Automated Analysis and Optimization of Distributed Self-Stabilizing Algorithms

by

Saba Aflaki

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2015

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis contains material published in [3] (Chapters 3, 6, 7) and [2] (Chapters 5, 6, 7). The material in Chapter 4 (and partly Chapters 6, 7) is under preparation for submission.

Synthesizing Self-stabilizing Protocols under Average Recovery Time Constraints. Saba Aflaki, Fathiyeh Faghih, Borzoo Bonakdarpour, In Proceedings of the 35th International Conference on Distributed Computing Systems (ICDCS), Columbus, Ohio, 2015.

Fathiyeh Faghih proof read the manuscript and helped with writing Chapter 3.1. Prof. Bonakdarpour proposed the problem, supervised the project and extensively edited the manuscript.

Automated Fine Tuning of Probabilistic Self-stabilizing Algorithms Saba Aflaki, Borzoo Bonakdarpour, Arne Storjohann, under preparation.

Prof. Storjohann implemented a code to compute Equation 4.3 for large matrices giving the results in Table 4.2. Prof. Storjohann also wrote Chapter 4.2.3. Prof. Bonakdarpour proposed the problem, supervised the project and extensively edited the manuscript.

Automated Analysis of Impact of Scheduling on Performance of Self-stabilizing Protocols. Saba Aflaki, Borzoo Bonakdarpour, Sébastien Tixeuil, In Proceedings of the 17th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Edmonton, AB, Canada, 2015.

Prof. Tixeuil proposed several ideas and helped with analyzing the results of the experiments. Prof. Tixeuil also greatly helped with writing Chapter 5.3. Prof. Bonakdarpour supervised the project and extensively edited the manuscript.

Abstract

Self-stabilization [14] is a versatile technique for recovery from erroneous behavior due to transient faults or wrong initialization. A system is self-stabilizing if (1) starting from an arbitrary initial state it can automatically reach a set of *legitimate states* in a finite number of steps and (2) it remains in legitimate states in the absence of faults. Weak-stabilization [25] and probabilistic-stabilization [32] were later introduced in the literature to deal with resource consumption of self-stabilizing algorithms and impossibility results. Since the system perturbed by fault may deviate from correct behavior for a finite amount of time, it is paramount to minimize this time as much as possible, especially in the domain of robotics and networking. This type of fault tolerance is called non-masking because the faulty behavior is not completely masked from the user [13].

Designing *correct* stabilizing algorithms can be tedious. Designing such algorithms that satisfy certain *average recovery time* constraints (e.g., for performance guarantees) adds further complications to this process. Therefore, developing an automatic technique that takes as input the specification of the desired system, and synthesizes as output a stabilizing algorithm with minimum (or other upper bound) average recovery time is useful and challenging. In this thesis, our main focus is on designing automated techniques to optimize the average recovery time of stabilizing systems using model checking and synthesis techniques.

First, we prove that synthesizing weak-stabilizing distributed programs from scratch and repairing stabilizing algorithms with average recovery time constraints are NP-complete in the state-space of the program. To cope with this complexity, we propose a polynomial-time heuristic that compared to existing stabilizing algorithms, provides lower average recovery time for many of our case studies.

Second, we study the problem of fine tuning of probabilistic-stabilizing systems to improve their performance. We take advantage of the two properties of self-stabilizing algorithms to model them as absorbing discrete-time Markov chains. This will reduce the computation of average recovery time to finding the weighted sum of elements in the inverse of a matrix.

Finally, we study the impact of scheduling policies on recovery time of stabilizing systems. We, in particular, propose a method to augment self-stabilizing programs with k-central and k-bounded schedulers to study different factors, such as geographical distance of processes and the achievable level of parallelism.

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Borzoo Bonakdarpour, for his continuous motivation, support and guidance. I learned a great deal from him during the past two years. I could not have imagined having a better supervisor for my Master's studies. Special thanks to my thesis readers: Dr. Arne Storjohann and Dr. Shai Ben-David for their helpful feedback and encouragement.

A special thank you to Dr. Storjohann for sharing his expertise and his collaboration in Chapter 4. My sincere thanks goes to Dr. Sébastien Tixeuil for his brilliant ideas and collaboration in Chapter 5. These works would not have been possible without their help and contribution.

Many thanks to my colleagues and friends at the University of Waterloo. I would not have survived without their support, encouragement and all the fun moments we had together. In particular, I would like to thank Fathiyeh Faghih for sharing her knowledge, constant encouragement and collaboration. I learned a lot from her during our invigorating discussions.

Dedication

This thesis is dedicated to:

My lovely parents Mina and Esmaeil Aflaki for their endless encouragement, devotion and understanding throughout the years. To my wonderful sister Mona for her unwavering support and kindness.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Distributed systems have conquered the digital world in recent years. Networks have been rapidly growing in size and the geographical area that they are distributed over. Their applications span a wide range including telecommunications, home appliances, medical equipment and real-time aircraft control. Due to the broad area and sensitiveness of their applications, reliability and robustness is a paramount issue in their design. As new generations of circuits continue to shrink in size, they become more prone to faults and sensitive to noise. Designing a fault-tolerant system requires predicting all possible scenarios of fault perturbation and their outcome which can be very challenging or impossible in very large networks. One way to deal with this difficulty is to design the system in a way that starting from any arbitrary initial state it will reach a correct behavior in a finite time. In this case, the system can recover from transient faults that throw the system in an unwanted state. Such systems are called *self-stabilizing* [14].

Self-stabilization is a non-masking fault-tolerance technique. A fault-tolerant system is non-masking if wrong behavior (violation of safety) is allowed for a finite amount of time. Self-stabilization can be used for recovery from erroneous behavior due to transient faults or wrong initialization in distributed systems. Pioneered by Dijkstra in 1974, he raised the question whether several processes can synchronize their behavior *in spite of distributed control* [14]. In his seminal paper, he presented self-stabilizing algorithms for the mutual exclusion problem. In distributed control, processes do not have access to the state of the whole system. Their information and consequently actions are limited to a small subset of the network. More specifically, each process shares information only with its neighbors. Consequently, reaching a desirable global behavior (*legitimate state*) and preserving it when each process has partial observation of the state of the system can be challenging or impossible.

Self-stabilizing systems are identified by two main properties:

- *strong convergence* ensures that starting from any arbitrary initial state, all computations will eventually reach a correct behavior in a finite number of steps,
- *Closure* guarantees that after reaching a legitimate state the system will remain in legitimate states in the absence of faults.

Strong convergence is a very strong condition as it requires all computations to reach a legitimate state. There are problems that are impossible to solve in non-probabilistic settings (e.g. token circulation and leader election in anonymous networks [32, 28, 12]). Furthermore, self-stabilizing algorithms with strong convergence usually require more space. Other variants of convergence, *weak* [25] and *probabilistic* [32] convergence, were introduced to solve a wider range of problems and tackle resource consumption issues. Weak convergence requires that starting from any state there should *exist* at least one computation that reaches a legitimate state. Similarly, in probabilistic convergence the probability of reaching a legitimate state starting from any arbitrary state should be one.

1.1 Challenges in Designing Self-Stabilizing Algorithms

Synthesizing correct distributed self-stabilizing algorithms from their specification is a difficult task. Below, we discuss three main challenges in this regard.

1.1.1 Proof of Correctness

Designing self-stabilizing algorithms and proving their correctness can be very demanding and prone to error to the point that Dijkstra asked readers to design an algorithm before reading his solution to the mutual exclusion problem [14, 16]. He, in fact, proved its correctness twelve years later [15].

1.1.2 Partial Observation

The main application of self-stabilization is in distributed systems. In such systems, processes can read and write only a small subset of the variables that determine the global state of the system in one atomic step. Processes need to reach a desired behavior using their partial information of the state. Partial observation causes dependency among some transitions of a process which we refer to them as *groups* of transitions. In Chapter 3 we prove that a consequence of this

dependency is that synthesis and repair of a weak-stabilizing algorithm to meet a given average recovery time bounded by a real number k is NP-complete.

1.1.3 Recovery Time Calculation

Analytical computation of recovery time of a self-stabilizing algorithm is challenging on its own. Adding average recovery time constraints to the system's specification can further complicate the design process.

Given the above challenges, it is crucial to develop automated techniques to synthesize correct and more preferably optimum self-stabilizing algorithms from their specification.

1.2 Contributions

In this thesis, our primary focus is on analysis and optimization of recovery time using model-checking and verification methods. Our contributions can be summarized as:

1. First, we prove that repair and synthesis of a weak-stabilizing algorithm with average recovery time constraints is NP-complete and present a polynomial-time heuristic that works for both repair and synthesis problems.
2. Given the popularity of randomized schedulers, we propose a technique to find the randomization values that give minimum average recovery time.
3. We study the effect of different types of schedulers through an empirical study of the vertex coloring problem. In this regard, we propose a method to augment self-stabilizing programs with k-central and k-bounded schedulers.

In the sequel, we explain the above contributions in more detail.

1.2.1 Complexity of Repair & Synthesis of Weak-Stabilizing Algorithms under Recovery Time Constraints

The first step was to analyze the complexity of the problem of synthesizing or repairing stabilizing systems to meet pre-specified average recovery time constraints. In [18], the authors

proposed an efficient automated method to add weak convergence to a distributed program without average recovery time bounds in linear time. Their heuristic uses the weakest program, a maximal program in the sense that all possible transitions are present in the system. All possible stabilizing programs for that specification are a subset of the maximal program. Therefore, synthesizing a stabilizing program can be reduced to repairing (eliminating transitions from) the maximal program. We employ this reduction to prove that synthesizing (equivalently repairing) a weak-stabilizing program with an average recovery time bounded by a real value is NP-complete. In this regard, we propose a polynomial-time heuristic that takes as input the state-space, set of legitimate states and the corresponding maximal program and repairs it with the goal of minimizing the average recovery time. Our heuristic is based on the intuition that cycles tend to increase recovery time and the closer they are to legitimate states the greater their impact.

1.2.2 Automated Fine-tuning of Probabilistic-Stabilizing Algorithms

Randomization can be used to break symmetry in anonymous networks in deterministic settings. In probabilistic distributed systems processes execute their actions randomly. Intuitively, the probability values based on which processes run their actions can affect the average recovery time. However, using biased coins (probability value other than 0.5) to minimize recovery time was a counter-intuitive result first demonstrated by Kwiatkowska et al. [37] through an empirical study. Following their line of work, we propose an automated technique to find the optimum randomization parameters that give minimum average recovery time for a specific network size. We use the two properties of stabilizing algorithms, closure and convergence, to model them with absorbing Markov chains and reduce the computation of average recovery time to calculating the weighted sum of elements in the inverse of the transition probability matrix among non-legitimate states. After computing the rational expression for the average recovery time in terms of randomization parameters, we use existing symbolic optimization tools (Maple) to find a valuation of the parameters that gives minimum average recovery time.

1.2.3 Automated Analysis of Impact of Scheduling on Performance of Self-Stabilizing Algorithms

Schedulers or daemons are another element of distributed systems that can significantly affect recovery time or even the possibility of convergence. A scheduler determines which subset of the enabled processes can execute their commands at each time. They can be classified based on several factors: *distribution*, *enabledness*, *boundedness* and *fairness* [17]. We study the effect

of different scheduler types on correctness and average recovery time of several stabilizing algorithms for the vertex coloring problem in arbitrary graphs [28]. The deterministic algorithm [28] does not converge in the worst case under a distributed unfair scheduler. One way to refine this algorithm to a scheduler-oblivious stabilizing algorithm is to compose it with a stabilizing mutual exclusion protocol with the cost of higher recovery time due to ensuring safety. Another way is to randomize the actions of processes [28, 29]. Our experiments showed that in general, deterministic algorithms have better performance. One deterministic algorithm is to assign unique IDs to each process and always have an enabled process with lowest ID to run. This protocol beats all the other deterministic and probabilistic algorithms we studied. However, it has the drawback of needing unique IDs for processes in advance. To run a ID-based self-stabilizing algorithm on an anonymous network, it should be composed with a unique naming self-stabilizing algorithm at the cost of performance hit. We studied three randomization strategies: 1) p=constant value 2) p=1/vertex degree 3) p=1/#conflicts. The first two strategies are static, i.e. the probability values do not change throughout the execution. The third approach is dynamic as the number of conflicts for each vertex varies during the execution. Our experiments demonstrate that this strategy outperforms the other two in most cases with the advantage of no pre-tuning requirements.

1.3 Organization

Chapter 2 presents our framework, formal definitions of distributed programs, Markov chains and average recovery time. In Chapter 3 we prove that repair and synthesis of a weak-stabilizing algorithm with average recovery time constraint is NP-complete and propose a polynomial-time heuristic. In Chapter 4, we present an automated technique for fine tuning of probabilistic distributed self-stabilizing algorithms to achieve minimum average convergence time. In Chapter 5, we study the effect of schedulers and randomized conflict managers on correctness and average recovery time of a stabilizing vertex coloring algorithm. Chapter 6 is dedicated to related work and finally we conclude and discuss future work in Chapter 7.

Chapter 2

Preliminaries

In this chapter, we present our formal framework and computation models used throughout the thesis.

2.1 Distributed Programs

A *distributed program* dp consists of a finite set Π of *processes* and a finite set V of discrete *variables*, where each variable $v \in V$ ranges over a finite domain D_v. A *state* s of dp is determined by a value assignment to all variables, denoted by a vector $s = \langle v_1, \ldots, v_{|V|} \rangle$. The value of a variable v in a state s is indicated by $v(s)$. The *state space* of dp (S) is the set of all possible states:

$$S = \prod_{v \in V} D_v.$$

A *state predicate* is a subset of S. We denote the initial state distribution, i.e., the probability of starting the program in each state by $\iota_{init}(s)$ such that

$$\sum_{s \in S} \iota_{init}(s) = 1.$$

Definition 1 (Process) *A* process $\pi \in \Pi$ *over a set V of variables is a tuple $\langle R_\pi, W_\pi, T_\pi \rangle$, where*

- *$R_\pi \subseteq V$ is the* read-set *of π; i.e., variables that π can read;*

- $W_\pi \subseteq R_\pi$ *is the* write-set *of* π*; i.e., variables that* π *can modify, and*
- T_π *is the* transition relation *of* π*, which is a set of ordered pairs* (s, s')*, where* $s, s' \in S$*, subject to the following constraints:*
  - Write restriction:

$$\forall (s, s') \in T_\pi : \forall v \in V : (v(s) \neq v(s')) \Rightarrow v \in W_\pi$$

  - Read restriction:

$$\begin{aligned} &\forall (s_0, s_1) \in T_\pi : \exists s_0', s_1' \in S \\ &\quad \forall v \notin W_\pi : (v(s_0) = v(s_1) \ \wedge \ v(s_0') = v(s_1'))) \wedge \\ &\quad (\forall v \in R_\pi : (v(s_0) = v(s_0') \ \wedge \ v(s_1) = v(s_1'))) \Rightarrow \\ &\qquad\qquad (s_0', s_1') \in T_\pi. \ \blacksquare \end{aligned}$$

The write restriction requires that a process can only change the value of a variable in its write-set, but not blindly. That is, it cannot write a variable that it is not allowed to read. The read restriction captures the fact that in distributed computing, processes have only a partial view of the global state. The read restriction imposes the constraint that for each process π, each transition in T_π depends only on reading the variables that π is allowed to read (i.e. R_π). Thus, each transition in T_π is in fact an equivalence class in T_π, which we call a *group* of transitions. This is because each process π should exhibit identical behavior in states where its read-set has equal valuation. The key consequence of read restriction is that during synthesis, if a transition is included (respectively, excluded) in T_π, then its corresponding group must also be included (respectively, excluded) in T_π.

Notation 1 *For a transition* $(s, s') \in T_\pi$*,* $\mathcal{G}(s, s')$ *denotes the set of all transitions in the group of* (s, s')*. Also,* $\mathcal{G}(X)$ *denotes* $\bigcup_{(s,s') \in X} \mathcal{G}(s, s')$*, where X is a set of transitions.*

Definition 2 (Distributed Program) *A* distributed program *over a set of variables V is a tuple* $dp = \langle \Pi_{dp}, T_{dp} \rangle$*, where*

- Π_{dp} *is a set of processes over the common set V of variables, such that for any two distinct processes* $\pi_1, \pi_2 \in \Pi_{dp}$*, we have* $W_{\pi_1} \cap W_{\pi_2} = \emptyset$*,*
- T_{dp} *is the transition relation of dp, such that*

$$T_{dp} = \bigcup_{\pi \in \Pi_{dp}} T_\pi \ \ \blacksquare$$

Algorithm 1 Probabilistic-stabilizing Vertex Coloring (process π)

1: **Variable:** $c_\pi : int \in [0, B]$
2: **Guarded Command:** $c_\pi \neq max(\{0, \cdots, B\} \setminus \bigcup_{\pi' \in N(\pi)} c_{\pi'}) \longrightarrow p : c_\pi := max(\{0, \cdots, B\} \setminus \bigcup_{\pi' \in N(\pi)} c_{\pi'}) + (1 - p) : c_\pi := c_\pi;$

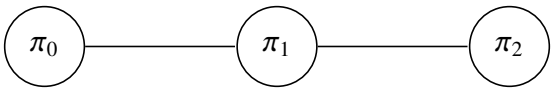

Figure 2.1: Example of a graph coloring problem

In our model, processes communicate through *shared memory*, i.e. several processes can read the same variable. However, only one process can write to a variable. It also models an asynchronous distributed program, where process transitions execute in an *interleaving* fashion resulting in T_{dp} being the union of transition relations of the processes in the program. We say that processes $\pi_1, \pi_2 \in \Pi_{dp}$ are *neighbors* iff $R_{\pi_1} \cap R_{\pi_2} \neq \emptyset$. Thus, the communication network of a distributed system can be modeled by a graph $G = (\Pi, E)$, where a node represents a process, and there is an edge between any two processes that are neighbors.

