Automated Fine Tuning of Probabilistic Self-Stabilizing Algorithms

Saba Aflaki
School of Computer Science
University of Waterloo, Canada
Email: saba.aflaki@uwaterloo.ca

Matthias Volk
Software Modeling and Verification Group
RWTH Aachen University, Germany
Email: matthias.volk@cs.rwth-aachen.de

Borzoo Bonakdarpour
Department of Computing and Software
McMaster University, Canada
Email: borzoo@mcmaster.ca

Joost-Pieter Katoen
Software Modeling and Verification Group
RWTH Aachen University, Germany
Email: katoen@cs.rwth-aachen.de

Arne Storjohann
School of Computer Science
University of Waterloo, Canada
Email: astorjoh@uwaterloo.ca

***Abstract*—Although randomized algorithms have widely been used in distributed computing as a means to tackle impossibility results, it is currently unclear what type of randomization leads to the best performance in such algorithms. This paper proposes three automated techniques to find the probability distribution that achieves minimum average recovery time for an input randomized distributed self-stabilizing protocol without changing the behavior of the algorithm. Our first technique is based on solving symbolic linear algebraic equations in order to identify fastest state reachability in parametric discrete-time Markov chains. The second approach applies parameter synthesis techniques from probabilistic model checking to compute the rational function describing the average recovery time and then uses dedicated solvers to find the optimal parameter valuation. The third approach computes over- and under-approximations of the result for a given parameter region and iteratively refines the regions with minimal recovery time up to the desired precision. The latter approach finds sub-optimal solutions with negligible errors, but it is significantly more scalable in orders of magnitude as compared to the other approaches.**

I. Introduction

Randomization techniques are frequently used in distributed algorithms mainly as a means of breaking symmetry when this is impossible in a deterministic setting. In particular, processes take an action based on a probabilistic function and communicate it to the rest of the network. The correctness of such algorithms stems from the zero probability of reaching a state infinitely often where the goal of the algorithm is not met. Even if processes can solve the problem without randomization (e.g., by employing unique IDs), it may lead to breaking symmetry "faster". Examples of probabilistic distributed algorithms include seminal algorithms for leader election [1], finding maximal independent set [2], crash consensus [3], and self-stabilization [4]. Most research efforts on probabilistic distributed algorithms focus on tackling impossibility results and not so much on how fast such algorithms converge to a solution. However, fast convergence times are crucial for the performance of the algorithms, as it decreases the number of computation steps and therefore the needed computing resources.

A crucial contributing factor in the performance of probabilistic algorithms is the choice of the employed probability distribution. For instance, an empirical study [5] shows a counter-intuitive result that in Herman's probabilistic-stabilizing token ring algorithm [4], for certain sizes of network, using a biased coin leads to faster average recovery time than using a fair coin. We argue that for most probabilistic distributed algorithms, it is currently unclear what choice of a probability distribution results in faster termination of the algorithm, or higher probability of obtaining a correct solution. Lack of this knowledge is primarily due to the incredible subtlety of purely analytical methods to characterize the performance of probabilistic distributed algorithms. The result in [5] motivates even a deeper and more challenging research problem that we call *fine tuning*:

Given a probabilistic distributed algorithm, is it possible to develop an automated technique that can generate a probability distribution function that results in deriving the best performance for the algorithm without changing its behavior?

This paper addresses the aforementioned problem in the context of probabilistic self-stabilizing systems. Self-stabilization is a versatile type of fault-tolerance, where the system is guaranteed to reach a good state after occurrence of a transient fault. Following the results in [6], [7], our choice of performance metric is *average recovery time*. In particular, we propose a fully automated technique that takes as input a probabilistic self-stabilizing algorithm, where processes execute their actions with some probability, and generates as output the probability function based on which, the input algorithm exhibits the *minimum* average recovery time. To better describe our approach, consider Algorithm 1 for solving distributed self-stabilizing *vertex coloring* [8]. Each process π calculates the maximum color available not taken by its

Algorithm 1 Probabilistic-stabilizing Vertex Coloring (process π)

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

neighbors (denoted by $N(\pi)$). If its own color c_π is not equal to this value, it will change its color to this value with probability p. Otherwise, with probability $1-p$ the color does not change. Our technique:

- First transforms such an algorithm into a parametric Markov chain (PMC). Fig. 1 shows the PMC of Algorithm 1 for two processes, where grey states are *legitimate*, since the colors of the two processes differ, and white states are non-legitimate[1]. Such a PMC can be represented by a symbolic transition probability matrix (TPM), where each element shows the probability of one-step reachability of each state from other states:

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

- Next, to compute the best probability function (e.g., p in Fig. 1), we apply one of the following three approaches:
  1) Using the theory of *absorbing* Markov chains [10], the first approach reduces the reachability analysis of legitimate states to computing the inverse of a sub-matrix of a canonical representation of the TPM obtained in the previous step. This inverse is then used to generate a rational function. 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 numerator of the derivative of the computed rational function.
  2) The second approach uses model-checking techniques for obtaining the rational function. Here, the language-theoretic approach of state elimination is applied on the PMC to compute the rational function representing the expected recovery time. Computing the roots of this function as before leads to the optimal probability values.
  3) Finally, the third approach computes over- and under-approximations of the average recovery time for all parameter values inside a given parameter region $\mathcal{R} \subseteq [0, 1]$. By iteratively refining the regions which lead to small convergence times the optimal probability values can be approximated up to the desired precision.

