Self-stabilizing Mutual Exclusion with Arbitrary Scheduler

Ajoy K. Datta[†] Maria Gradinariu[*] Sébastien Tixeuil[*]

[†] Department of Computer Science, University of Nevada Las Vegas (datta@cs.unlv.edu)
[*] Laboratoire de Recherche en Informatique, Université de Paris Sud, France ({mariag,tixeuil}@lri.fr)

Abstract

All existing uniform probabilistic self-stabilizing mutual exclusion algorithms designed to work under an unfair distributed scheduler suffer from the following common drawback: Once stabilized, there exists no upper bound of time between two executions of the critical section at a given processor. We present the first algorithm that guarantees such a bound ($O(n^3)$, where n is the network size) while working using an unfair distributed scheduler. Our algorithm works in an anonymous unidirectional ring of any size and has a polynomial expected stabilization time.

Keywords: Distributed algorithm, self-stabilization, mutual exclusion.

1 Introduction

Self-Stabilizing Mutual Exclusion. A self-stabilizing [5] algorithm, regardless of the initial system state, converges to one of the desirable states in finite time. A distributed system solving the mutual exclusion problem must guarantee the following two properties: *(i) Mutual Exclusion*: exactly one processor is allowed to execute its critical section at any time; and *(ii) Fairness*: Every processor must be able to execute its critical section infinitely often.

Related Work. Dijkstra's three self-stabilizing mutual exclusion algorithms [5] are deterministic and *non-uniform* (in such an algorithm, some processors are distinguished in the sense that they are allowed to execute a program that is different from that of the other processors). In [3], Burns and Pachl presented a deterministic algorithm for uniform unidirectional rings of prime size, and proved that no deterministic solution exists for rings of composite size.

Several papers investigated the mutual exclusion problem in the probabilistic (or randomized) setting. Randomization was used to reduce the space in [7, 9], and to cope with the case of anonymous networks in [1, 8]. However, a common problem to all these probabilistic algorithms is that once stabilized, there is no upper bound on the time between two entries into the critical section at a particular processor. In other words, although the expected time between two critical section executions is bounded, there exist computations in which a particular processor may not get the token infinitely often. We refer to this kind of algorithms as *weak probabilistic stabilizing* algorithms. Kakugawa and Yamashita [11] presented a probabilistic uniform self-stabilizing algorithm this class of algorithms *strong probabilistic stabilizing* algorithms. However, the algorithm of [11] works *only* under the *central scheduler* (which allows exactly one enabled processor at any time). All previously known algorithms solving the mutual exclusion problem ensure fairness using one of the two well-known methods: *(i)* by choosing an *ad hoc* scheduler (e.g., the fair scheduler in [7] or randomized central scheduler in [9]) and *(ii)* by requiring that the correctness of the system is probabilistic (as in [1] and [8]). The open question in [11] was to design a strong probabilistic stabilizing algorithm that solves the mutual exclusion problem under an unfair distributed scheduler.

Our Contributions. We answer the open question of [11] and provide a *strong probabilistic stabilizing* algorithm for the mutual exclusion problem in an anonymous unidirectional ring of any size running under an *unfair distributed scheduler.* (The distributed scheduler selects an arbitrary non-empty subset of enabled processors in a computation step at any time.) We describe the probabilistic self-stabilizing systems in Section 3. We then present a strong probabilistic stabilizing algorithm which works under a k-bounded scheduler (that bounds the ratio of relative speeds of executions of any two processors to k). Finally, we use the composition technique described in [2] to stabilize the algorithm under an unfair distributed scheduler (Section 4). We show that the maximum expected stabilization time is $O(n^3)$ under both schedulers. Once stabilized, the upper bound between two occurrences of the privilege at a given processor is $O(n^3)$ under the unfair scheduler and $O(kn)$ under the k-bounded scheduler. Further details about the algorithms and proofs can be found in [4].

2 Model

Distributed Systems 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.

