Synthesizing Self-stabilizing Protocols under Average Recovery Time Constraints

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

Fathiyeh Faghih
School of Computer Science
University of Waterloo, Canada
Email: ffaghihe@cs.uwaterloo.ca

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

***Abstract*—A *self-stabilizing* system is one that converges to a *legitimate state* from any arbitrary state. Such an arbitrary state may be reachable due to wrong initialization or the occurrence of transient faults. *Average recovery time* of self-stabilizing systems is a key factor in evaluating their performance, especially in the domain of network and robotic protocols. This paper introduces a groundbreaking result on automated repair and synthesis of self-stabilizing protocols whose average recovery time is required to satisfy certain constraints. We show that synthesizing and repairing weak-stabilizing protocols under average recovery time constraints is NP-complete. To cope with the exponential complexity (unless P = NP), we propose a polynomial-time heuristic.**

I. Introduction

Self-stabilization is a versatile technique to cope with transient faults in distributed systems. Pioneered by Dijkstra [1], a self-stabilizing system always ensures recovering a proper behavior (normally specified in terms of a set of *legitimate states*) when transient faults perturb the state of the system or the system starts operating from an arbitrary initial state. This property is called *strong convergence* (or *recovery*). Once a legitimate state is recovered, the system continues operating within the set of legitimate states in the absence of faults. This feature is called *closure*. Strong convergence is often too restrictive and there are known results on impossibility of designing self-stabilizing algorithms for token circulation and leader election in anonymous networks among others [2]–[4]. To tackle this problem, several relaxations have been introduced. One example is *weak-stabilization* [5], where there only exists the *possibility* of convergence; i.e., *some* (and not necessarily all) execution paths eventually reach a legitimate state. Thus, in a weak-stabilizing system, a recovery path may reach a live cycle on its way to the legitimate states.

Self-stabilization has a wide range of application domains, including networking [6] (specially in sensor networks [7]) and robotics [8] and automated *synthesis* of stabilizing programs from their formal specification has been an active area of research in the past few years. These efforts range over complexity analysis [9] to efficient synthesis heuristics design [10] as well as less efficient but complete techniques [11], [12]. However, none of these techniques take into account requirements on the performance (e.g., maximum or average convergence time) of the synthesized program. In other words, the existing approaches only synthesize *some* solution that respects only closure and convergence. We argue that this is a serious shortcoming, as some quantitative metrics such as recovery time are as crucial as correctness in practice (e.g., in developing stabilizing network protocols). Furthermore, it is common knowledge that designing correct distributed stabilizing programs is a challenging task and prone to errors. Adding recovery time constraints to the design process makes it even more daunting.

With this motivation, in this paper, we study the problem of synthesizing (from scratch) and repairing (existing) weak/strong-stabilizing programs under performance constraints. The constraint under investigation is, in particular, *average recovery time*. Following the work in [13], [14], we argue that average recovery time is a more descriptive metric than the traditional asymptotic complexity measure (e.g., the big O notation for the number of rounds) to characterize the performance of stabilizing programs. For example, in a long-running network protocol or in a robotic application, it is obviously desirable to maintain a minimum average recovery time than a good worst-case recovery time, which may rarely happen. Technically, average recovery time can be measured by giving weights to states and transitions of a stabilizing program and computing the expected value of the number of steps that it takes the program to reach a legitimate state. These weights can be assigned by a uniform distribution (in the simplest case), or by more sophisticated probability distributions. This technique has been shown to be effective in measuring the performance of weak-stabilizing programs [14] as well as cases where faults hit certain variables or locations more often.

Our contributions in this paper are the following:

- We show that the complexity of *synthesizing* a weak-stabilizing protocol with certain average recovery time, given the topology of the communication network and description of the set of legitimate states, is NP-complete in the state space. This result is especially counter-intuitive, as weak-stabilizing protocols (with no recovery time constraints) can be synthesized in linear time [9]. This, in turn, means that designing a weak/strong-stabilizing program whose recovery time satisfies certain performance constraints is significantly more challenging than synthesizing some solution.
- We show that the complexity of *repairing* an existing weak-stabilizing protocol to obtain either a weak or

strong stabilizing protocol, so that (1) only removal of transitions is allowed during repair, and (2) the repaired protocol satisfies a certain average recovery time, is also NP-complete.

