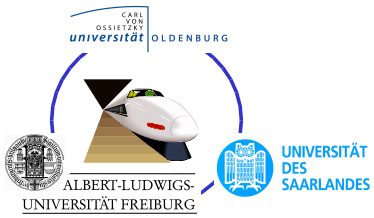

AVACS – Automatic Verification and Analysis of Complex Systems

REPORTS

of SFB/TR 14 AVACS

Editors: Board of SFB/TR 14 AVACS

Self-Stabilizing Mutual Exclusion Algorithm for Trees

by
Oday Jubran and Oliver Theel

AVACS Technical Report No. 91
May 2013
ISSN: 1860-9821

Publisher: Sonderforschungsbereich/Transregio 14 AVACS
(Automatic Verification and Analysis of Complex Systems)
Editors: Bernd Becker, Werner Damm, Bernd Finkbeiner, Martin Fränzle,
Ernst-Rüdiger Olderog, Andreas Podelski
ATRs (AVACS Technical Reports) are freely downloadable from `www.avacs.org`

Author(s) contact: Oday Jubran (`jubran@informatik.uni-oldenburg.de`).

Self-Stabilizing Mutual Exclusion Algorithm for Trees*

Oday Jubran and Oliver Theel

Carl von Ossietzky Universität Oldenburg
26111 Oldenburg, Germany
{jubran,theel}@informatik.uni-oldenburg.de

Abstract. Self-stabilizing mutual exclusion algorithms for ring topologies can be applied on trees by passing a token in an Euler cycle, thereby representing a virtual ring, through the processes in the tree. This virtual ring leads to a performance penalty because many processes have to be visited multiple times in order to pass the token to every single process. In this report, we present a self-stabilizing distributed mutual exclusion algorithm specialized for trees. The algorithm exploits the tree topology by propagating tokens from the root through the tree branches to search for processes ready to execute an action, namely active processes, and then to select one of them. We show that, on average, the proposed algorithm requires less time to search and select an active process than the state-of-the-art algorithms do.

1 Introduction

Self-stabilization is a system property reflecting that a system's desired behavior is eventually obtained regardless of its initial behavior. A system is said to be self-stabilizing if it guarantees two properties: (1) Regardless of the initial state, the system eventually reaches a safe state, and (2) if the system is in a safe state, then the system does not leave the set of safe states by its own. Self-stabilization is useful when the system initial state is arbitrary, or when the system's environment may exhibit transient faults or changes, such as memory allocation problems or dynamic topology changes.

Recent research concerned self-stabilizing mutual exclusion algorithms run by distributed processes, where each process can read the registers of its neighbors [1,2,3]. The aim of such algorithms is satisfying mutual exclusion, by allowing only one process to execute its action in each execution step. This is done by assigning privileges to processes, where having a privilege depends on the state of the process neighbors. When a process has a privilege, the process executes its action if it is an active process, and changes its state creating a privilege to other neighbor processes. This can be viewed as passing a token between processes, where a process is allowed to execute an action only if it holds the token.

* This work was partially supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, http://www.avacs.org/).

Developing self-stabilizing algorithms for different topologies is difficult. Thus, some approaches concern developing fair-composition of self-stabilizing algorithms applied on a class of topologies, to be applied on a larger class [4]. For example, a self-stabilizing mutual exclusion algorithm for ring topologies can be applied on general graphs by composing it with a spanning tree algorithm to form a virtual tree, and creating an Euler cycle through the tree reflecting a virtual ring, as in [4, p. 24]. This fair-composition achieves mutual exclusion, but it has low performance; in order to pass a token through a virtual ring, many processes have to be visited multiple times in order to pass the token to every single process. For example, if a leaf process in the tree holds the token, it is required to pass the token first to the parent in order to deliver the token to the leaf siblings. To pass a token to an active process in trees of high depth or branching factor using a virtual ring, the number of required steps is large in comparison to to the number of required steps for an actual ring topology.

Given that virtual trees can be formed from general graphs using spanning tree algorithms [5], our concern in this work is developing a self-stabilizing mutual exclusion algorithm for trees, that requires less steps to search and select active processes than other algorithms.

1.1 Related Work

Self-stabilizing distributed algorithms for mutual exclusion were pioneered by Dijkstra in [1], in which three self-stabilizing distributed algorithms for ring topologies are introduced. The algorithms are self-stabilizing under the assumption that there is a central daemon allowing at most one process to execute an action in each step, i.e. if two processes have privileges, only one may change its state in one step. Proofs of correctness of Dijkstra's algorithms were presented in [6,7]. Other works, as in [8], concern self-stabilizing algorithms for ring topologies allowing many processes to change their states in one step. Lamport presented self-stabilizing mutual exclusion systems in [3], where each process in a system can communicate to all other processes, reflecting a fully connected graph system topology. Kruijer presented a self-stabilizing token system for trees in [2], allowing many tokens to run concurrently, and mutual exclusion is achieved only within each path in the tree but not in the whole tree; i.e. for each path in the tree, only one process may execute an action, but there may exists processes on different paths that execute actions at the same time. In our approach, we require to achieve mutual exclusion in the whole tree, not only in the tree paths. The issue, that passing a token through a virtual ring requires a large number of steps to search and select an active process to execute an action, may also exist in any fair-composition of self-stabilizing algorithms, where an algorithm depends on passing a token through virtual rings. For example, in [9], the authors proposed a transformational approach of self-stabilizing algorithms working under strong schedulers to be applied on weak schedulers, where the transformational approach depends on virtual rings. The transformed algorithms by the transformational approach require larger number of steps to select and execute an active process than the pre-transformed algorithms do.

1.2 Contribution

In this work, we consider the issue that passing a token through a virtual ring requires a large number of steps to search and select an active process to execute an action. We present a self-stabilizing mutual exclusion algorithm for trees, where mutual exclusion is achieved among all processes in the tree, i.e. only one process in the tree may execute an action in each step. The algorithm extends the 4-state-machine algorithm of Dijkstra [1]. We show that, on average, the algorithm requires less time to search and select an active process than the state-of-the-art algorithms do, by propagating multiple tokens in parallel rather than passing only one token.

The following sections are organized as follows: Section 2 presents a formal model of the topology and the system that we consider. Next, Section 3 presents the algorithm and its properties. Section 4 demonstrates proofs that the algorithm satisfies its desired properties. Finally, Section 5 presents the conclusion and the discussion.

2 Formal Model

We consider a distributed algorithm executed by distributed processes. Such processes consist of variables. For each process in the topology, there exists some processes whose variables are visible to the process; i.e. each process in the topology can read the variables of some other processes. With respect to the visibility of variables, the topology can be modelled as a tree, where each process can read only the variables of its parent and children if they exist. Each process runs a sub-algorithm of the distributed algorithm, where a sub-algorithm reflects guarded commands that are enabled depending on the state of the process, its parent, and its children. In each execution step, if a process has an enabled guarded command, it executes an action. When an action of a guarded command is executed, it modifies the variables of its corresponding process. We model this framework as follows:

A *tree* is a directed graph $T = (P, E)$, such that P is a finite non-empty set of *nodes*, and $E \subseteq P \times P$ is a finite set of *edges*. An edge $(p_a, p_b) \in E$ is said to be *directed* from p_a to p_b. p_a is said to be a *parent* of p_b, denoted by $parent(p_b)$, and p_b is said to be a *child* of p_a. The set of children of a node p is denoted by $Children(p)$. Exactly one node $r \in P$ is called a *root*, to which no edge in E is directed. Each node $p \in P$, from which no edge in E is directed, is called a *leaf*. The set of leaves in P is denoted by $Leaves(T)$. A *path* in T is a finite sequence $p_0, ..., p_n \in P^*$ where $(p_i, p_{i+1}) \in E$ for $0 \leq i < n$. p_0 is said to be *linked* to p_n by this path. For each path $p_0, ..., p_n \in P^*$ in T, it holds that $p_0 \neq p_n$ *(acyclic)*. Let $p_0 \in P$ be the root. p_0 is linked to each node $p \in P \backslash \{p_0\}$ by a unique path $p_0, ..., p_n \in P^*$, such that $p_n = p$. n is called the *depth* of p. A path $p_i, ..., p_z$ is said to be a *maximal path* of p_i, denoted by $mPath(p_i)$, iff p_z is a leaf.

A *distributed algorithm* A is a non-empty set of *sub-algorithms* $\{A_{sub1}, ..., A_{subn}\}$ where a sub-algorithm A_{sub} is a set of *guarded commands*

$\{c_1, ..., c_n\}$, such that c : *guard* $\rightarrow$ *action* where: *guard* is a boolean expression over variables, and *action* is an assignment function.

