RR LRI 1459

Conflict Managers for Self-stabilization without Fairness Assumption

Maria Gradinariu* Sébastien Tixeuil†

* LIP6, Université Paris 6, INRIA REGAL Maria.Gradinariu@lip6.fr
† LRI-CNRS UMR 8623 & INRIA Grand Large, Université Paris Sud, tixeuil@lri.fr

Abstract

In this paper, we specify the *conflict manager* abstraction. Informally, a conflict manager guarantees that any two neighboring nodes can not enter their critical simultaneously (*safety*), and that at least one node is able to execute its critical section (*progress*). The conflict manager problem is strictly weaker than the classical local mutual exclusion problem, where any node that requests to enter its critical section eventually does so (*fairness*).

We argue that conflict managers are a useful mechanism to transform a large class of self-stabilizing algorithms that operate in an essentially sequential model, into self-stabilizing algorithm that operate in a completely asynchronous distributed model. We provide two implementations (one deterministic and one probabilistic) of our abstraction, and provide a composition mechanism to obtain a generic transformer. Our transformers have low overhead: the deterministic transformer requires one memory bit, and guarantees time overhead in order of the network degree, the probabilistic transformer does not require extra memory. While the probabilistic algorithm performs in anonymous networks, it only provides probabilistic stabilization guarantees. In contrast, the deterministic transformer requires initial symmetry breaking but preserves the original algorithm guarantees.

Résumé

Dans cet article, nous spécifions l'abstraction du *gestionnaire de conflits*. Informellement, un gestionnaire de conflits garantit que n'importe quel couple de nœuds en conflit ne peut entrer en section critique simultanément (*sûreté*), et qu'au moins un nœud est en mesure d'exécuter sa section critique (*progrès*). Le problème du gestionnaire de conflits est strictement plus faible que celui de l'exclusion mutuelle où tout nœud qui demande à entrer en section critique a la garantie de pouvoir le faire au bout d'un temps fini (*équité*).

Les gestionnaires de conflits constituent un mécanisme utile pour transformer une large classe d'algorithmes auto-stabilisants qui s'exécutent dans un modèle essentiellement séquentiel, en algorithmes qui supportent un modèle d'exécution totalement asynchrone. Nous proposons deux implantations (une déterministe et une probabiliste) de notre abstraction, ainsi qu'un mécanisme de composition permettant d'obtenir une transformation automatique. Le sourcoût en mémoire reste faible : la transformation déterministe requiert seulement un bit, et garantit un surcoût en temps de l'ordre du degré ; la transformation probabiliste ne requiert aucune mémoire supplémentaire.

Keywords: self-stabilization, central scheduler, local mutual exclusion, conflict manager, fairness

Chapter 1

Introduction

Self-stabilization [8, 9] is an elegant approach to forward recovery from transient faults as well as initializing a large-scale system. Informally, a self-stabilizing systems is able to recover from any transient fault in finite time, without restricting the nature or the span of those faults.

The historical model for presenting self-stabilizing algorithms [8] is the *central scheduler* model. This model is essentially sequential: at each step, exactly one process is activated and atomically executes a portion of its code. This model permits to write simple and elegant algorithms, and proving their correctness does not have to take into account more realistic system assumptions. Since algorithms are only able to read their neighbor state, this model is in fact equivalent to a model where exactly one process is activated at a certain time in its neighborhood (this model is also called the *locally central scheduler*).

To run algorithms written in the locally central scheduler model in more realistic systems (where processes are activated in a distributed manner), the approach so far consists in composing the initial algorithm with a (local) mutual exclusion algorithm. The (local) mutual exclusion mechanism, if self-stabilizing, eventually guarantees that all processes execute their critical sections in an atomic manner (in their neighborhood), so that if the initial algorithm is run in critical sections only, its correctness remains valid (the scheduler it relies on is simulated by the exclusion mechanism). As a result, the overhead induced by the simpler system hypothesis (locally central scheduler) is driven by the resources that are consumed by the implementation of the exclusion mechanism. There exist a large amount of self-stabilizing local mutual exclusion mechanisms [10, 2, 16, 7, 5]. In particular, [5] provides necessary and sufficient conditions to deterministically implement self-stabilizing local mutual exclusion. Fairness targeted atomicity refinement and adequate transformers are discussed in [17, 6].

In short, self-stabilizing local mutual exclusion on general networks requires, in the best case, an amount of memory per node that depends on the size of the network. Similarly, the stabilization time of such mechanisms could depend on the number of processes in the system. The reason for such high overhead is essentially that processes are guaranteed to eventually enter their critical section if they request so (that is, implementations of self-stabilizing mutual local exclusion mechanism guarantee *fairness* in addition to safety).

We argue that providing fairness to the initial algorithm is not necessary in all cases. In particular, all self-stabilizing algorithms whose complexity is evaluated in the *step* model (*i.e.* the number of atomic local steps that were performed in any computation) do not require fairness from the exclusion mechanism, but rather *progress* (*i.e.* at least one activatable process executes its code). So, for those algorithms, the overhead induced by regular exclusion mechanisms could be avoided if a different scheme is used.

Contributions. In this paper, we specify the *conflict manager* abstraction. Informally, a conflict manager provides the safety part of the local mutual exclusion specification, but not the fairness part (which is replaced by a progress property). Also, we provide both deterministic and probabilistic implementations of our abstraction, and use those implementations on the node coloring algorithm of [14] (the same scheme could also be applied to [12]) and the maximal matching algorithm of [15]. Using our tool, these algorithms, which perform in the locally central scheduler, can be run under an arbitrary distributed scheduler (where any subset of activatable processes could be selected at a given time to execute their action). The deterministic transformation requires only one extra bit of memory (zero for the probabilistic transformation), while the time overhead is in the order of the network degree.

Chapter 2

Model

Distributed Systems A distributed system is a set of state machines called processes. Each process can communicate with a subset of the processes called neighbors. We will use $\mathcal{N}_x$ to denote the set of neighbors of node x. The communication among neighboring processes is carried out using communication registers (called "shared variables" throughout this paper). The system's communication graph is drawn by representing processes as nodes and the neighborhood relationship by edges between the nodes.

We model a distributed system $\mathcal{S} = (C, T, I)$ as a *transition system* where C is the set of system configurations, T is a transition function from C to C, and I is the set of initial configurations. A *probabilistic distributed system* is a distributed system where a probabilistic distribution is defined on the transition function of the system.