We consider unidirectional ring networks where the processors maintain two types of variables: *local variables* and *field variables.* Each processor, P_i, has two neighbors named $left_i$ (its clockwise neighbor) and $right_i$ (its counter-clockwise neighbor). The local variables of P_i cannot be accessed by its right neighbor, whereas the field variables of P_i are part of the shared register which is used to communicate with P_i's right neighbor. A processor can write only into its own shared register and can read only from the shared registers, owned by its left neighbor or itself. The *state* of a processor is defined by the values of its local and field variables. A processor may change its state by executing its local *algorithm* (defined below). A *configuration* of a distributed system is an

The algorithm executed by each processor is described by a finite set of guarded actions of the form $\langle\text{guard}\rangle \longrightarrow \langle\text{statement}\rangle$. Each guard of processor P_i is a boolean expression involving P_i's variables and $left_i$'s field variables. A processor 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 processors in c. We denote by $\{c : CH\}$ the set of configurations that are *reachable* from c if every processor 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 $n \geq 2$ of computation e is denoted as $h_n = [h_{n-1}(c_{n-1}, CH_{n-1}, c_n)]_n$, where h_{n-1} is a history of length $n-1$ and (c_{n-1}, CH_{n-1}, c_n) is a computation step from the final configuration of the history h_{n-1}(for $n = 1$, $h_1 = (c_0, CH_0, c_1)$). In some context in this paper, the length n may not be used.

The probabilistic value of a history is the product of the probabilities of all the computation steps in the history. In the following, if h_n is a history such that $h_n = [(c_0, CH_0, c_1) \ldots (c_{n-1}, CH_{n-1}, c_n)]_n$, 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 (equal to c_n) is represented by $last(h_n)$, and the first configuration in h_n (equal to 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.

3 Probabilistic 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 [2].

Scheduler A scheduler can be considered as an adversary ([2, 6]). In this model, 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 nonempty subset of the enabled processors in configuration c_i. In every computation step, this subset is chosen by the scheduler. The interaction between a scheduler and the distributed system generates some special structures, called strategies. The scheduler strategy definition is

c be a system configuration and S a distributed system. The tree representing all computations in S starting from the configuration c is called the *S-tree* rooted at c and is denoted as $\mathcal{T}ree(c)$. Let n_1 be a processor in $\mathcal{T}ree(c)$. A *branch* rooted in n_1 indicates the set of all $\mathcal{T}ree(c)$ computations starting in n_1 with the same first transition. The degree of n_1 is the number of branches rooted in n_1. A *sub-S-tree of degree* 1 rooted in c is the pruned tree $\mathcal{T}ree(c)$ such that the degree of any $\mathcal{T}ree(c)$'s processor is at most 1. A scheduler strategy is defined as follows:

Definition 3.1 (Scheduler Strategy) *Let S a distributed system, let D be a scheduler and let c be a S configuration. We call a scheduler strategy rooted in c a* sub-S-tree *of degree 1 of $\mathcal{T}ree(c)$ such that any computation of the sub-tree verifies the scheduler D.*

Let st be a strategy. An *st-cone* $\mathcal{C}_h$ corresponding to a history h is the set of all possible st-computations which create the same history h. The probabilistic value of an st-cone $\mathcal{C}_h$ is the probabilistic value of the history h (i.e., the product of the probability of every computation step in h). An st-cone $\mathcal{C}_{h'}$ is called a *sub-cone* of $\mathcal{C}_h$ if and only if $h' = [hx]$, where x is a computation fragment.

Let S be a system, D be a scheduler, and st be a strategy. The set of st-computations that reach a configuration c' satisfying predicate P (denoted as $c' \vdash P$) is denoted as $\mathcal{EP}_{st}$, and its associated probabilistic value as represented by $Pr(\mathcal{EP}_{st})$. 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.

