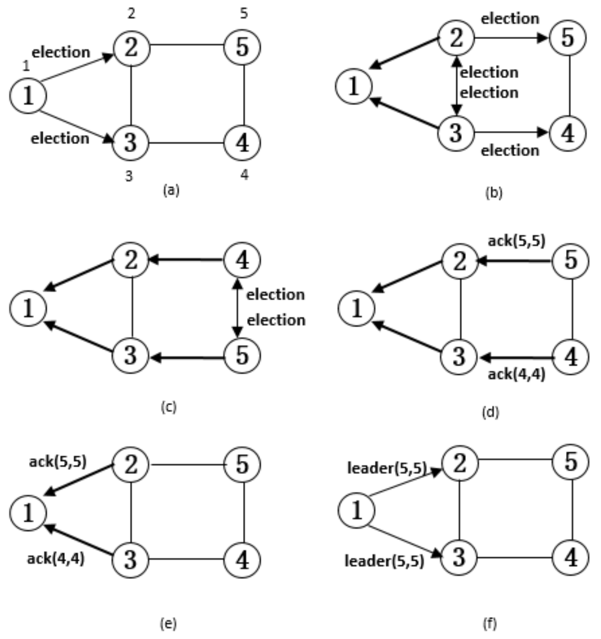


Fig. 1. An LE run in a static topology

### B. Brief Introduction to CSP

The CSP method, abbreviation for Communicating Sequential Processes, is first proposed by C.A.R Hoare [4]. It is a process algebra designed mainly for analyzing the behaviors of concurrent processes. In CSP, a synchronous communication mechanism holds when processes communicate with each other to coordinate the parallel executions. The syntax of a subset of the CSP language is given as follows.

$$P, Q = Skip \mid Stop \mid a \to P \mid c?x \to P \mid c!x \to P \mid P \,\square\, Q$$
$$P \,\Big|\Big|\, Q \mid P \,|||\, Q \mid Q \setminus M \mid P;\ Q \mid if\ b\ then\ P\ else\ Q$$

where:

- *Stop* represents that the process does nothing and its state is deadlock.
- *Skip* stands for a process which terminates successfully.
- $a \to P$ first performs the event $a$ then behaves like $P$.
- $c!v \ \to\ P$ sends message $v$ through channel $c$, then performs like $P$.
- $c?x \ \to\ P$ receives a message through channel $c$ and assigns it to a variable $x$, then does the subsequent behaviors like $P$.
- $P \,\square\, Q$ acts like either $P$ or $Q$ and lets the environment decide the selection.
- $P \,\Big|\Big|\, Q$ shows the parallel composition between $P$ and $Q$.
- $Q \setminus M$ acts like $Q$, except all events from the set $M$ are hidden.
- $P \,|||\, Q$ indicates the process chooses to perform actions in $P$ and $Q$ randomly.
- $P$; $Q$ executes $P$ and $Q$ sequentially.
- *if b then P else Q* denotes the conditional choice. If the value of $b$ is true then it behaves like $P$ else like $Q$.

## III. Modeling LE

In this section, we present a CSP model of LE. The formalization is carried out based on the introduction to LE which has been described in Section 2. Firstly, we give the whole structure of our model and then we model each precess respectively.

### A. Parameters in Model

To clarify the whole system, we give the channels and messages used in LE. They are described as follows.

- *election*: message for a node to pass an election message
- *ack*: to acknowledge receipt of an *election* message
- *leader*: to announce the new leader
- *probe*: to determine if a node is still alive

- *reply*: sent in response to a *probe* message

In our model, every process has some variables to help it make a decision when an action occurs. They are list as follows.