A *process* p_i is a tuple $(Vars_i, A_{subi})$ where A_{subi} is a sub-algorithm, and $Vars_i$ is a finite non-empty set of variables $\{i, x_1, x_2, .., x_n\}$, such that i has always a unique value, and is called the process *id*. A process p of id i is denoted by p_i.

A *topology* is a tree $T = (P, E)$ where P is a finite set of processes.

A *local state* V_i of a process $p_i = (Vars_i, A_{sub_i})$ is a valuation of $Vars_i$. A *global state* σ of a topology $T = (P, E)$ is a vector of local states $[V_1, ..., V_n]$ of $p_1, ..., p_n \in P$ respectively where $|P| = n$. The set of all global states (the global state space) is denoted by Σ. A guarded command c : *guard* $\rightarrow$ *action* is said to be *enabled* in a global state σ iff *guard* evaluates to *true*. A process $p_i = (Vars_i, A_{subi})$ is said to *have a privilege* in σ iff there exists an enabled guarded command in A_{sub_i}.

An *execution* Ξ over a topology T is a sequence of global states of T $\langle \sigma_1, \sigma_2, .. \rangle$ such that for each $i \geq 1$, σ_{i+1} is reachable from σ_i by executing a subset of enabled guarded commands. We call (σ_i, σ_{i+1}) an *execution step*. An *execution prefix* is a finite prefix $\sigma_1, ..., \sigma_n$ of an execution. Execution Ξ is said to be *consistent* iff for each execution step (σ_i, σ_{i+1}), each process with at least one privilege in σ_i executes an enabled guarded command.

A *system* Ω is a set of consistent executions over a topology T.

3 Algorithm

In this section, we present the self-stabilizing mutual exclusion algorithm using the model introduced in Section 2. We call the algorithm *SSMutex*.

3.1 Basic Idea

The basic idea for developing *SSMutex* is achieving the following execution scenario: (1) The root propagates a token only to search for an active process (Search token). (2) If an active process receives the token, the process sends back a token that is forwarded to the root, to inform the root about the active process (Ready token). (3) Once the root receives the Ready token sent by the active process, the root sends a token allowing the active process to execute its action (Execute token). (4) Once the active process receives the Execute token, the process executes its action, and then it sends back a token notifying the root that the execution is complete (Complete token). To achieve this scenario, we extend the 4-state-machine algorithm of Dijkstra explained in [1]: (1) In the 4-state-machine algorithm, given processes $p_1, ..., p_n$ organized in a ring topology, a token is sent back and forth between p_1 and p_n through $p_2, ..., p_{n-1}$. We use this movement to send tokens back and forth between the root and leaves in tree topologies. (2) The four states created by the 4-state-machine algorithm reflect the Search, Ready, Execute, and Complete tokens respectively.

process p_{id} {
  int *id*; **process** *parent*, *child*[0..*n*]; **boolean** *active*; // Basic Variables
  boolean *up*, *x*; **int** *rdyChild*; // Introduced Variables for *SSMutex*
  //Assert: $root.up = \top$, $\forall p \in Leaves(T) \bullet p.up = \bot$
}

Fig. 1. Process Variables

3.2 *SSMutex* Algorithm

SSMutex is a distributed algorithm run by processes forming a tree topology. *SSMutex* consists of two sub-algorithms: (1) A_{root} sub-algorithm run by the root, and (2) A_p sub-algorithm run by each non-root process. Each process has two boolean variables *up* and *x*, and one integer variable *rdyChild*. The variables *up* and *x* are used similarly as Dijkstra's algorithm ones to create four states. The variable *rdyChild* of a process p_i is used to point to (mark) process id's for execution purposes. We assert that always $up = \top$ for the root, and $up = \bot$ for each leaf (similar to Dijkstra's algorithm), and this can be practically managed by another algorithm, or by adding guarded commands to the sub-algorithms that reset the values of *up* and *x* to guarantee this assertion. Figure 1 illustrates process variables, that include the basic ones and *SSMutex*-related ones.

Definition 1 *SSMutex is a distributed algorithm* $\{A_{root}, A_p\}$.

For convenience, we present the guarded commands of A_{root} and A_p in an if-then-else style by omitting the labels; we present a guarded command "c : *guard* $\rightarrow$ *action*" as:

if *guard* **then**
  action;
end if

The root sub-algorithm A_{root} is shown by Algorithm 1, and the sub-algorithm of any non-root process is shown by Algorithm 2.

For convenient presentation, we introduce a notation that helps us to present the local states of processes. We basically annotate a process by the values of its introduced variables: Given a process p, we denote $p.x$, $p.up$, and $p.rdyChild$ by $p_x^{up}rdyChild$. For example, given a state where $p.x = \top$, $p.up = \bot$, and $p.rdyChild = 5$, we denote the state by $p_{\top}^{\bot}5$. If we specify only the value of one variable, we write the sign "?" for values of the other variables. For example, to specify that $p.x = \top$, we write $p_{\top}^{?}?$ or $p_{\top}^{?}$. To denote that $p.rdyChild \neq$ -1, given (for example) that $p.x = a$ and $p.up = b$, we write $p_a^b \neq$-1.

Each of the four states created by the variables *up* and *x* represents a meaningful token; given a process p: (1) $p_{\top}^{\top}$ is called a *Search token* sent from p to its children. (2) $p_{\top}^{\bot}$ is called a *Ready token* sent from p to its parent. (3) $p_{\bot}^{\top}$ is called an *Execute token* sent from p to its children. (4) $p_{\bot}^{\bot}$ is called a *Complete token* sent from p to its parent.

```
1: if x ∧ ∃ ch ∈ child • ch.x = x ∧ ¬ch.up ∧ ch.rdyChild ≠ −1 then
2:    rdyChild := ch.id; x := ¬x;
3: end if
4: if x ∧ ∀ ch ∈ child • ch.x = x ∧ ¬ch.up ∧ ch.rdyChild = −1 then
5:    x := ¬x;
6: end if
7: if ¬x ∧ ∀ ch ∈ child • ch.x = x ∧ ¬ch.up then
8:    update(active);
9:    if active then
10:      executeAction();
11:   else
12:      rdyChild := −1; x := ¬x;
13:   end if
14: end if
```

Algorithm 1: A_{root}

The stable behavior of the algorithm can be viewed as two phases. In the first phase, the root passes a Search token through the tree paths searching for an active process. Then, active processes send back a Ready token that is passed to the root to notify the root about active processes. The root can recognize a path containing an active process by the value of *rdyChild*. In the second phase, the root chooses one path containing an active process (by *root.rdyChild*), and sends an Execute token that is passed to the active process. The active process, then, runs the command *executeAction*() (which means execute an action) and sends back a Complete token that is passed to the root.

In more detail (see Algorithm 1 and Algorithm 2), the algorithm behavior is as follows:

Phase (1): The root sends a Search token ($root^{\top}_{\dagger}$-1) to its children. If a process p receives a Search token,

1. If p is not active, (a) if p is not a leaf, p switches into $p^{\top}_{\dagger}$-1 passing the Search token to its children. (b) if p is a leaf, p switches into $p^{\perp}_{\dagger}$-1 since for all leaves, $up = \perp$. Notice that *rdyChild* = -1 denotes that there is no active process in any subtree rooted by p. If there exists no active processes in the whole tree, a Ready token ($^{\perp}_{\dagger}$) with *rdyChild* = -1 is forwarded from the leaves to the root. The root recognizes that there is no active process in the tree from the value of *rdyChild* of each root child.
2. If there exists an active process p that receives a Search token, p switches into $p^{\perp}_{\dagger}p.id$ sending to its parent a Ready token, and pointing to itself by setting $rdyChild = p.id$. After that, $parent(p)$ recognizes the Ready token, and switches into $parent(p)^{\perp}_{\dagger}p.id$. The ancestors pass the Ready token to the root analogously as $parent(p)$.