Notation 2 *We denote the set of neighbors of a process π by $N(\pi)$, the shortest path between two vertices (processes) π and π' in G by $dist(\pi, \pi')$ and the diameter of the graph by $diam(G)$.*

Example 1 We use the probabilistic distributed *vertex coloring* algorithm of [28] as a running example. A solution to the vertex coloring problem is an assignment of colors to vertices (i.e., processes) of a graph from a given set of colors subject to the constraint that no two adjacent vertices share the same color. Here, adjacency is determined by the neighborhood relation. Algorithm 1 runs on each process π in the system. Each process π maintains a variable c_π, which represents the color of π and ranges over $D_{c_\pi} = [0, B]$, where B is the maximum vertex degree in the graph. Processes representing adjacent vertices in the graph are neighbors in the distributed program (they can read the color of their neighbors). States in which no guards are satisfied (i.e., the value of all variables equals the maximum available color) are considered legitimate states in this algorithm. Suppose we have a graph with three vertices connected in a row as shown in Figure 2.1. In this example, $\Pi_{dp} = \{\pi_0, \pi_1, \pi_2\}$ sharing a read variable with its neighbor process(es). Let $dp = \langle \Pi_{dp}, T_{dp} \rangle$ be the distributed program over variables $V = \{c_0, c_1, c_2\}$ solving our vertex coloring problem.

Now, consider the following transitions:

$$t_1 = (s_0 = [c_0 = 0, c_1 = 1, c_2 = 2], s_1 = [c_0 = 2, c_1 = 1, c_2 = 2])$$
$$t_2 = (s'_0 = [c_0 = 0, c_1 = 1, c_2 = 1], s'_1 = [c_0 = 2, c_1 = 1, c_2 = 1])$$

Observe that as far as process π_0 is concerned, states s_0 and s'_0 are identical, as the variables in the read-set of π_0 have the same value in both states. Consequently, since π_0 is changing its only write variable, c_0, from 0 to 2 in s_0, it should do the same in s'_0. In other words, if transition t_1 exists in the program, then so should t_2. If not, π_0's execution will depend on the value of c_2 which is not readable by π_0.

2.2 Discrete-time Markov Chains

Discrete-time Markov Chains (DTMCs) are transition systems equipped with probabilities. By modelling distributed programs with DTMCs, one can reason about their correctness and compute their recovery time in case of stabilizing programs.

2.2.1 Discrete-Time Markov Chains

Definition 3 (DTMC) *A* DTMC *is a tuple* $\mathcal{D} = (S, S_0, \iota_{init}, \mathbf{P}, L, AP)$ *where,*

- S *is a finite set of states*
- S_0 *is the set of initial states*
- $\iota_{init} : S \to [0, 1]$ *is the initial state distribution such that*

$$\sum_{s \in S} \iota_{init}(s) = 1$$

- $\mathbf{P} : S \times S \to [0, 1]$ *is the transition probability matrix (TPM) such that*

$$\forall s \in S : \sum_{s' \in S} \mathbf{P}(s, s') = 1$$

- $L : S \rightarrow 2^{AP}$ *is the labeling function that identifies which atomic propositions from a finite set AP hold in each state.*

■

In Definition 3, if the transition probabilities include symbolic values, it gives a *parametric DTMC (PDTMC)* [7].

Definition 4 (Parametric DTMC) *A* parametric DTMC *is a tuple* $\mathcal{PD} = (S, S_0, U, \iota_{init}, \mathbf{P}, L, AP)$ *where,*

- S, S_0, L *are as defined before,*
- $U = \{u_1, u_2, \cdots, u_r\}$ *is a finite set of real parameters,*
- $\iota_{init} : S \rightarrow F_U$ *is the initial state distribution and* F_U *is the set of multivariate polynomials in* $\mathbf{u} = (u_1, \cdots, u_r)$*,*
- $\mathbf{P} : S \times S \rightarrow F_U$ *is the transition probability matrix.* ■

An evaluation function $eval : U \rightarrow \mathbb{R}$ assigns real values to parameters in set U. Given an evaluation function $eval$ and a polynomial $f \in F_U$, $eval(f)$ denotes the value obtained by replacing each parameter u_i in f by $eval(u_i)$. An evaluation function is *valid* for a PDTMC with parameter set U if the induced TPM ($\mathbf{P}_{eval} = eval(\mathbf{P}) : S \times S \rightarrow [0, 1]$) and initial distribution ($\iota_{init_{eval}} = eval(\iota_{init}) : S \rightarrow [0, 1]$) satisfy the following conditions:

$$\forall s \in S : \sum_{s' \in S} \mathbf{P}_{eval}(s, s') = 1, \sum_{s \in S} \iota_{init_{eval}}(s) = 1.$$

2.2.2 Distributed Programs as DTMCs

It is straightforward to model the transition system of a distributed program with a DTMC. The state-space of the distributed program forms the set of states of the DTMC. S_0 and ι_{init} can be determined based on the program. If the distributed program is probabilistic, then the value of the elements of $\mathbf{P}$ are known. Otherwise, without loss of generality, we can consider uniform distribution over transitions. In that case:

$$\mathbf{P}(s, s') = \frac{1}{|\{(s, s'') \in T(dp)\}|}$$

L assigns atomic propositions to states which facilitates computation and verification of certain quantitative and qualitative properties. Later, in Section 2.3, we will see how a single atomic proposition ls can define an important class of distributed programs namely, self-stabilizing programs.

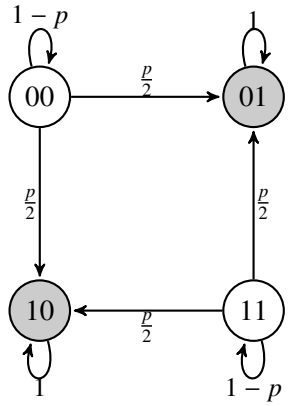

Figure 2.2: DTMC of Algorithm 1.

Example 2 The DTMC and TPMs of two processes running Algorithm 1 for a graph with two vertices π_1 and π_2 are shown in Figures 2.2 and 2.3 respectively. The TPM corresponding to Figure 2.2 is shown in Figure 2.4.

$$\begin{array}{c} \\ \mathbf{00} \\ \mathbf{11} \\ \mathbf{01} \\ \mathbf{10} \end{array} \begin{array}{c} \begin{array}{cccc} \mathbf{00} & \mathbf{11} & \mathbf{01} & \mathbf{10} \end{array} \\ \begin{bmatrix} 1-p & 0 & 0 & p \\ 0 & 1-p & p & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \qquad \begin{array}{c} \\ \mathbf{00} \\ \mathbf{11} \\ \mathbf{01} \\ \mathbf{10} \end{array} \begin{array}{c} \begin{array}{cccc} \mathbf{00} & \mathbf{11} & \mathbf{01} & \mathbf{10} \end{array} \\ \begin{bmatrix} 1-p & 0 & p & 0 \\ 0 & 1-p & 0 & p \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Figure 2.3: Transition probability matrices of π_1 and π_2.

Definition 5 (Computation) *A* computation σ *of a distributed program (equivalently a DTMC) is an infinite sequence of states:*

$$\sigma = s_0 s_1 s_2 \cdots$$

where,

- *for all* $i \geq 0 : s_i \in S$ *(called the state at time i),*

$$
\begin{array}{c} \\ \mathbf{00} \\ \mathbf{11} \\ \mathbf{01} \\ \mathbf{10} \end{array}
\begin{array}{c}
\begin{array}{cccc} \mathbf{00} & \mathbf{11} & \mathbf{01} & \mathbf{10} \end{array} \\
\left[\begin{array}{cccc}
1-p & 0 & \frac{p}{2} & \frac{p}{2} \\
0 & 1-p & \frac{p}{2} & \frac{p}{2} \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{array}\right]
\end{array}
$$

Figure 2.4: Transition probability matrix of Figure 2.2.

- $\forall i \geq 0 :\ (s_i, s_{i+1}) \in dp$, *or equivalently* $\mathbf{P}(s_i, s_{i+1}) > 0$. ■

A state with no outgoing transitions is a *terminating* state. We consider a self-loop on such states, so that any computation that reaches them stutters there infinitely. A prefix of σ is called a *finite computation*.

Notation 3 *σ_s indicates a computation that starts in state s. We use the notation σ_n for a finite computation of length n. We denote the set of all possible distributed programs by DP, the set of all computations of a distributed program by $\Sigma(dp)$ and the set of finite computations by $\Sigma_{fin}(dp)$.*

In the sequel, we review *cylinder sets* and *reachability probabilities* in Markov chains from [6].

Definition 6 *The* cylinder set *of a finite computation σ_n is the set of infinite computations which start with σ_n: $Cyl(\sigma_n) = \{\sigma \mid \sigma_n \in prefix(\sigma)\}$* ■

The probability of a cylinder set is given by:

$$Pr(Cyl(s_0 \cdots s_n)) = \iota_{init}(s_0) \prod_{0 \leq i < n} \mathbf{P}(s_i, s_{i+1}) \tag{2.1}$$

Reachability probability is a common quantitative property measured in Markov chains. Given a subset B of the state space S, we look for the probability of eventually reaching a state $s \in B$ (denoted by $(\Diamond)$). In other words, we are interested in computations with the initial sequence of states of the form $s_0 \cdots s_{n-1} \notin B$ and $s_n \in B$. All such computations can be indicated by the regular expression $R_B(M) = \Sigma_{fin}(M) \cap (S \setminus B)^* B$. Hence, reachability probability can be computed as follows:

$$Pr(\Diamond B) = \sum_{s_0 \cdots s_n \in \Sigma_{fin} \cap (S \setminus B)^* B} \iota_{init}(s_0) \prod_{0 \leq i < n} \mathbf{P}(s_i, s_{i+1}) \tag{2.2}$$

2.3 Self-stabilization and Convergence Time

A distributed program is *self-stabilizing* if (1) starting from an arbitrary initial state, all computations reach a state in the set LS of *legitimate states* in a finite number of steps without outside intervention, (2) after which remains there in the absence of faults. The first condition is known as *strong convergence* and the second as *closure*. An arbitrary state can be reached due to erroneous initialization or transient faults.

Definition 7 (self-stabilization) *A distributed program* $\mathcal{D} = (S, S_0 = S, \mathbf{P}, L, \{ls\})$ *is* self-stabilizing *iff the following conditions hold:*

- *Strong convergence:* $\forall s \in S$, all *computations* σ_s *eventually reach a state in* $LS = \{s \mid ls \in L(s)\}$,
- *Closure:* $\forall s \in LS : (\mathbf{P}(s, s') > 0) \Rightarrow (s' \in LS)$. ■

It has been shown that some problems do not have a self-stabilizing solution [29]. Thus, the strong convergence property has been relaxed in other variants of stabilization to tackle the impossibility issues and to reduce resource consumption. *Weak-stabilization* ensures the possibility of convergence and *probabilistic-stabilization* guarantees convergence with probability one.

Definition 8 (weak-stabilization) *A distributed program* $\mathcal{D} = (S, S_0 = S, \mathbf{P}, L, \{ls\})$ *is* weak-stabilizing *iff the following conditions hold:*

- *Weak convergence: For all* $s \in S$, *there* exists *a computation* σ_s *that eventually reaches a state in* LS,
- *Closure:* $\forall s \in LS : (\mathbf{P}(s, s') > 0) \Rightarrow (s' \in LS)$. ■

Definition 9 (probabilistic-stabilization) *A distributed program* $\mathcal{D} = (S, S_0 = S, \mathbf{P}, L, \{ls\})$ *is* probabilistic-stabilizing *iff the following conditions hold:*

- *Probabilistic convergence: For all* $s \in S$, *a computation* σ_s *reaches a state in* LS *with probability one,*
- *Closure:* $\forall s \in LS : (\mathbf{P}(s, s') > 0) \Rightarrow (s' \in LS)$. ■

In the sequel, we use the term *stabilizing* algorithm to refer to either of the three types of stabilization mentioned above. The expected convergence (recovery) time of a stabilizing algorithm is a key measure of its efficiency [21]. To define this metric, we use the concept of cylinder sets (Def. 6) and the reachability probability (Eq. 2.2) from Section 2.1.

The first time a computation of a stabilizing program reaches LS ($B = LS$) gives us the recovery time of that computation. More formally,

Definition 10 *For a stabilizing program $dp(\mathcal{D})$, the* convergence time *or* recovery time *of a computation σ with an initial fragment $s_0 s_1 \cdots s_n$ such that $s_0 s_1 \cdots s_{n-1} \notin LS$ and $s_n \in LS$ equals n.*

The expected recovery time of a stabilizing program is the expected value of the recovery time of all states in S.

$$ERT(dp) = \sum_{s \in S} \iota_{init}(s) ert(s). \tag{2.3}$$

$\iota_{init}(s)$ indicates the probability of starting the program in state s at time 0. If this initial distribution is unknown, we can consider a uniform distribution $\iota_{init}(s) = \frac{1}{|S|}$ for all $s \in S$. Given Def. 10 and Eq. 2.1, the expected recovery time (ERT) of a stabilizing program M is derived by the following equation:

$$ERT(\mathcal{D}) = \sum_{s_0 \in S_0} \sum_{\substack{\sigma_n \in R_B(M) \\ 0 \le n < \infty}} n.\iota_{init}(s_0). \prod_{0 \le i < n} \mathbf{P}(s_i, s_{i+1}) \tag{2.4}$$

Chapter 3

Synthesizing Self-stabilizing Protocols under Average Recovery Time Constraints

In this chapter, we formally define two problems of repair and synthesis of weak-stabilizing programs that meet certain average recovery time bounds. We prove that these problems are NP-complete in the state-space of the program and propose a polynomial-time heuristic in this regard.

3.1 Problem Statement

3.1.1 The Repair Problem

Given an existing stabilizing program and a real value *ert*, a *repair* algorithm generates another stabilizing program whose average recovery time is below *ert*. Moreover, the algorithm is required to preserve all properties of the input program. The latter can be achieved by allowing merely removing transitions from the original program. That is, we do not allow for adding transitions to avoid introducing new behaviors to the program. Since the new transition set of the repaired program will be a subset of the set of transitions of the input program, the set of computations in the new program will be a subset of the set of computations of the original one as well. Hence, any universal property satisfied by the input program (even during convergence) will be satisfied by the repaired program as well. Formally, the decision problem we study is as follows:

Instance. A weak-stabilizing program $dp = \langle \Pi_{dp}, T_{dp} \rangle$, and a real number ert.

Repair decision problem. Does there exist a weak-stabilizing program $dp' = \langle \Pi_{dp}, T'_{dp} \rangle$, such that:

- $T'_{dp} \subseteq T_{dp}$, where $T'_{dp} \neq \emptyset$, and
- $AvgRT(dp') \leq ert$.

3.1.2 The Synthesis Problem

A synthesis algorithm takes as input (1) an empty program, (2) the description of its set of legitimate states, and (3) a real value ert and generates as output the transition relation for each process, such that the average recovery time of the synthesized weak-stabilizing program is below ert.

Instance. An empty program $dp = \langle \Pi_{dp}, T_{dp} \rangle$, where $T_{dp} = \emptyset$, a state predicate LS, and a real number ert.

Synthesis decision problem. Does there exist a weak-stabilizing program $dp' = \langle \Pi_{dp}, T'_{dp} \rangle$, where $T'_{dp} \neq \emptyset$, for the set LS of legitimate states, such that:

- $AvgRT(dp') \leq ert$.

In Sections 3.2 and 3.3, we show that the above decision problems are NP-complete in the size of the state space.

3.2 The Complexity of Repairing Weak-Stabilizing Protocols with Respect to Average Recovery Time

In this section, we prove that the repair problem as introduced in Subsection 3.1.1 is NP-complete.

3.2.1 Weak-stabilizing Repair

We present a polynomial-time reduction from the *3-Dimensional Matching (3DM)* [22] problem to our repair problem.

Theorem 1 *The repair decision problem in Section 3.1.1 to obtain a weak-stabilizing programs is NP-complete.*

We show that the problem is in NP and it is NP-hard.

Proof of membership to NP
Given a program $dp' = \langle \Pi_{dp}, T'_{dp} \rangle$ as a solution, we should verify the following two conditions:

1. It is weak-stabilizing.
2. $AvgRT(dp') \leq ert$

To prove that a program is weak-stabilizing, we should verify weak-convergence and closure. This verification can be achieved through a simple graph exploration algorithm, such as BFS. Such an algorithm have polynomial-time complexity in the number of states. Calculation of expected recovery time, which in essence is reachability analysis in Markov chains (discussed in Section 2.3) can be solved in polynomial time as well [6].

Proof of NP-hardness

The 3-Dimensional Matching (3DM) problem: Given three disjoint sets X, Y, and Z each of equal size q, and m a subset of distinct tuples in $X \times Y \times Z$ of size M, the $3DM$ problem asks whether a subset $m_{sol} \subseteq m$ of size q exists such that none of the tuples in m_{sol} share a coordinate and m_{sol} is a cover for X, Y and Z. In other words,

- $\forall (x, y, z), (x', y', z') \in m_{sol} : \ (x \neq x') \ \wedge \ (y \neq y') \ \wedge \ (z \neq z')$
- $(\bigcup_{(x,y,z) \in m_{sol}} x = X) \ \wedge \ (\bigcup_{(x,y,z) \in m_{sol}} y = Y) \ \wedge \ (\bigcup_{(x,y,z) \in m_{sol}} z = Z)$

We present a polynomial-time mapping from an instance of $3DM$ to an instance of our repair problem, a distributed weak-stabilizing program $dp = \langle \Pi_{dp}, T_{dp} \rangle$, a set of legitimate states LS, and an upper bound of average recovery time ert. Figure 3.1 shows an example of an instance of the repair problem obtained from a $3DM$ instance, where $X = \{1, 2\}$, $Y = \{3, 4\}$, $Z = \{5, 6\}$, and $m = \{m_1 = (1, 3, 5), m_2 = (2, 4, 6), m_3 = (1, 4, 5)\}$, $q = 2$ and $M = 3$. In the sequel, we explain our mapping in detail.

Variables. $V_{dp} = \{v_1, v_2\}$, where $D_{v_1} = [0, M]$ and $D_{v_2} = [0, 9q - 1]$.

States. The state space of our instance is the set of all valuations of variables, resulting in $9q(M+1)$ states. The set of non-legitimate states is the following:

$$\{i_0, i_1 \mid i \in X \cup Y \cup Z\} \cup$$
$$\{i_0^{m_t}, j_0^{m_t}, k_0^{m_t}, i_1^{m_t}, j_1^{m_t}, k_1^{m_t} \mid m_t = (i,j,k) \in m\}$$

It is easy to see that there are $6q + 6M$ non-legitimate states: two per each element in $X \cup Y \cup Z$, and six per each tuple in m. We refer to them as *element states* and *tuple states*, respectively. They can be seen as the 30 white circles in Figure 3.1. The rest of the states are in LS. As shown in Figure 3.1, we include $3q + 3M$ states in LS denoted by $LS\,A_1$ and $LS\,B_0^{m_t}$, where A and B are integers that distinguish states from each other. State LS^* in the figures represents the remaining $9q(M+1) - 6q - 6M - 3q - 3M = 9M(q-1)$ states in LS. In all figures, shaded circles denote LS states.