We emphasize that our techniques do not change the semantics (i.e., the structure of the transition system) of the input protocol. They merely identify the probability[2] that results in the best average recovery time. Hence, our techniques can potentially be applied in dealing with network parameters and more importantly in identifying the best distribution for probabilistic schedulers. We also note that our techniques are completely independent of the probabilistic scheduling policy in the underlying PMC. In case the type of scheduler is of importance, one can easily combine the techniques introduced in this paper with the composition method in [11].

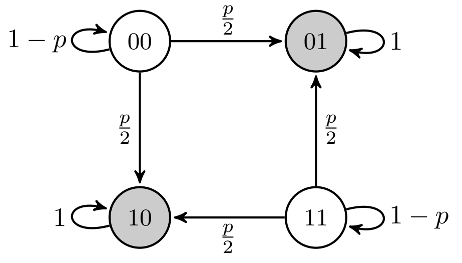

Fig. 1. PMC of probabilistic vertex coloring (Alg. 1) for two processes.

The algorithm of the first approach is implemented using symbolic computation techniques to calculate the optimum probability function that results in minimum average recovery time of stabilizing programs. For the second and third approaches, we extended the existing techniques of computing the rational function [12] and approximation via parameter lifting [13], respectively.

We demonstrate the effectiveness of the three approaches using two case studies: (1) Herman's probabilistic-stabilizing token ring algorithm [4], and (2) the above vertex color algorithm [8]. Our experiments showed that for the vertex coloring algorithm the two exact techniques yield the optimal solution within seconds. For Herman's algorithm with more than 10 processes the exact techniques do not scale as well. The third approximation approach scales better and we are orders of magnitude faster than the exact techniques while still ensuring an approximation error below 1%.

Organization: The rest of the paper is organized as follows. Section II describes our computational model and introduces (parametric) Markov chains. Section III presents the concept of self-stabilization and formally states our fine-tuning problem. Section IV explains the first symbolic algorithm and its efficient implementation. Section V explains the second approach using state elimination and Section VI introduces the approximation approach. Section VII presents the experimental results and evaluates the three approaches. Related

[1]We note that generating the Markov chain of a probabilistic algorithm can be accomplished using standard state exploration techniques [9].

[2]We consider single probabilities, i.e. one parameter, but the methods can also be extended to probability distributions with multiple parameters.

work is discussed in Section VIII. Finally, we make concluding remarks and discuss future work in Section IX.

II. Computational Model and Self-Stabilization

In this section, we review the preliminary concepts on the computational model of probabilistic distributed protocols and self-stabilization.

A. Computational Model

A *distributed program* $\mathcal{D}$ 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 $\mathcal{D}$ 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 $\mathcal{D}$ (denoted S) is the full set of all possible states:

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

A *state predicate* is a subset of S. Each process $\pi \in \Pi$ is associated with a *write-set* and a *read-set*. The read-set of a process π (denoted R_π) is a subset of V. Following the *shared-memory model*, two distinct processes $\pi, \pi' \in \Pi$ are called *neighbors* if $R_\pi \cap R_{\pi'} \neq \emptyset$. We denote the set of neighbors of a process π by $N(\pi)$. The write-set of a process π (denoted W_π) is a subset of R_π, such that for each two distinct processes $\pi, \pi' \in \Pi$, we have $W_\pi \cap W_{\pi'} = \emptyset$.

To concisely specify the behavior of a process π, we utilize a finite set of *probabilistic guarded commands* (denoted $\mathcal{G}_\pi$) of the following form:

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

where the *guard* is a Boolean expression over R_π. Each $statement_i$ updates variables in W_π with probability p_i and causes transitioning from one state s to another s', denoted by $g(s) = s'$, and when the guard is true, one statement is executed according to a probability distribution such that

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

Note that the probability p_i can be represented by a parameter instead of a concrete value.

Definition 1 (Computation): A *computation* σ of a distributed program 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 of $\mathcal{D}$ at time i),
- for all $i \geq 0$, there exists a guarded command g such that $g(s_i) = s_{i+1}$. ■

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.

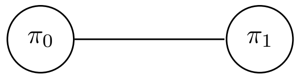

Fig. 2. Example of a vertex coloring problem

Example 1: Throughout the paper, we utilize the probabilistic distributed *vertex coloring* algorithm of [8] 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). For example, for Fig. 2, Algorithm 1 declares two processes and two variables each with domain $[0, 1]$. The state space of the program is

$$S = \{s_0 = \langle 0, 0 \rangle, s_1 = \langle 0, 1 \rangle, s_2 = \langle 1, 0 \rangle, s_3 = \langle 1, 1 \rangle\}.$$

B. Probabilistic Distributed Programs as Discrete-Time Markov Chains

Discrete-time Markov Chains (DTMCs) are transition systems equipped with probabilities. By modeling distributed programs with DTMCs, one can reason about their correctness and compute their expected performance.

Definition 2 (DTMC): A *Discrete-time Markov Chain* is a tuple $\mathcal{D} = (S, S_0, \iota_{init}, \mathbf{P}, L)$ where,

- S is a finite set of states
- S_0 is the set of initial states
- $\iota_{init} : S_0 \rightarrow [0, 1]$ is the initial state distribution such that
$$\sum_{s \in S_0} \iota_{init}(s) = 1$$
- $\mathbf{P} : S \times S \rightarrow [0, 1]$ 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 2, if the transition probabilities include symbolic values, it yields a *parametric Markov chain (PMC)* [14].