Phase (2): The root sends an Execute token ($root^{\top}_{\perp}$), such that if there is an active process p_j in the tree, the value of *root.rdyChild* is set to the id of the root child that is linked to the active process, and if there is no active process, the value of *root.rdyChild* is set to -1. The Execute token is passed to p_j, and

```
1: if parent.x ≠ x then
2:   if x then
3:     if parent.rdyChild = id then
4:       if active then
5:         executeAction(); up := ⊥;
6:       else if ∃ ch ∈ child • rdyChild = ch.id then
7:         up := ⊤;
8:       else
9:         rdyChild := −1; up := ⊥;
10:      end if
11:    else
12:      rdyChild := −1; up := ⊥;
13:    end if
14:  end if
15:  if ¬x then
16:    rdyChild := −1; up := ⊤; update(active);
17:    if active then
18:      up := ⊥; rdyChild := id;
19:    end if
20:  end if
21:  x := ¬x;
22: else if up ∧ ∃ ch ∈ child • ch.x = x ∧ ¬ch.up ∧ ch.rdyChild ≠ −1 then
23:  rdyChild := ch.id; up := ⊥;
24: else if up ∧ ∀ ch ∈ child • ch.x = x ∧ ¬ch.up ∧ ch.rdyChild = −1 then
25:  rdyChild := −1; up := ⊥;
26: end if
```

Algorithm 2: A_p

then p_j runs the command *executeAction*(). After that, p_j sends a Complete token ($p_j{}^{\perp}_{\perp}$) that is forwarded to the root. For any other process p receiving an Execute token from $parent(p)$, if $parent(p).rdyChild \neq p.id$, p sends a Complete token and set $p.rdyChild$ to -1.

3.3 Algorithm Performance

We show that, on average, *SSMutex* requires less time to search and select an active process, than the state-of-the-art algorithms do. The graph-related calculations are based on [10].

As discussed in Section 1.1, the state-of-the-art algorithms are applied on trees by passing a token in an Euler cycle. The minimum number of steps required to traverse a tree $T = (P, E)$ by an Euler cycle is $|P| + |E| - 1$. Thus, if there is only one active process in the tree, the average number of steps to reach the active process is $\frac{|P|+|E|-1}{2}$. Considering *SSMutex*, by Section 3, in Phase (1), a Search token is sent from the root and is passed through the tree paths in parallel. An active process sends back a Ready token once it receives the Search token. The number of steps required to pass the Search token to the process and to send back the Ready token depends on the depth of the active process. This

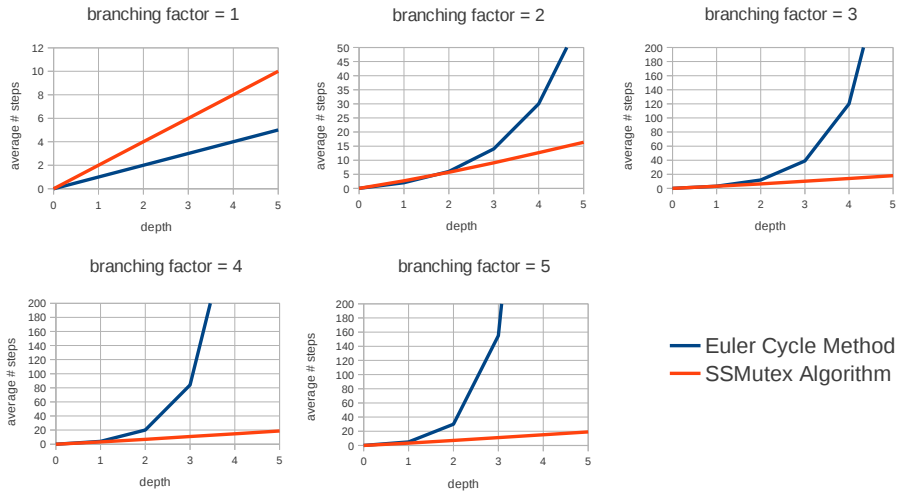

Fig. 2. Average Number of Steps required by the Euler Cycle Method and *SSMutex* for Trees with Different Branching Factors and Depth

applies analogously to Phase (2). Let the active process be at depth d. Given that there are two phases, where in each phase there are tokens sent back an forth between the root and the active process, then the number of steps required to complete the two phases is $4 \cdot d$. To compute the average number of steps required to complete the two phases, we compute the average depth of the tree. Therefore, we need to know the number of process at each depth. For simplicity, we consider trees with a fixed branching factor to know the number of processes at each depth. Given a tree $T = (P, E)$ of a depth d where $depth(root) = 0$, and a branching factor b, the average depth of T is given by $\frac{\sum_{i=0}^{d} b^i \cdot i}{|P|}$, and the average number of steps is $4 \cdot \frac{\sum_{i=0}^{d} b^i \cdot i}{|P|}$. Figure 2 illustrates a comparison of the average number of steps required to execute an active process between the Euler cycle method and *SSMutex*, for trees with different branching factors and depths. It is obvious that *SSMutex* has a better performance, especially for trees of high branching factor or high depth.

4 Algorithm Correctness

We show that *SSMutex* satisfies system progress, self-stability, and mutual exclusion properties.

In the following sections, we say that a process has an above (resp. bottom) privilege if the process receives a token from its parent (resp. child or children).

DEFINITION 2 *Given a topology T, a process $p_i = (Vars_i, A_{sub_i})$, and a global state σ, p_i is said to have an* above privilege *in σ iff $p_i \neq root \wedge p_i.x \neq parent(p_i).x$. Process p_i is said to have a* bottom privilege *in σ iff p_i has an enabled guarded command in A_{sub_i} that does not create an above privilege.*

4.1 System Progress

SSMutex is used to organize executions of processes. Thus, the algorithm is supposed to avoid deadlocks, i.e. in each global state, there must exist at least one process that has a privilege.

The proof idea is as follows: If there exists two processes p_1 and p_2 where $p_1.x \neq p_2.x$, then by having a tree topology and by Algorithm 1 and Algorithm 2, there exists at least one process with an above privilege. Otherwise, there exists a process p such that $p.up = \top$ and for each child $ch \in Children(p)$, it holds that $ch.up = \bot$ (Given that for all leaves, $up = \bot$). This implies that p has a bottom privilege.

THEOREM 1 (SYSTEM PROGRESS) *Given the global state space Σ of a topology $T = (P, E)$, for each global state $\sigma \in \Sigma$, there exists at least one process that has a privilege.*

Proof. (See Appendix, Section 5.1)

4.2 Self-Stabilization and Mutual Exclusion

SSMutex has to guarantee self-stabilization: (1) Convergence: Starting in any state, any execution leads to a safe state, and (2) Closure: Starting in a safe state, any execution does not leave the set of safe states. Mutual exclusion has to be satisfied while the system is stable.

We define a safe state, based on the desired progress of the algorithm. Roughly, a safe state has to guarantee the following: (1) When the root propagates a Search token ($root^{\top}_{\top}$), the value of *rdyChild* is -1. (2) Given an active process p in a state $p^{\perp}_{\top}$, if $p.rdyChild \neq$ -1, then there exists a path that leads to an active process with a Ready token. We call such a path an *active path*, denoted by $aPath_\sigma$. If $p.rdyChild =$ -1, then there exists no active process in any subtree rooted by p. (3) Whenever the root propagates a Search token, there must be neither an Execute token in the tree, nor a process that may create an Execute token while $root^{\top}_{\top}$. (4) When the root sends an Execute token, either there exists exactly one path from the root that is linked to an active process, where each parent points to its child by the variable *rdyChild*, or there exists no active process in the tree, and there is no possibility to create an Execute token by a process other than the root.

DEFINITION 3 *Given a global state σ, an* active path *at σ, denoted by $aPath_\sigma$, is a path $p_k, ..., p_n$ for $k \leq n$ such that σ is defined as follows:*

$$\forall\, k \leq j \leq n \bullet p_j{}^{\perp}_{\top} \;\wedge\; p_n.active \;\wedge\; p_n.rdyChild = p_n.id \;\wedge$$
$$\forall\, k \leq i < n \bullet p_i.rdyChild = p_{i+1}.id \;\wedge\; \neg p_i.active$$

DEFINITION 4 *Given a global state σ, an* execution path *at σ, denoted by $ePath_\sigma$, is a path $p_0, ..., p_n$ such that $p_0 = root$, and σ is defined as follows: for all $0 \leq i < n$:*

$$p_i.rdyChild = p_{i+1}.id \ \wedge \ p_n.rdyChild = p_n.id \ \wedge \ {p_n}^{\perp}_{?} \ \wedge$$
$${p_i}^{\top}_{?} \Rightarrow {p_i}^{\top}_{\perp} \ \wedge \ {p_{i+1}}^{\top}_{\perp} \Rightarrow {p_i}^{\top}_{\perp} \ \wedge \ {p_{i+1}}^{?}_{\perp} \Rightarrow {p_i}^{?}_{\perp} \ \wedge \ {p_i}^{\perp}_{?} \Rightarrow {p_{i+1}}^{\perp}_{?} \ \wedge$$
$${p_i}^{\perp}_{\top} \Rightarrow \neg p_i.active \ \wedge \ {p_n}^{\perp}_{\top} \Rightarrow p_n.active$$