Notation 4 *In the sequel, set $U = X \cup Y \cup Z$ is the universal set. We denote by $C(i)$ the set of tuples that contain element $i \in U$. For example, in Figure 3.1, we have $C(1) = \{m_1, m_3\}$. By $i_{0/1}$, we mean two states $\{i_0, i_1\}$. Likewise, $i_{0/1}^{m_t} = \{i_0^{m_t}, i_1^{m_t}\}$. Finally, $i_0^{C(i)} = \{i_0^{m_t} \mid (i \in U) \wedge (m_t \in C(i))\}$ denotes $|C(i)|$ states per element $i \in U$. From now on, we omit $\forall i \in U$ unless for emphasis.*

Values of v_1 and v_2 in non-legitimate states are as follows:

- $\forall i \in U:\ v_1(i_{0/1}) = 0$
- $\forall m_t = (i,j,k) \in m,\ t \in [1, M]:\ v_1(i_{0/1}^{m_t}) = v_1(j_{0/1}^{m_t}) = v_1(k_{0/1}^{m_t}) = t$
- $\forall i \in U,\ t \in [0, 3q-1]:\ v_2(i_0) = v_2(i_0^{C(i)}) = t \wedge v_2(i_1) = v_2(i_1^{C(i)}) = 3q + t$

Processes. Our mapping includes two processes π_1 and π_2. The read/write restrictions for each process are as follows:

$$R_{\pi_1} = \{v_1\} \qquad W_{\pi_1} = \{v_1\}$$
$$R_{\pi_2} = \{v_1, v_2\} \qquad W_{\pi_2} = \{v_2\}$$

Transition relation. Process π_1 has the following transitions: $T_{\pi_1} =$ outgoing:

$$\mathcal{G}(\{(i_0^{m_t}, i_0), (j_0^{m_t}, j_0), (k_0^{m_t}, k_0), (i_1^{m_t}, i_1), (j_1^{m_t}, j_1), (k_1^{m_t}, k_1) \mid m_t = (i,j,k) \in m\}) \cup$$

incoming:

$$\mathcal{G}(\{(i_0, i_0^{m_t}), (j_0, j_0^{m_t}), (k_0, k_0^{m_t}), (i_1, i_1^{m_t}), (j_1, j_1^{m_t}), (k_1, k_1^{m_t}) \mid m_t = (i, j, k) \in m\}) \tag{3.1}$$

The valuation of v_1 and v_2 in non-legitimate states is chosen in a way that the above six outgoing transitions from the six tuple states belong to the same group. To this end, the six incoming transitions (reverse direction) will be in the same group (but not in the same group as the outgoing transitions although the snake and shaded lines are on transitions on both direction).

Process π_2 has the following transitions:

$$T_{\pi_2} = \{(i_0^{m_t}, LS\,A_0^{m_t}),\ (j_0^{m_t}, LS\,B_0^{m_t}),\ (k_0^{m_t}, LS\,C_0^{m_t}) \mid m_t = (i, j, k) \in m\} \cup \{(i_1, LS\,D_1) \mid i \in U\}$$
$$\cup\ \{(s, s) \mid s \in LS\}$$

Process π_2 can read both variables, so none of its transitions form a group.

Average recovery time. In our mapping,

$$ert = \frac{21(q + M)}{9q(M + 1)}.$$

We now show that the answer to our decision problem is positive, if and only if the answer to the $3DM$ instance is affirmative:

($\Rightarrow$) Given a solution $m_{sol} \subseteq m$ to the instance of $3DM$, we show how to repair the corresponding weak-stabilizing instance to yield an average recovery time equal to ert. To this end, for every tuple m^* not in the solution, we remove the group of six transitions (pay attention to the subscripts $0/1$):

$$\{(i_{0/1}, i_{0/1}^{m^*}), (j_{0/1}, j_{0/1}^{m^*}), (k_{0/1}, k_{0/1}^{m^*}) \mid m^* = (i, j, k) \notin m_{sol}\}$$

The six transitions above on the reverse direction cannot be removed otherwise a computation that reaches a state in $\{i_1^{m^*}, j_1^{m^*}, k_1^{m^*}\}$ will not converge. Since we only remove transitions corresponding to the tuples not in m_{sol}, and the solution is a cover for sets X, Y, and Z, every state $i_{0/1}$ will have exactly one loop attached to it and $|C(i)| - 1$ incoming transitions. In our example, $m_{sol} = \{m_1, m_2\}$. Hence, $m^* = \{m_3\}$, and six (snake) transitions $\{(1_{0/1}, 1_{0/1}^{m_3}), (4_{0/1}, 4_{0/1}^{m_3}), (5_{0/1}, 5_{0/1}^{m_3})\}$ should be removed from Figure 3.1. In the repaired graph, we will have two types of connected components. One type can be seen among the states with subscript 0 (see Figure 3.2). It can be verified that the average recovery time of the top element state i_0 is 4 (there are $3q$ of them in

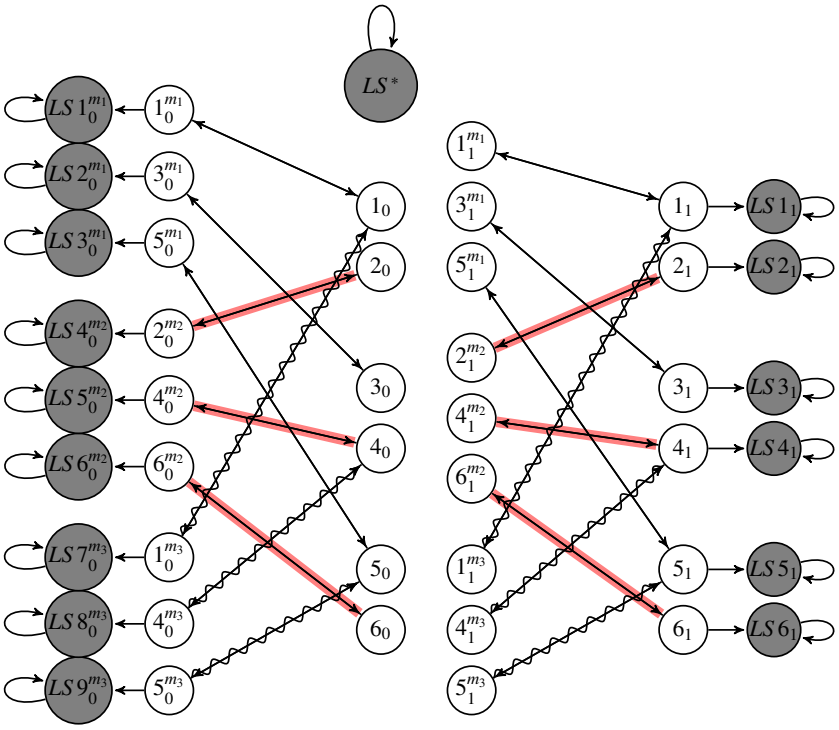

Figure 3.1: Weak repair instance mapped from $3DM$ instance $X = \{1, 2\},\ Y = \{3, 4\},\ Z = \{5, 6\}$ and $m = \{m_1 = (1, 3, 5), m_2 = (2, 4, 6), m_3 = (1, 4, 5)\}$.

total) and the tuple states $i_0^{m_t}$ is 3 (there are $3M$ of them). The other type can be seen among the states with subscript 1 (see Figure 3.3). The top state i_1's average recovery time is 3 (there are $3q$ of them) and for tuple states below $i_1^{m_t}$ is 4 (there are $3M$ of them). Hence, the average recovery time of the whole system is:

$$\frac{3q \times 4 + 3M \times 3 + 3q \times 3 + 3M \times 4}{9q(M+1)} = \frac{21(q+M)}{9q(M+1)}$$

which is exactly equal to the bound *ert*. Finally, closure of *LS* is ensured by the self-loops.

($\Leftarrow$) Now, we show that if we have a solution for the weak-stabilizing repair instance, we can find a solution for the corresponding $3DM$ instance. The bound above, *ert*, is in fact the minimum average recovery time. The obvious way to reduce the average recovery time of our instance is to eliminate loops. Observe that transitions $(i_0^{m_t}, i_0)$ cannot be removed because they are the only way that states $i_1^{m_t}$ could converge to *LS*. On the other hand, for states i_0 to converge, they should have at least one outgoing transition. Particularly, those states must have at least one

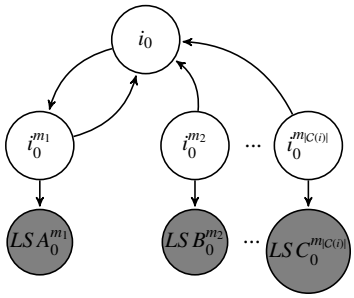

Figure 3.2: Connected blocks with subscript 0.

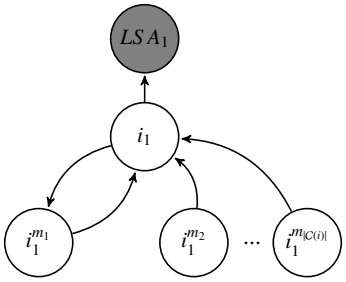

Figure 3.3: Connected blocks with subscript 1.

loop attached to them. Hence, the minimum recovery time is achieved when they have exactly one loop attached to them. As a result, the solution to the $3DM$ instance is the set of tuples m_t for which transitions $(i_0, i_0^{m_t})$, $(j_0, j_0^{m_t})$, $(k_0, k_0^{m_t})$ exist (have a loop attached to them). For example, in Figure 3.1, $(1_{0/1}^{m_3}, 1_{0/1})$, $(4_{0/1}^{m_3}, 4_{0/1})$, $(5_{0/1}^{m_3}, 5_{0/1})$ are the only transitions that can be removed without violating convergence, correctly suggesting that $m_{sol} = \{m_1, m_2\}$ is the solution to the $3DM$ instance.

Note that the outgoing/incoming transitions of the six states presenting a tuple are in the same group, so they are either removed together or kept together. In other words, all three coordinates of a tuple are simultaneously eliminated or selected implying that the loops remaining in the program correctly represent the tuples of a solution.

3.3 The Complexity of Synthesizing Weak-Stabilizing Protocols with Certain Average Recovery Time

In this section, we prove that the synthesis problem defined in Subsection 3.1.2 is NP-complete.

Theorem 2 *The problem of* synthesizing *a weak stabilizing program with a constrained average recovery time is NP-complete.*

Observe that synthesizing a weak-stabilizing program from scratch for a given set *LS* of legitimate states is equivalent to repairing a weak-stabilizing program with *LS* whose set of transitions is *maximal*. That is, the set of transitions includes all possible transitions (and their groups) except for the ones that violate the closure of *LS*.

Definition 11 *A weak-stabilizing program* $dp = \langle \Pi_{dp}, T_{dp} \rangle$ *is* maximal *in an asynchronous setting if*

$$T_{dp} = \{(s, s') \mid \forall s, s' \in S \ \ s.t.\ s \text{ and } s' \text{ differ in one variableonly}\} - \{\mathcal{G}(s, s') \mid s \in LS \wedge s' \notin LS\}$$

Before we present the proof, we note that the mapping presented in proof of Theorem 1 cannot be applied to the synthesis case, as the mapped program is not maximal. For example, it lacks transition $(1_0, 2_0^{m_2})$ where neither itself nor its group transitions violate closure. Thus, in order to make our mapped instance a maximal program, we must find a way to cause at least one of the transitions in $\mathcal{G}((1_0, 2_0^{m_2}))$ violate closure. One way to defeat this problem is to add the source state (1_0) to *LS* and leave the destination state $2_0^{m_2}$ in non-legitimate states. However, that will cause transitions $(1_0, 1_0^{m_1}), (1_0, 1_0^{m_3})$ to be removed which is not desirable. The other solution is to create group transition(s) for *only* $(1_0, 2_0^{m_2})$ that violate closure. We demonstrate through an example how one can eliminate some transitions appearing in a (possibly maximal) program by adding variables and processes (and inevitably transitions of the new process) to the system.

Consider a system consisting of a single process π_1, where $R_{\pi_1} = W_{\pi_1} = v_1$ with domain $D_{v_1} = [0, 2]$ and $LS = \{\langle 0 \rangle\}$. Each $\langle\rangle$ denotes a state. the corresponding maximal program dp_{max} contains 4 transitions:

$$T_{\pi_1,max} = \{(\langle 1 \rangle, \langle 0 \rangle), (\langle 2 \rangle, \langle 0 \rangle), (\langle 1 \rangle, \langle 2 \rangle), (\langle 2 \rangle, \langle 1 \rangle)\}$$

We examine a situation in which we require a maximal program where $(\langle 1 \rangle, \langle 2 \rangle) \notin T_{\pi_1,max}$. In a system with the above specification, $(\langle 1 \rangle, \langle 2 \rangle)$ cannot be avoided in the maximal program since it does not violate closure and it has no group transitions to do so. For this purpose, we add v_2 with $D_{v_2} = [0, 1]$ to the system such that $v_2 \notin R_{\pi_1}$. Since our model does not permit blind write, $v_2 \notin W_{\pi_1}$ is also true. As a result, π_2 is introduced ($R_{\pi_2} = W_{\pi_2} = \{v_2\}$). The new state space will consist of 6 states. Furthermore, transitions $(\langle 1, 0 \rangle, \langle 2, 0 \rangle)$ and $(\langle 1, 1 \rangle, \langle 2, 1 \rangle)$ of π_1 will belong to the same group. The first and second integers in $\langle, \rangle$ denote values of v_1 and v_2 respectively. If we choose $LS = \{\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 1 \rangle \in LS\}$, transition $(\langle 1, 1 \rangle, \langle 2, 1 \rangle)$ of π_1 will

violate closure. Consequently, both transitions $(\langle 1,0\rangle, \langle 2,0\rangle), (\langle 1,1\rangle, \langle 2,1\rangle)$ cannot exist in the corresponding maximal program dp'_{max}. The transition relation of π_1 and π_2 in the new maximal program are as follows:

$$\begin{aligned} T'_{\pi_1,max} = \{&(\langle 1,0\rangle, \langle 0,0\rangle), (\langle 2,0\rangle, \langle 1,0\rangle), (\langle 2,0\rangle, \langle 0,0\rangle),\\ &(\langle 1,1\rangle, \langle 0,1\rangle), (\langle 2,1\rangle, \langle 1,1\rangle), (\langle 2,1\rangle, \langle 0,1\rangle)\}\\ T'_{\pi_2,max} = \{&(\langle 0,0\rangle, \langle 0,1\rangle), (\langle 1,0\rangle, \langle 1,1\rangle), (\langle 2,0\rangle, \langle 2,1\rangle)\}. \end{aligned}$$

Note that it is not always possible to create groups that violate closure for a specific transition without affecting other transitions (whether they belong to the same process or not). The main challenge of this work was to find a maximal program with a substructure similar to Figure 3.1.

Proof of membership to NP is identical to that of Theorem 1.

Proof of NP-hardness. To prove that synthesis is NP-hard, we provide a mapping from $3DM$ to a maximal weak-stabilizing program. We present a polynomial-time mapping from an instance of $3DM$ to an instance of the synthesis problem, a distributed maximal weak-stabilizing program $dp = \langle \Pi_{dp}, T_{dp}\rangle$, legitimate state set LS, and

$$ert = \frac{(18M+30)q^2 + (21M+11)q + 4M}{36(M+1)q^2 + 12(M+1)q}.$$

Variables. $V_{dp} = \{v_1, v_2, v_3, v_4\}$, where, $D_{v_1} = [0, M]$, $D_{v_2} = [0, 9q-1]$, $D_{v_3} = [0, 6q+1]$, and $D_{v_4} = [0, 1]$.

States. The state space of our instance is the set of all valuations of variables, resulting in $108(M+1)q^2 + 36(M+1)q$ states which is polynomial in the size of the $3DM$ instance.

Processes. We declare 4 processes $\pi_1, \cdots, \pi_4$, with the following read/write restrictions:

$$\begin{aligned} R_{\pi_1} &= \{v_1, v_4\}, W_{\pi_1} = \{v_1\}\\ R_{\pi_2} &= \{v_1, v_2\}, W_{\pi_2} = \{v_2\}\\ R_{\pi_3} &= \{v_1, v_2, v_3\}, W_{\pi_3} = \{v_3\}\\ R_{\pi_4} &= \{v_1, v_4\}, W_{\pi_4} = \{v_4\} \end{aligned}$$

Starting from $V = \{v_1, v_2\}$, $\Pi = \{\pi_1, \pi_2\}$ with specifications defined above, we illustrate how to design a distributed system whose maximal weak-stabilizing program has a substructure resembling Figure 3.1.

First, we determine LS and transition relations of π_1, π_2. Similar to Figure 3.1, we have $6q+6M$ non-legitimate states. All other states are in LS. The valuation of v_1, v_2 in non-legitimate states and the transition relation T_{π_1} (Equation 3.1) also remain the same. We modify T_{π_2} to contain transitions only among the states in LS^*:

$$T_{\pi_2} = \{(s, s') \mid s, s' \in LS^*, s, s' \text{differ only in value of } v_2\}$$

For now, it appears that the non-legitimate states do not converge (See Figure 3.4). As we will discuss shortly, this problem is solved by including variable v_3 and process π_3 to the system. Figure 3.4 cannot represent a maximal program since there are many transitions that can exist without violating closure, e.g., $(1_0, 2_0)$. To construct a maximal program with the desired structure, we need to group the missing transitions with some other transitions that violate closure.

Including v_3, π_3 ($gadget_{v_3}$): Variable v_3 and process π_3 are incorporated in our mapping to preserve two-way transitions (loops) between every two state in $\{i_{0/1}, i_{0/1}^{C(i)}\}$ and eliminate all other transitions to/from them.

We group the undesirable transitions of π_1 and π_2 with transitions that violate closure. To do so, we append $\forall v \in [0, 6q+1],\ v_3 = v$ to all states in the state space of $\{v_1, v_2\}$ and call it $gadget_v$. The transitions in each gadget form a group with the corresponding transitions of every other gadget because π_1 and π_2 cannot read v_3. We add a second subscript v to the label of every state in $gadget_v$ when referring to them. In every gadget $v \in [1, 3q]$, in addition to LS^*, we will have the following states in LS: $i_{0,v}$, $i_{0,v}^{C(i)}$. Similarly, $\forall v \in [3q+1, 6q]$ we have additional states $i_{1,v}$, $i_{1,v}^{C(i)}$ in LS (see Figure 3.5). From this point forward, we add LS to their superscripts for clarity.