The *state* of a process is defined by the values of its variables. A process may change its state by executing its local *algorithm* (defined below). A *configuration* of a distributed system is an instance of the state of its processes.

The *algorithm* executed by each process is described by a finite set of guarded actions of the form $\langle \text{guard} \rangle \longrightarrow \langle \text{statement} \rangle$. Each guard of process P_i is a boolean expression involving the variables of P_i and its neighbors. A process P_i is *enabled* in configuration c if at least one of the guards of the program of P_i is *true* in c. Let c be a configuration and CH be a subset of enabled processes in c. We denote by $\{c : CH\}$ the set of configurations that are *reachable* from c if every process in CH executes an action starting from c. A *computation step* is a tuple (c, CH, c'), where $c' \in \{c : CH\}$. Note that all configurations $\in \{c : CH\}$ are reachable from c by executing *exactly one* computation step. In a probabilistic distributed system, every computation step is associated with a probabilistic value (the sum of the probabilities of the computation steps determined by $\{c : CH\}$ is 1). A *computation* of a distributed system is a maximal sequence of computation steps. A *history* of a computation is a finite prefix of the computation. A history of length n (denoted by h_n) can be defined recursively as follows:

$$h_n \equiv \begin{cases} (c_0, CH_0, c_1) & \text{if } n = 1 \\ [h_{n-1}(c_{n-1}, CH_{n-1}, c_n)] & \text{otherwise} \end{cases}$$

If the system is probabilistic, the probabilistic value of a history is the product of the

probabilities of all the computation steps in the history. If h_n is a history such that

$$h_n = [(c_0, CH_0, c_1) \dots (c_{n-1}, CH_{n-1}, c_n)]$$

then we use the following notations: the length of the history h_n (equal to n) is denoted as $length(h_n)$, the last configuration in h_n (which is c_n) is represented by $last(h_n)$, and the first configuration in h_n (which is c_0) is referred to as $first(h_n)$ ($first$ can also be used for an infinite computation). A *computation fragment* is a finite sequence of computation steps. Let h be a history, x be a computation fragment such that $first(x) = last(h)$, and e be a computation such that $first(e) = last(h)$. Then $[hx]$ denotes a history corresponding to the computation steps in h and x, and (he) denotes a computation containing the steps in h and e.

Scheduler. A *scheduler* is a *predicate* over the system computations. In a computation, a transition (c_i, c_{i+1}) occurs due to the execution of a non-empty subset of the enabled processes in configuration c_i. In every computation step, this subset is chosen by the scheduler. We refer to the following types of schedulers in this paper: the *locally centralized scheduler* ([10, 2, 3]) ensures that in every computation step, neighboring processes are not chosen concurrently by the scheduler; the *distributed scheduler* ensures that during a computation step, any nonempty subset of the enabled processes is chosen by the scheduler. We generalize the *locally centralized scheduler* to *$\mathcal{R}$-restricted scheduler* where $\mathcal{R}$ is a conflict relation defined on the processes. During a computation step a non-empty set of enabled processes is chosen by this scheduler such that no two chosen processes verify the relation $\mathcal{R}$. The interaction between a scheduler and the distributed system generates some special structures, called *strategies*. The scheduler strategy definition is based on the tree of computations (all the computations having the same initial configuration). Let c be a system configuration and S a distributed system. The tree representing all computations in S starting from the configuration c is the tree rooted at c and is denoted as $\mathcal{T}ree(S, c)$. Let n_1 be a process in $\mathcal{T}ree(S, c)$. A *branch* originating from n_1 represents the set of all $\mathcal{T}ree(S, c)$ computations starting in n_1 with the same first transition. The degree of n_1 is the number of branches rooted in n_1.

Definition 1 (Strategy) *Let S be a distributed system, D a scheduler, and c a configuration in S. We define a strategy as the set of computations represented by the tree obtained by pruning $Tree(S, c)$ such that the degree of any process is at most* 1.

Definition 2 (Cone) *Let s be a strategy of a scheduler D. A* cone $\mathcal{C}_h(s)$ *corresponding to a history h is defined as the set of all possible computations under D which create the same history h.*

When the system is probabilistic, the probabilistic value of a cone $\mathcal{C}_h(s)$ is the probabilistic value of the history h (*i.e.*, the product of the probabilities of all computation steps in h).

Definition 3 (Subcone) *A cone $\mathcal{C}_{h'}(s)$ is called a* subcone *of $\mathcal{C}_h(s)$ if and only if $h' = [hx]$, where x is a computation fragment.*

Let S be a system, D a scheduler, and s a strategy of D. The set of computations under D that reach a configuration c' satisfying predicate P (denoted as $c' \vdash P$) is denoted as $\mathcal{EP}_s$. When the system is probabilistic, the associated probabilistic value of $\mathcal{EP}_s$ is represented by $Pr(\mathcal{EP}_s)$. We call a predicate P a *closed predicate* if the following is true: If P holds in configuration c, then P also holds in any configuration reachable from c.

2.1 Deterministic self-stabilization

In order to define self-stabilization for a distributed system, we use two types of predicates: the legitimacy predicate—defined on the system configurations and denoted by $\mathcal{L}$—and the problem specification—defined on the system computations and denoted by $\mathcal{SP}$.

Let $\mathcal{P}$ be an algorithm. The set of all computations of the algorithm $\mathcal{P}$ is denoted by $\mathcal{E}_\mathcal{P}$. Let $\mathcal{X}$ be a set and $Pred$ be a predicate defined on the set $\mathcal{X}$. The notation $x \vdash Pred$ means that the element x of $\mathcal{X}$ satisfies the predicate $Pred$ defined on the set $\mathcal{X}$.

Definition 4 (Deterministic self-stabilization) *An algorithm $\mathcal{P}$ is self-stabilizing for a specification $\mathcal{SP}$ if and only if the following two properties hold:*

1. *convergence — all computations reach a configuration that satisfies the legitimacy predicate. Formally,*
   $\forall e \in \mathcal{E}_\mathcal{P} :: e = ((c_0, c_1)(c_1, c_2)\ldots) : \exists n \geq 1, c_n \vdash \mathcal{C}$;

2. *correctness — all computations starting in configurations satisfying the legitimacy predicate satisfy the problem specification $\mathcal{SP}$. Formally, $\forall e \in \mathcal{E}_\mathcal{P} :: e = ((c_0, c_1)$ $(c_1, c_2)\ \ldots) : c_0 \vdash \mathcal{L} \Rightarrow e \vdash \mathcal{SP}$.*

2.2 Probabilistic Self-Stabilizing Systems.

In this section, we give an outline of the probabilistic model used in the rest of the paper. A detailed description of this model is available in [4].