Definition 3 (PMC): A *parametric Markov chain* is a tuple $\mathcal{PD} = (S, S_0, U, \iota_{init}, \mathbf{P}, L)$ where,

- S, S_0, L are as defined in Definition 2,
- $U = \{u_1, u_2, \cdots, u_r\}$ is a finite set of real parameters,
- $\iota_{init} : S_0 \rightarrow F_U$ 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 $eval : U \to \mathbb{R}$ assigns real values to parameters in the 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 PMC with parameter set U if the induced TPM ($\mathbf{P}_{eval} = eval(\mathbf{P}) : S \times S \to [0,1]$) and initial distribution ($\iota_{init_{eval}} = eval(\iota_{init}) : S \to [0,1]$) satisfy the following conditions:

- $\forall s \in S : \sum_{s' \in S} \mathbf{P}_{eval}(s, s') = 1$
- $\sum_{s \in S_0} \iota_{init_{eval}}(s) = 1.$

The transition system of a process and, hence, a distributed program can be modeled by a DTMC. The TPM of a process can be trivially derived from its set of probabilistic guarded commands.

Example 2: The TPMs of Algorithm 1 for processes π_1 and π_2 in Fig. 2 are respectively the following:

$$\begin{array}{c} \begin{array}{c|cccc} & \mathbf{00} & \mathbf{11} & \mathbf{01} & \mathbf{10} \\ \hline \mathbf{00} & 1{-}p & 0 & 0 & p \\ \mathbf{11} & 0 & 1{-}p & p & 0 \\ \mathbf{01} & 0 & 0 & 1 & 0 \\ \mathbf{10} & 0 & 0 & 0 & 1 \end{array} \\ \text{TPM}(\pi_1) \end{array} \qquad \begin{array}{c} \begin{array}{c|cccc} & \mathbf{00} & \mathbf{11} & \mathbf{01} & \mathbf{10} \\ \hline \mathbf{00} & 1{-}p & 0 & p & 0 \\ \mathbf{11} & 0 & 1{-}p & 0 & p \\ \mathbf{01} & 0 & 0 & 1 & 0 \\ \mathbf{10} & 0 & 0 & 0 & 1 \end{array} \\ \text{TMP}(\pi_2) \end{array}$$

Given an asynchronous scheduler, Fig. 1 illustrates the transition system of Algorithm 1 for processes π_1 and π_2 in Fig. 2.

III. Fine Tuning of Probabilistic Self-stabilization Programs

A. Probabilistic Self-Stabilization and Recovery Time

A distributed program $\mathcal{D}$ is called *self-stabilizing* for state predicate LS (called the set of *legitimate states*) iff: (1) starting from any arbitrary initial state, every computation automatically converges to a *legitimate state* in a finite number of steps, (2) after which it is guaranteed to remain in a legitimate state as long as a fault does not occur. The first condition is known as *strong convergence* and the second as *closure*. Since strong convergence is a rather strict condition and has led to several impossibility results, *probabilistic convergence* and, hence, *probabilistic stabilization* under a probabilistic scheduler was introduced [4].

Example 3: For example, in our vertex coloring problem, the set of legitimate states is where each two neighboring processes do not share a color:

$$LS = \{s_1 = \langle 0, 1\rangle, s_2 = \langle 1, 0\rangle\}.$$

Our focus in this paper is on probabilistic stabilization. We represent a probabilistic stabilizing program by a DTMC $\mathcal{D} = (S, S_0, \iota_{init}, \mathbf{P}, LS)$, where $S = S_0$ (i.e., any arbitrary state can be an initial state), and $LS \subseteq S$ is the set of legitimate states. Thus, in the sequel, we omit S_0.

Definition 4 (Recovery Path): Let $\mathcal{D} = (S, \iota_{init}, \mathbf{P}, LS)$ be the DTMC of a probabilistic stabilizing program, and s be a state in S. A *recovery path* of $\mathcal{D}$ from state s is a finite computation $\sigma_r = s_0 s_1 \cdots s_n$, such that

- $s_0 = s$;
- for all $i \in [0, n)$, we have
  - $s_i \notin LS$
  - $\mathbf{P}(s_i, s_{i+1}) > 0$
- $s_n \in LS$ ■

We denote the set of all recovery paths from a state s by $\sigma_r(s)$. The *recovery probability* of a recovery path σ_r is

$$\mathbb{P}(\sigma_r) = \prod_{i \in [0,n)} \mathbf{P}(s_i, s_{i+1})$$

Thus, the probability of recovery from a state s is

$$\mathbb{P}(s) = \sum_{\sigma \in \sigma_r(s)} \mathbb{P}(\sigma)$$

Definition 5 (Probabilistic stabilization): A distributed program $\mathcal{D}$ is *probabilistic stabilizing* iff the following conditions hold:

- *Probabilistic recovery:* For all $s \in S$, we have $\mathbb{P}(s) = 1$.
- *Closure:* $\forall s \in LS : \forall \pi \in \Pi : \forall g \in \mathcal{G}_\pi : g(s) \in LS$. ■

The *recovery time* of a recovery path $\sigma_r = s_0 \cdots s_n$ is $\mathbf{R}(\sigma_r) = n$. The *expected recovery time* for a state s is

$$\mathbf{R}(s) = \sum_{\sigma \in \sigma_r(s)} \mathbb{P}(\sigma) \cdot \mathbf{R}(\sigma)$$

and the expected recovery time for a program $\mathcal{D} = (S, \iota_{init}, \mathbf{P}, L)$ is

$$ERT(\mathcal{D}) = \sum_{s \in S} \iota_{init}(s) \cdot \mathbf{R}(s) \qquad (1)$$

Eq. 1 states that the expected recovery time of a stabilizing program is calculated by summing the expected recovery time of a state times the probability of starting the program in that state over all states.