- $d_i$: a binary variable indicating if $i$ is currently in an election or not
- $p_i$: $i$'s parent node in the spanning tree
- $D_i$: a binary variable indicating if $i$ has sent an *ack* to $p_i$ or not
- $lid_i$: $i$'s leader
- $N_i$: $i$'s current neighbors
- $S_i$: set of nodes from which $i$ is yet to hear an *ack* from
- $src_i$: $i$'s priority
- $max_i$: the most-valued node $i$ has record so far

In our model the status of each node can be in one of eight statuses: *BeginElection*, *WaitFor*, *SendElection*, *AwaitAck*, *AwaitLeader*, *SendInfo*, *SendLeader* and *Running*. Their state transition diagram is shown in Fig. 2. Node $i$ can start an election by sending message *election*. After sending message to all of its immediate neighbors, it enters into state *AwaitAck*. In this state, $i$ sets a expire time $T$ for each of node in its waiting list. If a node has no answer in $T$ units, $i$ removes it from its waiting list. After receiving all of the *ack* messages, node $i$ sends message *ack* to its parent and enters into state *SendInfo*. In state *SendInfo*, if the election is start by its own, it will enter into state *SendLeader* to send the *leader* message, otherwise it will enter into state *AwaitLeader* to wait a *leader* message from its parent. When a node finishes sending *leader* message or receives a *leader* message, it will enter into state *Running*. We will discuss each process respectively.

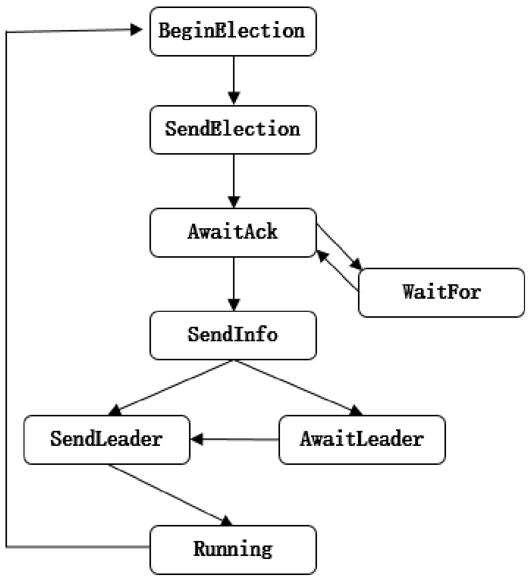


Fig. 2. State Transition Diagram

### B. Modeling each Process

*1) BeginElection:* In process *BeginElection*, $n$ denotes its id number and $N$ represents its neighbors.

$$\begin{aligned} &BeginElection(n, N) \widehat{=} \\ &\qquad SendElection(n, True, n, false, -1, N, N, n, n, N) \end{aligned}$$

*BeginElection* starts an election by sending *election* messages to its one-hop neighbors through process *SendElection*.

*2) SendElection:* Process *SendElection* is shown in Fig. 3. In process *SendElection*, $n$ is the id number of process *SendElection*. The eight variables between $n$ and $N'$ have been explained in previous section. $N'$ is a set that records all nodes which have not been sent *election* message by the current node. Action $election.n!i : N'!src$ represents that node $n$ sends *election* message to its neighbor node $i$ with election priority *src*. When $N'$ is empty, it enters into process *AwaitAck*. During its sending action, it may receive another *election* message from its neighbor and it will compare the priority of these two elections (i.e., $s.p > src.p$ in Fig. 4) and decide which election to take part in. If it receives an *ack* message from its neighbor, it removes it from its waiting list $S$. The last three statements, i.e., *probe*, *fail* and *tock*, describe three basic actions of each process. A node can be tested whether it is alive or not by its parent via channel *probe*. A node may fail and enters into a state *Fail* or it just does nothing but wait the time pass by via channel *tock*.

*3) AwaitAck:* Process *AwaitAck*, its main job is to wait *ack* message from all neighbors in $S$. The process is shown in Fig. 3. When it receives an *ack* message from its neighbor, it removes the neighbor from its waiting list. *AwaitAck* tests whether a node is still alive or not by sending message *probe* and sets a expire time $T$ then it enters into process *WaitFor*. If it receives an *election* message, like process *SendElection*, it compares the priority of these two elections and decides which election to take part in. In our model, since every node can start an election, so there may exist a local leader and therefore when a node receives a *leader* message, it will compare the priority of these two elections and decide to take part in which election.