Probabilistic Self-Stabilizing Systems 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 3.2 (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 st of S under D, the two following conditions hold:*
(i) The probability of the set of st-computations, starting from c, reaching a configuration c', such that c' satisfies L is 1 (probabilistic convergence). (Formally, $\forall st, Pr(\mathcal{EL}_{st}) = 1$).
(ii) All computations, starting from a configuration c' such that c' satisfies L, satisfy SP (strong correctness).(Formally, $\forall st : \forall e, e' \in st : e = (e'e'') ::\ last(e') \vdash L \Rightarrow e'' \vdash SP$).

probabilistic convergence of the algorithms presented in this paper. This result is built upon some previous work on probabilistic automata ([12, 13, 14, 15]) 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* denoted 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 3.3 (Local Convergence) *Let st be a strategy, $PR1$ and $PR2$ be two predicates on configurations, where $PR1$ is a closed predicate. Let δ be a positive probability and N a positive integer. Let $\mathcal{C}_h$ be a st-cone with $last(h) \vdash PR1$ and let M denote the set of sub-cones $\mathcal{C}_{h'}$ of $\mathcal{C}_h$ such that $last(h') \vdash PR2$ and $length(h') - length(h) \leq N$. Then the cone $\mathcal{C}_h$ satisfies LC $(PR1, PR2, \delta, N)$ if and only if $Pr(\bigcup_{\mathcal{C}_{h'} \in M} \mathcal{C}_{h'}) \geq \delta$.*

Now, if in strategy st, there exist $\delta_{st} > 0$ and $N_{st} \geq 1$ such that any st-cone $\mathcal{C}_h$ with $last(h) \vdash PR1$ satisfies $LC(PR1, PR2, \delta_{st}, N_{st})$, then the result of [2] states that the probability of the set of st-computations reaching configurations satisfying $PR1 \wedge PR2$ is 1. Formally:

Theorem 3.1 ([2]) *Let st be a strategy. Let $PR1$ and $PR2$ be closed predicates on configurations such that $Pr(\mathcal{EPR}1_{st}) = 1$. If $\exists \delta_{st} > 0$ and $\exists N_{st} \geq 1$ such that any st-cone $\mathcal{C}_h$ with $last(h) \vdash PR_1$ satisfies the LC $(PR1, PR2, \delta_{st}, N_{st})$ property, then $Pr(\mathcal{EPR}12) = 1$, where $PR12 = PR1 \wedge PR2$.*

4 Strong Probabilistic Stabilizing Mutual Exclusion

In this section, we present two solutions to the mutual exclusion problem: one solution for the k-bounded scheduler (Section 4.1) and the other to work under an unfair scheduler (Section 4.2). We consider a solution to be satisfying the specification, $\mathcal{SP}_{ME}$, of the mutual exclusion problem if the solution satisfies the mutual exclusion and fairness properties.

4.1 k-Bounded Scheduler

In Algorithm 4.1, every processor in the system has a field variable t. A processor is *privileged* if and only if the difference between t and t_l (the t variable of its left neighbor) is not 1. It was proven in [1] that if operations on t variables are made *modulo* $mnd(n)$[1], where n is the number of processors in the ring, then at least one privilege is always present in the ring. Every processor has a local clock variable (noted *go_ahead*), taking values between 0 and $k+1$ (where k is a parameter

[1] $mnd(n)$ denotes the minimum non-divisor of n. For example, $mnd(5) = 2$.

changes its clock value. Otherwise, the processor strictly increases its clock. All the internal clock values between 0 and k are *wait* states and the value $k+1$ is a *pass* state. We will use $Priv(c)$ to denote the number of privileged processors in configuration c.

We define the legitimacy predicate, $\mathcal{L}_{ME}$, as follows: $\mathcal{L}_{ME} \equiv$ There exists exactly one privilege.