B. The Fine Tuning Problem

Our *fine tuning* problem takes as input the parametric Markov chain 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 PMC $\mathcal{PD} = (S, S_0 = S, U, \iota_{init}, \mathbf{P}, LS)$.

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

$$eval_{min} = \underset{eval}{\operatorname{argmin}}\, ERT(\mathcal{PD})$$

In the following we present three approaches to solve this problem.

IV. Approach 1: A Symbolic Linear Algebraic Technique for Computing Minimum Expected Recovery Time

A. Absorbing DTMCs

We use the theory of *absorbing* DTMCs to reduce the computation of expected recovery time of stabilizing programs to the computation of absorption time in DTMCs. 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.

Definition 6 (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:
$$\exists s \in S : \mathbf{P}(s, s) = 1$$
- *Reachability*: All non-absorbing (called *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 it 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 7 (Absorption Time): The *absorption time* of a state $s \in S$ in a computation $\sigma = s_0 s_1 \cdots$ starting from s is:

$$T(\sigma_s) = \min\{i \mid s_0 = s \wedge \forall k \in [0, i) : s_k \notin \mathcal{A} \wedge s_i \in \mathcal{A}\}$$

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

B. Computing Recovery Time in Absorbing DTMCs

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:

$$\mathbf{P} = \begin{bmatrix} Q_{t\times t} & R_{t\times r} \\ \mathbf{0}_{r\times t} & I_{r\times r} \end{bmatrix}, \qquad (2)$$

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, $\mathbf{0}$ is a $r \times t$ zero sub-matrix and I is a $r \times r$ identity matrix resulting from the self-loops at absorbing states.

The expected absorption time of the transient states of an absorbing DTMC can be computed as follows [10]:

$$N = \Big(\sum_{i=0}^{\infty} Q^i\Big)\tilde{\mathbf{e}} = (I - Q)^{-1}\tilde{\mathbf{e}}, \qquad (3)$$

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, i.e., $N(i) = T(\sigma_i)$.

C. Stabilizing Programs as Absorbing DTMCs

We exploit the two fundamental properties of stabilizing programs, convergence and closure, to model a stabilizing program $\mathcal{D}$ as an absorbing DTMC (denoted $\mathcal{D}^*$). This mapping helps us employ Eq. 3 to compute the expected recovery time of stabilizing programs.

a) Closure $\Longrightarrow$ Absorption: The closure property of stabilizing programs states that once a program reaches a legitimate state, it is trapped in the set of legitimate states unless a transient fault occurs. This implies that entire set of legitimate states can be viewed as an absorbing state. We will justify this assumption shortly.

b) Convergence $\Longrightarrow$ Reachability: Convergence ensures 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 6.

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:

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

where Q, R, $\mathbf{0}$, and C are the one-step transition probability among $\neg LS$ to $\neg LS$, $\neg LS$ to LS, LS to $\neg LS$, and LS to 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:

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

where, $I_{r\times r}$ is the identity matrix. Observe that $\mathbf{P}^*$ is in the form of an absorbing DTMC (Eq. 2). 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. 3, Eq. 1, and the fact that legitimate states have *zero* recovery time to calculate the expected recovery time of a stabilizing program as follows:

$$ERT(\mathcal{D}^*) = \vec{init}(I - Q)^{-1}\tilde{\mathbf{e}}, \qquad (4)$$

where, $\vec{init}$ is a $1 \times t$ row vector containing $\iota_{init}(s)$ for each $s \notin LS$. Recall from Eq. 3 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. 1. For example, the average recovery time of the program of Fig. 2 is derived as follows:

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

c) Discussion: We modeled stabilizing programs with absorbing DTMCs to use the fundamental matrix (Eq. 3) to calculate the expected recovery time. Calculating the sum of powers of a matrix, especially for large powers, is computationally expensive. Eq. 3 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 the weighted sum of elements of the inverse matrix in Section IV-D. The similarity, however, implies that as far as absorption time of transient states (equivalently, recovery time of non-LS states) is concerned, we can represent the stabilizing program $\mathbf{P}$ with $\mathbf{P}^*$.

D. Symbolic Linear Algebraic Technique

We now explain how we compute Eq. 4. 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 [15], 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

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

Fast polynomial interpolation [16, Section 10.2] and rational function reconstruction [16, 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 product $\vec{init}\ v$. In our implementation we solved the systems $Av = \tilde{\mathbf{e}}$ using the Integer Matrix Library (IML) [17], 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[3]. 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.

V. Approach 2: Parameter Synthesis via Rational Function

In this section, we apply an already existing parameter synthesis approach [12] used in probabilistic model checking to compute the rational function representing the expected recovery time. Again, we use the parametric Markov chain $\mathcal{D}^*$ obtained from the stabilizing program $\mathcal{D}$ as input. The approach then operates on the PMC $\mathcal{D}^*$ and is twofold:

1) On the given PMC $\mathcal{D}^*$, we first compute the expected recovery time $ERT(\mathcal{D}^*)$ in the form of a rational function $\frac{r_1}{r_2}$.
2) Next, we compute the first derivative of the rational function $(\frac{r_1}{r_2})' = \frac{r_3}{r_4}$ and check for roots of the numerator r_3 to obtain the optimal parameter value(s).