*4) WaitFor:* Process *WaitFor* will wait the *reply* message from $Node_{pid}$ until the time exceeds $T$ or it receives a *reply* message.

$$\begin{aligned} &WaitFor(n, d, p, D, lid, N, S, src, max, pid, T) \widehat{=} \\ &\quad if\ T == 0 \\ &\quad then\ AwaitAck(n, d, p, D, lid, N, S \setminus \{pid\}, src, max) \\ &\quad else\ tock \rightarrow WaitFor(n, d, p, D, lid, N, S, src, max, pid, T-1) \\ &\quad \Box reply.n.pid \rightarrow AwaitAck(n, d, p, D, lid, N, S, src, max) \end{aligned}$$

*5) SendInfo:* It judges whether the election is start by its own. If is, it enters into *SendLeader*. Otherwise, it sends *ack*

```
$SendElection(n,d,p,D,lid,N,S,src,max,N') \mathrel{\widehat{=}}$
    $if\ empty(N')\ then\ AwaitAck(n,d,p,D,lid,N,S,src,max)$
    $else\ election.n!i : N'!src \to SendElection(n,d,p,D,lid,N,S,src,max,N' \setminus \{i\})$
  $\Box election?c : N!n?s \to (if\ s.p > src.p\ then\ SendElection(n,d,c,False,-1,N,N \setminus \{c\},s,max,N \setminus \{c\}))$
              $else\ ack!n!c!max \to SendElection(n,d,p,D,lid,N,S,src,max,N'))$
  $\Box ack?c : S!n?v \to SendElection(n,d,p,D,lid,N,S \setminus \{c\},src,Max(max,v),N')$
  $\Box leader?c : N!n?s \to (if\ s.p > src.p\ then\ SendElection(n,d,c,False,-1,N,N \setminus \{c\},s,max,N \setminus \{c\})$
             $else\ SendElection(n,d,p,D,lid,N,S,src,max,N'))$
  $\Box probe?c \in N!n \to reply!c!n \to SendElection(n,d,p,D,lid,N,S,src,max,N')$
  $\Box fail.n \to Faild(n,N)$
  $\Box tock \to SendElection(n,d,p,D,lid,N,S,src,max,N')$
```

Fig. 3. Process SendElection

```
$AwaitAck(n,d,p,D,lid,N,S,src,max) \mathrel{\widehat{=}}$
    $if\ empty(S)\ then\ SendInfo(n,d,p,D,lid,N,S,src,max)$
    $else\ tock \to AwaitAck(n,d,p,D,lid,N,S,src,max)$
  $\Box ack?c : S!n?v \to AwaitAck(n,d,p,D,lid,N,S \setminus \{c\},src,Max(max,v))$
  $\Box probe?c : N!n \to reply!c!n \to AwaitAck(n,d,p,D,lid,N,S,src,max)$
  $\Box probe!n?j : S \to WaitFor(n,d,p,D,lid,N,S,src,max,j,T)$
  $\Box tock \to AwaitAck(n,d,p,D,lid,N,S,src,max)$
  $\Box fail.n \to Faild(n,N)$
  $\Box election?c : N!n?s \to (ifs.p > src.p$
      $then\ SendElection(n,d,c,False,-1,N,N \setminus \{c\},s,max,N \setminus \{c\})$
      $else(if\ s == src\ then\ ack!n!c!max \to AwaitAck(n,d,p,D,lid,N,N \setminus \{c\},src,max)$
          $else\ AwaitAck(n,d,p,D,lid,N,S,src,max)))$
  $\Box leader?c : N!n?s \to (if\ s.p > src.p$
      $then\ SendElection(n,d,c,False,-1,N,N \setminus \{c\},s,max,N \setminus \{c\})$
      $else\ election.n!c!src \to AwaitAck(n,d,p,D,lid,N,S,src,max))$
```