A probabilistic self-stabilizing system is a probabilistic distributed system satisfying two important properties: *probabilistic convergence* (the probability of the system to converge to a configuration satisfying a *legitimacy predicate* is 1) and *correctness* (once the system is in a configuration satisfying a legitimacy predicate, it satisfies the system specification). In this context, the correctness comes in two variants: *weak correctness*—the system correctness is only probabilistic, and *strong correctness*—the system correctness is certain.

Definition 5 (Strong Probabilistic Stabilization) *A system S is* strong self-stabilizing *under scheduler D for a specification SP if and only if there exists a closed legitimacy predicate L such that in any strategy s of S under D, the two following conditions hold:*

(i) The probability of the set of computations under D, starting from c, reaching a configuration c', such that c' satisfies L is 1 (probabilistic convergence). (Formally, $\forall s, Pr(\mathcal{EL}_s) = 1$).
(ii) All computations, starting from a configuration c' such that c' satisfies L, satisfy SP (strong correctness).(Formally, $\forall s, \forall e, e' \in s, e = (he') :: last(h) \vdash L \Rightarrow e' \vdash SP$).

Convergence of Probabilistic Stabilizing Systems. We borrow a result of [4] to prove the probabilistic convergence of the algorithms presented in this paper. This result is built upon previous work on probabilistic automata ([18, 19, 20, 21]) and provides a complete framework for the verification of self-stabilizing probabilistic algorithms. We need to introduce a few terms before we are ready to present this result. First, we explain a key property, called *local convergence* and denoted by LC. Informally, the LC property characterizes a probabilistic self-stabilizing system in the following way: The system reaches a configuration which satisfies a particular predicate, in a bounded number of computation steps with a positive probability.

Definition 6 (Local Convergence) *Let s be a strategy, and P_1 and P_2 be two predicates defined on the system configurations, where P_1 is a closed predicate. Let δ be a positive number $\in]0,1[$ and N a positive integer. Let $\mathcal{C}_h(s)$ be a cone with $last(h) \vdash P_1$ and let M denote the set of subcones $\mathcal{C}_{h'}(s)$ of $\mathcal{C}_h(s)$ such that $last(h') \vdash P_2$ and $length(h') - length(h) \leq N$. Then $\mathcal{C}_h(s)$ satisfies the local convergence property denoted as $LC(P_1, P_2, \delta, N)$ if and only if $Pr(\bigcup_{\mathcal{C}_{h'}(s) \in M} \mathcal{C}_{h'}(s)) \geq \delta$.*

Now, if in strategy s, there exist $\delta_s > 0$ and $N_s \geq 1$ such that any cone $\mathcal{C}_h(s)$ with $last(h) \vdash P_1$ satisfies $LC(P_1, P_2, \delta_s, N_s)$, then the result of [4] states that the probability of the set of computations under D reaching configurations satisfying $P_1 \wedge P_2$ is 1. Formally:

Theorem 1 ([4]) *Let s be a strategy. Let P_1 and P_2 be two closed predicates on configurations such that $Pr(\mathcal{EP}1_s) = 1$. If $\exists \delta_s > 0$ and $\exists N_s \geq 1$ such that any cone $\mathcal{C}_h(s)$ with $last(h) \vdash P_1$ satisfies $LC(P_1, P_2, \delta_s, N_s)$, then $Pr(\mathcal{EP}12_s) = 1$, where $P_{12} = P_1 \wedge P_2$.*

Chapter 3

Self-stabilizing Conflict Manager

A *conflict manager* is an abstraction for a distributed oracle that is queried by processes wanting to execute possibly conflicting actions. A conflicting action could consist in accessing a non-sharable resource or executing a critical section of code (*i.e.* code that cannot be executed by nodes in the same physical or logical neighborhood). When queried, the conflict manager gives execution permission to exactly one of the queriers. Conflict managers are extensively used in distributed computing in general, and in the self-stabilizing setting in particular. A locally centralized scheduler for example is a conflict manager in a neighborhood while a centralized scheduler is a conflict manager when all the processes in the system have to execute conflicting actions or a critical section. In this paper, we propose a formal specification for the conflict manager abstraction that unifies these schedulers. Additionally we provide both probabilistic and deterministic implementations.

Let $\mathcal{R}$ be the conflict relation defined over $V \times V$ defining the conflicts in the system (V is the set of processes in the system[1]). For example, if the semantics of the conflict is related to un-sharable resources in a system then two processes willing to simultaneously use this kind of resource are in $\mathcal{R}$.

Definition 7 (Deterministic Conflict Manager) *Let $\mathcal{R}$ be a conflict relation over the set of processes of a system. A* deterministic conflict manager *restricted to $\mathcal{R}$ is defined by the following two properties:*

- **safety** *no two processes, p, q such that $p\ \mathcal{R}\ q$, execute their actions simultaneously.*
- **progress** *if in a configuration there are some enabled processes, $S = \{p_1, \ldots p_k\}$ such that $\forall (p_i, p_j), p_i\ \mathcal{R}\ p_j$, then at least one process in S executes its actions in a finite number of steps.*

Obviously, the main difference between the conflict manager abstraction and the local mutual exclusion or mutual exclusion problem is the lack of fairness requirement. That is, the conflict manager is not required to fairly grant access to critical section to all

[1]Note that the knowledge of V is not mandatory

queriers. Hence, it does not solve the starvation problem traditionally solved by the (local) mutual exclusion. Note that there is no deterministic implementation of conflict managers in anonymous systems (this is a direct consequence of Angluin [1] impossibility results), therefore probabilistic conflict managers are used whenever the anonymity of the network has to be preserved.

Definition 8 (Probabilistic Conflict Manager) *Let $\mathcal{R}$ be a conflict relation over the set of processes of a system. A probabilistic conflict manager is defined by the following two properties:*

- **probabilistic safety** *no two processes, p, q such that $p\ \mathcal{R}\ q$, execute their actions simultaneously with positive probability.*
- **probabilistic progress** *if in a configuration there are some enabled processes, $S = \{p_1, \ldots p_k\}$ such that $\forall (p_i, p_j), p_i\ \mathcal{R}\ p_j$, then at least one of the enabled processes eventually executes its actions with high probability.*

A *local conflict manager* is a conflict manager restricted to the local neighborhood. That is, two processes p, q are in local conflict iff $p \in \mathcal{N}_q$ and $q \in \mathcal{N}_p$. A local conflict manager has only a local cover (*i.e.* it solves the conflicts only in a local neighborhood). In the following we propose deterministic and probabilistic memory optimal and distributed implementations of conflict managers, along with an application for automatic transformation of self-stabilizing algorithms.