Notice that the reason the mapping in the proof of weak repair works is that states i_0 and $i_1^{m_t}$ are not directly connected to LS. To prevent transitions of form (i_0, LS) and $(i_1^{m_t}, LS)$ for π_1 or π_2, we flip the non-legitimate and LS states of $gadget_0$, in $gadget_{6q+1}$ (see Figure 3.6). In Figure 3.6, state $\overline{LS}_{V,0}^*$ represents a graph with $9qM - 3q - 6M$ non-legitimate states with all possible transitions of π_1 and π_2.

The problem that arises by adding v_3 and π_3 to the system is that inter-gadget transitions of π_3 will cause states $i_{0,0}$ and $i_{1,0}^{m_t}$ to connect directly to LS states of other gadgets ($i_{0,v}^{LS}$ and $i_{0,v}^{LS}$, $v \neq 0$, respectively). To eliminate these transitions, we need to add another variable and process.

Including v_4, π_4 ($Gadget_{v_4}$): Let $Gadget_0$ and $Gadget_1$ be all the $9q(6q+2)(M+1)$ states that we had so far in the mapped program appended with $v_4 = 0$ and $v_4 = 1$, respectively. A third subscript in a state's label shows the value of v_4 in that state (equivalently, which Gadget it belongs to).

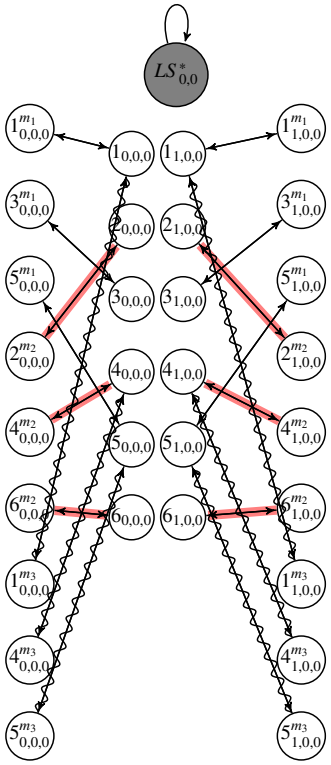

Figure 3.4: $v_3 = v_4 = 0$ in $gadget_0 \in Gadget_0$.

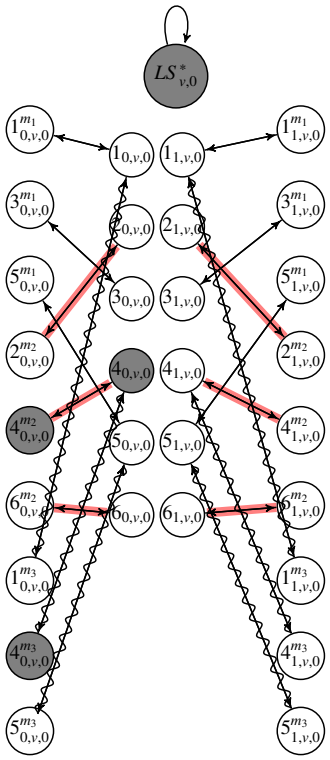

Figure 3.5: $v_3 \in [1, 6q], v_4 = 0$ in $gadget_v \in Gadget_0$.

In $Gadget_0$, everything remains the same, as described before. However, in $Gadget_1$, states $i^{LS}_{0,0,1}$ and $i^{m_t,LS}_{1,0,1}$ are the only states in LS (see Figure 3.7). All other gadgets $gadget_v$, $v \in [1, 6q+1]$ in $Gadget_1$ have the same structure as in Figure 3.7 except that they have no LS states. Observe that there are only one-way transitions of π_1 from non-LS to LS states. This will not affect gadgets in $Gadget_0$ where $v_4 = 0$ because π_1 can read v_4 which is 1 in $Gadget_1$. Transitions $(i_{0,0,1}, i^{LS}_{0,v,1}), v \in [1, 3q]$ and $(i^{m_t}_{1,0,1}, i^{m_t,LS}_{1,v,1}), v \in [3q+1, 6q]$ of π_3 in $Gadget_1$ violate closure, so do their corresponding transitions in $Gadget_0$ (in the same group). This implies that the only way for states $i_{0,0,0}$ and $i^{m_t}_{1,0,0}$ to directly connect to LS is through transitions $(i_{0,0,0}, i^{LS}_{0,0,1})$ and $(i^{m_t}_{1,0,0}, i^{m_t,LS}_{1,0,1})$ of π_4. That is not possible since their group transitions $(i^{LS}_{0,v,0}, i_{0,v,1})$ and $(i^{m_t,LS}_{1,v,0}, i_{1,v,1})$, $(v \in [1, 6q])$ violate closure.

Note that π_1 should be able to read v_4, otherwise the transitions of π_1 in $Gadget_0$ will inevitably be eliminated due to violation of closure by their group transitions $(i^{LS}_{0,0,1}, i^{C(i)}_{0,0,1})$ and $(i^{m_t,LS}_{1,0,1}, i_{1,0,1})$ in $Gadget_1$.

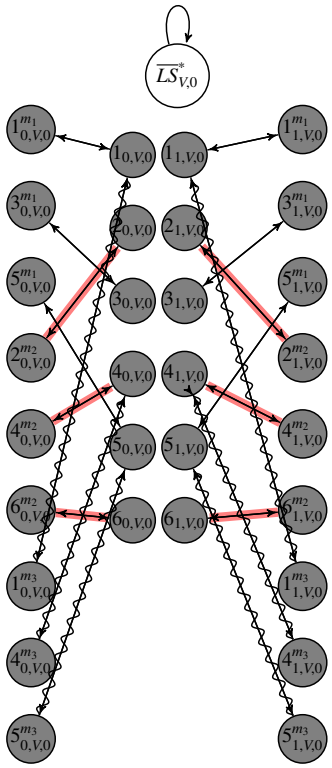

Figure 3.6: $V = v_3 = 6q + 1, v_4 = 0$ in $gadget_{6q+1} \in Gadget_0$.

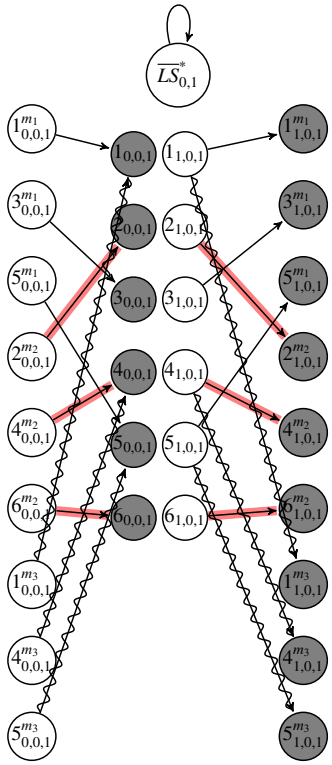

Figure 3.7: $= v_3 = 0, v_4 = 1$ in $gadget_0 \in Gadget_1$.

Legitimate States. The list of states in LS are summarized below with the number of them in brackets:

$\{LS^*_{v,0} \mid v \in [0, 6q]\}$	$[(9Mq + 3q$
	$-6M)(6q + 1)]$
$\{i^{LS}_{0/1,6q+1,0} \mid i \in U,\ m_t \in m\}$	$[6q + 6M]$
$\{i^{LS}_{0,v,0},\ i^{C(i),LS}_{0,v,0} \mid i \in U,\ v \in [1, 3q]\}$	$[3q + 3M]$
$\{i^{LS}_{1,v,0},\ i^{C(i),LS}_{1,v,0} \mid i \in U,\ v \in [3q + 1, 6q]\}$	$[3q + 3M]$
$\{i^{LS}_{0,0,1},\ i^{C(i),LS}_{1,0,1} \mid i \in U\}$	$[3q + 3M]$

We now have to show that our mapped instance has a solution if and only if there is a solution to the $3DM$ problem. To this end, note that $gadget_0$ of $Gadget_0$ has the same structure as the weak repair instance in Figure 3.1. Thus, all arguments in the proof of weak repair apply here. Hence, it will suffice to show how to compute the new bound ert. The bound is obtained by repairing the maximal program, i. e. removing as many loops as possible. All loops are removable except for at least one between tuple states and element states as mentioned in the proof of weak repair. Therefore, $gadget_0$ of $Gadget_0$ has $3q + 3M$ states with average recovery time equal to 4 and the same number with average recovery time of 3. All other states (total of $18q(6q + 2)(M + 1) - 9q(M + 1)$) in other gadgets are either in LS for which average recovery time is 0, or are directly connected to LS (by breaking the loops their average recovery time is 1). The sum of expected recovery times of states will add up to:

$$\begin{aligned}&[(90 + 54M)q^2 + (63M + 12)q + 12M] \times 1 + \\ &(3q + 3M) \times 3 + (3q + 3M) \times 4 \\ &= (54M + 90)q^2 + (63M + 33)q + 12M\end{aligned}$$

Finally, the average recovery time of the repaired maximal program that determines the bound ert is:

$$ert = \frac{(18M + 30)q^2 + (21M + 11)q + 4M}{36(M + 1)q^2 + 12(M + 1)q}$$

3.4 Polynomial-time Heuristic and Case Studies

Following our NP-completeness results, in this section, we present a polynomial-time heuristic that can be employed both for synthesis and repair.

3.4.1 The Heuristic

We take into account two factors that contribute to average recovery time: (1) existence of *loops* in the transition system, and (2) the position of loops with regard to LS states. In general, loops increase the average recovery time of its successors. Thus, it is beneficial to decrease average recovery time of states closer to LS that more states depend on them. Due to grouping of transitions, loops may not be avoidable. Our strategy is to eliminate as many loops as possible in near proximity of LS.

Algorithm 2 takes as input a stabilizing program dp, and a set LS of legitimate states. In the case of synthesis, dp must be maximal. Algorithm 2 works as follows. We consider $Diam + 1$

classes of states, where *Diam* is the diameter of the underlying graph of dp. States are assigned to a class based on their shortest path distance to *LS* states. If a state s has distinct shortest path distances to different states in *LS*, it will be assigned to several classes accordingly. Classification can be done by performing *BFS* several times, each time starting from a state in *LS*. After classification, we keep a Boolean value for every edge that shows whether it has been visited before or not. Initially, this value is false for all edges. Starting from class 1, first, we store transitions in a priority queue based on how much average recovery time will decrease if that transition and its group are deleted. Then, we remove only transitions (and their group) of form (s, t), where $s \in d[i]$, $t \notin LS$, and $t \notin d[i-1]$, that have not been visited before. Every time a group of transitions are removed, we check if convergence is violated. If so, we put them back and mark that group as *visited*. We repeat this step for all classes up to $d[Diam]$. Condition $t \notin d[i-1]$ tends to keep a shorter path and lets longer paths to be removed.

Algorithm 2 Synthesis/Repair Heuristic

```
1:  Input: dp, LS
2:  d[0..Diam(dp)] ← BFS(dp, LS)
3:  for i = 1 to Diam(dp) do
4:      pq ← PriorityQueue({(s,t) | s ∈ d[i], (s,t) ∈ T_d p})   ▷ Keep transitions based on effect on avg recovery
    time in priority queue.
5:      while ¬pq.empty do
6:          edge ← pq.get()                                    ▷ edge[0]=s, edge[1]=t
7:          if edge[1] ∉ LS ∧ ¬Visited(edge) ∧ edge[1] ∉ d[i − 1] then
8:              DeleteTransitionGroup(dp, edge)
9:              if ¬Converge(dp) then
10:                 InsertTransitionGroup(dp, edge)
11:                 VisitedTransitionGroup(edge)
12:             end if
13:         end if
14:     end while
15: end for
```

The total running time of our algorithm is $O(n(n+m) + 2m + gn(n+m) + gn^{2.3})$ contributed by classification, visiting edges and calculation of average recovery time. In the worst case $g = O(m)$, while in an asynchronous setting $m = O(nlogn)$. To conclude, Algorithm 2 has running time $O(n^3 log^2 n + n^{3.3} logn)$.

One might think keeping transitions in a priority queue considerably improves average recovery time. Although there are cases for which the above statement is true, in our case studies we saw only negligible impact (even in cases detrimental) with evident average recovery time calculation overhead. The results in tables 3.1, 3.2, and 3.3 come from an implementation without a priority queue.

P	Algorithm 2	Gradinariu et al [28]	Synthesis
2	0.89	1	9 (ms)
3	1.62	1.85	40 (ms)
4	2.01	2.21	0.5 (s)
5	2.87	2.82	5 (s)
6	4.06	3.34	97 (s)
7	3.89	3.88	26 (m)

Table 3.1: Vertex Coloring (line)

P	Algorithm 2	Adamek et al [1]	Synthesis
3	0.25	0.56	20 (ms)
4	0.5	0.5	273 (ms)
5	0.69	0.36	2 (s)
6	1.13	0.23	19 (s)
7	1.44	0.14	182 (s)

Table 3.2: Dining Philosophers (tree)

P	Algorithm 2	Devismes et al [12]	Synthesis
3	0.25	0.25	10 (ms)
4	2.02	2.63	2 (s)
5	1.49	1.49	0.2 (s)
6	7.99	9.05	24 (hr)
7	3.64	3.64	11 (s)

Table 3.3: Token circulation (ring)

3.4.2 Case Studies and Experimental Results

We ran Algorithm 2 on three problems: vertex coloring, token circulation in rings and dining philosophers. We used a system with 8GB RAM and Intel Core i5 2.60GHz CPU for our experiments. A brute-force approach for determining optimum average recovery time for our instances was not feasible on the system used in our experiments. We compared our results with average recovery time of existing stabilizing algorithms for the three problems in the literature using the technique in [21]. In all case studies, we used exactly the same setting (processes, read/write restrictions, variables, variable domains and *LS*) as described in the existing algorithms. In most cases, the output of our heuristic provides lower average recovery time. For each case study, we present an example of a program that our algorithm synthesizes.

Case Study 1: Vertex Coloring

The *vertex coloring* problem asks for a coloring of vertexes from a set of given colors such that no to adjacent vertices have identical colors. We compare our results with the deterministic weak-stabilizing algorithm of [28] (see Table 3.1) for the line topology, where P is the number of processes. As can be seen, in half cases, the program synthesized by Algorithm 2 outperforms the algorithm proposed in [28], as far as the average recovery time is concerned. The synthesized program for a line graph of size 3 is as follows:

$D_{v_i} = [0, 2],\ i = 1, 2, 3,\ LS = \{\langle 1, 2, 1\rangle, \langle 2, 1, 2\rangle\}$
$W_{\pi_1} = \{v_1\},\ R_{\pi_1} = \{v_1, v_2\}$
$W_{\pi_2} = \{v_2\},\ R_{\pi_2} = \{v_1, v_2, v_3\}$
$W_{\pi_3} = \{v_3\},\ R_{\pi_3} = \{v_2, v_3\}$

$$T_{\pi_1} = \{(\langle 0, 1\rangle, \langle 2, 1\rangle), (\langle 0, 2\rangle, \langle 1, 2\rangle), (\langle 2, 2\rangle, \langle 1, 2\rangle), (\langle 1, 1\rangle, \langle 2, 1\rangle), (\langle 0, 0\rangle, \langle 1, 0\rangle), (\langle 0, 0\rangle, \langle 2, 0\rangle)\}$$

$$T_{\pi_2} = \{(\langle 0, 0\rangle, \langle 0, 1\rangle), (\langle 1, 0\rangle, \langle 1, 2\rangle), (\langle 1, 1\rangle, \langle 1, 2\rangle), (\langle 2, 2\rangle, \langle 2, 1\rangle), (\langle 0, 0\rangle, \langle 0, 2\rangle), (\langle 2, 0\rangle, \langle 2, 1\rangle)\}$$

Case Study 2: Dining Philosophers

The *dining philosophers* is a classic problem in concurrent algorithms design that deals with synchronization. A solution to this problem must guarantee two conditions: (1) neighbor processes should not enter their critical sections simultaneously (in the same state), and (2) each process that requests to enter its critical section must eventually be allowed to do so.

We compared our results to the stabilizing dining philosophers of Adamek et al. [1] (see Table 3.2). We considered tree structures with height 1 in our experiments. Our results show that Adamek's algorithm has lower average recovery time for trees with degree higher than 3.

$D_{v_i} = [0, 1],\ i = 1, 2, 3$
$LS = \{\langle 0, 0, 0\rangle, \langle 0, 0, 1\rangle, \langle 0, 1, 0\rangle, \langle 1, 0, 1\rangle, \langle 1, 1, 0\rangle, \langle 1, 1, 1\rangle\}$
$W_{\pi_1} = \{v_1\},\ R_{\pi_1} = \{v_1, v_2, v_3\}$
$W_{\pi_2} = \{v_2\},\ R_{\pi_2} = \{v_1, v_2\}$
$W_{\pi_3} = \{v_3\},\ R_{\pi_3} = \{v_1, v_3\}$

$$
\begin{aligned}
T_{\pi_1} =&\{(\langle 1,0,1\rangle, \langle 0,0,1\rangle), (\langle 0,0,1\rangle, \langle 1,0,1\rangle), (\langle 0,1,0\rangle,\\
&\langle 1,1,0\rangle), (\langle 0,1,1\rangle, \langle 1,1,1\rangle), (\langle 1,1,0\rangle, \langle 0,1,0\rangle),\\
&(\langle 1,0,0\rangle, \langle 0,0,0\rangle)\}
\end{aligned}
$$

$$
\begin{aligned}
T_{\pi_2} =&\{(\langle 1,1,1\rangle, \langle 1,0,1\rangle), (\langle 0,1,1\rangle, \langle 0,0,1\rangle), (\langle 1,0,0\rangle,\\
&\langle 1,1,0\rangle), (\langle 0,0,0\rangle, \langle 0,1,0\rangle), (\langle 0,1,0\rangle, \langle 0,0,0\rangle),\\
&(\langle 1,0,1\rangle, \langle 1,1,1\rangle)\}
\end{aligned}
$$

$$
\begin{aligned}
T_{\pi_3} =&\{(\langle 1,1,1\rangle, \langle 1,1,0\rangle), (\langle 0,1,1\rangle, \langle 0,1,0\rangle), (\langle 1,1,0\rangle,\\
&\langle 1,1,1\rangle), (\langle 0,0,1\rangle, \langle 0,0,0\rangle), (\langle 0,0,0\rangle, \langle 0,0,1\rangle),\\
&(\langle 1,0,0\rangle, \langle 1,0,1\rangle)\}
\end{aligned}
$$

Case Study 3: Token circulation in anonymous networks

Our last case study is the famous *token circulation* problem which ensures that only one process holds the token and each process holds the token infinitely often. Table 3.3 compares the average recovery time of our synthesized solutions to that of the algorithm in [12], where P is the number of processes. As can be seen, in most cases, the synthesized program has the same average recovery time as the algorithm proposed in [12]. We note that since the second condition requires the existence of a cycle in LS, we slightly modified Algorithm 2 to preserve a cycle in LS.