DEFINITION 5 *A* safe state *is a global state σ that satisfies the following conditions:*

1. $\forall\, p \in P \bullet {p}^{\top}_{\top} \Rightarrow {p}^{\top}_{\top}$-1 $\wedge \neg p.active$
2. $\forall\, p_s \in P \bullet {p_s}^{\perp}_{\top} \Rightarrow$
   - $\exists\, n \geq s \bullet p_s, ..., p_n : aPath_\sigma \ \vee$
   - $\forall\, mPath : p_s, ..., p_z \bullet \forall\, e \in [s, z] \bullet \neg p_e.active \wedge {p_e}^{\perp}_{\top}$-1
3. ${root}^{\top}_{\top} \Rightarrow$
   - $\forall\, p \in P \bullet {p}^{\top}_{?} \Rightarrow {p}^{\top}_{\top}-1\wedge$
   - $\forall\, p \neq root \bullet ({p}^{?}_{\top} \wedge parent(p).rdyChild = p.id) \Rightarrow {parent(p)}^{?}_{\top}$
4. ${root}^{\top}_{\perp} \Rightarrow$
   - $\exists\, p \in P \bullet p.active \Rightarrow \exists\, !\, ePath_\sigma \wedge$
   - $\neg \exists\, p \in P \bullet p.active \Rightarrow \forall\, p \neq root \bullet$
     - ${p}^{\top}_{?} \Rightarrow {p}^{\top}_{\top}-1\wedge$
     - ${p}^{?}_{\top} \wedge {parent(p)}^{?}_{\perp} \Rightarrow parent(p).rdyChild \neq p.id$

The set of safe states is denoted by Σ_{safe}.

Convergence In this section, we prove that given a topology, each execution with a non-safe initial state, eventually reaches a safe state.

The proof idea is as follows: Regardless of the initial state, any execution keeps switching the root local state between ${root}^{\top}_{\top}$ and ${root}^{\top}_{\perp}$. By Algorithm 1, when the root switches from ${root}^{\top}_{\perp}$ into ${root}^{\top}_{\top}$, the value of *root.rdyChild* is -1, and the local state of each root child *ch* is ${ch}^{\perp}_{\perp}$. This enables propagating a Search token that resets the value of *rdyChild* for the receiving processes, leading the execution into a safe state.

THEOREM 2 (CONVERGENCE) *Given a system over a topology T, for each global state $\sigma_0 \in \Sigma$, any execution $\sigma_0, \sigma_1, ..$ reaches a safe state $\sigma_n \in \Sigma_{safe}$ after a finite number of execution steps.*

Proof. (See Appendix, Section 5.2)

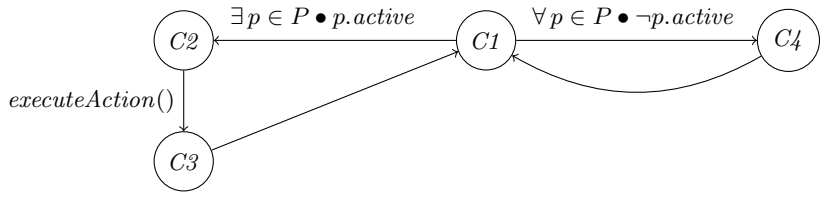

Fig. 3. System Progress through Safe States

Closure and Mutual Exclusion In this section we prove that given a topology, each execution with a safe initial state does not reach any non-safe state. The proof idea is as follows: We categorize the safe states into four categories: Category 1 contains all the safe states in which the root sends a Search token ($root^{\top}_{\top}$). Category 2 (resp. 3) contains all safe states in which the root sends an Execute token and there exists an active process that still did not execute (resp. has executed). Category 4 includes all safe states in which the root sends an Execute token and there is no active process. A system evolves through safe states wrt. the defined categories as follows (see Figure 3): If an execution has an initial state in Category 1, then the execution eventually reaches a state in Category 2 if there exists an active process, or Category 4 if there exists no active process, only through states from Category 1. No process runs the command *executeAction*() while $root^{\top}_{\top}$. Starting from any state in Category 2, any execution reaches a state in Category 3 through states in Category 2, by running the command *executeAction*() mutual exclusively, i.e. only one active process runs *executeAction*(). From any state in Category 3 (resp. 4), any execution reaches a state in Category 1 through states in Category 3 (resp. 4), where no process runs the command *executeAction*(). This guarantees closure and mutual exclusion.

Definition 6 *Given the global safe state space $\Sigma_{safe} \subset \Sigma$, four subsets of states C1, C2, C3, C4 where*

$$C1 \,\dot\cup\, C2 \,\dot\cup\, C3 \,\dot\cup\, C4 \subseteq \Sigma_{safe}$$

are defined as follows:

- $C1 ::= \{\sigma \in \Sigma_{safe} | \sigma : root^{\top}_{\top}\}$
- $C2 ::= \{\sigma \in \Sigma_{safe} | \sigma : \exists\, ePath_{\sigma} : root, ..., p_n \bullet {p_n}^{?}_{\top}\}$
- $C3 ::= \{\sigma \in \Sigma_{safe} | \sigma : \exists\, ePath_{\sigma} : root, ..., p_n \bullet {p_n}^{?}_{\bot}\}$
- $C4 ::= \{\sigma \in \Sigma_{safe} | \sigma : root^{\top}_{\bot} \wedge \neg \exists\, p \in P \bullet p.active\}$

Corollary 1 $\Sigma_{safe} = C1 \,\dot\cup\, C2 \,\dot\cup\, C3 \,\dot\cup\, C4$

Theorem 3 *Given a system over a topology T, and a safe state $\sigma_0 \in C1$, for each execution $\sigma_0, \sigma_1, ..$, there exists finally a state σ_k for $k > 0$, such that*

1. *$\sigma_k \in C2$ iff there is at least one active non-root process in σ_0, or*
2. *$\sigma_k \in C4$ iff there is no active process in σ_k,*

such that $\forall\, 0 \leq i \leq k-1 \bullet \sigma_i \in C1$.

Proof. (See Appendix, Section 5.3)

THEOREM 4 *Given a system over a topology* T*, and a safe state* $\sigma_0 \in C2$*, then for each execution* $\sigma_0, \sigma_1, ..$*, there exists a state* σ_k *for* $k > 0$*, such that*

- $\sigma_k \in C3$ *and* $\forall\, 0 \leq i \leq k-1 \bullet \sigma_i \in C2$*, and*
- *for the execution step* (σ_{k-1}, σ_k)*, there exists exactly one process* $p \neq root$ *such that* $\sigma_{k-1} : p^{\perp}_{\top} p.id$ *and* $\sigma_k : p^{\perp}_{\perp} p.id$*, and* p *runs the command execute*() *in the execution step.*

Proof. (See Appendix, Section 5.4)

THEOREM 5 *Given a system over a topology* T*, and a safe state* $\sigma_0 \in C3$*, for each execution* $\sigma_0, \sigma_1, ..$*, there exists a state* σ_k *for* $k > 0$*, such that* $\sigma_k \in C1$ *and* $\forall\, 0 \leq i \leq k-1 \bullet \sigma_i \in C3$.

Proof. (See Appendix, Section 5.5)

THEOREM 6 *Given a system over a topology* T *and a safe state* $\sigma_0 \in C4$*, for each execution* $\sigma_0, \sigma_1, ..$*, there exists a state* σ_k *for* $k > 0$*, such that* $\sigma_k \in C1$ *and* $\forall\, 0 \leq i \leq k-1 \bullet \sigma_i \in C4$.

Proof. (See Appendix, Section 5.6)

5 Conclusion and Discussion

We presented a self-stabilizing mutual exclusion algorithm for tree topologies, that overcomes the performance penalty caused by creating an Euler cycle representing a virtual ring in tree topologies using other algorithms. The algorithm exploits the tree structure by propagating tokens from the root through the tree branches to search and select an active process. The average time required to search and select an active process depends on the average depth of active processes in the tree.