3.1 Self-stabilizing Deterministic Conflict Managers

In this section we propose a deterministic implementation of a conflict manager, presented as Algorithm 3.1.1. Algorithm 3.1.1 accepts as inputs the initial algorithm (using guarded commands) and the conflict relation $\mathcal{R}$. Additionally, the algorithm needs the specification of a total order relation, $\prec$, defined on the processes in $\mathcal{R}$. The algorithm uses a shared variable *want_to_act* in order to capture the state of the process guards. The algorithm idea is very simple. A process i executes its actions if it has one of the guards of the original algorithm enabled and is maximal in the set of conflicting processes (*i.e.* processes in relation $\mathcal{R}$ with i and that have their *want_to_act* variables set to *true*) [2].

In the following we prove that Algorithm 3.1.1 is a self-stabilizing implementation of a conflict manager as specified by Definition 7.

Lemma 1 (Safety) *In any configuration, no two processes in $\mathcal{R}$ execute their actions simultaneously.*

[2]For the local conflict manager, the maximal has to be computed over the set of neighboring processes.

Algorithm 3.1.1 Self-stabilizing Deterministic Conflict Manager

Input Conflict Relation:
$\mathcal{R}$: conflict relation over processes;
Input Order Relation:
$\prec$: total order relation over processes in $\mathcal{R}$;
Input Guards:
$guard_{\mathcal{A}}$: boolean;
Input Actions(Code):
$actions_{\mathcal{A}}$: actions;
Variables:
$want\_to\_act_i$: boolean;
Predicates:
$Allowed\_To\_Act(i)$: true iff i is maximal in $\mathcal{R}$ with respect to $\prec$, formally ($\forall j,\ i\ \mathcal{R}\ j$, s.t. $want\_to\_act_j = want\_to\_act_i = true$, $j \prec i$
Actions:
$R_1 :: \langle want\_to\_act_i = \vee guard_{\mathcal{A}} \wedge Allowed\_To\_Act(i) \rangle \rightarrow actions_{\mathcal{A}}$
$actions_{\mathcal{A}}$ corresponds to the only guard true in $\mathcal{A}$

$R_2 :: \langle want\_to\_act_i \neq \vee guard_{\mathcal{A}} \rangle \rightarrow want\_to\_act_i = \vee guard_{\mathcal{A}}$

Proof: Assume that two processes $p_i\ \mathcal{R}\ p_j$ execute their actions simultaneously. This implies that $want\_to\_act_{p_i} = want\_to\_act_{p_j} = true$, and that simultaneously $p_i \prec p_j$ and $p_j \prec p_i$,which is impossible since $\prec$ is a total order on the processes in $\mathcal{R}$. □

Lemma 2 (Progress) *If in a configuration there are some enabled processes in $\mathcal{R}$ then at least one process executes its actions in a finite number of steps.*

Proof: Assume some processes in $\mathcal{R}$ are enabled yet none of them executes its actions. Let p be such process. There are two possible cases: the *want_to_act* variable of p is not up to date or p is not allowed to execute ($Allow\_To\_Act$) predicate is *false*. In the former case the process, after the execution of rule R_2, p updates the variable *want_to_act*. In the latter case, let S be the set of processes that have *want_to_act* set to true and greater than p with respect to $\prec$ relation. S has a maximum since the set of processes is finite. The maximum in S verifies $Allowed\_To\_Act$ predicate and consequently is allowed to execute rule R_1 if at least one of its $\mathcal{A}$ guards is enabled. If the maximum executes R_1, the system verifies the progress property.

However, it is possible that the maximum in S has a wrong value of *want_to_act* and none of its $\mathcal{A}$ guards enabled. After the execution of R_2 the maximum resets its *want_to_act* variable and the size of the set S is decreased by 1. Following the same reasoning, in a finite number of steps a process greater than p with respect to $\prec$ relation executes its $\mathcal{A}$ actions or, once S is empty, p itself executes its $\mathcal{A}$ actions. In the worst case, until at least one process executes $\mathcal{A}$ actions the system takes at least Δ steps where Δ is the maximal size of S. □

Theorem 2 *Algorithm 3.1.1 is a deterministic self-stabilizing implementation of a conflict manager under an unfair scheduler.*

Proof: The proof trivially follows from Lemmas 1 and 2. Note that the safety property is always guaranteed. The progress property is guaranteed only after the neighbors *want_to_act* variables are checked. □

In the following we propose an application of the Conflict Manager to reinforce self-stabilizing algorithms. That is, algorithms that are self-stabilizing under $\mathcal{R}$-restricted schedulers (where $\mathcal{R}$ is the conflict relation) are transformed into self-stabilizing algorithms that perform under a distributed scheduler. The following theorem proves that using the hierarchical composition [11] with Algorithm 3.1.1 any algorithm self-stabilizing under a $\mathcal{R}$-restricted scheduler transforms into an algorithm self-stabilizing under a distributed scheduler.

Theorem 3 *Let $\mathcal{R}$ be a conflict relation over the set of processes of a system. Let $\mathcal{A}$ be a self-stabilizing algorithm for a given specification $\mathcal{S}$ under a $\mathcal{R}$-restricted scheduler. Let $\mathcal{AT}$ be the hierarchical composition of $\mathcal{A}$ with Algorithm 3.1.1. Algorithm $\mathcal{AT}$ is self-stabilizing under an unfair distributed scheduler.*

Proof: Let e be an execution of the transformed algorithm under distributed scheduler. Let e_p be the projection of e on the variables and actions of Algorithm $\mathcal{A}$. Following Lemmas 1 and 2, e_p converges to an execution of $\mathcal{A}$ under a $\mathcal{R}$-restricted scheduler. That is, e_p has an infinite suffix such that no two processes in $\mathcal{R}$ execute their actions simultaneously. Since $\mathcal{A}$ is self-stabilizing under the $\mathcal{R}$-restricted scheduler then e_p converges to a legitimate configuration for $\mathcal{A}$ and subsequently verifies specification $\mathcal{S}$. e_p is the projection of e on the variables and actions of Algorithm $\mathcal{A}$. The additional variables and actions of the transformation do not interfere with the variables, guards and actions of $\mathcal{A}$. Consequently, e converges to a legitimate configuration for A and verifies the specification $\mathcal{S}$. □