Fig. 4. Process AwaitAck

message to its parent and enters into process *AwaitLeader*.

```
$SendInfo(n,d,p,D,lid,N,S,src,max) \mathrel{\widehat{=}}$
    $if\ src == n$
    $then\ SendLeader(n,d,p,D,max,N,S,src,max,N)$
    $else\ ack!n!p!max \to$
        $AwaitLeader(n,d,p,True,max,N,S,src,max)$
  $\Box tock \to SendInfo(n,d,p,D,lid,N,S,src,max)$
  $\Box fail.n \to Faild(n,N)$
  $\Box probe.p.n \to reply.p.n \to$
        $SendInfo(n,d,p,D,lid,N,S,src,max)$
  $\Box election?c : N!n?s \to (if\ s.p > src.p\ then$
      $SendElection(n,d,c,False,-1,N,$
        $N \setminus \{c\},s,max,N \setminus \{c\})$
    $else\ SendInfo(n,d,p,D,lid,N,S,src,max))$
```

*6) Fail:* At any time, one node may enter into state *Fail*. In this process, it absorbs all messages. It is shown as follows.

```
$Faild(n,N) \mathrel{\widehat{=}} tock \to Faild'(n,N)$
        $\Box probe?c : N!n \to Faild(n,N)$
        $\Box election?c : N!n?v \to Faild(n,N)$
        $\Box ack?c : N!n?v \to Faild(n,N)$
        $\Box leader?c : N!n?s \to Faild(n,N)$
$Faild'(n,N) \mathrel{\widehat{=}} tock \to Faild'(n,N)$
        $\Box probe?c : N!n \to Faild(n,N)$
        $\Box election?c : N!n?v \to Faild(n,N)$
        $\Box ack?c : N!n?v \to Faild(n,N)$
        $\Box leader?c : N!n?s \to Faild(n,N)$
        $\Box revive.n \to BeginElection(n,N)$
```

A process enters into *Fail* at least one unit then it can enter into process *Fail'*. In this state, it also absorbs all the actions but it can revive as well.

*7) SendLeader:* Process *SendLeader* sends the *leader* message to all of its one-hop neighbors and then enters into process *Running*. If it receives an *election* message, it compares the priority of the two elections and decides whether to take part in the new election or to ignore this the message.

$$
\begin{aligned}
& \mathit{SendLeader}(n,d,p,D,lid,N,S,src,max,N') \mathrel{\widehat{=}} \\
& \quad \mathit{if\ empty}(N') \\
& \quad \mathit{then\ Running}(n,\mathit{false},n,D,lid,N,S,src,max) \\
& \quad \mathit{else\ leader}.n!i : N'!max \rightarrow \\
& \qquad \mathit{SendLeader}(n,d,p,D,lid,N,S,src,max,N' \setminus \{i\}) \\
& \Box \mathit{tock} \rightarrow \mathit{SendLeader}(n,d,p,D,lid,N,S,src,max,N') \\
& \Box \mathit{fail}.n \rightarrow \mathit{Faild}(n,N) \\
& \Box \mathit{election}?c : N!n?s \rightarrow \mathit{if}\ s.p > src.p\ \mathit{then} \\
& \quad \mathit{SendElection}(n,d,c,\mathit{False},-1,N,N \setminus \{c\},s,max,N \setminus \{c\}) \\
& \quad \mathit{else\ SendLeader}(n,d,p,D,lid,N,S,src,max,N')
\end{aligned}
$$

*8) AwaitLeader:* Process *AwaitLeader* just waits for *leader* message from its parent. When it receives the leader information, it will enter into process *SendLeader* to pass this message to its neighbors.

$$
\begin{aligned}
& \mathit{AwaitLeader}(n,d,p,D,lid,N,S,src,max) \mathrel{\widehat{=}} \\
& \quad \mathit{leader}.p.n?v \rightarrow \\
& \qquad \mathit{SendLeader}(n,d,p,D,v,N,S,src,v,N \setminus \{p\}) \\
& \Box \mathit{tock} \rightarrow \mathit{AwaitLeader}(n,d,p,\mathit{True},lid,N,S,src,max) \\
& \Box \mathit{fail}.n \rightarrow \mathit{Faild}(n,N) \\
& \Box \mathit{probe}.p.n \rightarrow \mathit{reply}.p.n \rightarrow \\
& \qquad \mathit{AwaitLeader}(n,d,p,D,lid,N,S,src,max) \\
& \Box \mathit{election}?c : N!n?s \rightarrow \mathit{if}\ s.p > src.p\ \mathit{then} \\
& \quad \mathit{SendElection}(n,d,c,\mathit{False},-1,N,N \setminus \{c\}),s,max,N \setminus \{c\}) \\
& \quad \mathit{else\ if}\ s == src\ \mathit{then\ ack}!n!c!max \rightarrow \\
& \qquad \mathit{AwaitLeader}(n,d,p,D,lid,N,S,src,max) \\
& \qquad \mathit{else\ AwaitLeader}(n,d,p,D,lid,N,S,src,max)
\end{aligned}
$$