Table 3.3 shows a trend that for rings with odd length, Algorithm 2 and [12] have the same average recovery time. This is because they both generate the maximal program. Algorithm 2 could not remove any group of transitions due to violation of convergence suggesting that the maximal program is the only stabilizing program.

Algorithm 2 synthesizes a strong-stabilizing program for a ring consisting of 4 processes which means the program can be used on non-anonymous networks [12]. It was not feasible to analyze the output for even-length networks of size 6 and higher due to very long execution time.

$D_{v_i} = [0,1],\ i = 1,2,3$
$LS = \{\langle 0,0,0\rangle, \langle 0,0,1\rangle, \langle 0,1,0\rangle, \langle 1,0,1\rangle, \langle 1,1,0\rangle, \langle 1,1,1\rangle\}$
$W_{\pi_1} = \{v_1\},\ R_{\pi_1} = \{v_1, v_2, v_3\}$
$W_{\pi_2} = \{v_2\},\ R_{\pi_2} = \{v_1, v_2\}$
$W_{\pi_3} = \{v_3\},\ R_{\pi_3} = \{v_1, v_3\}$

$$T_{\pi_1} = \{(\langle 0,0,0\rangle, \langle 1,0,0\rangle), (\langle 1,0,1\rangle, \langle 0,0,1\rangle), (\langle 1,1,1\rangle, \langle 0,1,1\rangle), (\langle 0,1,0\rangle, \langle 1,1,0\rangle)\}$$

$$T_{\pi_2} = \{(\langle 1,1,1\rangle, \langle 1,0,1\rangle), (\langle 0,0,0\rangle, \langle 0,1,0\rangle), (\langle 1,1,0\rangle, \langle 1,0,0\rangle), (\langle 0,0,1\rangle, \langle 0,1,1\rangle)\}$$

$$T_{\pi_2} = \{(\langle 0,1,1\rangle, \langle 0,1,0\rangle), (\langle 0,0,0\rangle, \langle 0,0,1\rangle), (\langle 1,1,1\rangle, \langle 1,1,0\rangle), (\langle 1,0,0\rangle, \langle 1,0,1\rangle)\}$$

Chapter 4

Automated Fine Tuning of Probabilistic Self-Stabilizing Algorithms

In this chapter, we study fine tuning of probabilistic-stabilizing programs. The fine tuning problem assigns probability distributions to transition executions such that the average recovery time of the program is minimized. In this chapter, first we formally define the fine tuning problem. Then, We show how stabilizing programs can be modelled by absorbing DTMCs. We use the mapping to reduce the calculation of average recovery time to computing the weighted sum of elements in the inverse sub-matrix of the transition probability matrix to generate a symbolic expression. We find the real roots of this expression using existing methods and tools to find the optimum probability values.

4.1 Fine Tuning of Probabilistic Models

Our *tuning* problem takes as input the parametric DTMC of a probabilistic-stabilizing distributed program $\mathcal{PD}$ and outputs a valid evaluation function that minimizes the expected recovery time of $\mathcal{PD}$.

Instance. A probabilistic-stabilizing program modeled by a parametric DTMC $\mathcal{PD} = (S, S_0 = S, U, \iota_{init}, \mathbf{P}, \{LS\})$.

Fine tuning problem. An evaluation function $eval_{min} : U \rightarrow \mathbb{R}$ that is valid for $\mathcal{PD}$ and minimizes its expected recovery time. That is,

$$eval_{min} = \operatorname*{argmin}_{eval} ERT(\mathcal{PD})$$

4.2 Calculating the Expected Recovery Time of Stabilizing Programs

We use the theory of Absorbing Discrete-Time Markov Chains to reduce the computation of expected recovery time of stabilizing programs to matrix inversion. In this section, we first define absorbing DTMCs. Next, we present a mapping from stabilizing programs to absorbing DTMCs such that the expected absorption time of transient states in the absorbing DTMC is equivalent to the expected recovery time of the stabilizing program.

4.2.1 Absorbing Discrete-Time Markov Chains

A DTMC should have two properties, which we call *absorption* and *reachability*, to be considered absorbing.

Definition 12 (Absorbing DTMC) *A DTMC* $\mathcal{D} = (S, S_0, \iota_{init}, \mathbf{P}, L)$ *is* absorbing *iff*

- *Absorption: It contains at least one* absorbing state *from where there exists one and only one self-loop:*
$$\exists s \in S : (\mathbf{P}(s, s) = 1 \ \wedge \ \forall s' \neq s : \mathbf{P}(s, s') = 0)$$
- *Reachability: All non-absorbing (transient) states can reach at least one absorbing state.* ■

We denote the set of absorbing states of an absorbing DTMC by $\mathcal{A}$. Note that transient states are not required to be in the set of initial states S_0. The reachability condition requires for each transient state s the existence of a computation that once arrives in s at time i ($s_i = s$ for $i \geq 0$), it should be able to reach an absorbing state at time $j > i$ ($s_j \in \mathcal{A}$).

Definition 13 (Absorption Time) *The* absorption time *of a state $s \in S$ in a computation $\sigma = s_0 s_1 \cdots$ starting from s is:*

$$T(\sigma_s) = min\{i \mid s_0 = s,\ 0 \leq k < i,\ s_k \notin \mathcal{A},\ s_i \in \mathcal{A}\}$$

We consider $T(\sigma_s) = \infty$ for computations that never reach an absorbing state. ■

Since there can be more than one computation starting from s that reaches absorbing states, each with a distinct absorption time, the absorption time of a state is a discrete random variable and we can calculate its expected value. To do so, we use the canonical representation of the transition probability matrix of the absorbing DTMC. The canonical form of representing an absorbing Markov chain with t transient and r absorbing states is as follows [30]:

$$\mathbf{P} = \begin{bmatrix} Q_{t\times t} & R_{t\times r} \\ 0_{r\times t} & I_{r\times r} \end{bmatrix}, \tag{4.1}$$

where Q is a $t \times t$ sub-matrix of one-step transition probability among transient states, R is a $t \times r$ sub-matrix of one-step transition probability from transient states to absorbing states, 0 is a $r \times t$ zero sub-matrix and I is a $r \times r$ identity matrix. The expected absorption time of the transient states of an absorbing DTMC can be computed as follows [30]:

$$N = \Big(\sum_{i=0}^{\infty} Q^i\Big)\tilde{\mathbf{e}} = (I - Q)^{-1}\tilde{\mathbf{e}}, \tag{4.2}$$

where $\tilde{\mathbf{e}}$ is a column vector of size t filled with ones, N is a column vector of size t such that $N(i)$ is the expected absorption time of the ith transient state.

4.2.2 Stabilizing Programs as Absorbing DTMCs

We exploit the two fundamental properties of stabilizing programs, convergence and closure, to model every stabilizing program (modelled by a DTMC $\mathcal{D}$) with an absorbing DTMC ($\mathcal{D}^*$). This mapping helps us employ Eq. 4.2 to compute the expected recovery time of stabilizing programs.

Closure $\implies$ Absorption: The closure property of stabilizing programs states that once a program reaches a legitimate state, it is trapped in legitimate states. Hence, we can assume each legitimate state is an absorbing state. We will justify this assumption shortly.

Convergence $\Longrightarrow$ Reachability: All types of convergence ensure that every non-legitimate state has at least one path to a legitimate state (now absorbing), which satisfies the reachability property of transient states of absorbing DTMCs in Def. 12.

The transition probability matrix of a stabilizing program with t non-legitimate and r legitimate states can be divided into 4 sub-matrices as follows if written in the proper order:

$$\mathbf{P} = \begin{array}{c} \\ \neg LS \\ LS \end{array} \begin{array}{c} \begin{array}{cc} \neg LS & \quad LS \end{array} \\ \left[\begin{array}{c|c} Q_{t\times t} & R_{t\times r} \\ \hline \mathbf{0}_{r\times t} & C_{r\times r} \end{array}\right] \end{array},$$

where Q, R, $\mathbf{0}$, and C are the one-step transition probability among $\neg LS - \neg LS$, $\neg LS - LS$, $LS - \neg LS$ and $LS - LS$ states, respectively. The closure property ensures that the transition probability from LS states to non-LS states is zero leaving the lower left quarter of $\mathbf{P}$ all zeros. Converting legitimate states to absorbing ones will modify $\mathbf{P}$ as follows:

$$\mathbf{P}^* = \begin{array}{c} \\ \neg LS \\ LS \end{array} \begin{array}{c} \begin{array}{cc} \neg LS & \quad LS \end{array} \\ \left[\begin{array}{c|c} Q_{t\times t} & R_{t\times r} \\ \hline \mathbf{0}_{r\times t} & I_{r\times r} \end{array}\right] \end{array},$$

where, $I_{r\times r}$ is the identity matrix. Observe that $\mathbf{P}^*$ is in the form of an absorbing DTMC (Eq. 4.1). It is now easy to draw a connection between the absorption time of transient states in $\mathcal{D}^*$ and the recovery time of states in $\neg LS$ in $\mathcal{D}$. As a matter of fact, they are equivalent (the absorption time of a transient state is the first time it reaches an absorbing state, just as the recovery time of a non-legitimate state is the first time it reaches a legitimate state). Hence, we use Eq. 4.2, Eq. 2.3, and the fact that legitimate states have *zero* recovery time to calculate the expected recovery time of a stabilizing program as follows:

$$ERT(\mathcal{D}^*) = \vec{init}(I - Q)^{-1}\tilde{\mathbf{e}}, \tag{4.3}$$

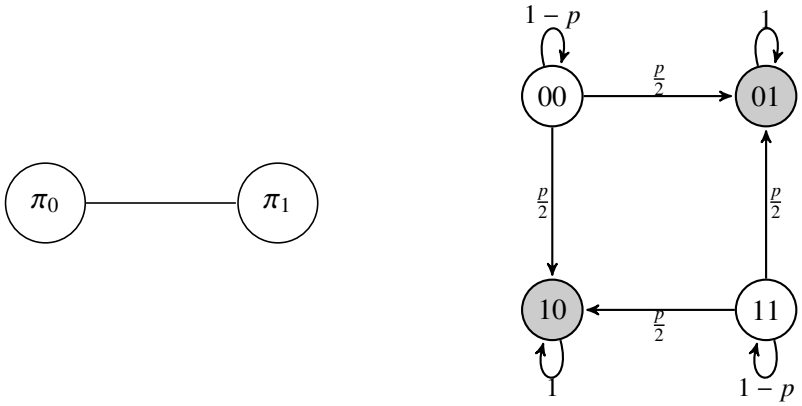

Figure 4.1: The DTMC of the vertex coloring Algorithm 1 (right) run on a graph with two vertices (left).

where, $\vec{init}$ is a $1 \times t$ row vector containing $\iota_{init}(s)$ for each $s \notin LS$.

Recall from Eq. 4.2 that $(I - Q)^{-1}\tilde{\mathbf{e}}$ produces a $t \times 1$ column vector N of expected recovery times of non-LS states. Thus, the dot product of $\vec{init}$ with N gives Eq. 2.3. For example, the average recovery time of the program of Figure 4.1 is derived as follows:

$$ERT(D) = \begin{bmatrix} \frac{1}{4} & \frac{1}{4} \end{bmatrix} . \begin{bmatrix} p & 0 \\ 0 & p \end{bmatrix}^{-1} . \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{2p}.$$

Discussion We modeled stabilizing programs with absorbing DTMCs to use the fundamental matrix (Eq. 4.2) to calculate the expected recovery time. Calculating the sum of powers of a matrix, especially for large powers, is computationally expensive. Eq. 4.2 becomes particularly more interesting in our case because it reduces calculating the sum of powers of a matrix to calculating the weighted sum of the elements of the inverse matrix. The latter does not involve finding the inverse itself explicitly and takes less computational time. We elaborate on the computation of weighted sum of elements of inverse matrix in Sec. 4.2.3. The similarity, however, implies that as far as absorption time of transient states (equivalently, recovery time of non-LS states) are concerned, we can represent the stabilizing program with $\mathbf{P}^*$ instead of $\mathbf{P}$.

4.2.3 A Symbolic Linear Algebraic Technique for Computing Recovery Time

In this section we explain how we compute $ERT(\mathcal{D}^*) = \vec{init}(I-Q)^{-1}\tilde{\mathbf{e}}$ as defined in Eq. 4.3. Observe that $ERT(\mathcal{D}^*)$ is the weighted sum of the elements of $(I-Q)^{-1}$. The dot product $(I-Q)^{-1}\tilde{\mathbf{e}}$ produces a column vector N whose elements are the sum of the elements in each row of $(I-Q)^{-1}$. Computing the dot product of $\vec{init}$ with N gives the weighted sum of the elements of N, which by linearity of addition is equal to the weighted sum of elements of $(I-Q)^{-1}$. In the sequel, we describe the techniques we used from the literature of symbolic computation.

We begin by estimating the size of the symbolic expression $\vec{init}(I-Q)^{-1}\tilde{\mathbf{e}}$, a rational function in the variable p. The matrix Q is filled with integer polynomials in the variable p. Let n be the dimension of Q, let d be the degree of Q (i.e., the maximal degree in p of all entries of Q), and let α be an upper bound for the number of bits in the binary representation of any integer coefficient of any entry of Q. The total size of (number of bits to represent) Q is then bounded by $n^2(d+1)\alpha$. Now consider $(I-Q)^{-1}$. By Cramer's rule, each entry $(I-Q)^{-1}_{i,j}$ is equal to $\pm B_{i,j}/\det(I-Q)$, where $B_{i,j}$ is a minor of $I-Q$ of dimension $n-1$ and $\det(I-Q)$ is the determinant of $I-Q$. The degrees of $B_{i,j}$ and $\det(I-Q)$ are thus bounded by $(n-1)d$ and nd respectively, so on the order of n times larger than the degree of Q. Moreover, the integer coefficients of $B_{i,j}$ and $\det(I-Q)$ are bounded in length by $O(n(\alpha+\log n+\log d))$ bits [24], so on the order of roughly a factor of n larger than the length of the integer coefficients in Q. This analysis shows that we can expect the size of the single rational function $\vec{init}(I-Q)^{-1}\tilde{\mathbf{e}}$ to be about the same as the size of the entire input matrix Q, and the size of the entire inverse $(I-Q)^{-1}$ will be on the order of n^2 times the size of Q. Because of the growth in degrees and bitlengths, computing the entire inverse $(I-Q)^{-1}$ explicitly, let alone storing it in memory, would become prohibitively expensive as n grows.

Fortunately, we can avoid the computation of $(I-Q)^{-1}$ entirely by exploiting a simple homomorphic imaging scheme. We choose a collection $\mathcal{X} = (x_i)_{0\le i\le nd+(n-1)d}$ of distinct integer points that are not roots of $\det(I-Q)$, and compute the rational numbers

$$\mathcal{Y} = \left((\vec{init}(I-Q)^{-1}\tilde{\mathbf{e}})\mid_{p=x_i}\right)_{0\le i\le nd+(n-1)d}.$$

Fast polynomial interpolation [23, Section 10.2] and rational function reconstruction [23, Section 5.7] can recover $\vec{init}(I-Q)^{-1}\tilde{\mathbf{e}}$ from the list of independent and dependent values $\mathcal{X}$ and $\mathcal{Y}$, respectively. By far the dominant cost of this scheme is to compute the evaluations $\mathcal{Y}$. To this end, note that

$$(\vec{init}(I-Q)^{-1}\tilde{\mathbf{e}})\mid_{p=x_i} = \vec{init}((I-Q)\mid_{p=x_i})^{-1}\tilde{\mathbf{e}},$$

that is, we can first evaluate $I-Q$ at the point $p=x_i$ to obtain an integer matrix $A=(I-Q)\mid_{p=x_i}$, then solve the nonsingular rational system $Av=\tilde{\mathbf{e}}$ for v, and finally compute the dot prod-

uct $\vec{init}\ v$. In our implementation we solved the systems $Av = \tilde{\mathbf{e}}$ using the Integer Matrix Library (IML) [10], a highly optimized C library for integer matrix computations. The evaluations $(I-Q)\mid_{p=x_i}$ and the reconstruction of the rational function $\vec{init}(I-Q)^{-1}\tilde{\mathbf{e}}$ from $\mathcal{X}$ and $\mathcal{Y}$ were performed using the Maple computer algebra system (`http://www.maplesoft.com`). We were able to call IML directly from Maple using Maple's `DefineExternal` facility to link to the library.

Once $ERT(\mathcal{D}^*)$ has been computed, the values of p in the interval $[0,1]$ that minimize the expected recovery time must be a subset of the set of real roots of the derivative of the numerator of $ERT(\mathcal{D}^*)$. These roots can be found using Maple's `realroots` or `RootFinding[Isolate]` procedures.

4.3 Experiments and Analysis

We use Eq. 4.3 to find the optimum probability value for two probabilistic-stabilizing algorithms: (1) Herman's token circulation in synchronous anonymous rings, and (2) Gradinariu and Tixeuil's vertex coloring of arbitrary graphs.

4.3.1 Vertex Coloring of Arbitrary Graphs

Algorithm 1 is a probabilistic-stabilizing vertex coloring algorithm designed for arbitrary graphs. The challenge is to find the optimum value of p that minimizes the expected recovery time. A higher value of p increases recovery time by reducing the possibility of making progress, while it decreases recovery time by reducing the probability of simultaneous execution of two enabled neighbors. Our experiments for line and ring structures of size up to 6 showed that the expected recovery time is of form

$$\frac{c_1}{c_2 p},$$

where $c_1, c_2 \in \mathbb{N}$. Therefore, the effect of reducing the probability of simultaneous execution prevails and $p = 1$ produces minimum expected recovery time in case of a central scheduler (See Table 4.1). In an anonymous network under a distributed scheduler Algorithm 1 does not work in a deterministic setting. In that case, p must be strictly less than 1.