Lemma 3 *Let $O(f(n))$ be the stabilization time (in number of steps) of an Algorithm $\mathcal{A}$ under a $\mathcal{R}$-restricted scheduler. The stabilization time of the transformed algorithm, Algorithm $\mathcal{AT}$, is $O(f(n) \times \Delta)$ steps where Δ is the maximum size of $\mathcal{R}$.*

Proof: The stabilization time of Algorithm $\mathcal{AT}$ depends on the stabilization time of $\mathcal{A}$ and the stabilization time of the conflict manager. Following Lemma 2 the stabilization time of $\mathcal{AT}$ is $O(f(n) \times \Delta)$ steps. □

Since the transformation requires that a total order is defined over conflicting processes, the system may not be both uniform and anonymous. In the following we propose a transformation that preserves the anonymity of the network, however the specification of the algorithm is verified only with high probability. The proposed transformation uses a probabilistic implementation of a conflict manager (see Definition 8).

3.2 Self-stabilizing Probabilistic Conflict Manager

In this section we propose a *probabilistic conflict manager* that takes as input the original algorithm (as guarded commands) $\mathcal{A}$. The transformer general scheme is very simple: a process executes the actions of $\mathcal{A}$ if it is enabled and the toss of a random coin return true.

Algorithm 3.2.1 Self-stabilizing Probabilistic Local Conflict Solver

Input Guards:
$guard_{\mathcal{A}}$: boolean
Input Actions(Code):
$actions_{\mathcal{A}}$: actions
Predicates:
$Allowed\_To\_Act(i)$: true iff random(0,1)=1
Actions:
$R_1 :: \langle \vee guard_{\mathcal{A}} \wedge Allowed\_To\_Act(i) \rangle \rightarrow critical\_actions_i$
$actions_{\mathcal{A}}$ corresponds to the only guard true in $\mathcal{A}$

We now prove that Algorithm 3.2.1 is a self-stabilizing implementation of a probabilistic conflict manager as specified by Definition 8. In the following, critical actions refer to the input actions of Algorithm 3.2.1.

Lemma 4 (Probabilistic Safety) *Let $\mathcal{R}$ be a conflict relation over the set of processes of a system. In any configuration, no two processes in $\mathcal{R}$ execute their critical actions simultaneously with positive probability.*

Proof: Let S be the set of processes with $\mathcal{A}$ guards enabled such that $\forall i, j \in S, i\mathcal{R}j$. The probability that exactly one of the processes executes the critical actions is $|S| \times [\frac{1}{2} \times (\frac{1}{2})^{(|S|-1)}] = |S| \times (\frac{1}{2})^{|S|}$. □

Lemma 5 (Probabilistic Progress) *If in a configuration there are some enabled processes then at least one process executes its critical section with high probability in a finite number of steps.*

Proof: Let S be the set of processes with at least one guard of $\mathcal{A}$ enabled. The probability that no process executes $\mathcal{A}$ actions at the first trial is $(\frac{1}{2})^{|S|}$. The probability no process executes its actions after the xth trial is $(\frac{1}{2})^{x \times |S|}$. The probability that none of the processes executes its actions is $lim_{x \rightarrow \infty}(\frac{1}{2})^{x \times |S|} = 0$. □

Theorem 4 *Algorithm 3.2.1 is a self-stabilizing implementation of a probabilistic conflict manager (see Definition 8).*

Proof: The proof follows from Lemmas 4 and 5. □

In the following we show that given an Algorithm $\mathcal{A}$, that is self-stabilizing for a specification $\mathcal{S}$ under a $\mathcal{R}$-restricted scheduler, the hierarchical composition of $\mathcal{A}$ with Algorithm

3.2.1 is a self-stabilizing algorithm under an unfair distributed scheduler that verifies Specification $\mathcal{S}$ with high probability. In the following $\mathcal{A}TP$ denotes the hierarchical composition of $\mathcal{A}$ with Algorithm 3.2.1.

Lemma 6 *Let e be an execution of $\mathcal{A}$ under a scheduler $\mathcal{D}$. There exists, with positive probability, an execution e_t of $\mathcal{A}TP$ under the unfair scheduler such that e_t emulates e.*

Proof: In the following we show that each transition in e can be emulated by $\mathcal{A}TP$ with some positive probability. In the following e_t is the execution emulating e. Let c_0 be the first configuration in e and let $p_{i_1}, \ldots, p_{i_k}$ be the set of processes allowed by the scheduler to execute their actions at c. Let c_1 be the configuration obtained after the execution of $p_{i_1}, \ldots, p_{i_k}$. Following Lemmas 4 and 5, in e_t we can construct a transition with positive probability c_0, c_1. That is, in this transition only processes $S = \{p_{i_1}, \ldots, p_{i_k}\}$ execute their action. Let S' be the set of processes having their $\mathcal{A}$ guards true in c. Since $\mathcal{A}TP$ is bound to unfair scheduling all these processes may be chosen by the scheduler. The probability of choosing only the subset S and consequently the probability of the transition c_0, c_1 is $(\frac{1}{2})^{|S|} \times (\frac{1}{2})^{|S' \setminus S|}$. Following the same scenario, each transition in e can be emulated with high probability in e_t. The probability of emulating e is the the product of the probability of each emulated transition of e. □

Theorem 5 *Let $\mathcal{A}$ be an algorithm that is self-stabilizing for the specification $\mathcal{S}$ under a $\mathcal{R}$-restricted scheduler. Let the convergence time of $\mathcal{A}$ be finite (in number of steps). Let $\mathcal{A}TP$ be the hierarchical composition of $\mathcal{A}$ with Algorithm 3.2.1. $\mathcal{A}TP$ is probabilistically self-stabilizing for $\mathcal{S}$.*

Proof: In the following we prove that the set of executions of $\mathcal{A}TP$ that converges to a legitimate configuration of $\mathcal{A}$ and that verifies the specification $\mathcal{S}$ has probability 1. Let c be an arbitrary initial configuration of $\mathcal{A}TP$. Following Lemma 6 there exists with positive probability α a finite execution of $\mathcal{A}TP$ that emulates an execution of $\mathcal{A}$ under a $\mathcal{R}$-restricted scheduler. $\mathcal{A}$ converges towards a legitimate configuration, consequently the emulated execution converges towards a legitimate configuration in a finite number of steps. Following Theorem 1, $\mathcal{A}TP$ is probabilistically self-stabilizing for $\mathcal{S}$. □

Chapter 4

Case Studies

In this section we transform the algorithms proposed and proved correct in [14] and [15] under a locally centralized scheduler into algorithms that are self-stabilizing under an unfair distributed scheduler. Additionally we propose the anonymous probabilistic counterpart of those algorithms.

