UE ARA — M2 SAR
– Self-Stabilization –

Franck Petit

1 Model and Definition

Model. We consider a network as an undirected connected graph $G = (V, E)$ where V is a set of processors and E is a binary relation that denotes the ability for two processors to communicate ($(p, q) \in E$ if and only if p and q are *neighbors*). Every processor p can distinguish its neighbors and locally label them, and we assume that p maintains N_p, the set of its neighbors local labels. In the following, n denotes the number of processors, and Δ the maximal degree. If p and q are two processors of the network, we denote by $d(p, q)$ the length of the shortest path between p and q (*i.e* the *distance* from p to q).

We consider the classical local shared memory model of computation where communications between neighbors are modeled by direct reading of variables instead of exchange of messages. In this model, the program of every processor consists in a set of shared variables (henceforth, referred to as *variables*) and a finite set of *rules*. A processor can write to its own variables only, and read its own variables and those of its neighbors. Each rule consists of: $<$*label*$>$::$<$*guard*$>\longrightarrow<$*statement*$>$. The label of a rule is simply a name to refer the action in the text. The guard of a rule in the program of p is a boolean predicate involving variables of p and its neighbors. The statement of a rule of p updates one or more variables of p. A statement can be executed only if the corresponding guard is satisfied (the processor rule is then *enabled*). The state of a processor is defined by the value of its variables. The state of a system (*a.k.a.* the *configuration*) is the product of the states of all processors. We note Γ the set of all configurations of the system.

Processor p is *enabled* in $\gamma \in \Gamma$ if and only if at least one rule is enabled for p in γ. Let a distributed protocol $\mathcal{P}$ be a collection of binary transition relations denoted by $\rightarrow$, on Γ. An execution of a protocol $\mathcal{P}$ is a maximal sequence of configurations $\epsilon = \gamma_0\gamma_1 \ldots \gamma_i\gamma_{i+1} \ldots$ such that, $\forall i \geq 0, \gamma_i \rightarrow \gamma_{i+1}$ ($(\gamma_i, \gamma_{i+1}) \in \rightarrow$ is called a *step*) if γ_{i+1} exists (else γ_i is a *terminal* configuration). *Maximality* means that the sequence is either finite (and no action of $\mathcal{P}$ is enabled in the terminal configuration) or infinite. $\mathcal{E}$ is the set of all possible executions of $\mathcal{P}$. A processor p is *neutralized* in step $\gamma_i \rightarrow \gamma_{i+1}$ if p is enabled in γ_i and is *not* enabled in γ_{i+1}, yet did not execute any rule in step $\gamma_i \rightarrow \gamma_{i+1}$.

A *scheduler* (also called *daemon*) is a predicate over the executions. In any execution, each step $\gamma \longrightarrow \gamma'$ results from a *non-empty* subset of enabled processors *atomically executing* a rule. This subset is chosen by the scheduler. A scheduler is *central* if it chooses *exactly one* enabled processor in any particular step, it is *distributed* if it chooses *at least one* enabled processor, and *locally central* if it chooses *at least one* enabled processor yet ensures that no two neighbors are chosen concurrently. A scheduler is *synchronous* if it chooses *every* enabled processor in every step. A scheduler is *asynchronous* if it is either central, distributed or locally central.

A scheduler may also have some *fairness* properties. A scheduler is *strongly fair* (the strongest fairness assumption for asynchronous schedulers) if every processor that is enabled *infinitely often* is eventually chosen to execute a rule. A scheduler is *weakly fair* if every *continuously* enabled processor is eventually chosen to execute a rule. Finally, the *unfair* scheduler has the weakest fairness assumption: it only guarantees that at least one enabled processor is eventually chosen to execute a rule. As the strongly fair scheduler is the strongest fairness assumption, any problem that cannot be solved under this assumption cannot be solved for all weaker fairness assumptions. In contrast, any algorithm performing under the unfair scheduler also works for all stronger fairness assumptions.

Definition. We consider *self-stabilizing* algorithms, that is algorithms that recover a correct behavior in a finite time after an unbounded number of transient failures. The formal definition of self-stabilization follows.

Definition 1 (Self-Stabilization) *An algorithm $\mathcal{A}$ is self-stabilizing for a specification $\mathcal{S}$ if and only if the two following properties hold:*

Closure: Starting from any configuration that satisfies $\mathcal{S}$, any execution of $\mathcal{A}$ always satisfies $\mathcal{S}$.

Convergence: Starting from any configuration that does not satisfy $\mathcal{S}$, any execution of $\mathcal{A}$ reaches in a finite time a configuration which satisfies $\mathcal{S}$.

2 Token Circulation

In this exercise, we consider a mutual exclusion protocol by token circulation. The specifications of these problems follow.

Specification 1 (Mutual Exclusion)

Liveness: Any processor can enter in critical session infinitely often.

Safety: In any configuration, at most one processor is in critical session.

Specification 2 (Token Circulation)

Liveness: Any processor holds infinitely often the token.

Safety: In any configuration, at most one processor holds the token.

The network is a ring of n processors denoted by $p_0, \ldots, p_{n-1}$. We assume a distributed unfair scheduling. Each processor p_i has only one variable $c_i \in \{0, 1, \ldots, k-1\}$ (where k is a given number such that $k > n$). The processor p_0 holds the token if $c_0 = c_{n-1}$. A processor p_i ($i \neq 0$) holds the token if $c_i \neq c_{i-1}$. The protocol is the following.