*9) Running:* Process *Running* represents a normal state of a node. The CSP code is shown as follows.

$$
\begin{aligned}
& \mathit{Running}(n,d,p,D,lid,N,S,src,max) \mathrel{\widehat{=}} \\
& \Box \mathit{election}?c : N!n?v \rightarrow (p(v) > p(lid) \mathit{or} \neg d) \& \\
& \quad \mathit{SendElection}(n,d,c,\mathit{False},-1,N, \\
& \qquad N \setminus \{c\},v,max,N \setminus \{c,v\})) \\
& \Box \mathit{tock} \rightarrow \mathit{Running}(n,d,p,D,lid,N,S,src,max) \\
& \Box \mathit{fail}.n \rightarrow \mathit{Faild}(n,N) \\
& \Box \mathit{probe}?c : N!n \rightarrow \mathit{reply}!c!n \rightarrow \\
& \qquad \mathit{Running}(n,d,p,D,lid,N,S,src,max)
\end{aligned}
$$

When it receives an *election* message, if it has no leader (it has not taken in any election) or the priority of the node which starts this election is greater than its leader, it will take part in this election by entering into state *SendElection*.

## IV. Verification

In this section, we implement CSP model and verify some important properties in FDR. Before we do the verification, we should construct our network. To illustrate our model is correct. We choose a topology Fig. 1 in Section 2 to do verification.

$$
\begin{aligned}
& \mathit{Proc} = \{1..5\} \\
& \mathit{NEIGHBORS} \mathrel{\widehat{=}} \{(1,2),(1,3),(2,3),(2,5),(3,4),(4,5)\} \\
& \mathit{Neighbors}(x) \mathrel{\widehat{=}} \{k \mid k \leftarrow \mathit{Proc}, \mathit{member}((x,k),\mathit{NEIGHBORS}) \\
& \qquad \mathit{or\ member}((k,x),\mathit{NEIGHBORS})\}
\end{aligned}
$$

*Proc* is a set which contains five numbers. We stores all the edges in set *NEIGHBORS* and *Neighbors*$(x)$ is a function which returns a set of all neighbors of $x$. We formalize a node as follows.

$$
\begin{aligned}
& \mathit{Node}(x) \mathrel{\widehat{=}} \mathit{if}\ x == 1\ \mathit{then\ BeginElection}(x,\mathit{Neighbors}(x))\ \mathit{else} \\
& \mathit{Running}(x,\mathit{false},x,\mathit{false},x,\mathit{Neighbors}(x),\mathit{Neighbors}(x),x,x)
\end{aligned}
$$

Therefore, we build our network as follows. *Alpha*$(n)$ denotes the synchronized action set of node $n$. For instance *Alpha*$(1) = \{ack.1.2, ack.2.1, \ldots, tock\}$ where "..." indicates some channels like *reply*, *probe* and actions between 1 and 3.

$$\mathit{Network} \mathrel{\widehat{=}} \|\ n : \mathit{Proc} \bullet [\mathit{Alpha}(n)]\mathit{Node}(n)$$

**(1) DeakLock Freedom**

In LE, deadlock freedom means process network can move on at any time. In FDR, we use statement below to do that.

$$\mathit{assert\ Network} : [\mathit{deadlock\,free}\ [F]]$$

The verification result is shown in Fig. 5. The first statement in the *Assertions* block is for deadlock freedom and the green dot on the left side shows that this assertion is passed, which means our system is deadlock free. The block under the *Assertions* is the *Tasks* block, in which the actual checking steps lay here.