4.1 Vertex Coloring

In the algorithm of [14], each process maintains a *color*, whose domain is the set $\{0, \ldots, \delta\}$, where δ is the node's degree. The neighborhood agreement of a particular process p is defined as follows:

Definition 9 (Agreement) *A process p* agrees *with its neighborhood if the two following conditions are verified:*

1. *p's color is different from any of p's neighbors,*
2. *p's color is minimal within the set $\{0, ..., \delta\} \setminus \cup_{j \in N_i}(R_j)$.*

A system is in a legitimate configuration if each process agrees with its neighborhood.

When any of these two conditions is falsified, p performs the following two actions: *(i)* p removes colors used by its neighbors from the set $\{0, \ldots, \delta\}$ and *(ii)* takes the minimum color of the resulting set as its new color. The resulting set is always non-empty. Core of the algorithm is presented in Algorithm 4.1.1.

Theorem 6 ([14]) *The stabilization time of Algorithm 4.1.1 is $O(\Delta \times n)$ steps, where Δ is the maximal degree of the network.*

The deterministic transformation is presented in the Appendix. The randomized variant of Algorithm 4.1.1 is now presented. This algorithm works on anonymous networks and

Algorithm 4.1.1 Self-stabilizing Deterministic Coloring Algorithm

Shared Variable:
R_i: integer $\in \{0, \ldots, \delta\}$;
Function:
$Agree(i) : R_i = min\left(\{0, \ldots, \delta\} \setminus \bigcup_{j \in \mathcal{N}_i} \{R_j\}\right)$
Actions:
$\mathcal{C} : \neg Agree(i) \longrightarrow R_i := min\left(\{0, \ldots, \delta\} \setminus \bigcup_{j \in \mathcal{N}_i} \{R_j\}\right)$

Algorithm 4.1.2 Self-stabilizing Randomized Coloring Algorithm

Shared Variable:
$\forall j \in \mathcal{N}_i, R_j$: integer $\in \{0, \ldots, \delta\}$;
Function:
$Agree(i) : R_i = min\left(\{0, \ldots, \delta\} \setminus \bigcup_{j \in \mathcal{N}_i} \{R_j\}\right)$
Actions:
$\mathcal{C} : \neg Agree(i) \wedge random(0, 1) = 1 \longrightarrow$
$R_i := min\left(\{0, \ldots, \delta\} \setminus \bigcup_{j \in \mathcal{N}_i} \{R_j\}\right)$

stabilizes with an unfair distributed scheduler. The algorithm is obtained *via* hierarchical composition between Algorithm 4.1.1 and Algorithm 3.2.1.

Compared to Algorithm 4.1.1, a process which does not agree with one of its neighbors tosses a coin before changing its color. Even if neighboring processes would compete for executing their action, by randomization there exists a positive probability that only one of those processes executes. The correctness of Algorithm 4.1.2 directly follows from the correctness of the probabilistic conflict manager.

Lemma 7 *Algorithm 4.1.2 probabilistically self-stabilize for the coloring specification (Definition 9).*

Proof: Algorithm 4.1.2 is the hierarchical composition between Algorithm 4.1.1 and Algorithm 3.2.1. Algorithm 4.1.1 is self-stabilizing for the coloring specification under a locally centralized scheduler. Following Theorem 5, Algorithm 4.1.2 is probabilistically self-stabilizing for the coloring specification under a distributed scheduler. □

Lemma 8 *The average number of computations steps to reach a configuration c where all processes agree with their neighbors is $O(\Delta n \log_2 n)$ where Δ is the maximal degree of the network.*

Proof: Let A be the set of processes which agree with their neighbors (see Definition 9). Let R be a round during which each process in the system executes Δ trials. The probability for process i to move to A after a round is

$$p_i \geq \left(\frac{1}{2}\right)^{\Delta} \times \left(\frac{1}{2}\right)^{\Delta^2}$$

Therefore, for n-sized networks, the average number of processes in A after a round is at least $n \times \left(\frac{1}{2}\right)^{\Delta+\Delta^2}$. This also means that at most $n \times \left(1 - \left(\frac{1}{2}\right)^{(\Delta+\Delta^2)}\right)$ processes are not in A.

After x rounds, the average number of processes in A is at least $n \times \left(1 - \left(\frac{1}{2}\right)^{(\Delta+\Delta^2)}\right)^x$. The algorithm would stop when all processes agree. Then x is a solution of the following equation

$$\begin{aligned} \left[n \times \left(1 - \left(\frac{1}{2}\right)^{(\Delta+\Delta^2)}\right)^x = 1\right] &\Rightarrow \left[x = \log_{\frac{1}{1-\frac{1}{2}^{(\Delta+\Delta^2)}}} n\right] \\ &\Rightarrow \left[x = \frac{\log_2 n}{\log_2 \frac{1}{1-\frac{1}{2}^{(\Delta+\Delta^2)}}}\right] \\ &\Rightarrow x = O(\log_2 n) \end{aligned}$$

Therefore, on average, all processes agree with their neighbors within $O(\Delta n \log n)$ computation steps. □

4.2 Maximal Matching

We first recall the maximal matching algorithm of [15]. The algorithm presented in this section requires a locally centralized scheduler. Given an undirected graph, $G = (V, E)$, a *matching* is defined to be a subset of pairwise disjoint edges. A matching is *maximal* if it is not included in another matching.

Each node i maintains a single pointer variable which is either null (denoted in the following $\perp$) or points to one of its neighbors j, denoted in the following $i \rightarrow j$. A node i is *matched* if and only if i and one of its neighbors j verifies the following relation $i \rightarrow j \wedge j \rightarrow i$.