[3]http://www.maplesoft.com

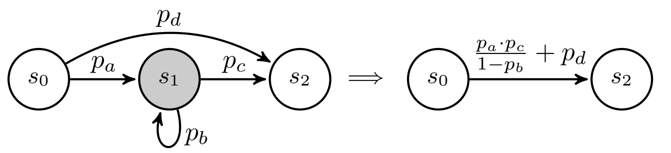

Fig. 3. State elimination

To compute the expected recovery time, we use Eq. 1:

$$ERT(\mathcal{D}^*) = \sum_{s \in S} \iota_{init}(s) \cdot \Big(\sum_{\sigma \in \sigma_r(s)} \mathbb{P}(\sigma) \cdot \mathbf{R}(\sigma) \Big)$$

In the Markov chain, this translates to the expected time of eventually reaching a legitimate state LS for each possible path in $\mathcal{D}^*$. We solve this using the algorithm from [18], which is based on the state elimination idea of [19]. Similar to the transformation of a finite state automaton into a regular expression the state elimination algorithm successively eliminates states of the PMC. An example of this elimination is depicted in Fig. 3, where state s_1 is eliminated and the transition probability from s_0 to s_2 is updated accordingly. The crucial part during the state elimination is the simplification of the intermediate rational functions. By keeping them as small as possible the performance can be significantly increased.

After eliminating all possible states only transitions from initial to legitimate states remain and we obtain a rational function representing the expected recovery time of the model. This rational function is then used in the second step to automatically synthesize optimal parameter values using standard root finding techniques.

Notice that the elimination of a state in the PMC $\mathcal{D}^*$ corresponds to a Gaussian elimination step in the transition probability matrix $\mathbf{P}$. Thus approaches 1 and 2 compute the same rational function.

VI. Approach 3: Approximation via Parameter Lifting

In this section, we apply the *parameter lifting algorithm (PLA)* of [13] to approximate optimal parameter values. Instead of computing the exact optimal parameter values this approach computes *regions* containing the optimal parameter values. By iteratively refining and tightening these regions we can approximate the optimal values up to the desired precision. Note that for ease of presentation we only consider one parameter in the following, but the algorithm is applicable to several parameters in the same manner.

For our parameter synthesis problem, we start with the PMC $\mathcal{D}^*$, an initial region $\mathcal{R} := [\ell, u] \subseteq [0, 1]$ of the values for parameter p and a threshold $\lambda \geq 0$ as input. The question is now to find all sub-regions of $\mathcal{R}$ where the expected recovery time on $\mathcal{D}^*$ is below the given threshold λ:

$$\mathcal{R}_{\leq\lambda}(\mathcal{D}^*) = \{\mathcal{R}' \mid \mathcal{R}' \subseteq \mathcal{R} \wedge \forall v \in \mathcal{R}' : ERT(\mathcal{D}^*|_{p=v}) \leq \lambda\}$$

We start by computing an over- and under-approximation of the exact result for the given region. The idea here is to choose for each transition either the minimal or maximal possible probability value and therefore computing upper and lower bounds for the result. This is done by transforming the PMC into a *Markov Decision Process (MDP)*. An MDP is an extension of a DTMC containing additional non-deterministic choices over the probability distributions in each state. In our transformation for each state in the MDP a non-deterministic choice is introduced, modelling the choice of either taking the lower bound ℓ for the given parameter or the upper bound u. After the transformation an over- (under-) approximation of the actual result can be computed by model checking the MDP and maximizing (minimizing) the reachability result. This approximation can be compared to the given threshold λ to determine if all or none or some parameter values inside the region fulfil the threshold. If only some parameter values fulfil the threshold the approximation is too coarse and a refinement can be made for instance by splitting the region $[\ell, u]$ into two regions $[\ell, m], [m, u]$ with $m := \frac{\ell+u}{2}$.

Notice that the approach is only sound if the extremal probabilities occur at the region bounds. We can ensure this by splitting the region at the extreme points into multiple regions and check each region independently.

Using the parameter lifting approach the optimal parameter values can be approximated up to a desired precision using Algorithm 2. The algorithm gets a PMC $\mathcal{D}^*$ and a desired precision ε as input and returns the region(s) containing the optimal parameter values and the bounds for the optimal recovery time. It starts by computing a coarse initial approximation of the best expected recovery time using the parameter lifting algorithm PLA (Lines 3-5). Next, it iteratively refines the approximation until the desired precision is reached (Lines 6-15). In each loop the parameter lifting algorithm PLA is called for the PMC, a set of regions and the new threshold and returns whether parameter values satisfy the threshold (Line 8). If at least one region fulfils the threshold, we keep the satisfying regions (Line 10) and the upper bound is decreased as there exists a parameter value corresponding to this recovery time (Line 11). If all regions violate the threshold, the lower bound is increased since the given recovery time cannot be realized for any possible parameter value (Line 12). Otherwise, we