Exploiting the tree structure has a noticeable impact on the average time required for searching and selecting active processes as shown in Section 4, where the time depends on the tree depth. This observation can be exploited to derive other properties of the algorithm. We mention two of them: (1) The algorithm can be applied on trees having processes, for which being active or not depends on the states of all other processes in the tree. This is achieved by passing a global snapshot reflecting a copy of a global state together with the tokens; each process receiving the token copies the snapshot from the process that sends the token, and then the process receiving the token updates the values of its corresponding variables in the snapshot. (2) In the current version of the algorithm, selecting active processes is done by the root, and the selection depends on the depth of the active processes; the active processes with the least depth receive the

Search token before other processes, and they send Ready tokens before others do. This selection procedure can be extended to selecting processes depending on some metrics to support other QoS measurements. This is done by letting Search tokens reach the leaves before sending Ready tokens. Then, Ready tokens are sent starting from the leaves, where each process receiving the token decides which process, among its children, to be chosen depending on the metric value that each child holds.

References

1. E. W. Dijkstra. Self-Stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, 17(11):643–644, 1974.
2. H. S. M. Kruijer. Self-Stabilization (in Spite of Distributed Control) in Tree-Structured Systems. *Information Processing Letters*, 8(2):91–95, 1979.
3. L. Lamport. The Mutual Exclusion Problem: Part II - Statement and Solutions. *Journal of the ACM*, 33(2):327–348, 1986.
4. S. Dolev. *Self-Stabilization*. The MIT Press, 2000.
5. F. C. Gärtner. A Survey of Self-Stabilizing Spanning-Tree Construction Algorithms. Technical report, Swiss Federal Institute of Technology (EPFL), 2003.
6. E. W. Dijkstra. A Belated Proof of Self-Stabilization. *Distributed Computing*, 1(1):5–6, 1986.
7. S. Ghosh. An Alternative Solution to a Problem on Self-Stabilization. *ACM Transactions on Programming Languages and Systems*, 15(4):735–742, 1993.
8. G. M. Brown, M. G. Gouda, and C.-l. Wu. Token Systems that Self-Stabilize. *IEEE Transactions on Computers*, 38(6):845–852, 1989.
9. A. Dhama and O. Theel. A Tranformational Approach for Designing Scheduler-Oblivious Self-stabilizing Algorithms. In *SSS*, volume 6366 of *Lecture Notes in Computer Science*, pages 80–95. Springer, 2010.
10. R. Diestel. *Graph Theory*. Springer-Verlag, 2005.

Appendix

5.1 Proof of Theorem 1

Proof. We categorize the state space Σ into two categories, and we show that for each category, for each global state, there exists at least one process that has a privilege.

- Category 1: $\exists\, p_1, p_2 \in P \bullet p_1.x \neq p_2.x$,
  Since the tree is a connected graph, then there exists a process p where $p.x \neq \mathit{parent}(p).x$. By Algorithm 2 (Line 1), p has an above privilege.
- Category 2: $\forall\, p_1, p_2 \in P \bullet p_1.x = p_2.x$
  We categorize this category of states into into two categories:
  - (a) $\exists\, p \neq \mathit{root} \bullet p.\mathit{up} = \top$.
    In this category, a non-root process has a privilege if one of the following guards holds:
    - $\mathit{up} \wedge \exists\, \mathit{ch} \in \mathit{child} \bullet \mathit{ch}.x = x \wedge \neg \mathit{ch}.\mathit{up} \wedge \mathit{ch}.\mathit{rdyChild} \neq -1$ (Line 22)

$*$ $up \wedge \forall\, ch \in child \bullet ch.x = x \wedge \neg ch.up \wedge ch.rdyChild = -1$ (Line 24)

Now consider a process p of the highest depth where $p.up = \top$. Given that for all leaves $up = \bot$, then p is not a leaf, and all the children of p have $up = \bot$. This implies that one of the guarded commands is enabled, which implies that p has a bottom privilege.

- (b) $\forall\, p \neq root \bullet p.up = \bot$

  By considering Algorithm 1, the root has a privilege if one of the following guards holds:

  $*$ $x \wedge \exists\, ch \in child \bullet ch.x = x \wedge \neg ch.up \wedge ch.rdyChild \neq -1$ (Line 1)

  $*$ $x \wedge \forall\, ch \in child \bullet ch.x = x \wedge \neg ch.up \wedge ch.rdyChild = -1$ (Line 4)

  $*$ $\neg x \wedge \forall\, ch \in child \bullet ch.x = x \wedge \neg ch.up$ (Line 7)

  In this category, it holds that $\forall\, ch \in root.child \bullet ch.x = root.x \wedge ch_?^{\perp}$. This implies that at least one of the root guarded commands is enabled. So the root has a privilege.

□

5.2 Proof of Theorem 2

LEMMA 1 *Given a system over a topology T, a path $p_1, ..., p_n$, and a global state $\sigma_0 : p_i{}_b^?{}^?$ for $1 \leq i \leq n$ and $b \in$ B. For each execution prefix $\sigma_0, ..., \sigma_m$, where $\sigma_j : p_1{}_b^?$ for $0 \leq j \leq m$, each of the processes $p_2, ..., p_n$ does not have an above privilege.*

Proof. A process p may have an above privilege only if $p.x \neq parent(p).x$. In σ_0, $p.x = parent(p).x = b$, and therefore p does not have an above privilege. p may have an above privilege in a consequent state only if $parent(p)$ has changed the value of x. Privileges from bottom may exist. However, execution steps following bottom privileges do not change the value of x for all processes except the root. Notice that if the root exists in the sequence, then by definition, it can be only p_1. Therefore, it is required that p_1 changes its state into $p_1{}_{\neg b}^?$ in order to create an above privilege. □

COROLLARY 2 *Given a system over a topology T, a path $p_1, ..., p_n$ and a global state $\sigma_0 : p_i{}_b^?$ for $1 \leq i \leq n$ and $b \in$ B. For each execution $\sigma_0, \sigma_1, ..$, each of $p_2, ..., p_n$ has no above privilege unless p_1 changes its x value through:*

- *a bottom privilege if $p_1 = root$, or*
- *an above privilege*

Proof. Lemma 1. □

LEMMA 2 *Given a system over a topology T, a path $p_1, ..., p_n$, and a global state $\sigma_0 : p_1{}_b^?$ where $b \in$ B. For each execution $\sigma_0, \sigma_1, ..$, there exists finally a global state σ_k, where $k \geq 0$, such that*

$$\textit{if} \quad \forall\, j \in [0, k] \bullet \sigma_j : p_1{}_b^?, \qquad \textit{then} \quad \forall\, i \in [1, n] \bullet \sigma_k : p_i{}_b^?$$

i.e. there is an upper bound of the number of execution steps where for each state, $p_1{}_b^?$ holds, and not all other processes p_i have $p_i{}_b^?$

Proof. Consider the path $p_1, ..., p_n$. If in σ_0, $p_i.x = b$ for $1 \leq i \leq n$, then the claim holds trivially. Otherwise, consider ${p_i}^?_{\neg b}$ with the least depth d: By Corollary 2, and given that ${p_1}^?_b$ holds, p_{i-1} does not have an above privilege. Now p_i may have an above privilege and possibly bottom. In the next execution step (σ_0, σ_1), p_i takes the above privilege and switches from ${p_i}^?_{\neg b}$ into ${p_i}^?_b$. Now in σ_1, p_i does not have an above privilege, given that $p_1.x = b$. Note that the value of x is equal to b in all the processes in the lower depths. Inductively, the same procedure applies to the processes with depth $d + 1$ in the next execution steps. Since the path is finite, then eventually $x = b$ for all the processes, given that $p_1.x$ remains equal to b. □

Lemma 3 *Given system over a topology T, and a non-root process p with a global state $\sigma_0 : {p_b}^\top$. For each execution $\sigma_0, \sigma_1, ..$, there exists finally a global state σ_k, where $k > 0$, such that*

$$\textit{if} \quad \forall\, j \in [0, k] \bullet \sigma_j : \textit{parent}(p)^?_b, \qquad \textit{then} \quad \sigma_k : {p_b}^\bot$$

i.e. there is an upper bound on the number of execution steps where ${p_b}^\top$ holds, if $\textit{parent}(p)^?_b$ holds in the states.

Proof. By Lemma 2, after a finite number of execution steps, at a state σ_j, it holds that all processes p_i in the maximal paths $p, .., p_z$ switch into ${p_i}^?_b$, and in particular, ${p_z}^\bot_b$ for leaves. At this point, by definition, there is no above privilege for any process in the subtree rooted by p. Moreover, by Lemma 1, steps following bottom privileges do not enable above privileges as long as $p^?_b$.