The matching algorithm proposed in [15] works as follows. When a node is unmatched (it points to null) and one of its neighbors proposed it a matching then the node accepts the proposal ($\mathcal{R}_1$). When a node is unmatched and none of its neighbors proposed it a matching then the node propose a matching to one of its unmatched neighbors ($\mathcal{R}_2$). When a node proposed a matching to another node which proposed a matching to a different node then the initial node withdraws its proposal ($\mathcal{R}_3$).

Theorem 7 *[15] Algorithm 4.2.1 is self-stabilizing for the maximal matching specification. The stabilization time of Algorithm 4.2.1 is $2m + n$ steps where m is the number of edges in the network.*

A transformation of [15] under an asynchronous scheduler was also proposed in [13]. However, this transformation is not generic. It uses a probabilistic naming underlying module, the memory additional cost is n^2 and the transformation cannot allow to easily derive the time complexity of the transformed algorithm.

Algorithm 4.2.1 Self-stabilizing Deterministic Maximal Matching under a locally centralized scheduler

Predicates:

$Want\_To\_Engage(i,j):\ true$ iff $i \to \perp \wedge \exists j \in \mathcal{N}(i), j \to i$

$Want\_To\_Propose(i,j):\ true$ iff $i \to \perp \wedge\ \nexists j \in \mathcal{N}(i), j \to i \wedge \exists j \in \mathcal{N}(i), j \to \perp$

$Want\_To\_Desengage(i):\ true$ iff $i \to j \wedge j \to k, k \neq i \neq \perp$

Actions:

$\mathcal{R}_1 : Want\_To\_Engage(i,j) \longrightarrow i \to j$

$\mathcal{R}_2 : Want\_To\_Propose(i,j) \longrightarrow i \to j$

$\mathcal{R}_3 : Want\_To\_Desengage(i) \longrightarrow i \to \perp$

We transform Algorithm 4.2.1 such that it stabilizes in spite of any unfair distributed scheduler. Using the technique proposed in Chapter 3 we hierarchically compose Algorithm 4.2.1 with Algorithm 3.1.1 where the $\mathcal{R}$ relation is instantiated to the local neighborhood and the $\prec$ relation is instantiated to the $<$ relation defined on the identifiers of the neighboring nodes. The transformation result is presented as Algorithm 4.2.2.

Algorithm 4.2.2 Self-stabilizing Deterministic Maximal Matching Algorithm

Constants:

id_i: integer identifier of the process (chromatic)

Variables:

$want\_to\_act_i$: boolean;

Predicates:

$Allowed\_To\_Act(i)$: true iff i has the maximal id in its neighborhood restricted to the neighbors that want to execute their action, formally ($\forall j \in N_i$ s.t. $want\_to\_act_j = want\_to\_act_i = true$, $id_j < id_i$

$Want\_To\_Engage(i,j):\ true$ iff $i \to \perp \wedge \exists j \in \mathcal{N}(i), j \to i$

$Want\_To\_Propose(i,j):\ true$ iff $i \to \perp \wedge\ \nexists j \in \mathcal{N}(i), j \to i \wedge \exists j \in \mathcal{N}(i), j \to \perp$

$Want\_To\_Desengage(i):\ true$ iff $i \to j \wedge j \to k, k \neq i \neq \perp$

Actions:

$R_0 :: \langle want\_to\_act_i = Want\_To\_Engage(i,j) \wedge Allowed\_To\_Act(i) \rangle \to\ i \to j$

$R_1 :: \langle want\_to\_act_i = Want\_To\_Propose(i,j) \wedge Allowed\_To\_Act(i) \rangle \to\ i \to j$

$R_2 :: \langle want\_to\_act_i = Want\_To\_Desengage(i) \wedge Allowed\_To\_Act(i) \rangle \to\ i \to \perp$

$R_1 :: \langle want\_to\_act_i \neq Want\_To\_Engage \vee Want\_To\_Propose \vee Want\_To\_Desengage \rangle \to$ $want\_to\_act_i := Want\_To\_Engage \vee Want\_To\_Propose \vee Want\_To\_Desengage$

Lemma 9 *Algorithm 4.2.2 is self-stabilizing under an unfair distributed scheduler for the maximal matching specification.*

Proof: The proof is a consequence of Theorem 3. □

Lemma 10 *The stabilization time of the transformed algorithm is $O(\Delta \times m)$ steps.*

Proof: The proof follows from Lemma 3 and Theorem 7. □

Chapter 5

Conclusions

In this paper we proposed the specification and both probabilistic and deterministic self-stabilizing implementations of a new abstraction, the *Conflict Manager*. The conflict manager generalizes various kinds of schedulers used in the self-stabilizing literature. Additionally we presented as an application of our abstraction two transformers, that transform algorithms written for essentially sequential schedulers into algorithms that can perform under the general unfair distributed scheduler. The transformation cost is only one memory bit for the deterministic algorithms and zero memory bit for the probabilistic ones. The time complexity is order of the maximal degree of the network for the deterministic transformer. We demonstrate the effectiveness with two case studies: the coloring algorithm of [14] and the maximal matching algorithm of [15]. Of course, any self-stabilizing algorithm that has bounded step complexity can be transformed using our approach. There remains the interesting open question of the necessity of the step complexity boundedness for our scheme to permit transformation under the unfair distributed scheduler.

Bibliography

[1] D. Angluin. Local and global properties in networks of processors. In *Proc. Symp. on Theory of Computing(STOC'80)*, pages 82–93, 1980.

[2] A. Arora and M. Nesterenko. Stabilization-preserving atomicity refinement. *DISC'99*, pages 254–268, 1999.

[3] J. Beauquier, A. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. In *Proceedings of DISC'2000*, October 2000.

[4] J Beauquier, M Gradinariu, and C Johnen. Crossover composition. In *Proceedings of the Fifth Workshop on Self-stabilizing Systems (WSS 2001)*, pages 19–34, 2001.

[5] Christian Boulinier, Franck Petit, and Vincent Villain. When graph theory helps self-stabilization. In *PODC*, pages 150–159, 2004.

[6] Sébastien Cantarell, Ajoy Kumar Datta, and Franck Petit. Self-stabilizing atomicity refinement allowing neighborhood concurrency. In *Self-Stabilizing Systems, 6th International Symposium, SSS 2003, San Francisco, CA, USA, June 24-25, 2003, Proceedings*, volume 2704 of *Lecture Notes in Computer Science*, pages 102–112. Springer, 2003.

[7] Alain Cournier, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Enabling snap-stabilization. In *ICDCS*, pages 12–19, 2003.

[8] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[9] S. Dolev. *Self-stabilization*. MIT Press, March 2000.