**(2) Divergence Freedom**

A divergence of a process is that any trace of the process has a point after which the process behaves chaotically. Divergence freedom assures our model is well-defined without ambiguousness. We use statement below to complete the verification.

$$\mathit{assert\ Network} : [\mathit{divergence\,free}\ [F]]$$

The checking result is shown in Fig. 5., the second statement in *Assertion* block.

**(3) Unique Leader**

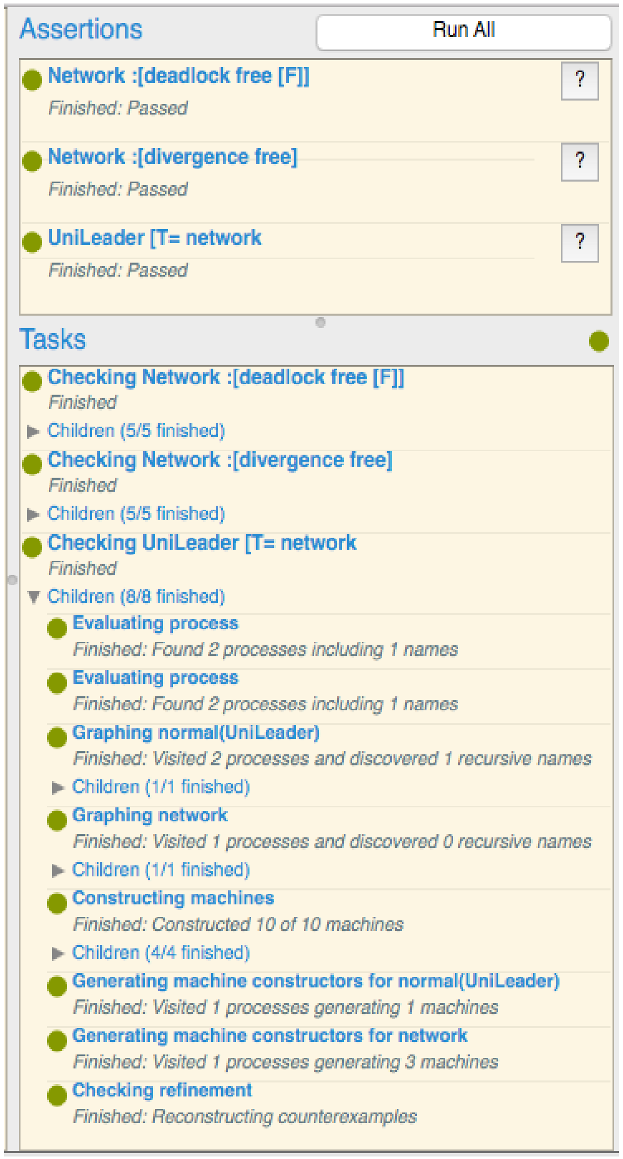
The main goal of LE is that it will eventually select a unique leader, which is the most-valued-node among the nodes in the component. In our model, leader information is sent by *leader* message, and therefore we only concentrate on message *leader* and hide the other channels.

$$
\begin{aligned}
\mathit{network} \mathrel{\widehat{=}} \mathit{Network} \setminus \{|\ & \mathit{probe},\mathit{fail},\mathit{ack}, \\
& \mathit{election},\mathit{tock},\mathit{reply},\mathit{revive}\ |\}
\end{aligned}
$$

In the topology which we prepare to verify, we know that the most-valued-node is 5 and its value is 5 so we formalize the property as follows.