Algorithm 4.1 Mutual exclusion under a k-bounded scheduler (for p)

```
Field: $t_p \in [0, mnd(n) - 1]$ (the privilege)
Variables: $rand\_bool_p$ holds any value in $\{0, 1\}$. Each value has a probability of 1/2.
           $go\_ahead_p$ holds any integer value in [0..(k+1)].
Predicate: $Privilege(p) \equiv t_p - t_{lp} \neq 1 \bmod mnd(n)$
Macro: $Pass\_privilege(p)$ : $t_p := (t_{lp} + 1) \bmod mnd(n)$
Actions: $\mathcal{A}_1$:: $Privilege(p) \wedge go\_ahead_p \neq$ (k+1) $\longrightarrow$
             if ($rand\_bool_p = 1$) then $go\_ahead_p$:=(k+1) else $go\_ahead_p := go\_ahead_p + 1$;
         $\mathcal{A}_2$:: $Privilege(p) \wedge go\_ahead_p$=(k+1) $\longrightarrow$
             $Pass\_privilege(p)$; if ($rand\_bool_p = 0$) then $go\_ahead_p$:=random(0..k+1);
```

Lemma 4.1 (Strong Correctness) *Starting from any legitimate configuration, each processor is privileged within $(k+2) \times n$ computation steps, where n is the size of the ring.*

Proof: The worst scenario in terms of a privilege being held at processor p is as follows: p has its $go\_ahead_p = 0$ and every time p is chosen, $rand\_bool_p = 0$. In the worst case, the privilege remains at p for $(k+2)$ times (for $k+1$ steps, the clock is incremented, and finally, in step $k+2$, the privilege is passed). □

Lemma 4.2 (Probabilistic Convergence) *Let st be a strategy of Algorithm 4.1 under a k-bounded scheduler starting in configuration c. There exists $\epsilon > 0$ and $N \geq 1$ such that any cone of computations of st satisfies $LC(true, \mathcal{L}_{ME}, \epsilon, N)$.*

Proof: In the following, $dist(p, q)$ denotes the distance between Processors p and q, which is equal to the number of the processors between p and q in the clockwise orientation, plus 1. Assume that in c, there are m privileges $t_1, \ldots, t_m$, where $dist(t_m, t_1) = min_{1 \leq i \leq m-1}(dist(t_i, t_{i+1}))$. Let d_i be the distance in c between t_i and t_{i+1}. In the worst case, every processor p between t_m and t_1 is in the "worst" *wait* state ($go\_ahead = 0$), and every processor p' between t_1 and t_2 is in the *pass* state ($go\_ahead_{p'} = k+1$).

1. We calculate the probability ϵ_0 to obtain a cone $\mathcal{C}_{h0}$, where $last(h0)$ satisfies the following three properties: *(1)* t_m reached the processor which held t_1 in c, *(2)* t_1 reached the processor which held t_2 in c, and *(3)* all the processors visited by t_1 set their variable *go_ahead* to $k+1$, and

$dist(t_m, t_1) \leq (k+2) \times d_m$, $length(h0) \leq max(d_1, (k+2)d_m)$ and $\epsilon_0 \geq (\frac{1}{2})^{n^2}(\frac{1}{k+2})^{2\times n^2}$.

2. We calculate the probability ϵ_1 that cone $\mathcal{C}_{h0}$ has a sub-cone $\mathcal{C}_{h1}$ such that $Priv(last(h1)) \leq Priv(c) - 1$. The probability to obtain $\mathcal{C}_{h1}$ is $\epsilon_1 \geq \epsilon_0(\frac{1}{2})^{k\times d_m}(\frac{1}{2\times(k+2)})^{2\times d_m}(\frac{1}{4} \times \frac{1}{k+2})^{(m-2)\times d_m}$. The length of the history of cone $h1$ is $length(h1) \leq max(d_1, (k+2) \times d_m) + d_m$ and $\epsilon_1 \geq (\frac{1}{2})^{4\times n^2}(\frac{1}{k+2})^{3\times n^2}$.