Algorithm 2 Algorithm for approximating optimal values

```
Input: PMC $\mathcal{D}^*$, Precision $\varepsilon > 0$
Output: Region $\mathcal{R}$, Bounds for optimal value $[\lambda_\ell, \lambda_u]$
 1: $[\lambda_\ell, \lambda_u] := [0, 1]$;
 2: $\mathcal{R} := \{[0, 1]\}$;
 3: while PLA($\mathcal{D}^*, \mathcal{R}, \lambda_u$) = None do
 4:     $[\lambda_\ell, \lambda_u] := [\lambda_u, \lambda_u \cdot 2]$;
 5: end while
 6: while $\lambda_u - \lambda_\ell > \varepsilon$ do
 7:     $m := \frac{\lambda_\ell + \lambda_u}{2}$;
 8:     res := PLA($\mathcal{D}^*, \mathcal{R}, m$);
 9:     if res = Some then
10:         $\mathcal{R}$ := KeepSatRegions($\mathcal{R}$);
11:         $[\lambda_\ell, \lambda_u] := [\lambda_\ell, m]$;
12:     else if res = None then $[\lambda_\ell, \lambda_u] := [m, \lambda_u]$;
13:     else $\mathcal{R}$ := Split($\mathcal{R}$);                ▷ result Unknown
14:     end if
15: end while
16: return $\mathcal{R}, [\lambda_\ell, \lambda_u]$;
```

still have regions where the result is *Unknown* meaning we split these regions to get finer regions and therefore also better approximations (Line 13). In the end, the algorithm returns the set of satisfying regions $\mathcal{R}$ which correspond to a recovery time in the interval $[\lambda_\ell, \lambda_u]$.

VII. Experiments and Analysis

We compare the three approaches on two probabilistic-stabilizing algorithms: (1) Herman's token circulation in synchronous anonymous rings [4], and (2) Gradinariu and Tixeuil's vertex coloring of arbitrary graphs [8].

We performed all experiments on a HP BL685C G7 restricted to 64GB RAM and used a single 2.0GHz core.

A. Vertex Coloring of Arbitrary Graphs

Algorithm 1 is a probabilistic-stabilizing vertex coloring algorithm designed for arbitrary graphs. When synchronous execution of neighbors is possible, 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 increasing the probability of simultaneous execution of two enabled neighbors, while it decreases recovery time by increasing the probability of making progress. Our experiments verified that as the value of p increases, the average recovery time decreases up to a certain point. Beyond that value, the recovery time increases again. The optimum probability values for graphs with line topology and sizes up to 5 under a synchronous scheduler are demonstrated in the first part of Table I. For the approximation approach we used a precision of 10^{-2}. Note that for the first approach, we only list the number of transient states $|Q|$ whereas for the second approach we count all states.

All three approaches found an optimal solution within a short time. However for larger instances the computation time of the exact approaches grows exponentially whereas the PLA still returns results within seconds.

We also study Algorithm 1 under an asynchronous (central) scheduler (second part of Table I). In this case, no two neighbors ever execute their commands at the same time which brings us to the trivial conclusion that a deterministic algorithm recovers faster. In fact, the deterministic algorithm recovers twice as fast as a probabilistic algorithm with a fair coin. 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 $c_1, c_2 \in \mathbb{N}$, which is a proof of the aforementioned fact. The performance of all three approaches on the asynchronous examples is fast as the resulting rational function is linear and therefore the computation scales well. Note that for the line structure the input matrix for the first approach and the input model for the other two approaches are slightly different leading to different recovery times.

B. 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 has been shown that a non-probabilistic self-stabilizing algorithm for the token circulation problem in anonymous networks does not exist [4], [20].

Algorithm 3 Probabilistic-stabilizing Token Circulation (process i)

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

Herman's probabilistic algorithm [4] (see Algorithm 3) is designed for distributed systems in which an odd number of identical processes are connected in a ring. It breaks the symmetry by randomizing processes actions. Every process owns a binary variable x and looks at the value of its own variable and that of its left neighbor. 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 $O(\frac{2^n}{n})$, since, approximately, every n distinct states of the state space represent the same topology. Notice that approaches 2 and 3 first build the complete state space and then reduce it via bisimulation minimization.

An interesting observation made in [5] was that for more than 9 processes, p=0.5 does not yield worst-case recovery time anymore. We calculate the precise value of p that results in the minimum average recovery time for networks of sizes $3-15$ (see Table II). Based on our results, we see that for networks of size over 7, 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. Thus, 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 and one towards higher values) and a biased coin can significantly reduce the expected recovery time.

The performance of the three approaches on Herman's algorithm is depicted in Fig. 4 where the computation time (in seconds) to find the optimal probabilities is displayed in log-scale. For PLA, we give the time needed to reach a precision of 10^{-2} and 10^{-4}, respectively. The comparison of the exact approaches shows that the symbolic approach performs better than the rational function approach. This is mainly due to the algorithm being specifically optimized for the fine-tuning problem and the mature techniques implemented in Maple. However, as the computation time grows exponentially in the number of processes the exact approaches do not scale well. In fact, for more than 11 processes, it was not even feasible to obtain a solution on our experimental platform. For a network with 15 processes (not shown in Fig. 4), the symbolic approach consumed a total CPU time of half a year running several instances of Maple in parallel on a machine with 64 cores. On the contrary, for larger numbers of processes the approximation method performs orders of magnitude better than the exact approaches. If an approximation error of 1%