$$UniLeader \mathrel{\widehat{=}} leader?c : Proc?d : Proc!5 \to UniLeader$$

Process *UniLeader* means if there exists a *leader* message the *id* which it sends must be 5. Process *UniLeader* contains all the traces which elect node 5 as the leader. We use refinement to complete this check. If $A[= B$ is true (where $[=$ represents refinement) then the behaviours of B are contained within the behaviours of A. By showing $UniLeader[T = network$, we can conclude that our model satisfies the unique leader scheme. The verification result is shown in Fig. 5.



```
Assertions                                  Run All
● Network :[deadlock free [F]]                   ?
  Finished: Passed
● Network :[divergence free]                     ?
  Finished: Passed
● UniLeader [T= network                          ?
  Finished: Passed

Tasks                                            ●
● Checking Network :[deadlock free [F]]
  Finished
▶ Children (5/5 finished)
● Checking Network :[divergence free]
  Finished
▶ Children (5/5 finished)
● Checking UniLeader [T= network
  Finished
▼ Children (8/8 finished)
  ● Evaluating process
    Finished: Found 2 processes including 1 names
  ● Evaluating process
    Finished: Found 2 processes including 1 names
  ● Graphing normal(UniLeader)
    Finished: Visited 2 processes and discovered 1 recursive names
  ▶ Children (1/1 finished)
  ● Graphing network
    Finished: Visited 1 processes and discovered 0 recursive names
  ▶ Children (1/1 finished)
  ● Constructing machines
    Finished: Constructed 10 of 10 machines
  ▶ Children (4/4 finished)
  ● Generating machine constructors for normal(UniLeader)
    Finished: Visited 1 processes generating 1 machines
  ● Generating machine constructors for network
    Finished: Visited 1 processes generating 3 machines
  ● Checking refinement
    Finished: Reconstructing counterexamples
```

Fig. 5. Verification Results

## V. Conclusion

In this paper, we have constructed the formal models for LE, modeled the operations of each state of LE in CSP. We also verified the LE model using FDR. We constructed the specific models based on three properties including deadlock freedom, divergence freedom and unique leader scheme. The results show that our model satisfies all those properties, indicating the LE get a strong robustness and consisting with the specification.

In mobile ad hoc network, node can transfer from one position to another which results that a node will connect to a new network and disconnect from the old one. Therefore, in the future, we will modify the model to fit the situation like this and based on the new model we do other checks to verify LE.

**Acknowledgement.** This work was supported by Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things (No. ZF1213).

## References

[1] A. Ansari. Verification of Peterson's Algorithm for Leader Election in a Unidirectional Asynchronous Ring Using NuSMV. *CoRR*, abs/0808.0962, 2008.

[2] S. Bhattacharya, J. Kulkarni, and V. S. Mirrokni. Coordination mechanisms for selfish routing over time on a tree. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, pages 186–197, 2014.

[3] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In E. Abraham and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.

[4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[5] S. Kim, V. Vasireddy, and K. Harfoush. Scalable coordination for sensor networks in challenging environments. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11-15, 2007*, pages 214–221, 2007.

[6] A. Mazeev, A. Semenov, and A. Simonov. A Distributed Parallel Algorithm for Minimum Spanning Tree Problem. *CoRR*, abs/1610.04660, 2016.

[7] A. Mehmood, M. M. Umar, and H. Song. ICMDS: secure inter-cluster multiple-key distribution scheme for wireless sensor networks. *Ad Hoc Networks*, 55:97–106, 2017.

[8] Y. Nakamura, M. Louvel, and H. Nishi. Coordination middleware for secure wireless sensor networks. In *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society, Florence, Italy, October 23-26, 2016*, pages 6931–6936, 2016.

[9] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2010.

[10] J. Sun, Y. Liu, and J. S. Dong. Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15, 2008. Proceedings*, pages 307–322, 2008.

[11] S. Vasudevan, J. F. Kurose, and D. F. Towsley. Design and Analysis of a Leader Election Algorithm for Mobile Ad Hoc Networks. In *12th IEEE International Conference on Network Protocols (ICNP 2004), 5-8 October 2004, Berlin, Germany*, pages 350–360, 2004.

[12] L. Wang, F. Sui, Y. Huang, and H. Zhu. Modeling and Verifying the Ballooning in Xen with CSP. In *16th IEEE International Symposium on High Assurance Systems Engineering, HASE 2015, Daytona Beach, FL, USA, January 8-10, 2015*, pages 18–25, 2015.

[13] L. Xu and P. Jeavons. Simple algorithms for distributed leader election in anonymous synchronous rings and complete networks inspired by neural development in fruit flies. *Int. J. Neural Syst.*, 25(7), 2015.