Structure	Size	|Q|	p_{opt}	Ert_{opt}
Line	**3**	25	1	$\frac{146}{81p} = 1.80$
	4	77	1	$\frac{2357}{972p} = 2.42$
	5	237	1	$\frac{31295857}{10497600p} = 2.98$
	6	721	1	$\frac{8401071143}{2361960000p} = 3.56$
Ring	**3**	21	1	$\frac{1}{p} = 1$
	4	79	1	$\frac{228}{81p} = 2.81$
	5	233	1	$\frac{1690}{729p} = 2.32$
	6	715	1	$\frac{981097}{291600p} = 3.36$

Table 4.1: Randomized Vertex Coloring

4.3.2 Herman's Token Circulation

The token circulation problem ensures that only one process holds a token (privilege) at any time and every process is infinitely often granted the token. It's been shown that a non-probabilistic self-stabilizing algorithm for the token circulation problem in anonymous networks does not exist [4, 32].

Herman's probabilistic algorithm [32] (see Algorithm 3) is designed for distributed systems in which an odd number of identical processes are connected in a ring. Herman's algorithm breaks the symmetry by randomizing processes actions. It declares a binary variable per process. Each process looks at the value of its own variable and that of its left neighbour. If they are identical, the process holds a token and it sets its corresponding variable to 0 with probability p and to 1 with probability $1 - p$. Otherwise, it flips its value with probability 1. The size of the state space of this program is 2^n, where n is the number of processes. By taking advantage of topological symmetry in anonymous rings, we were able to reduce this size to $O(\frac{2^n}{n})$, which made the computations feasible. Since, approximately, every n distinct states of the state space represent the same topology. For instance, consider a ring of size 3. States $\langle 1, 0, 0\rangle$, $\langle 0, 1, 0\rangle$ and $\langle 0, 0, 1\rangle$ basically, exhibit similar behavior because the network is anonymous and the fact that which two variables are 0 and which one is 1 does not affect the global behavior of the system.

An interesting observation made in [37] was that when the size of network is greater than 9, $p = 0.5$ does not yield minimum worst-case expected recovery time anymore. We calculate the precise value of p that results in the minimum average-case expected recovery time for networks

Algorithm 3 Probabilistic-stabilizing Token Circulation (process i)

1: **Variable:** x_i : *boolean* $\in [0, 1]$
2: **Guarded Commands:**
$x_i = x_{i-1} \longrightarrow p : x_i := 0 \; + \; 1 - p : x_i := 1;$
$x_i \neq x_{i-1} \longrightarrow p : x_i := x_{i-1};$

Size	**|Q|**	$\mathbf{p_{opt}}$	$\mathbf{ERT_{opt}}$	$\mathbf{ERT_{p=0.5}}$	**diff(%)**
3	2	0.5	0.33	0.33	0
5	6	0.5	1.93	1.93	0
7	18	0.5	4.49	4.49	0
9	58	0.53	7.9210	7.9215	0.006
11	186	0.37,0.64	12.1020	12.2058	0.85
13	630	0.33,0.67	16.95	17.35	2.31
15	2190	0.31,0.69	22.46	23.34	3.77

Table 4.2: Herman's Randomized Token Circulation

of sizes $3 - 15$ (see Table 4.2). Based on our results, we see that for networks of size over 9, $p = 0.5$ is a local maximum and there are two points, one smaller and one larger than $p = 0.5$, that minimize expected recovery time. Furthermore, for larger networks a biased coin is more effective. We conjecture that as the network size grows, the two minimum points grow farther (one towards lower values and one towards larger values) and a biased coin can significantly reduce the expected recovery time.

Implementation Notes We should note that for a network with 15 processes, we used a machine with 64 cores and ran several instances of the computer algebra system Maple in parallel. Moreover, for rings of size 13 and 15, we found the optimum points by plugging values in the final symbolic expression.

Chapter 5

Automated Analysis of Impact of Scheduling on Performance of Self-Stabilizing Protocols

We study the effect of different scheduling schemes on possibility of convergence and performance of stabilizing distributed programs. There are several criteria to consider: *distribution*, *boundedness*, *enabledness* and *fairness* [17]. There are algorithms that stabilize under a probabilistic scheduler [32, 42], a fair scheduler [12] and a non-distributed scheduler [28]. Schedulers can also affect the performance of self-stabilizing algorithms [8].

In this chapter, we review the formal definitions of schedulers [17]. We propose a method to augment stabilizing algorithms with k-central and k-bounded scheduling policies. In particular, we studied the impact of schedulers on three vertex coloring stabilizing algorithms [28] and a few other variants considering both deterministic and probabilistic algorithms. Our experiments showed that in general, deterministic algorithms have lower average recovery time. We compared three randomization strategies, two static and one dynamic. The adaptive dynamic strategy had better performance in most cases with the advantage of no pre-tuning requirements.

5.1 Scheduler Types

Schedulers determine the degree of parallelism in a distributed program. They are specifically important in stabilizing programs as they affect both the possibility of convergence and convergence time. A detailed survey of schedulers in self-stabilization can be found in [17]. In this

section, we review the four classification factors of schedulers that we consider in this paper, namely *distribution*, *fairness*, *boundedness* and *enabledness* studied in [17].

A *scheduler d* is a function that associates to each distributed program a subset of its computations: $d : DP \rightarrow \mathcal{P}(\Sigma(dp))$, where $\mathcal{P}$ denotes the powerset. Thus, $d(dp)$ denotes the computations of dp constrained by scheduler d. A transition $t = (s, s')$ *activates* a process π in state s iff it updates at least one of the variables in W_π. *Act* associates to each transition $t = (s, s')$ the set of processes that it activates:

$$Act(s, s') = \{\pi \in \Pi \mid \exists v \in W_\pi : \ v(s) \neq v(s')\}$$

In a distributed program dp, we say that a process π is *enabled* in state s iff there exists a transition in dp that originates in s and activates π. Hence, the set of processes enabled in state s is given by:

$$En(s, dp) = \{\pi \in \Pi \mid \pi \text{ is enabled by dp in s}\}$$

5.1.1 Distribution

Distribution imposes spatial constraints on the selection of processes whose transitions will be executed.

Definition 14 (k-central) *Given a distributed system $G = (\Pi, E)$, a scheduler d is* k-central *iff:*

$$\forall dp \in DP : \forall \sigma = s_0 s_1 \cdots \in d(dp) : \forall i \in \mathbb{N} : \forall \pi_1 \neq \pi_2 \in \Pi :$$
$$[\pi_1 \in Act(s_i, s_{i+1}) \wedge \pi_2 \in Act(s_i, s_{i+1})] \rightarrow dist(\pi_1, \pi_2) > k$$

■

In other words, a k-central scheduler allows processes in distance at least k to execute simultaneously. In particular, 0-central and $diam(G)$-central schedulers are called *distributed* and *central* respectively. In the former, any subset of the enabled processes can be scheduled at any time. In the latter, a single process can execute at a time (i.e., the interleaving semantics).

5.1.2 Fairness

Schedulers are classified into four categories based on fairness assumptions. A *weakly fair* scheduler ensures that a continuously enabled process is eventually scheduled.

Definition 15 (weakly fair) *Given a distributed system $G = (\Pi, E)$, a scheduler d is* weakly fair *iff*

$\forall dp \in DP : \forall \sigma = s_0 s_1 \cdots \in \Sigma(dp) :$
$[\exists \pi \in \Pi : \exists i \geq 0 : \forall j \geq i : (\pi \in En(s_j, dp) \wedge \pi \notin Act(s_j, s_{j+1}))] \Rightarrow \sigma \notin d(dp)$ ■

A *strongly fair* scheduler ensures that a process that is enabled infinitely often is eventually scheduled.

Definition 16 (strongly fair) *Given a distributed system $G = (\Pi, E)$, a scheduler d is* strongly fair *iff*

$$\begin{aligned}&\forall dp \in DP : \forall \sigma = s_0 s_1 \cdots \in \Sigma(dp) :\\ &\quad [\exists \pi \in \Pi : \exists i \geq 0 : (\forall j \geq i : \exists k \geq j : \pi \in En(s_k, dp)) \wedge (\forall j \geq i : \pi \notin Act(s_j, s_{j+1}))] \Rightarrow\\ &\quad \sigma \notin d(dp)\end{aligned}$$

■

A third type of fairness is *Gouda fairness* which is beyond the scope of this paper. Any other scheduler that does not satisfy any type of fairness constraints is considered *unfair*.

5.1.3 Boundedness and Enabledness

A scheduler is *k-bounded* if it does not schedule a process more than k times between any two schedulings of any other process.

Definition 17 (k-bounded) *Given a distributed system $G = (\Pi, E)$, a scheduler d is* k-bounded *iff:*

$$\begin{aligned}\forall \pi \in \Pi : \exists k > 0 : \forall dp \in DP : \forall \sigma = s_0 s_1 \cdots \in d(dp) : \forall (i, j) \in \mathbb{N}^2 :\\ [[\pi \in Act(s_i, s_{i+1}) \wedge (\forall l < i : \pi \notin Act(s_l, s_{l+1}))] \Rightarrow\\ \forall \pi' \in \Pi \backslash \{\pi\} : |\{l \in \mathbb{N} \mid l < i \wedge \pi' \in Act(s_l, s_{l+1})\}| \leq k] \wedge\\ [[i < j \wedge \pi \in Act(s_i, s_{i+1}) \wedge \pi \in Act(s_j, s_{j+1}) \wedge\\ (\forall l \in \mathbb{N} : i < l < j \Rightarrow \pi \notin Act(s_l, s_{l+1}))] \Rightarrow\\ \forall \pi' \in \Pi \backslash \{\pi\} : |\{l \in \mathbb{N} \mid i \leq l < j \wedge \pi' \in Act(s_l, s_{l+1})\}| \leq k]\end{aligned}$$

■

A scheduler is *k-enabled* if a process cannot be enabled more than k times before being scheduled.

Definition 18 (k-enabled) *Given a distributed system G, a scheduler d is* k-enabled *iff:*

$$\begin{aligned}\forall dp \in DP : \exists k > 0 : \forall \sigma = s_0 s_1 \cdots \in d(dp) : \forall (i,j) \in \mathbb{N}^2 : \forall \pi \in \Pi : \\ [[\pi \in Act(s_i, s_{i+1}) \wedge (\forall l \in \mathbb{N}, l < i \Rightarrow \pi \notin Act(s_l, s_{l+1}))] \Rightarrow \\ |\{l \in \mathbb{N} \mid l < i \wedge \pi \in En(s_l, \sigma)\}| \leq k] \wedge \\ [[i < j \wedge \pi \in Act(s_i, s_{i+1}) \wedge \pi \in Act(s_j, s_{j+1}) \wedge \\ (\forall l \in \mathbb{N}, i < l < j \Rightarrow \pi \notin Act(s_l, s_{l+1}))] \Rightarrow \\ |\{l \in \mathbb{N} \mid i < l < j \wedge \pi \in En(s_l, \sigma)\}| \leq k] \wedge \\ [[pi \in Act(s_i, s_{i+1}) \wedge (\forall l \in \mathbb{N}, l > i \Rightarrow \pi \notin Act(s_l, s_{l+1}))] \Rightarrow \\ |\{l \in \mathbb{N} \mid l > i \wedge \pi \in En(s_l, \sigma)\}| \leq k]\end{aligned}$$

■

5.2 Augmenting a Distributed Program with a Scheduler

To concisely specify the behavior of a process π, we utilize a finite set of *guarded commands* ($\mathcal{G}_\pi$). A guarded command has the following syntax:

$$\langle label \rangle \ : \ \langle guard \rangle \ \rightarrow \ \langle statement \rangle;$$

The *guard* is a Boolean expression over the read-set of the process. The statement is executed whenever the guard is satisfied. Execution of guarded commands updates variables and causes transitioning from one state to another. In *probabilistic programs* commands are executed with a probability. Hence, transitions among states in the system are executed according to a probability distribution.

$$\langle label \rangle \ : \ \langle guard \rangle \ \rightarrow \ p_1 : \langle statement_1 \rangle \ + \ \cdots \ + \ p_n : \langle statement_n \rangle;$$

where

$$\sum_{i=1}^{n} p_i = 1$$

A guarded command is *enabled* if its guard evaluates to true. A process is enabled if at least one of its guarded commands is enabled. The set of guarded commands of a distributed program is formed by the union of the guarded commands of its constituent processes. In a parallel (i.e., simultaneous) execution, all enabled processes execute their enabled commands. In contrast, in a serial (i.e., interleaving) execution, only one enabled process runs its enabled commands. We use labels to synchronize (parallelized) guarded commands of different processes. More specifically, if all guarded commands (possibly belonging to different processes) that have identical labels are enabled, they will all be executed. If at least one of them is not enabled, none of them will be executed. The synchronization of guarded commands with guards $g_1, \cdots, g_n$ is equivalent to having one guarded command with guard $g_1 \wedge \cdots \wedge g_n$ and the union of all statements. We omit the label from a guarded command whenever it is not used.

5.2.1 Encoding Schedulers in a Distributed Program

In this section, we describe how we modify a distributed program dp to obtain a program that behaves as if dp was executed under a certain type of scheduler, when only serial executions are available.

k-Central Scheduler

Given a distributed system composed of processes Π. Let each process π in Π consist of a set of guarded commands $\mathcal{G}_\pi$. We augment Π with a k-central scheduler as follows. For every process $\pi \in \Pi$, let

$$KValid_\pi = \{\pi' \mid dist(\pi, \pi') > k\}$$

be the set of processes that are at least $k + 1$ hops away from π. To encode a k-central scheduler, we synchronize every guarded command of a process π with the guarded commands of every subset of $KValid_\pi$. Thus, each process of the new program consists of the following guarded commands:

$for\ all\ \langle g_{\pi,i} \rangle \rightarrow \langle s_{\pi,i} \rangle \in \mathcal{G}_\pi : for\ all\ kval_\pi \subseteq KValid_\pi : for\ all\ \pi' \in kval :$
$for\ all\ \langle g_{\pi',j} \rangle \rightarrow \langle s_{\pi',j} \rangle \in \mathcal{G}_{\pi'} :$
$\langle g_{\pi,i} \wedge (\bigwedge g_{\pi',j}) \rangle \rightarrow \langle s_{\pi,i} \rangle;$

Note that $kval_\pi = \emptyset$ yields the original guarded command of the process. It is necessary to include this case to model a central scheduler.

k-Bounded and k-Enabled Schedulers

To simulate the behavior of a k-bounded (similarly, k-enabled) scheduler, we add a counter (variable) per every ordered pair of processes in the system. A command of a process is allowed to execute only if it has been executed less than k times between any two executions of every other process. Once a process π executes a command, the variables which count the number of executions of other processes between any two executions of π are reset to zero. In a distributed system with n processes $\Pi = \{\pi_1, \cdots, \pi_n\}$, we add variables $\{count_{\pi_i,\pi_j} \mid 1 \leq i, j \leq n\}$. Thus, we replace each guarded command $\langle g_{\pi_i} \rangle \rightarrow \langle s_{\pi_i} \rangle$ in $\mathcal{G}_{\pi_i}$ with the following:

$$\langle g_{\pi_i} \wedge (\bigwedge_{\substack{1 \leq j \leq n \\ i \neq j}} count_{\pi_i,\pi_j} < k) \rangle \rightarrow \langle s_{\pi_i} \rangle; \{\langle count_{\pi_j,\pi_i} := 0 \rangle;\}_{\substack{1 \leq j \leq n \\ i \neq j}}$$

Fairness

Schedulers that generate the worst and best cases are unfair. They can be achieved by modelling the program with a Markov decision process (MDP) instead of a DTMC (for more information see [41]). A probabilistic scheduler which uniformly chooses transitions produces average case expected recovery time, which is both fair and unfair.

5.2.2 Transformation to Scheduler-oblivious Self-stabilization

Refining self-stabilizing algorithms to work under weaker scheduling constraints or no constraints at all has been studied before [26, 27, 9]. A solution to this problem is to compose the algorithm with a self-stabilizing local mutual exclusion algorithm that prevents neighbors from executing their commands simultaneously. In this section, we review the composition method used in [9].

First, we define the specification of the local mutual exclusion problem [9].

Definition 19 *A distributed program dp satisfies the* local mutual exclusion *problem specification iff:*

- (safety) *A process and none of its neighbors do not hold a* privilege *simultaneously,*
- (liveness) *Each process holds the privilege infinitely often.* ■

In the composed algorithm, each process basically contains the guarded commands of the local mutual exclusion algorithm ($\mathcal{LME}$) and the algorithm dp as its critical section. In particular, for every guarded command of $\mathcal{LME}$ and dp, we include the following:

$\langle guard_{\mathcal{LME}} \rangle \rightarrow \langle statement_{\mathcal{LME}} \rangle;$

$\langle guard_{\mathcal{LME}} \wedge privilege = 1 \wedge guard_{dp} \rangle \rightarrow \langle statement_{\mathcal{LME}} \rangle; \langle statement_{dp} \rangle;$

where *privilege* is a variable that determines if the process holds the privilege. Our experimental results in Section 5.3 show that ensuring safety imposes overhead on the recovery time of the stabilizing algorithm.

5.3 Experiments and Analysis

We use probabilistic model-checking (in particular, the tool PRISM [36]) to investigate the significance of the choice of a scheduler on the expected recovery time of a stabilizing distributed algorithm. We chose the *vertex coloring in arbitrary graphs* problem as our case study. It is a classic problem in graph theory that has many applications in scheduling, pattern matching, etc. Furthermore, we study several stabilizing programs that solve this problem. One non-probabilistic algorithm that requires a network, where each process must have a unique id. A probabilistic algorithm where a static probability is assigned to each process, and one with adaptive probability. We compare the expected recovery time of these strategies under all types of schedulers. In our experiments, the choice of graph structure/size and some other parameters was influenced and limited by the computational power of the machine used to do the experiments.

5.3.1 Self-stabilizing Vertex Coloring in Arbitrary Graphs

Definition 20 (Vertex Coloring) *In a graph $G = (V, E)$, the* vertex coloring *problem asks for a mapping from V to a set C of colors, such that no two adjacent vertices (connected directly by an edge) share the same color.* ■

Definition 21 (Vertex Conflict) *Two vertices are in* conflict *iff they are neighbors and they have the same color. Thus, the number of conflicts for a vertex is the number of its neighbors that have the same color as the vertex.* ■

The first deterministic self-stabilizing vertex coloring program of [28] is designed for an anonymous network with an arbitrary underlying communication graph structure $G = (\Pi, E)$ and a non-distributed scheduler. We call this program *deterministic*. Each process has a variable c_π representing its color with domain $c_\pi \in [0, B]$, where B is the maximum degree (number of neighbors) of a vertex (process) in G. In every state, if a process's color is not equal to the maximum available color (the maximum number not taken by any of its neighbors) max_π, it changes its color to max_π. Otherwise, it does not do anything. In this algorithm, a legitimate state is one that the color of each process is equal to max_π. We denote the neighbors of a process by $N(\pi)$ (see Algorithm 1).