Let p_r be a process in the subtree rooted by p, such that p_r is the process with the largest depth d where ${p_r}^\top_b$ holds. This process has either one child which is a leaf p_z, such that ${p_z}^\bot_b$, or it has multiple children as leaves or other processes p_s with ${p_s}^\bot_b$. By Algorithm 2, a non-root process switches *up* into $\bot$ if one of the following guards hold:

- $\textit{up} \wedge \exists\, \textit{ch} \in \textit{child} \bullet \textit{ch}.x = x \wedge \neg \textit{ch}.\textit{up} \wedge \textit{ch}.\textit{rdyChild} \neq -1$ (Line 22)
- $\textit{up} \wedge \forall\, \textit{ch} \in \textit{child} \bullet \textit{ch}.x = x \wedge \neg \textit{ch}.\textit{up} \wedge \textit{ch}.\textit{rdyChild} = -1$ (Line 24)

In σ_j, one of the guards holds for p_r. Therefore, in the next step, p_r and similarly all processes at depth d switch into ${p_r}^\bot_b$. Keep in mind that, by Algorithm 2, there is no privileges from above or bottom for all processes in the subtree rooted by p at depth $\geq d$. Inductively, the processes in $d - 1$ or less will perform the same action in the next step. Since the depth is finite, p switches into ${p_b}^\bot$ in some state σ_k. □

Lemma 4 *Given a system over a topology T and a global state $\sigma_0 : \textit{root}^\top_\top$. After a finite number of execution steps, the system switches into $\sigma_k : \textit{root}^\top_\bot$.*

Proof. By Lemma 3, after a finite number of steps, the system switches into a global state σ_{k-1} in which: either one of the root children ch_i is ${ch_i}^\bot_\top \neq 1$, or each root child ch is ${ch}^\bot_\top$-1. By Algorithm 1 (Lines 1 and 4), in σ_{k-1}, the root has a bottom privilege, which enables a guarded command that switches the root into $\textit{root}^\top_\bot$ in σ_k.

□

LEMMA 5 *Given a system over a topology T and a global state $\sigma_0 : root^{\top}_{\bot}$. After a finite number of execution steps, the system switches into σ_k, where:*

- $root^{\top}_{\top}$*-1*
- *For each process p_i in depth 1, it holds that $p_i{}^{\bot}_{\bot}$*
- *For each p_j in depth 2, it holds that $p_j{}^{?}_{\bot}$*

Proof. The root does not change its state until all root children ch are $ch^{\bot}_{\bot}$. Consider the root children ch:

- In the current state, either all root children are $ch^{\bot}_{\bot}?$, or
- Regardless of the current state, in the next step, each ch switches into $ch^{?}_{\bot}?$. Then, by Lemma 3 and Corollary 2, after a finite number of steps, each root child becomes $ch^{\bot}_{\bot}?$.

Now each root child ch is $ch^{\bot}_{\bot}?$. In the next execution step, the root executes finitely if it is active, then it switches into $root^{\top}_{\top}$-1. At the same time, and by definition, the children of depth 2 at least (if they exist) switch into $ch^{?}_{\bot}?$. □

LEMMA 6 *Given a system over a topology T and a global state*

$$\sigma_0 : \quad root^{\top}_{\top}\text{-}1 \quad \wedge \quad \forall p \in P \bullet (depth(p) = 1) \rightarrow p^{\bot}_{\bot}? \quad \wedge \quad (depth(p) = 2) \rightarrow p^{?}_{\bot}?,$$

after a finite number of execution steps, the system reaches a safe state.

Proof. In the next execution step (σ_0, σ_1), by Algorithm 2, line 1:

- Processes p of depth 1 switch into $p^{\top}_{\top}$-1 or $p^{\bot}_{\top} \neq 1$.
- Processes p of depth 3 switch into $p^{\top}_{\bot}$ or $p^{\bot}_{\bot}$. Now it is obvious that the subtree rooted by the root and including only all the processes at depth 1 satisfy the safe state conditions if we ignore that fact that all leaves have $up = \bot$.

In σ_1, the processes in depth 2 do not create privileges to the processes in depth 1, because each of the $x's$ of the processes of depth 1 is equal to $\top$, and each of the $x's$ of the processes of depth 2 is equal to $\bot$.

Analogously, in the next execution step (σ_1, σ_2), Processes at depth 2 and depth 4 perform similar actions as processes at depth 1 and depth 3 in (σ_0, σ_1) done, respectively. Again in σ_2, the subtree rooted by the root and including only all the processes at depths 1 and 2 satisfy the safe state conditions if we ignore that fact that all leaves have $up = \bot$.

Analogously, this execution step is repeated for larger depths. Since the depth is finite, then after some steps, the system reaches a safe state.

□

Proof of Theorem 2:

Proof. By Theorem 1, the system is a set of infinite executions. For each execution, in the initial state σ_0, the root is either $root^{\top}_{\top}$ or $root^{\top}_{\bot}$. If $root^{\top}_{\top}$, then by Lemma 4, after a finite number of steps, the root switches into $root^{\top}_{\bot}$. Given $root^{\top}_{\bot}$, by Lemma 5, the root switches into $root^{\top}_{\top}$-1 where:

$$\forall\, p \in P \bullet (depth(p) = 1) \rightarrow p^{\bot}_{\bot}? \quad \wedge \quad (depth(p) = 2) \rightarrow p^{?}_{\bot}?,$$

By Lemma 6, after a finite number of steps, the system switches into a safe state. □

5.3 Proof of Theorem 3

LEMMA 7 *Given a system over a topology T, for each execution $\sigma_0, \sigma_1, ...$ if $\sigma_0 \in \Sigma_{safe}$, then for each state σ_i for $i \geq 0$:*

$$\forall\, p \in P \bullet p^{\top}_{\top} \Rightarrow p^{\top}_{\top}\text{-}1 \wedge \neg p.active \tag{1}$$

$$\forall\, p_s \in P \bullet p_s{}^{\bot}_{\top} \Rightarrow \tag{2}$$

$$\exists\, n \geq s \bullet p_s, ..., p_n : aPath_{\sigma} \vee \tag{3}$$

$$\forall\, mPath : p_s, ..., p_z \bullet \forall\, e \in [s, z] \bullet \neg p_e.active \wedge p_e{}^{\bot}_{\top}\text{-}1 \tag{4}$$

Proof. We show that the conditions (1-4) hold in each σ_i by induction. For the base Case σ_0, the conditions hold by Definition 5. For the induction step: We show that if the conditions hold in some state σ_{j-1}, then they hold in σ_j for $j \in \mathrm{N}^{+}$.

- Condition (1): By definition, in σ_{j-1}, for each process $p \in P$ where $p^{\top}_{\top}$, it holds that $p.rdyChild = -1 \wedge \neg p.active$. By definition, the value of *rdyChild* changes in σ_j only if:
  - A process $p^{\top}_{\bot}$ switches into $p^{\bot}_{\bot}$, and
  - A process $p^{?}_{\bot}$ switches into $p^{\top}_{\top}$-1 or $p^{\bot}_{\top} \neq 1$.

  In any of the above cases, $\sigma_j : p^{\top}_{\top} \Rightarrow p^{\top}_{\top}$-1.
  Now regarding being active or not, this depends on the output of *update*(*active*), which is called only at Algorithm 1 (Line 7), or Algorithm 2 (Line 16). In both cases, if for a process p the output of *update*(*active*) is *true*, then p switches into $p^{\bot}_{\top}$, but not into $p^{\top}_{\top}$. Condition (1) holds in σ_j.
- Condition (2): By hypothesis, the condition is satisfied by σ_{i-1}. By definition, always $root.up = \top$. Therefore the condition holds trivially for the root. Let $p \in P$ be a non-root process such that $\sigma_{j-1} : \neg(p^{\bot}_{\top})$. p switches into $p^{\bot}_{\top}$ in σ_j iff
  - (Lines 18, 21) $\sigma_{j-1} : p^{\bot}_{\bot} \wedge p.active \wedge parent(p)^{?}_{\top}$. p switches into $p^{\bot}_{\top} id$ in σ_j. Notice that the new state σ_j has an active path with one process p satisfying Sub-condition (3).
  - (Lines 22, 23) $\sigma_{j-1} : p^{\top}_{\top} \wedge \exists\, ch \in p.child \bullet ch.x = p.x \wedge ch^{\bot}_{?} \wedge ch.rdyChild \neq -1$. In this case, p switches into $p^{\bot}_{\top} ch.id$ forming an active path $p, ch, ..., p_n$ for $p_n \in P$, satisfying Sub-condition (3).