Algorithm 1 Self-stabilizing token circulation for processor p_i

Rule for $i = 0$:
$(R_1) :: c_0 = c_{n-1} \longrightarrow c_0 := c_0 + 1 \text{ modulo } k$

Rule for $i \neq 0$:
$(R_2) :: c_i \neq c_{i-1} \longrightarrow c_i := c_{i-1}$

Questions:

1. Give an example of execution of the algorithm starting from the following initial configuration: $n = 5$ and $\forall i \in \{0, \ldots, n-1\}, c_i = 0$.
2. Prove that at least one processor holds the token in any configuration.
3. Prove that any processor holds the token infinitely often in any execution starting from any configuration.
4. Prove that the specification of token circulation is satisfied in any execution starting from a configuration in which exactly one token exists.
5. Prove that any execution starting from any configuration reach a configuration in which exactly one token exists.
6. Conclude.

3 Maximal Matching

In this exercise, we consider the problem of maximal matching. Specification of this problem follows.

Definition 2 (Matching) *Given a graph $G = (V, E)$, a matching M on G is a subset of E such that any node of V belongs to at most one edge of M. A matching is maximal if there exists no matching M' such that $M \subsetneq M'$.*

Specification 3 (Maximal Matching)

Liveness: The protocol ends in a finite time.

Safety: In the terminal configuration, there exists a maximal matching on the system.

The network is an arbitrary graph. We assume a central unfair scheduling. Each processor p has a variable $pref_p$ which belongs to the set $N_p \cup \{null\}$. This variable refers to the preferred neighbor of p for a matching. For example, if $pref_p = q$ then p wants to add the edge $\{p, q\}$ to the matching. For any processor p, we define the following set of predicates over the system:

$$\begin{array}{rcc} waiting_p & \equiv & (pref_p = q) \wedge (pref_q = null) \\ matched_p & \equiv & (pref_p = q) \wedge (pref_q = p) \\ chaining_p & \equiv & (pref_p = q) \wedge (pref_q = r) \vee (r \neq p) \\ dead_p & \equiv & (pref_p = null) \wedge (\forall q \in N_p, matched(q) = true) \\ free_p & \equiv & (pref_p = null) \wedge (\exists q \in N_p, matched(q) = false) \end{array}$$

It is easy to verify that for any configuration γ and for any processor p, exactly one of these predicates holds for p in γ. The algorithm follows.

Algorithm 2 Self-stabilizing maximal matching for processor p

Rules:

/* Matching rule */
$(M) :: (pref_p = null) \wedge (\exists q \in N_p, pref_q = p) \longrightarrow pref_p := q$
/* Seduction rule */
$(S) :: (pref_p = null) \wedge (\forall q \in N_p, pref_q \neq p) \wedge (\exists q \in N_p, pref_q = null) \longrightarrow pref_p := q$
/* Abandonment rule */
$(A) :: (pref_p = q) \wedge (pref_q \neq p) \wedge (pref_q \neq null) \longrightarrow pref_p := null$

Questions:

1. Prove that the set of edges $\{\{p, pref_p\} | pref_p \neq null\}$ is a maximal matching in a configuration which satisfies $\forall p \in V, matched_p \vee dead_p$.
2. Prove that any terminal configuration satisfies $\forall p \in V, matched_p \vee dead_p$.
3. Prove that any configuration which satisfies $\forall p \in V, matched_p \vee dead_p$ is a terminal configuration.
4. Prove that any execution starting from any configuration reaches in a finite time a terminal configuration.
   Help: Consider the number of waiting processors (w), the number of chaining processors (c) and the number of free processors (f). Prove that the function $P : \gamma \in \Gamma \rightarrow (w + c + f, 2c + f)$ is strictly decreasing in any execution (consider the lexicographic order).
5. Conclude.

4 Local Mutual Exclusion

In this exercise, we consider the problem of local mutual exclusion. The specification of this problem follows.

Specification 4 (Local Mutual Exclusion)

Liveness: Any processor can enter in critical session infinitely often.

Safety: In any configuration, no two neighbors are in critical session.

The network is an oriented tree. Each processor p can access to its father in the tree by F_p and to the set of its children in the tree by C_i. We assume a distributed weakly fair scheduling. Each processor p has a boolean variable b_p. The algorithm follows.

Algorithm 3 Self-stabilizing local mutual exclusion for processor p

Rule for the root ($p = r$):
$(R) :: \forall q \in C_r, b_r = b_q \longrightarrow b_r := \neg b_r$

Rule for a leaf ($p = l$):
$(L) :: b_l \neq b_{F_l} \longrightarrow b_l := \neg b_l$

Rule for an internal processor:
$(I) :: (b_p \neq b_{F_p}) \wedge (\forall q \in C_p, b_p = b_q) \longrightarrow b_p := \neg b_p$

Questions:

1. Prove that no neighbor of an enabled processor is enabled in any configuration.
2. Prove that, if a processor p has the value b_p in a configuration γ, then any child of p takes the value b_p in a finite time in any execution starting from γ.
3. Prove that any processor is infinitely often activated in any execution starting from any configuration.
4. Conclude.