The Effect of Schedulers on Expected Recovery Time

We investigate the effect of four attributes of schedulers: centrality, boundedness, enabledness, and fairness, on the expected recovery time of Algorithm 1.

***k*-Centrality:** We calculate the average case expected recovery time for a linear graph, where the size varies from $5 - 7$ and k varies from $0 - Diam(G)$. In a linear graph, $Diam(G) = size - 1$. As expected, Table 5.1(a) validates that, in average, parallelism helps improve the recovery time. However, there can be cases in which it shows detrimental effect. The impact of centrality also depends on the fairness of the scheduler. In the worst case this program does not stabilize under a distributed ($0 - central$) scheduler.

Boundedness/Enabledness: We study the effect of boundedness/enabledness on graphs of size 4 with complete, star, and linear structures for $k = 1, 2, 3$. For each graph structure and each value of k, Table 5.1(b) contains three numbers: R_{min} (best case expected recovery time), R_{avg} (average case expected recovery time), and R_{max} (worst case expected recovery time). Table 5.2(c) demonstrates that as k increases so does the gap between the best case and the worst case. This is the result of allowing more computations as we increase k. That is, all executions corresponding to a k-bounded (respectively, k-enabled) scheduler are also included in the executions of a $(k - 1)$-bounded (respectively, $(k - 1)$-enabled) scheduler.

Fairness: Fairness alongside centrality can determine possibility of convergence. An unfair distributed scheduler can prevent Algorithm 1 from converging. Consider, for example, a state in which two neighbors have identical colors ($\langle 1, 1 \rangle$) and the same maximum available color (2). A computation that infinitely alternates between states $\langle 1, 1 \rangle$ and $\langle 2, 2 \rangle$ never converges to a

(a) Effect of centrality on expected recovery time (average case)

Size\k	0	1	2	3	4	5	6
5	5.6	9.1	13.1	14.4	15.7	-	-
6	7.1	10.7	14.6	18.6	19.6	20.9	-
7	7.6	11.6	16.1	21.2	24.5	24.6	25.8

(b) Effect of boundedness/enabledness on expected recovery time

	Complete			Star			Linear		
	R_{min}	R_{max}	R_{exp}	R_{min}	R_{max}	R_{exp}	R_{min}	R_{max}	R_{exp}
1	3.24	4.64	3.88	10.25	13.33	11.78	6.84	10.30	8.48
2	2.68	7.14	4.10	6.74	24.20	12.52	4.48	19.40	9.03
3	2.57	9.63	4.28	5.74	35.04	13.39	3.87	28.61	9.60

(c) $R_{max} - R_{min}$

	Complete	Star	Linear
1	1.40	3.08	3.46
2	4.46	17.46	14.92
3	7.06	29.30	24.74

Table 5.1: Effect of centrality, boundedness and enabledness.

correct state. Such a computation can be produced by a distributed unfair scheduler. In the rest of our experiments, by unfair scheduler we mean a scheduler that results in worst case expected recovery time, unless otherwise specified.

5.3.2 Composition with Dining Philosophers & the Cost of Ensuring Safety

Recall that Algorithm 1 needs to be refined to work under distributed unfair schedulers. We compose Algorithm 1 with an optimal *snap-stabilizing* (i.e., zero recovery time) dining philosophers distributed program for trees of [33] and refer to it as the *composed strategy*. The solution to the dining philosophers problem provides local mutual exclusion. Since this algorithm is designed specifically for tree structures, in the rest of this section, we use balanced trees in our experiments to ensure fair comparison. Figs. 5.1 and 5.2 depict the expected recovery time of the composed algorithm under fair (central, 1-central, distributed) and unfair (central, 1-central, distributed) schedulers, respectively.

Deg	Fair		Unfair	
	deterministic 1-central	composed distributed	deterministic 1-central	composed distributed
2	1.72	2.54	2.44	2.91
3	2.29	3.73	3.52	4.75
4	2.72	4.69	4.61	6.56
5	3.05	5.49	5.69	8.34
6	3.33	6.16	6.77	10.08
7	3.56	6.72	7.82	11.77
8	3.76	7.21	8.87	13.43

Table 5.2: Cost of ensuring safety in executions

Observe that composing a distributed program with dining philosophers and running the composition under a distributed scheduler is in principle equivalent to running the original distributed program under a 1-central scheduler. However, the 1-central scheduler that is produced by the dining philosopher algorithm may only be able to produce a subset of the possible schedules. Table 5.2 shows the expected recovery time of the deterministic algorithm under 1-central scheduler and the composed algorithm under distributed scheduler (both fair and unfair) for trees with height one and degrees $2-8$. The difference is explained by the fact that the dining philosophers layer itself forces processes that are normally not activatable (that is, they already have a non-conflicting color) to act; that is, the enforcement of fairness between nodes induces unnecessary computation steps.

5.3.3 ID-Based Prioritization

This strategy corresponds to the second deterministic self-stabilizing algorithm of [28], and requires an identified network where each process has a unique id. When several processes are in conflict with the same color, only the process with the highest id will execute its command. As a result, no two similarly colored enabled neighbors will ever execute their commands simultaneously. In some rare cases, this algorithm may not produce 1-central schedules: consider a line of 4 processes c, a, b, d (where identifiers are ordered alphabetically), such that c and a have the same color α, and b and d have the same color β (with $\alpha \neq \beta$). Then, a distributed scheduler may schedule both a and b in a particular step from this situation, resulting in neighboring nodes executing their actions simultaneously. In trees of height 1, this situation cannot occur, and all produced schedules are 1-central. This explains in our results (see Figs. 5.3 and 5.4) why running this program under 1-central and distributed schedulers produces the same

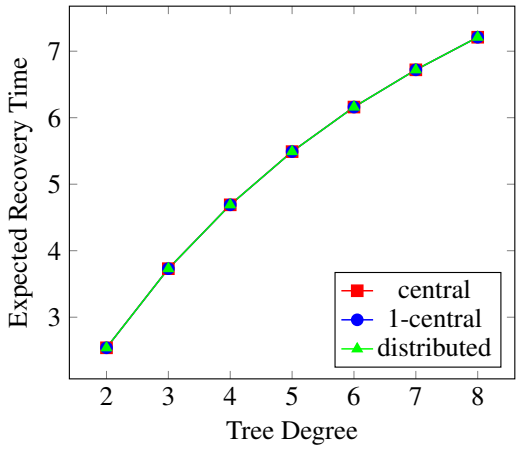

Figure 5.1: Composed program with a fair scheduler

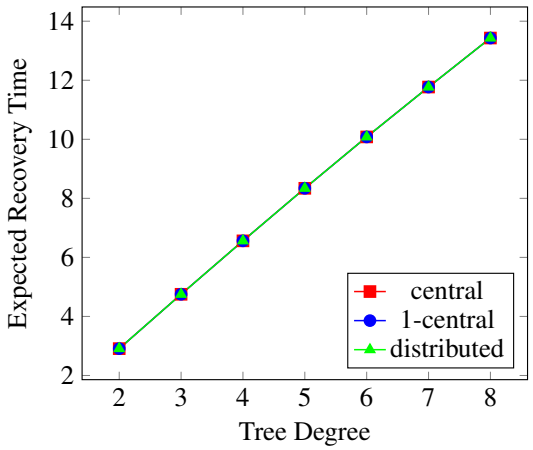

Figure 5.2: Composed program with an unfair scheduler

expected recovery time.

5.3.4 Probabilisitic-Stabilizing Vertex Coloring Programs

The random conflict manager [29] is a lightweight composition scheme for self-stabilizing programs that amounts to executing the original algorithm with some probability p (rather than always executing it). The probabilistic conflict manager does *not* ensure that two neighboring nodes are *never* scheduled simultaneously, but anytime the (possibly unfair) scheduler activates two neighboring nodes u and v, there is a $1-p^2$ probability that u and v do *not* execute simultaneously. Composing the random conflict manager with the deterministic coloring protocol yields a probabilistic coloring algorithm. Fine tuning the parameter p is challenging: a higher p reduces the possibility that a conflict persists when two neighboring conflicting nodes are activated simultaneously (reducing the stabilization time), but also reduces the possibility to make progress by executing the algorithm (increasing the stabilization time). Thus, we consider three strategies for choosing p: (1) p is a constant, for all nodes, throughout the entire execution; (2) p depends on local topology (*i.e.* the current node degree); (3) p is dynamically computed (*i.e.* depending on the current number of conflicts at the current node).

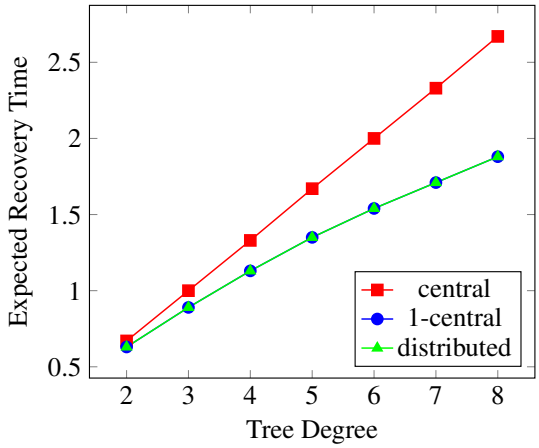

Figure 5.3: ID-based deterministic program with a fair scheduler

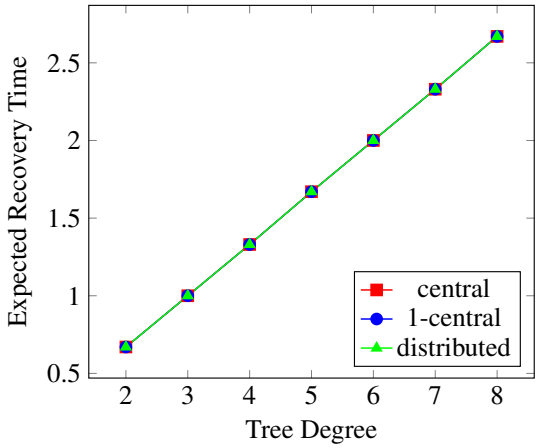

Figure 5.4: ID-based deterministic program with an unfair scheduler

Constant Randomization Parameter

In this strategy, p is a fixed constant for all processes during the program execution. In the original third probabilistic algorithm [28], this probability is equal to 0.5. Figs. 5.5 and 5.6 show that with fixed probability of execution, the stabilization time increases as the number of potential initial conflicts rises. Figs 5.7 and 5.8 demonstrate that for a fixed topology, fine tuning the probability used can result in significantly lower stabilization time. We observe that the stabilization time is not necessarily monotonous with respect to the probability used, as the unfair case demonstrates that increasing the probability of execution too much may have detrimental effects (more conflicts can be preserved in the worst case).

Vertex Degree

This strategy depends on the local structure of the network to let a process execute its commands. It is based on the intuitive reasoning that nodes with fewer neighbors have a lower chance of being in conflict with one of them. The protocol gives higher priority to processes with less number of neighbors. Although processes can have distinct values of p, their values are statically chosen and fixed during the execution. Fig. 5.5 shows that this strategy works remarkably better than a fair coin under a fair scheduler. However, it gradually falls behind a fair coin in the worst case under an unfair scheduler. This is explained by the existence of a central node with an increasing number of neighbors. If executed, this central node can resolve many conflicts at the same time

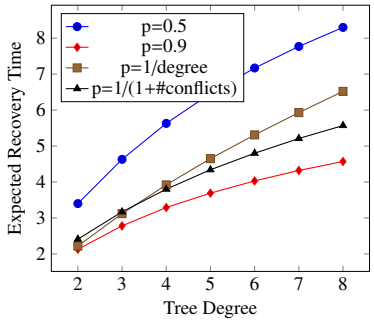

Figure 5.5: Probabilistic programs with a fair distributed scheduler

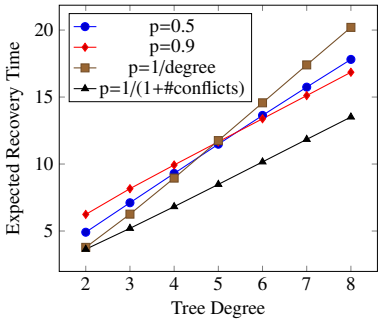

Figure 5.6: Probabilistic programs with an unfair distributed scheduler

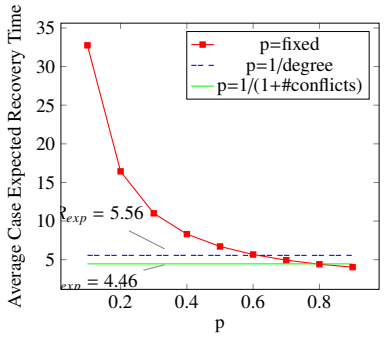

Figure 5.7: Evaluating the effect of the randomization parameter on expected recovery time (on a tree of height=2, degree=2) with a fair distributed scheduler

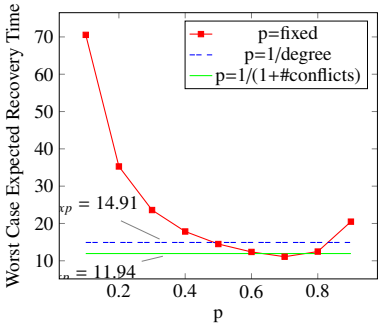

Figure 5.8: Evaluating the effect of the randomization parameter on expected recovery time (on a tree of height=2, degree=2) with an unfair distributed scheduler

(expediting stabilization) in the initial case where it has many conflicts. However, the vertex degree approach pushes towards that these many conflicts are resolved by satellite nodes with a higher probability, causing stabilization to require additional steps, in the worst case.

Number of Conflicts

This strategy refines the vertex degree approach to dynamically take into account the number of potential conflicts. It prioritizes processes with more conflicts over processes with fewer conflicts. Figures 5.5- 5.8 indicate that except for a few biased coins, this adaptive method defeats the other two strategies. It also has the clear advantage of no pre-tuning of the system.

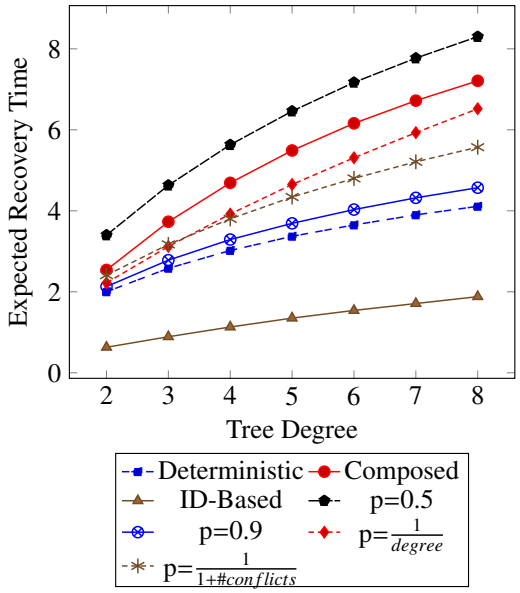

Figure 5.9: Fair distributed scheduler

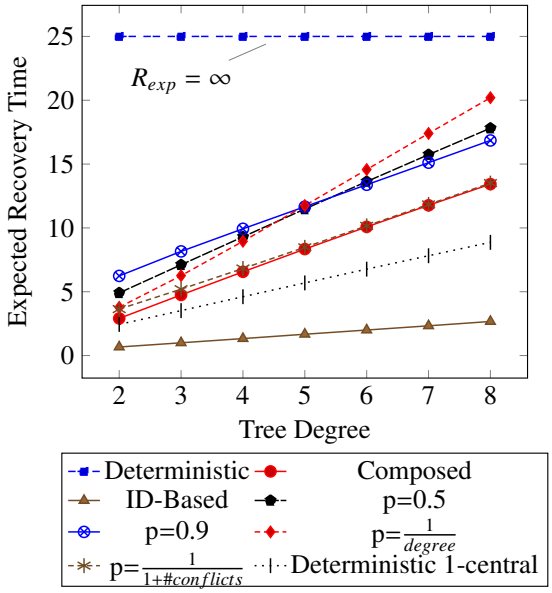

Figure 5.10: Unfair distributed scheduler

5.3.5 Comparing Strategies & Schedulers

This section is devoted to analyzing the results of our technique to select a protocol variant for a particular environment (topology and scheduler). Figure 5.9 presents a comparison of protocol variants (deterministic, composed, id-based, and the three probabilistic ones) when the scheduler is fair, varying the number of nodes in the network. One interesting lesson learned is that the original protocol (deterministic), which is not self-stabilizing for the distributed scheduler (only weakly stabilizing) performs in practise better than actually self-stabilizing protocols (composed, and the three probabilistic variants), so there is a price to pay to ensure (actual or probabilistic) self-stabilization. Overall, the id-based deterministic protocol performs the best (but requires the additional assumption that nodes are endowed with unique identifiers). We also observe that smarter probabilistic variants outperform the composed deterministic protocol, so probabilistic stabilization can come cheaper than a deterministic one.

Figure 5.10 describes the performance of the same protocols in the worst case (unfair scheduler). As deterministic is only weak-stabilizing, its stabilization time with unfair scheduler is infinite. In that case, all probabilistic protocols perform worse than composed, as there exists computations with longer incorrect paths of execution. We also represent the performance of deterministic under the 1-central scheduler as a reference (all other protocols are presented for

the distributed scheduler) for best case situation where only 1-central execution are present. It turns out that both probabilistic variants and composed introduce overhead. The overhead of composed has been discussed in Section 5.3.2, while the overhead of probabilistic variants is that more executions (including executions that are not 1-central) remain possible (with respect to 1-central ones). Again, id-based outperforms all others, including those of deterministic under 1-central scheduler.