- (Lines 24, 25) $\sigma_{j-1} : p^{\top}_{\top} \wedge \forall\, ch \in p.child \bullet ch.x = p.x \wedge ch^{\perp}_{?} \wedge ch.rdyChild = -1$. In this case, p switches into $p^{\perp}_{\top}$-1 satisfying Sub-condition (4).

□

Lemma 8 *Given a system over a topology T, for each execution prefix $\sigma_0, ..., \sigma_n$ if $\sigma_0 \in C1$ and $\sigma_i : root^{\top}_{\top}$ for $0 \leq i \leq n$, then $\sigma_i \in C1$.*

Proof. Given that $\sigma_i : root^{\top}_{\top}$, then by Definition 5 and 6, $\sigma_i \in C1$ if each of the following holds:

$$\forall\, p \in P \bullet p^{\top}_{\top} \Rightarrow p^{\top}_{\top}\text{-}1 \wedge \neg p.active \tag{5}$$

$$\forall\, p_s \in P \bullet p_s{}^{\perp}_{\top} \Rightarrow \tag{6}$$

$$\exists\, n \geq s \bullet p_s, ..., p_n : aPath_{\sigma} \vee \tag{7}$$

$$\forall\, mPath : p_s, ..., p_z \bullet \forall\, e \in [s, z] \bullet \neg p_e.active \wedge p_e{}^{\perp}_{\top}\text{-}1 \tag{8}$$

$$\forall\, p \in P \bullet p^{\top}_{?} \Rightarrow p^{\top}_{\top}\text{-}1 \wedge \tag{9}$$

$$\forall\, p \neq root \bullet (p^{?}_{\top} \wedge parent(p).rdyChild = p.id) \Rightarrow parent(p)^{?}_{\top} \wedge \tag{10}$$

We show that each σ_i satisfies each of the above conditions separately by induction: The base case holds trivially: $\sigma_0 \in C1$. Regarding the induction step: We show the that if $\sigma_{j-1} \in C1$ for $0 < j \leq n$, then $\sigma_j \in C1$.

- Conditions (5-8) hold in σ_j by Lemma 7.
- Condition (10): Given σ_{j-1}, let p be a non-root process such that $p^{?}_{\top}$. p switches into $p^{?}_{\perp}$ in σ_j only if $parent(p)$ is $parent(p)^{?}_{\perp}$ in σ_{j-1}. By Condition (10), if $parent(p)^{?}_{\perp}$, then $parent(p).rdyChild \neq p.id$, which implies that p switches only into $p^{\perp}_{\perp}$-1, which also satisfies Condition (10) and the other conditions.
- Condition (9): No process p may switch into $p^{\top}_{\perp}$ in σ_j because switching a non-root process into $^{\top}_{\perp}$ requires that

$$p^{?}_{\top} \wedge parent(p).rdyChild = p.id \wedge parent(p)^{?}_{\perp} \tag{11}$$

  which does not hold in *C1* by Condition (10).

□

Corollary 3 *Given a system over a topology T, and a non-root process p with a global state $\sigma_0 : p^{\top}_{\top}$ where $\sigma_0 \in C1$, for each execution $\sigma_0, \sigma_1, ..$, there exists finally a global state σ_k, where $k > 0$, such that*

$$\textit{if} \quad \forall\, j \in [0, k] \bullet \sigma_j : parent(p)^{\top}_{\top}, \qquad \textit{then} \quad \sigma_k : p_1{}^{\perp}_{\top} v \wedge \sigma_j \in C1$$

where

1. *$v = -1$ iff there exists no active process in any of p descendants.*
2. *$v \neq -1$ iff there exists at least one active process in one of p descendants.*

Proof. Lemmas 3 and 8.

□

Proof of Theorem 3:

Proof. By Lemma 8, $\sigma_i \in C1$. By Algorithm 1, an execution step switches $root^{\top}_{\top}$ into $\sigma_k : root^{\top}_{\bot}$ where:

1. $\sigma_k \in C2$ if and only if
$$\sigma_{k-1} : root^{\top}_{\top} \wedge \exists\, ch \in root.child \bullet ch^{\bot}_{\top} \wedge ch.rdyChild \neq -1 \qquad (12)$$
2. $\sigma_k \in C4$ if and only if
$$\sigma_{k-1} : root^{\top}_{\top} \wedge \forall\, ch \in root.child \bullet ch^{\bot}_{\top} \wedge ch.rdyChild = -1 \qquad (13)$$

We show that either (12) or (13) eventually holds. We distinguish 3 cases:

1. If there exists a root child ch such that $ch^{\bot}_{\top} \neq -1$, then (12) holds, and the root switches into $root^{\top}_{\bot}$ in $\sigma_2 \in C2$.
2. Given a root child ch, If there exists an active process in a maximal path $ch, ..., p_z$, then by Corollary 3 and Definition 5, after a finite number of execution steps, ch switches into $ch^{\bot}_{\top} \neq -1$, and then (12) holds, and the root switches into $root^{\top}_{\bot}$ in a state $\sigma_k \in C2$.
3. If there exists no active process in any maximal path starting from a a root child, then by Corollary 3 and Definition 5, after a finite number of steps, for all root children ch, it holds that $ch^{\bot}_{\top} = -1$, and then (13) holds, and the root switches into $root^{\top}_{\bot}$ in a state $\sigma_k \in C4$.

□

5.4 Proof of Theorem 4

LEMMA 9 *Given a system over a topology T, for each execution prefix $\sigma_0, ..., \sigma_n$, if $\sigma_0 \in C2 \cup C3$ and $\sigma_i : root^{\top}_{\bot}$ for $0 \leq i \leq n$, then $\sigma_i \in C2 \cup C3$.*

Proof. By Definitions 5 and 6 a state σ is in $C2 \cup C3$ if and only if the following conditions hold:

$$\forall\, p \in P \bullet p^{\top}_{\top} \Rightarrow p^{\top}_{\top}\text{-}1 \wedge \neg p.active \qquad (14)$$

$$\forall\, p_s \in P \bullet p_s{}^{\bot}_{\top} \Rightarrow \qquad (15)$$
$$\exists\, n \geq s \bullet p_s, ..., p_n : aPath_{\sigma} \;\vee$$
$$\forall\, mPath : p_s, ..., p_z \bullet \forall\, e \in [s, z] \bullet \neg p_e.active \wedge p_e{}^{\bot}_{\top}\text{-}1$$

$$\exists !\, ePath_{\sigma} : p_0, ..., p_n \qquad (16)$$
$$(\forall\, i \in [0, n-1] \bullet p_i.rdyChild = p_{i+1}.id \;\wedge\; p_n.rdyChild = p_n.id \;\wedge\; p_n{}^{\bot}_{?} \;\wedge$$
$$p_i{}^{\top}_{?} \Rightarrow p_i{}^{\top}_{\bot} \;\wedge\; p_{i+1}{}^{\top}_{\bot} \Rightarrow p_i{}^{\top}_{\bot} \;\wedge\; p_{i+1}{}^{?}_{\bot} \Rightarrow p_i{}^{?}_{\bot} \;\wedge\; p_i{}^{\bot}_{?} \Rightarrow p_{i+1}{}^{\bot}_{?} \;\wedge$$
$$p_i{}^{\bot}_{\top} \Rightarrow \neg p_i.active \;\wedge\; p_n{}^{\bot}_{\top} \Rightarrow p_n.active)$$

We show that each σ_i satisfies each of the above conditions separately by induction: For the base case, $\sigma_0 \in C2 \cup C3$ by definition. For the induction step, we show that if $\sigma_{j-1} \in C2 \cup C3$ for $0 < j \leq n$, then $\sigma_j \in C2 \cup C3$.

- Conditions (14) and (15) are satisfied in σ_j by Lemma 7.
- Condition (16): In state σ_{j-1}, by Condition (15), the $ePath_{\sigma_{j-1}}$ can be expressed as one of the following:

$$p_0 {}^{\top}_{\perp} p_1, p_1 {}^{\top}_{\perp} p_2, ..., p_k {}^{\top}_{\perp} p_{k+1}, p_{k+1} {}^{\perp}_{\top} p_{k+2}, ..., p_n {}^{\perp}_{\top} p_n \tag{17}$$

$$p_0 {}^{\top}_{\perp} p_1, p_1 {}^{\top}_{\perp} p_2, ..., p_r {}^{\top}_{\perp} p_{r+1}, p_{r+1} {}^{\perp}_{\perp} p_{r+2}, ..., p_n {}^{\perp}_{\perp} p_n \tag{18}$$

  Recall that in the premise, we consider states where $root {}^{\top}_{\perp}$. In (17), by definition, the only process with a privilege is p_{k+1} which has an above privilege. The execution step (σ_{j-1}, σ_j) switches p_{k+1} into $p_{k+1} {}^{\top}_{\perp} p_{k+2}$, which does not violate the conditions. In (18), the execution step (σ_{j-1}, σ_j) switches p_r into $p_r {}^{\perp}_{\perp} p_{r+1}$. Other processes in the $ePath_{\sigma_{j-1}}$ do not have privileges.