[10] M. Gouda and F. Hadix. The alternator. *In Proceedings of the Third Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems)*, pages 48–53, 1999.

[11] Mohamed G. Gouda and Ted Herman. Adaptive programming. *IEEE Trans. Software Eng.*, 17(9):911–921, 1991.

[12] M. Gradinariu and S. Tixeuil. Self-stabilizing vertex coloring of arbitrary graphs. In *Proceedings of OPODIS 2000, STUDIA INFORMATICA*, pages 55–70, 2000.

[13] Maria Gradinariu and Colette Johnen. Self-stabilizing neighborhood unique naming under unfair scheduler. In *Euro-Par*, pages 458–465, 2001.

[14] Stephen T. Hedetniemi, David Pokrass Jacobs, and Pradip K. Srimani. Linear time self-stabilizing colorings. *Inf. Process. Lett.*, 87(5):251–255, 2003.

[15] Su-Chu Hsu and Shing-Tsaan Huang. A self-stabilizing algorithm for maximal matching. *Inf. Process. Lett.*, 43(2):77–81, 1992.

[16] Hirotsugu Kakugawa and Masafumi Yamashita. Self-stabilizing local mutual exclusion on networks in which process identifiers are not distinct. In *SRDS*, pages 202–211, 2002.

[17] Mehmet Hakan Karaata. Self-stabilizing strong fairness under weak fairness. *IEEE Trans. Parallel Distrib. Syst.*, 12(4):337–345, 2001.

[18] A. Pogosyants, R. Segala, and N. Lynch. Verification of the randomized consensus algorithm of Aspen and Herlihy: a case study. *Distributed Computing*, 13(4):155–186, 2000.

[19] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, MIT, Departament of Electrical Engineering and Computer Science, 1995.

[20] R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. In Springer-Verlag, editor, *Proceedings of the Fifth International Conference on Concurrency Theory (CONCUR'94) LNCS:836*, Uppsala, Sweden, August 1994.

[21] S. H. Wu, S. A. Smolka, and E. W. Stark. Composition and behaviors of probabilistic i/o automata. In *Proceedings of the Fifth International Conference on Concurrency Theory (CONCUR'94) LNCS:836*, pages 513–528, 994.

Appendix A

Deterministic Coloring under Distributed Scheduler

We now transform Algorithm 4.1.1 such that it stabilizes in spite of any unfair distributed scheduler. Using the technique we proposed in Chapter 3, we hierarchically compose Algorithm 4.1.1 with Algorithm 3.1.1 where the $\mathcal{R}$ relation is instantiated to the local neighborhood and the $\prec$ relation is instantiated to the $<$ relation defined on the identifiers of the neighboring nodes. The transformation result is presented as Algorithm A.0.3.

Algorithm A.0.3 Self-stabilizing Deterministic Coloring Algorithm

Shared Variable:
R_i: integer $\in \{0, \ldots, \delta\}$;
Constants:
id_i: integer identifier of the process (chromatic)
Variables:
$want\_to\_act_i$: boolean;
Predicates:
$Allowed\_To\_Act(i)$: true iff i has the maximal id in its neighborhood restricted to the neighbors that want to execute their action, formally ($\forall j \in N_i$ s.t. $want\_to\_act_j = want\_to\_act_i = true$, $id_j < id_i$
Function:
$Agree(i) : R_i = min\left(\{0, \ldots, \delta\} \setminus \bigcup_{j \in \mathcal{N}_i} \{R_j\}\right)$
Actions:
$R_0 :: \langle want\_to\_act_i = \neg Agree(i) \wedge Allowed\_To\_Act(i)\rangle \rightarrow \ R_i := min\left(\{0, \ldots, \delta\} \setminus \bigcup_{j \in \mathcal{N}_i} \{R_j\}\right)$
$R_1 :: \langle want\_to\_act_i \neq \neg Agree(i)\rangle \rightarrow want\_to\_act_i := \neg Agree(i)$

Lemma 11 *Any computation of Algorithm A.0.3 under a distributed scheduler eventually achieves a legitimate configuration.*

Proof: The proof is a consequence of Theorem 3. □

Lemma 12 *The stabilization time of the transformed algorithm is $O(\Delta^2 \times n)$ steps, where Δ is the maximal degree of the network.*

Proof: The proof trivially follows from Lemma 3 and Theorem 6. □

Appendix B

Probabilistic Maximal Matching with Distributed Scheduler

In this section we present the randomized variant of Algorithm 4.2.1. This algorithm works on anonymous networks and stabilizes with an unfair scheduler. The algorithm is obtained *via* the hierarchical composition between Algorithm 4.2.1 with Algorithm 3.2.1.

Algorithm B.0.4 Self-stabilizing Randomized Maximal Matching Algorithm

Predicates:

$Want\_To\_Engage(i,j) :\ true$ iff $i \to \perp \wedge \exists j \in \mathcal{N}(i), j \to i$

$Want\_To\_Propose(i,j) :\ true$ iff $i \to \perp \wedge\ \nexists j \in \mathcal{N}(i), j \to i \wedge \exists j \in \mathcal{N}(i), j \to \perp$

$Want\_To\_Desengage(i) :\ true$ iff $i \to j \wedge j \to k, k \neq i \neq \perp$

Actions:

$\mathcal{R}_1 : Want\_To\_Engage(i,j) \wedge random(0,1) = 1 \longrightarrow i \to j$

$\mathcal{R}_2 : Want\_To\_Propose(i,j) \wedge random(0,1) = 1 \longrightarrow i \to j$

$\mathcal{R}_3 : Want\_To\_Desengage(i) \wedge random(0,1) = 1 \longrightarrow i \to \perp$

Compared to Algorithm 4.2.1, an enabled process tosses a coin before changing its status. Even if neighboring processes would compete for executing their action, by randomization there exists a positive probability that only one of those processes executes.

The correctness of Algorithm B.0.4 directly follows from the correctness of the probabilistic conflict manager.

Lemma 13 *Algorithm B.0.4 probabilistically self-stabilizes for the maximal matching specification.*

Proof: Algorithm B.0.4 is the hierarchical composition between Algorithm 4.2.1 and Algorithm 3.2.1. Algorithm 4.2.1 is self-stabilizing for the maximal matching specification under a locally centralized scheduler. Following Theorem 5, Algorithm B.0.4 is probabilistically self-stabilizing for the coloring specification under a distributed scheduler. □