Model			Matrix			Rational Function					Approximation (1E-2)		
Sched.	Topol.	Size	$\|Q\|$	p_{opt}	Ert_{opt}	# States	# Trans.	p_{opt}	Ert_{opt}	Time (s)	p_{opt}	Ert_{opt}	Time (s)
Sync.	Line	2	2	0.50	1.00	5	14	0.50	1.00	0.31	[0.48, 0.52]	[1.00, 1.01]	0.51
		3	25	0.69	2.74	28	159	0.69	2.74	0.36	[0.67, 0.71]	[2.73, 2.75]	1.20
		4	77	0.64	2.95	82	753	0.64	2.95	1.49	[0.63, 0.65]	[2.94, 2.95]	1.49
		5	237	0.64	3.44	244	3671	0.64	3.44	80.94	[0.61, 0.66]	[3.44, 3.47]	4.41
Asynchronous	Line	3	25	1.00	$\frac{146}{81p} = 1.80$	28	108	1.00	1.85	0.37	[0.94, 1.00]	[1.84, 1.86]	0.43
		4	77	1.00	$\frac{2357}{972p} = 2.42$	82	378	1.00	2.21	0.50	[0.94, 1.00]	[2.20, 2.22]	0.38
		5	237	1.00	$\frac{31295857}{10497600p} = 2.98$	244	1296	1.00	2.82	0.60	[0.94, 1.00]	[2.81, 2.83]	0.55
		6	721	1.00	$\frac{8401071143}{2361960000p} = 3.56$	730	4374	1.00	3.34	0.94	[0.94, 1.00]	[3.31, 3.34]	0.77
	Ring	3	21	1.00	$\frac{1}{p} = 1.00$	28	108	1.00	1.00	0.39	[0.94, 1.00]	[1.00, 1.01]	0.34
		4	79	1.00	$\frac{228}{81p} = 2.81$	82	378	1.00	2.81	0.50	[0.94, 1.00]	[2.81, 2.83]	0.32
		5	233	1.00	$\frac{1690}{729p} = 2.32$	244	1296	1.00	2.32	0.51	[0.94, 1.00]	[2.31, 2.33]	0.49
		6	715	1.00	$\frac{981097}{291600p} = 3.36$	730	4374	1.00	3.36	0.75	[0.94, 1.00]	[3.34, 3.38]	0.52

TABLE I
RANDOMIZED VERTEX COLORING

Model	Matrix					Rational Function					Approximation (1E-2)		
Size	$\|Q\|$	p_{opt}	ERT_{opt}	$ERT_{p=0.5}$	diff(%)	# States	# Trans.	p_{opt}	Ert_{opt}	Time (s)	p_{opt}	Ert_{opt}	Time (s)
3	2	0.50	0.33	0.33	0	9	36	0.50	0.33	0.37	[0.44, 0.56]	[0.33, 0.33]	0.41
5	6	0.50	1.93	1.93	0	33	276	0.50	1.93	0.26	[0.43, 0.57]	[1.92, 1.94]	0.79
7	18	0.50	4.49	4.49	0	129	2316	0.50	4.49	1.01	[0.43, 0.57]	[4.47, 4.50]	2.29
9	58	0.46, 0.54	7.9210	7.9215	0.006	513	20196	0.46, 0.54	7.92	115.45	[0.39, 0.61]	[7.88, 7.94]	7.18
11	186	0.37, 0.64	12.1020	12.2058	0.85	2049	179196	0.37, 0.64	12.10	97257.30	[0.33, 0.44], [0.60, 0.67]	[12.00, 12.12]	32.49
13	630	0.33, 0.67	16.95	17.35	2.31	8193	1602516	-	-	-	[0.29, 0.38], [0.62, 0.71]	[16.88, 17.00]	275.15
15	2190	0.31, 0.69	22.46	23.34	3.77	32769	14381676	-	-	-	[0.29, 0.34], [0.66, 0.71]	[22.38, 22.50]	3332.92

TABLE II
HERMAN'S RANDOMIZED TOKEN CIRCULATION

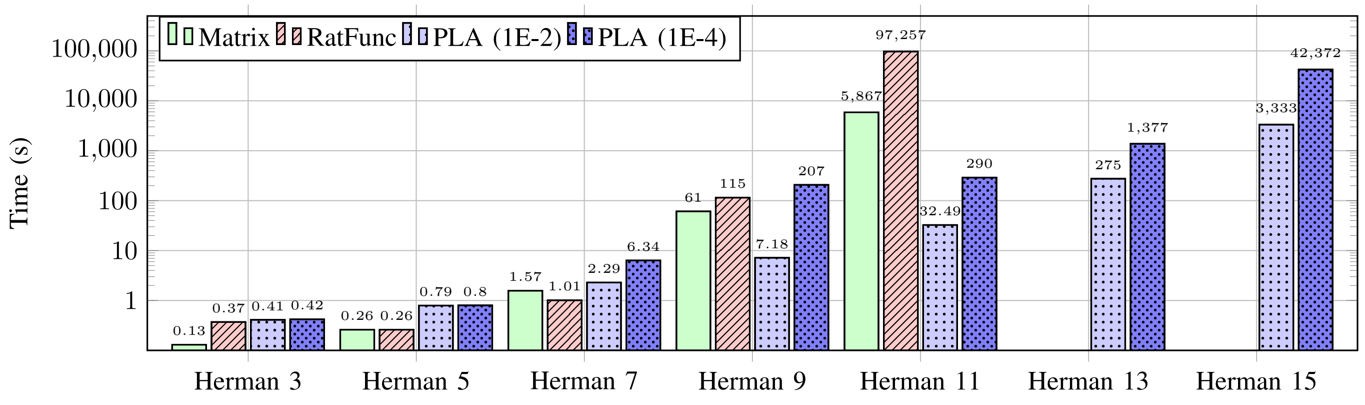

Fig. 4. Solving times on Herman's algorithm

suffices, it gives a solution within 1 hour for the largest instance. Here we also benefit from the iterative nature of the approximation approach as we can stop the approximation at any time if the precision suffices.