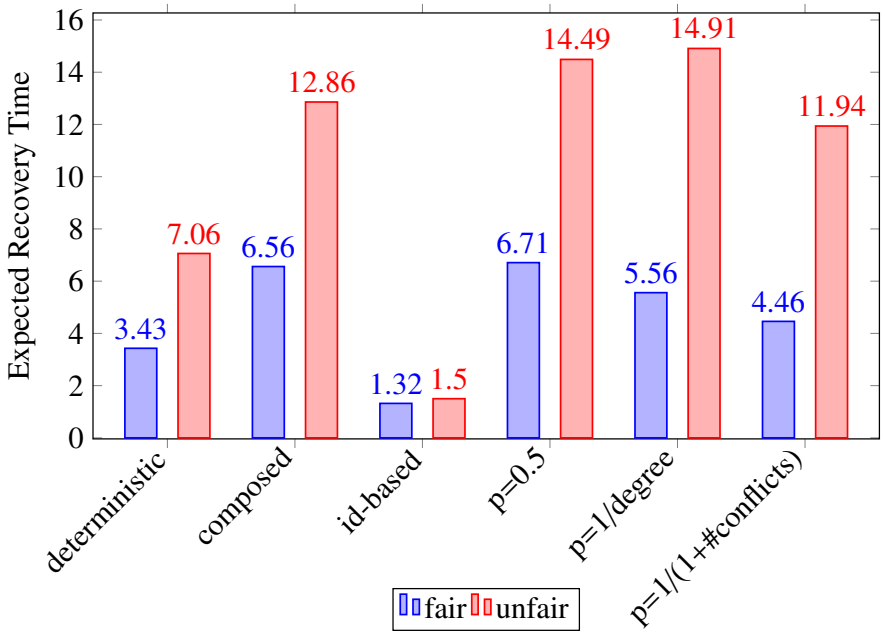

Figure 5.11: Expected recovery time of the six algorithms under fair and unfair schedulers for a tree of height 2 and degree 2. Deterministic is presented for the 1-central scheduler. All others are presented for the distributed scheduler.

The most complex topology is presented in Fig. 5.11, and the relative order of strategies is preserved also for this setting. If the scheduling is fair and identifiers are not available, leaving the algorithm unchanged is the best option. Otherwise, the choice can be to use the refined probabilistic option (that is depending on the current number of conflicts) when there are no identifiers, and the id-based deterministic protocol whenever they are available.

Chapter 6

Related Work

There has been extensive research on self-stabilizing systems from design to analysis in the past years. In this chapter, we discuss some important published work related to our work.

6.1 Complexity of Repair & Synthesis of Weak-Stabilizing Algorithms under Recovery Time Constraints

Designing self-stabilizing distributed programs and proving their correctness is challenging and prone to errors. Therefore, a lot of effort has been put in automating the design and verification process. In terms of complexity, it has been shown adding strong convergence to a distributed program is NP-Complete [34]. An automated efficient sound correct-by-construction method for designing convergence was proposed in [18]. They use the maximal stabilizing program corresponding to the system specifications as an approximation of strong convergence. They rank states based on shortest path to legitimate states in the maximal program. Using the ranks, they find backward reachable states starting from legitimate states attempting to eliminate deadlocks. Their algorithm is sound but not complete as such their solution might output failure although a solution exists.

Other less efficient but complete techniques were introduced [19, 35]. In [19], the synthesis problem is reduced to solving a satisfiability problem by modeling read/write restrictions, closure and convergence properties as an SMT instance. This method is sound (correct-by-construction) and complete. It guarantees finding a solution if one exists if enough computational resources are available. A sound and complete method was proposed in [35]. Their method is based on variable superposition and backtracking search. They introduce computational redundancy by

adding variables to the original system and perform parallel backtracking with a fixed number of threads. One distinction of their method compared to others is including new behavior into the system.

None of the aforementioned techniques, however, impose performance constraints, e.g. average convergence time on the output. Average recovery time is shown to be an important performance metric [21, 20]. Therefore, in this study, we considered the problem of synthesizing self-stabilizing algorithms under average recovery time constraints. To the best of our knowledge, no prior work in this area exists.

6.2 Automated Fine-tuning of Probabilistic-Stabilizing Algorithms

Randomization was introduced to tackle impossible cases in designing stabilizing programs [32]. There has been active research in design and repair of probabilistic systems. In [11], Daws presents a language-theoretic approach to symbolic model checking of reachability properties in DTMCs. In this approach, a finite state automaton is derived from the DTMC from which a regular expression is obtained that amounts to the probability measure of a set of paths satisfying a formula. The regular expression is recursively converted to a rational function over the set of parameters. This approach lacks scalability especially that in stabilizing systems all states are initials states and the above procedure must be repeated for every state. The parametric model checker PARAM (`http://depend.cs.uni-sb.de/tools/param/`) [31], is an improved version of [11]. For our case studies, this tool was not able to compute the expected recovery time.

In [7], the authors modify the probability of controllable transitions to achieve a new model of the program that satisfies a desired property represented in the form of a rational function over a set of parameters while minimizing the cost function. They use the recursive method presented in [11] to obtain the rational parametric function. They show that this problem can be reduced to a non-linear optimization problem. While they present a solution to the model repair problem, their work differs from ours in two aspects. First, they consider only one initial state. This significantly reduces the memory and computation time. Second, their solution works for models representing a single process or networks that are not necessarily anonymous since their approach does not guarantee the preservation of anonymity of processes. Recall that the TPM of a distributed program is a function of the TPMs of the underlying processes which is not accounted for in this work. Furthermore, we take advantage of properties of stabilizing programs to reduce the calculation of the rational function to weighted sum of inverse matrix elements as opposed to the recursive language-theoretic approach of [11].

In [40], instead of performing non-linear optimization as done in [7], which is not scalable, the authors take a greedy approach to finding the optimal evaluation of parameters that results in satisfying the property. Our work is different from this work also in the aforementioned two aspects. Moreover, we compute the symbolic expression of expected recovery time as well as optimum numerical values for parameters.

In [37], the authors verified the asymptotic bounds on the worst-case recovery time of Herman's token circulation algorithm with probabilistic model checking. By calculating the worst-case expected recovery time for different probabilities and network sizes, they made an interesting and surprising observation that a fair coin does not lead to minimum worst-case expected recovery time for networks of size greater than 9. In this paper, for each network size, we compute the parametric average-case expected recovery time of the algorithm, a symbolic rational function over p, and find the exact optimum value of p.

6.3 Automated Analysis of Impact of Scheduling on Performance of Self-Stabilizing Algorithms

A vital factor in designing self-stabilizing protocols is the scheduling assumptions. A detailed survey of schedulers in self-stabilization can be found in [17]. They classify schedulers based on four characteristics (distribution, boundedness, enabledness and fairness). Schedulers can affect the speed and possibility of convergence. For instance, certain protocols are stabilizing under a (1) fair scheduler [12], (2) probabilistic scheduler [32, 42], or (3) scheduler that disallows fully parallel execution of processes [28]. In [28], authors propose a deterministic stabilizing vertex coloring algorithm for arbitrary graphs in an anonymous network. This algorithm, does not converge under a synchronous scheduler in the worst case (unfair scheduler). One way to overcome this problem is to use a less distributed k-central scheduler, where $k > 0$, or use a probabilistic scheduler that randomizes the actions.

One of the challenges of our study was augmenting schedulers in the program. The schedulers we augmented with our stabilizing programs were not maximal. In other words, they did not generate all valid schedules of that specific type of scheduler. Previous work [39] has shown it is impossible to generate all strongly fair schedules when tasks are interleaved. The same authors presented a distributed maximal scheduler for strong fairness [38]. They used unbounded counters to generate all valid schedules. The unbounded counters create infinite state-space in the DTMC model of the program while our models are limited to finite space Markov chains. Therefore, we were not able to use their maximal scheduler in our experiments.

In addition to the issue of correctness, different scheduling policies may have a totally different impact on the *performance* of a self-stabilizing protocol [8]. To the best of our knowledge, there is no work on rigorous analysis of how a scheduling policy alters the performance of self-stabilization.

Chapter 7

Conclusion

Given the rising interest in self-stabilization in the area of fault-tolerant distributed program design, we explored this field from an optimization perspective. Our main focus in this study was minimizing the average recovery time of such systems. In this chapter, we summarize our results and discuss possible extensions and future work.

7.1 Summary

Our first contribution in this study was the counter-intuitive result on the complexity of repairing and synthesizing weak-stabilizing programs under recovery time constraints. We proved that this problem is NP-complete although the synthesis problem without average recovery time constraints has a polynomial time solution [18]. In this regard, we proposed a polynomial-time heuristic for repair and synthesis of stabilizing algorithms under average recovery time constraints. Our heuristic synthesized programs with lower average recovery time in networks of smaller sizes.

We studied the randomization parameter in probabilistic-stabilizing programs. Following the work in [37], we proposed an automated method to calculate the average recovery time of a stabilizing algorithm. We use this method to find the optimum randomization parameters that give the minimum average recovery time for an instance of a probabilistic-stabilizing program. Our method is based on the observation that stabilizing programs can be modelled by absorbing DTMCs due to their convergence and closure properties. By modelling them with absorbing DTMCs we reduced the average recovery time calculation to finding the weighted sum of elements in the inverse of the transition probability matrix among non-legitimate states. We find the

real roots of the derivative of the rational expression computed in the previous step, the optimum probability values, using existing symbolic algebra techniques.

Finally, we evaluated the effect of different types of schedulers on stabilizing deterministic and probabilistic vertex coloring algorithms. We proposed a method to augment stabilizing programs with k-central and k-bounded schedulers. Our experiments showed that parallelism improves the recovery time in general. However, it depends on the fairness of the scheduler as well. One method used to allow weaker scheduling constraints is composing the algorithm with a self-stabilizing mutual exclusion algorithm. We composed the deterministic anonymous vertex coloring algorithm [28] with a snap-stabilizing dining philosophers algorithm [33] and showed that ensuring safety imposes an overhead on recovery time. Another way to tackle this problem is randomization [28, 29]. Our experiments demonstrated that deterministic algorithms have better performance, especially when there are unique IDs assigned to each process. ID-based self-stabilizing algorithms should be composed with self-stabilizing unique naming algorithms to be able to run on anonymous networks at the cost of higher recovery time.

We considered three randomization schemes to choose the probability p with which the processes execute their enabled commands: (1) p is a constant value for all nodes throughout the execution (2) p depends on local topology (i.e.the current node degree); (3) p is dynamically computed (i.e. depending on the current number of conflicts at the current node). Our experiments demonstrated promising performance for the last strategy. The dynamic adaptive strategy had lower average recovery time than some constant values of p and higher than others. However, it has the advantage of no pre-tuning requirements for the system.

7.2 Future Work

7.2.1 Complexity of Repair & Synthesis of Weak-Stabilizing Algorithms under Recovery Time Constraints

For future work, we reckon that there exists no heuristic with constant approximation ratio to solve the synthesis/repair problem under average recovery time constraints. Existing sound and complete algorithms for synthesizing any stabilizing algorithm (without recovery time constraints) are computationally expensive. An interesting future work is finding efficient heuristics for the synthesis problem (both with and without recovery time constraints).

Our heuristic (Algorithm 2 in Section 3.4) can be improved both in efficiency of the heuristic and performance of the synthesized algorithms using other methods such as A^* search or ILP.

There are other interesting problems to consider, such as synthesizing snap-stabilizing, ideal-stabilizing algorithms and parametrized stabilizing protocols under recovery time constraints.

7.2.2 Automated Fine-tuning of Probabilistic-Stabilizing Algorithms

An interesting challenging problem for future is parametrizing the number of processes as well as execution probabilities. Developing methods that scale better and studying the problem in the context of other stabilizing algorithms with larger state-spaces are also challenging problems.

7.2.3 Automated Analysis of Impact of Scheduling on Performance of Self-Stabilizing Algorithms

Given the importance of schedulers, a challenging problem to consider is finding a technique that augments stabilizing programs with the maximal version of a type of scheduler. For example, building a scheduler that its computations generates all k-bounded/enabled schedulers. Another open challenge is using advanced composition techniques [5] to analyze the precise performance hit when a weak-stabilizing algorithm is composed with a self-stabilizing mutual exclusion algorithm to work under distributed schedulers.

References

[1] J. Adamek, M. Nesterenko, and S. Tixeuil. Comparing self-stabilizing dining philosophers through simulation. Technical Report TR-KSU-CS-2013-01, Kent State University, 2013.

[2] S. Aflaki, B. Bonakdarpour, and S. Tixeuil. Automated analysis of impact of scheduling on performance of self-stabilizing protocols. In *Proceedings of 17th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'15)*, pages 156–170, 2015.

[3] S. Aflaki, F. Faghih, and B. Bonakdarpour. Synthesizing self-stabilizing protocols under average recovery time constraints. In *Proceedings of 35th IEEE International Conference on Distributed Computing Systems (ICDCS'15)*, pages 579–588, 2015.

[4] D. Angluin. Local and global properties in networks of processors (extended abstract). In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing (STOC)*, pages 82–93, 1980.

[5] A. Arora, M. G. Gouda, and T. Herman. Composite routing protocols. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 70–78, 1990.

[6] C. Baier and J-.P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[7] E. Bartocci, R. Grosu, P. Katsaros, C. R. Ramakrishnan, and S. A. Smolka. Model repair for probabilistic systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 326–340, 2011.

[8] J. Beauquier and C. Johnen. Analyze of probabilistic algorithms under indeterministic scheduler. In *IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 553–558, 2008.

[9] Joffroy Beauquier, Ajoy Kumar Datta, Maria Gradinariu, and Frederic Magniette. Self-stabilizing local mutual exclusion and daemon refinement. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, pages 223–237, London, UK, UK, 2000. Springer-Verlag.

[10] Z. Chen and A. Storjohann. A BLAS based C library for exact linear algebra on integer matrices. In M. Kauers, editor, *Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC'05*, pages 92–99. ACM Press, New York, 2005.

[11] C. Daws. Symbolic and parametric model checking of discrete-time markov chains. In *Proceedings of the First International Conference on Theoretical Aspects of Computing (ICTAC)*, pages 280–294, 2005.

[12] S. Devismes, S. Tixeuil, and M. Yamashita. Weak vs. self vs. probabilistic stabilization. In *Proceedings of the 28th International Conference on Distributed Computing Systems (ICDCS)*, pages 681–688, 2008.

[13] Abhishek Dhama. A compositional framework for designing self-stabilizing distributed algorithms. 2013.

[14] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[15] Edsger W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, January 1986.

[16] Shlomi Dolev. *Self-stabilization*. MIT press, 2000.

[17] S. Dubois and S. Tixeuil. A taxonomy of daemons in self-stabilization. *CoRR*, abs/1110.0334, 2011.

[18] A. Ebnenasir and A. Farahat. A lightweight method for automated design of convergence. In *IPDPS*, pages 219–230, 2011.

[19] F. Faghih and B. Bonakdarpour. SMT-based synthesis of distributed self-stabilizing systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*. To appear.

[20] N. Fallahi and B. Bonakdarpour. How good is weak-stabilization? In *Proceedings of the 15th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 148–162, 2013.

[21] N. Fallahi, B. Bonakdarpour, and S. Tixeuil. Rigorous performance evaluation of self-stabilization using probabilistic model checking. In *Proceedings of the 32nd IEEE International Conference on Reliable Distributed Systems (SRDS)*, pages 153 – 162, 2013.

[22] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[23] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 3rd edition, 2013.

[24] A. J. Goldstein and R. L. Graham. A Hadamard-type bound on the coefficients of a determinant of polynomials. *SIAM Review*, 16:394–395, 1974.

[25] M. G. Gouda. The theory of weak stabilization. In *International Workshop on Self-Stabilizing Systems*, pages 114–123. Springer, 2001.

[26] M. G. Gouda and F. F. Haddix. The linear alternator. In *Proceedings of the 3rd Workshop on Self-stabilizing Systems (SSS)*, pages 31–47, 1997.

[27] M. G. Gouda and F. F. Haddix. The alternator. In *Proceedings of the 5th Workshop on Self-stabilizing Systems (SSS)*, pages 48–53, 1999.

[28] M. Gradinariu and S. Tixeuil. Self-stabilizing vertex coloring of arbitrary graphs. In *Proceedings of the 4th International Conference on Principles of Distributed Systems (OPODIS)*, pages 55–70, 2000.

[29] M. Gradinariu and S. Tixeuil. Conflict managers for self-stabilization without fairness assumption. In *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS)*, pages 46–46, 2007.

[30] C. M. Grinstead and J. L. Snell. *Introduction to probability*. American Mathematical Soc., 1997.

[31] E. M. Hahn, T. Han, and L. Zhang. Synthesis for PCTL in parametric markov decision processes. In *Proceedings of the 3rd NASA Formal Methods International Symposium*, pages 146–161, 2011.

[32] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.

[33] C. Johnen, L. O. Alima, A. K. Datta, and S. Tixeuil. Optimal snap-stabilizing neighborhood synchronizer in tree networks. *Parallel Processing Letters*, 12(3–4):327–340, 2002.

[34] A. Klinkhamer and A. Ebnenasir. On the complexity of adding convergence. In *Proceedings of the International Conference Fundamentals of Software Engineering*, pages 17–33, 2013.

[35] A. Klinkhamer and A. Ebnenasir. Synthesizing self-stabilization through superposition and backtracking. In *Proceedings of the 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 252–267, 2014.

[36] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, pages 585–591, 2011.

[37] M. Z. Kwiatkowska, G. Norman, and D. Parker. Probabilistic verification of herman's self-stabilisation algorithm. *Formal Aspects of Computing*, 24(4-6):661–670, 2012.

[38] Matthew Lang and Paolo AG Sivilotti. A distributed maximal scheduler for strong fairness. In *Distributed Computing*, pages 358–372. Springer, 2007.

[39] Matthew Lang and Paolo AG Sivilotti. On the impossibility of maximal scheduling for strong fairness with interleaving. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 482–489. IEEE, 2009.

[40] S. Pathak, E. Ábrahám, N. Jansen, A. Tacchella, and J.-P. Katoen. A greedy approach for the efficient repair of stochastic models. In *Proceedings of the 7th NASA Formal Methods International Symposium (NFM)*, pages 295–309, 2015.

[41] A. Simaitis. *Automatic Verification of Competitive Stochastic Systems*. PhD thesis, Department of Computer Science, University of Oxford, 2014.

[42] Y. Yamauchi, S. Tixeuil, S. Kijima, and M. Yamashita. Brief announcement: Probabilistic stabilization under probabilistic schedulers. In *Proceedings of the 26th International Conference on Distributed Computing (DISC)*, volume 7611 of *Lecture Notes in Computer Science*, pages 413–414. Springer, 2012.