□

Proof of Theorem 4:

Proof. By Lemma 9, if $\sigma_i : root {}^{\top}_{\perp}$, then $\sigma_i \in C2 \cup C3$. In general, given a state $\sigma \in C2 \cup C3$ and an $ePath : p_0, ..., p_n$:

$$p_n {}^{\perp}_{\top} \Rightarrow \sigma \in C2 \quad \wedge \quad p_n {}^{\perp}_{\perp} \Rightarrow \sigma \in C3$$

Given that σ_0 is in $C2$, then for the $ePath_{\sigma_0} : p_0, ..., p_n$,

$$\sigma_0 : p_n {}^{\perp}_{\top} p_n$$

By Definition 4, for the $ePath_\sigma$:

$$\begin{aligned}(\forall\, i \in [0, n-1] \bullet\ & p_i.rdyChild = p_{i+1}.id \quad \wedge \quad p_n.rdyChild = p_n.id \quad \wedge \quad p_n {}^{\perp}_{?} \quad \wedge \\ & p_i {}^{\top}_{?} \Rightarrow p_i {}^{\top}_{\perp} \quad \wedge \quad p_{i+1} {}^{\top}_{\perp} \Rightarrow p_i {}^{\top}_{\perp} \quad \wedge \quad p_{i+1} {}^{?}_{\perp} \Rightarrow p_i {}^{?}_{\perp} \quad \wedge \quad p_i {}^{\perp}_{?} \Rightarrow p_{i+1} {}^{\perp}_{?} \quad \wedge \\ & p_i {}^{\perp}_{\top} \Rightarrow \neg p_i.active \quad \wedge \quad p_n {}^{\perp}_{\top} \Rightarrow p_n.active)\end{aligned}$$

The $ePath_\sigma$ can be expressed as follows:

$$p_0 {}^{\top}_{\perp} p_1, p_1 {}^{\top}_{\perp} p_2, ..., p_k {}^{\top}_{\perp} p_{k+1}, p_{k+1} {}^{\perp}_{\top} p_{k+2}, ..., p_n {}^{\perp}_{\top} p_n \tag{19}$$

By Algorithm 2 and Lemmas 2 and 3:

- The processes $p_0, ..., p_k$, $p_{k+2}, ..., p_n$ have no privileges from above.
- The processes $p_0, ..., p_n$ have no privileges from bottom.
- The only process which has a privilege is p_{k+1} (Line 1), and it is an above privilege.

In σ_1, p_{k+1} switches into $p_{k+1} {}^{\top}_{\perp} p_{k+2}$. Notice that $\sigma_1 \in C2$. Analogously, p_{k+2} performs the same action in σ_2. Since the number of $p_{k+1}, ..., p_n$ is finite, then a state σ_{k-1} is reached where

$$p_0 {}^{\top}_{\perp} p_1, p_1 {}^{\top}_{\perp} p_2, ..., p_{n-1} {}^{\top}_{\perp} p_n, p_n {}^{\perp}_{\top} p_n \tag{20}$$

In this state, by Algorithm 2, the only process with a privilege is p_n (Line 1), which is an above privilege. In the execution step (σ_{k-1}, σ_k), p_n runs the command *execute*() and switches into $p_n {}^{\top}_{\perp} p_n$, which implies that $\sigma_k \in C3$. □

5.5 Proof of Theorem 5

Proof. By Lemma 9, if $root^{\top}_{\perp}$ in σ_i, then $\sigma_i \in C2 \cup C3$. In state $\sigma_0 \in C3$, the $ePath_{\sigma_0}$ can be expressed as follows:

$$p_0 {}^{\top}_{\perp} p_1, p_1 {}^{\top}_{\perp} p_2, ..., p_k {}^{\top}_{\perp} p_{k+1}, p_{k+1} {}^{\perp}_{\perp} p_{k+2}, ..., p_n {}^{\perp}_{\perp} p_n \tag{21}$$

By definition and Lemmas 2 and 3:

- The processes $p_0, ..., p_n$ have no privileges from above.
- The processes $p_0, ..., p_{k-1}$, $p_{k+1}, ..., p_n$ have no privileges from bottom.
- By Algorithm 2, the only process which has a privilege is p_k (Line 22), and it is a bottom privilege.

In the execution step (σ_0, σ_1), p_k switches into $p_k {}^{\perp}_{\perp} p_{k+1}$. Analogously, p_{k-1} performs the same step. Since the number of processes $p_1, ..., p_k$ is finite, then at a state σ_r for $r \geq 0$, the $ePath$ is expressed as follows:

$$p_0 {}^{\top}_{\perp} p_1, p_1 {}^{\perp}_{\perp} p_2, ..., p_k {}^{\perp}_{\perp} p_{k+1}, p_{k+1} {}^{\perp}_{\perp} p_{k+2}, ..., p_n {}^{\perp}_{\perp} p_n \tag{22}$$

By Lemma 3, at some state σ_l for $l \geq r$, each root child ch switches into $ch^{\perp}_{\perp}$. This create a bottom privilege for the root (Line 4). In the next execution step, the root executes finitely if it is active, then it switches into a state $\sigma_k : root^{\top}_{\top}\text{-}1 \in C1$. □

5.6 Proof of Theorem 6

LEMMA 10 *Given a system over a topology T, for each execution prefix $\sigma_0, ..., \sigma_n$, if $\sigma_0 \in C4$ and $\sigma_i : root^{\top}_{\perp}$ for $0 \leq i \leq n$, then $\sigma_i \in C4$.*

Proof. Proving

$$\forall\, p \in P \bullet p^{\top}_{\top} \Rightarrow p^{\top}_{\top}\text{-}1 \wedge \neg p.active \tag{23}$$

$$\forall\, p_s \in P \bullet p_s {}^{\perp}_{\top} \Rightarrow \tag{24}$$

$$\exists\, n \geq s \bullet p_s, ..., p_n : aPath_{\sigma} \ \vee$$

$$\forall\, mPath : p_s, ..., p_z \bullet \forall\, e \in [s, z] \bullet \neg p_e.active \wedge p_e {}^{\perp}_{\top}\text{-}1$$

$$\forall\, p \neq root \bullet \tag{25}$$

$$p^{\top}_{?} \Rightarrow p^{\top}_{\top}\text{-}1 \ \wedge \tag{26}$$

$$p^{?}_{\top} \wedge parent(p)^{?}_{\perp} \Rightarrow parent(p).rdyChild \neq p.id \tag{27}$$

Condition (23) and (24) hold in σ_i by Lemma 7. We show that Condition (25) holds in σ_i by induction: For the base case, Condition (25) holds in σ_0 by Definition 5. For the induction step: We show that if it holds in σ_{j-1}, then it holds in σ_j for $j \in \mathrm{N}^+$:

- Part (26): Given σ_{j-1}, let p be a non-root process such that $p^{?}_{\top}$. p switches into $p^{?}_{\perp}$ in σ_j only if $parent(p)$ is $parent(p)^{?}_{\perp}$ in σ_{j-1}. By (27), if $parent(p)^{?}_{\perp}$, then $parent(p).rdyChild \neq p.id$, which implies that p switches only into $p^{\perp}_{\perp}$-1, which also satisfies Condition (27) and the other conditions.

– Part (27): No process p may switch into $p^{\top}_{\perp}$ in σ_j because switching a non-root process into $\overset{\top}{\perp}$ requires that

$$p^{?}_{\top} \wedge \mathit{parent}(p).\mathit{rdyChild} = p.\mathit{id} \wedge \mathit{parent}(p)^{?}_{\perp} \tag{28}$$

which does not hold in *C1* by (27).

□

Proof of Theorem 6

Proof. By Lemma 10, if $\sigma_0 \in C4$ and $\sigma_i : \mathit{root}^{\top}_{\perp}$, then $\sigma_i \in C4$. By Lemma 5, the root finally switches into $\mathit{root}^{\top}_{\top}$-1. □