VIII. Related Work

In [14], 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 state elimination method presented in [19] to obtain the rational parametric function. They show that this problem can be reduced to a non-linear optimization problem. Their work is similar to our second approach which also uses the rational function. However as seen in the experiments this approach does not always scale well making the use of dedicated methods like the symbolic approach or approximation methods necessary.

In [21], instead of performing non-linear optimization as done in [14], 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 as we are not only interested in satisfying the property, e.g., having a recovery time below a certain threshold, but in achieving the optimal recovery time. Moreover, we compute the symbolic expression of expected recovery time as well as optimal numerical values for parameters.

In [5], 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.

In a recent work [22], the authors approached our problem with genetic algorithms. Their results for Algorithm 3 is almost identical to ours. The drawback of using genetic algorithms is that there are no guaranteed theoretical bounds on the optimality of the result. In contrast our approximation approach gives sound error bounds on the result.

In [11], the effect of schedulers, not the internal behavior of the program, on possibility and speed of convergence is studied through an empirical study. Our experiments in Subsection VII-A confirm the results presented in [11]. A higher value of p improves convergence time by increasing the probability of making progress, while a lower value worsens it by increasing the probability of simultaneous execution of two enabled neighbors.

IX. Conclusion

In this paper, we proposed three automated methods to compute the probability values that result in the minimum average recovery time in a given probabilistic-stabilizing algorithm. In the first approach, using parametric absorbing Markov chains, we reduced the problem of computing expected recovery time to the problem of calculating the weighted sum of elements in an inverse matrix which is computationally cheaper than computing the inverse itself or other iterative approaches. We used existing symbolic algebraic algorithms to compute the weighted sum and to find optimum values. The second approach computes a rational function representing the recovery time via state elimination. The third approach approximates the optimal values by use of the parameter lifting algorithm and scales best for large numbers of processes. Our results systematically confirmed the previous empirical method [5] that a fair coin (p=0.5) does not necessarily yield minimum expected recovery time in Herman's randomized stabilizing token circulation. Given the observed trend in our experiments, we conjecture that as the network size grows the choice of a better p becomes more effective. In the case of the vertex coloring problem, a deterministic approach (p=1) is optimum for an asynchronous scheduler whereas a biased coin is best for a synchronous scheduler.

Future work includes the study of the problem in the context of other distributed algorithms such as randomized leader election and consensus, and for programs with multiple parameters. One can also study the same problem in the presence of different scheduling schemes (modeled as a Markov decision process). A more challenging avenue of research is to not only parameterize the probability function, but also make the computational model parametric in terms of the number of processes. Finally, we can use our techniques to automatically generate state encoding [6], [7] schemes to orthogonally improve the recovery time.

References

[1] A. Itai and M. Rodeh, "Symmetry breaking in distributed networks," *Information and Computation*, vol. 88, no. 1, pp. 60–87, 1990.

[2] M. Luby, "A simple parallel algorithm for the maximal independent set problem," *SIAM Journal of Computing*, vol. 15, no. 4, pp. 1036–1053, 1986.

[3] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of the ACM*, vol. 32, no. 4, pp. 824–840, 1985.

[4] T. Herman, "Probabilistic self-stabilization," *Information Processing Letters*, vol. 35, no. 2, pp. 63–67, 1990.

[5] M. Z. Kwiatkowska, G. Norman, and D. Parker, "Probabilistic verification of herman's self-stabilisation algorithm," *Formal Aspects of Computing*, vol. 24, no. 4-6, pp. 661–670, 2012.

[6] 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)*, 2013, pp. 153 – 162.

[7] 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)*, 2013, pp. 148–162.

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

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

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

[11] 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)*, 2015, pp. 156–170.

[12] C. Dehnert, S. Junges, N. Jansen, F. Corzilius, M. Volk, H. Bruintjes, J.-P. Katoen, and E. Ábrahám, "Prophesy: A probabilistic parameter synthesis tool," in *Proceedings of the 27th International Conference on Computer-Aided Verification (CAV)*, 2015, pp. 214–231.

[13] T. Quatmann, C. Dehnert, N. Jansen, S. Junges, and J.-P. Katoen, "Parameter synthesis for Markov models: Faster than ever," in *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2016, pp. 50–67.

[14] 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)*, 2011, pp. 326–340.

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

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

[17] Z. Chen and A. Storjohann, "A BLAS based C library for exact linear algebra on integer matrices," in *Proceedings of the International Symposium on Symbolic and Algebraic Computation*. ACM Press, New York, 2005, pp. 92–99.

[18] E. M. Hahn, H. Hermanns, and L. Zhang, "Probabilistic reachability for parametric Markov models," *International Journal on Software Tools for Technology Transfer*, pp. 3–19, 2010.

[19] 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)*, 2004, pp. 280–294.

[20] 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)*, 1980, pp. 82–93.

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

[22] L. Zhu, J. Chen, and S. S. Kulkarni, "Refinement of probabilistic stabilizing programs using genetic algorithms," in *Proceedings of the 17th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2015, pp. 217–232.