Reapplying (2) for the cone $\mathcal{C}_{h1}$, there is a positive probability to obtain a sub-cone $\mathcal{C}_{h2}$ such that $Priv(last(h2)) \leq m-2$, and then a positive probability ϵ to obtain a sub-cone $\mathcal{C}_{hm-1}$, where the number of privileges is 1. $\epsilon \geq (\frac{1}{2})^{4\times n^3}(\frac{1}{k+2})^{3\times n^3}$ and $length(hm-1) \leq (k+2) \times n^2$. □

From Lemmas 4.1 and 4.2, and Theorem 3.1, we can claim the following result:

Theorem 4.1 *Algorithm 4.1 is strong probabilistic stabilizing for $\mathcal{SP}_{ME}$.*

4.2 Unfair Scheduler

In this section, we extend Algorithm 4.1 so that it can cope up with an unfair scheduler. For this purpose, the idea of *cross-over* composition (introduced in [2]) is used. That is, Algorithm 4.2 results from crossover composition of Algorithm 4.1 and the deterministic token passing algorithm of [2] (the tokens related to this algorithm are thereafter mentioned as *fair privileges*). Algorithm 4.2 combines the best of both algorithms, retaining the strong probabilistic stabilization of Algorithm 4.1 and the unfair distributed scheduler support of the token passing algorithm. In the worst case, a $(n-1)$-bounded scheduler is guaranteed, which gives a $O(n^3)$ computation steps time complexity.

Theorem 4.2 *Algorithm 4.2 is strong probabilistic stabilizing for $\mathcal{SP}_{ME}$ under an unfair scheduler.*

Proof: The proof directly follows from Theorem 4.1 and [2]. □

5 Complexity

Space Complexity. The minimum non-divisor of n is $O(\log(n))$ [10]. Therefore, Algorithm 4.2 needs $O(\log(n-1) + 2 \times \log(\log(n)))$ bits per processor. Algorithm 4.1 uses $O(\log(k) + \log(\log(n)))$ bits per processor.

Stabilization Time Complexity. As the convergence of our algorithms is only probabilistic, we can only guarantee maximum expected stabilization time. In the literature (e.g., [6]), the maximum expected stabilization time is expressed in terms of *rounds*, where a round is a minimal computation fragment during which every processor executes one action.

From Lemma 4.1, a round consists of $O(nk)$ computation steps. We now calculate the maximum expected number of rounds for Algorithm 4.1 (where $k = n-1$) to stabilize when started in the worst possible configuration (with m tokens).

```
Fields: $t_p \in [0, mnd(n) - 1]$ (the privilege)
        $ft_p \in [0, mnd(n) - 1]$ (the fair privilege)
Variables: $rand\_bool_p$ holds any value in {0, 1}. Each value has a probability of 1/2.
           $go\_ahead_p$ holds any integer value in [0..(k+1)].
Predicates: $Privilege(p) \equiv t_p - t_{lp} \neq 1 \bmod mnd(n)$
            $Fair\_privilege(p) \equiv ft_p - ft_{lp} \neq 1 \bmod mnd(n)$
Macros: $Pass\_privilege(p) : t_p := (t_{lp} + 1) \bmod mnd(n)$
        $Pass\_Fair\_privilege(p) : ft_p := (ft_{lp} + 1) \bmod mnd(n)$
Actions: $\mathcal{A}_1$:: $Fair\_privilege(p) \wedge Privilege(p) \wedge go\_ahead_p \neq$ (k+1) $\longrightarrow$
             $Pass\_Fair\_privilege(p)$
             if ($rand\_bool_p = 1$) then $go\_ahead_p$=(k+1) else $go\_ahead++$;
         $\mathcal{A}_2$:: $Fair\_privilege(p) \wedge Privilege(p) \wedge go\_ahead_p$=k+1 $\longrightarrow$
             $Pass\_Fair\_privilege(p)$ ; $Pass\_privilege(p)$;
             if ($rand\_bool_p = 0$) then $go\_ahead_p$=random(0..k+1);
         $\mathcal{A}_3$:: $Fair\_privilege(p) \wedge \neg Privilege(p) \longrightarrow Pass\_Fair\_privilege(p)$
```

First, we find an upper bound on the expected number of rounds needed to reach a configuration where the number of tokens has decreased by 1. We consider the behavior of the m pairs of successive tokens within one round. The probability that no two consecutive tokens merge is less than $p = \left(\frac{1}{2} \times \frac{1}{(k+2)}\right)^{m \times n}$. Thus the probability that at least one pair of two consecutive tokens merge is more than $q = 1 - p$. Then, the expected number of rounds before the system reaches a configuration with $m - 1$ tokens is $E[m, m-1] < \frac{1}{q}$, *i.e.* $E[m, m-1] < \frac{2(k+2)^{m \times n}}{2(k+2)^{m \times n} - 1}$.

The maximum expected number of rounds $T_{4.1}$ before stabilization of Algorithm 4.1 (where $k = n - 1$) from a configuration with m privileges to a configuration with 1 privilege is given by the formula $T_{4.1} \leq \sum_{i=2}^{m} E[i, i-1] = \sum_{i=2}^{m} \frac{2(n+1)^{in}}{2(n+1)^{in} - 1} \leq m$. Since $T_{4.1}$ is $O(m)$ rounds and $m \leq n$, $T_{4.1}$ is $O(n^3)$ computation steps. A processor executing Algorithm 4.2 executes a rule of Algorithm 4.1 if and only if it holds a fair token. For the time complexity analysis, the worst number of fair tokens is 1. Hence, the bounds provided for Algorithm 4.1 hold for Algorithm 4.2 too. Therefore, its time complexity is $O(n^3)$.

6 Conclusions

We presented a solution to the open problem of having a strong probabilistic self-stabilizing mutual exclusion algorithm under an unfair distributed scheduler. Once the system is stabilized, a processor only waits a bounded (polynomial) amount of time. Bounding the coin tossing as presented in this paper can be applied to several other probabilistic algorithms (e.g., [10]) in order to provide a bound of the service time. The service time provided by our solutions is $(k + 2) \times n$ (Algorithm 4.1) and n^3(Algorithm 4.2) respectively. On an average, the service time is $\frac{(k+3) \times n}{2}$ and $\frac{n^2(n+1)}{2}$ for

References

[1] J. Beauquier, S. Cordier, and S. Delaët. Optimum probabilistic self-stabilization on uniform rings. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 15.1–15.15, 1995.

[2] J. Beauquier, M. Gradinariu, and C. Johnen. Randomized self-stabilizing optimal leader election under arbitrary scheduler on rings. Technical Report 1225, Laboratoire de Recherche en Informatique (`http://www.lri.fr/~mariag/articles.html`), September 1999.

[3] J. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11:330–344, 1989.

[4] A. K. Datta, M. Gradinariu, and S. Tixeuil. Self-stabilizing mutual exclusion using unfair distributed scheduler. Technical report, Laboratoire de Recherche en Informatique, Université de Paris Sud, France (`http://www.lri.fr/~tixeuil/Recherche_publi_us.htm`).

[5] E. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[6] S. Dolev, A. Israeli, and S. Moran. Analyzing expected time by scheduler-luck games. *IEEE Transactions on Software Engineering*, 21:429–439, 1995.

[7] M. Flatebo and A. Datta. Two-state self-stabilizing algorithms for token rings. *IEEE Transactions on Software Engineering*, 20:500–504, 1994.

[8] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35:63–67, 1990.

[9] T. Herman. Self-stabilization: randomness to reduce space. *Distributed Computing*, 6:95–98, 1992.

[10] A. Israeli and M. Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *PODC90 Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pages 119–131, 1990.

[11] H. Kakugawa and M. Yamashita. Uniform and self-stabilizing token rings allowing unfair daemon. *IEEE Transactions on Parallel and Distributed Systems*, 8:154–162, 1997.

[12] A. Pogosyants, R. Segala, and N. Lynch. Verification of the randomized consensus algorithm of aspen and herlihy: a case study.

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

[14] R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. In Springer-Verlag, editor, *CONCUR '94, Concurrency Theory, 5th International Conference , LNCS:836*, Uppsala, Sweden, August 1994.

[15] S. H. Wu, S. A. Smolka, and E. W. Stark. Composition and behaviors of probabilistic i/o automata. In *concur94*, pages 513–528, 994.