- In order to cope with the exponential complexity (unless P = NP), we propose a polynomial-time heuristic. We demonstrate that our heuristic synthesizes stabilizing programs with lower or equal average recovery time compared to existing algorithms for the majority of instances of our case studies: Dining philosophers and token circulation in rings.

Organization: The rest of the paper is organized as follows. Section II presents our computation model for distributed programs. The notion of self-stabilization and average convergence time are presented in Section III. We present the formal statement of the problem in Section IV. The complexity of the repair and synthesis problems are discussed in Sections V and VI, respectively, while the polynomial-time heuristic is presented in Section VII. Related work is discussed in Section VIII. Finally, we make concluding remarks and discuss future work in Section IX.

II. Distributed Programs

In our formal framework, a distributed program is comprised of a finite set V of *discrete variables*, and a finite set Π of *processes* operating on V. Each variable $v \in V$ has a finite domain D_v. A *state* of the program is identified by a valuation of all variables v in V. We call the set of all possible states the *state space*, denoted by S. We indicate the value of a variable v in a state s with $v(s)$.

Definition 1: 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 write (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:*
  $$\begin{aligned}&\forall (s_0, s_1) \in T_\pi : \exists s'_0, s'_1 \in S\\ &\quad \forall v \notin W_\pi : (v(s_0) = v(s_1) \ \wedge \ v(s'_0) = v(s'_1))) \ \wedge\\ &\quad (\forall v \in R_\pi : (v(s_0) = v(s'_0) \ \wedge \ v(s_1) = v(s'_1))) \Rightarrow\\ &\qquad\qquad (s'_0, s'_1) \in T_\pi. \ \blacksquare\end{aligned}$$

Notice that Definition 1 resembles the *shared-memory model*. 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.

We say that a process π is *enabled* in state s if there exists a state s', such that $(s, s') \in T_\pi$.

Definition 2: A *distributed program* over a set of variables V is a transition system $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$, where

- $\Pi_\mathcal{D}$ is a set of processes over the common set V of variables, such that for any two distinct processes $\pi_1, \pi_2 \in \Pi_\mathcal{D}$, we have $W_{\pi_1} \cap W_{\pi_2} = \emptyset$,
- $T_\mathcal{D}$ is the transition relation of $\mathcal{D}$, such that
$$T_\mathcal{D} = \bigcup_{\pi \in \Pi_\mathcal{D}} T_\pi \quad \blacksquare$$

Note that Definition 2 resembles an asynchronous distributed program, where process transitions execute in an *interleaving* fashion resulting in $T_\mathcal{D}$ being the union of transition relations of the processes in the program. We say that processes $\pi_1, \pi_2 \in \Pi_\mathcal{D}$ are *neighbors* iff $R_{\pi_1} \cap R_{\pi_2} \neq \emptyset$.

Starting from an initial state $s_0 \in S$ in a distributed program $\mathcal{D}$, following the footsteps of the transition system during an execution, yields a *computation*.

Definition 3: A *computation* σ of a distributed program $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ with state space S, is an infinite sequence of states:
$$\sigma = s_0 s_1 s_2 \cdots$$

- For all $i \geq 0$, we have $(s_i, s_{i+1}) \in T_\mathcal{D}$,
- if σ reaches a state s_i, such that (s_i, s_i) is the only outgoing transition from s_i in $T_\mathcal{D}$, then σ is called a *terminating computation*. Such a computation stutters at s_i indefinitely. $\blacksquare$

We denote by σ_s a computation with initial state s. A prefix of σ is called a *finite computation*. We use the notation σ_n for a finite computation of length n.

III. Self-Stabilization and Convergence Time

A. Self-stabilization

A self-stabilizing program is one that starting from any arbitrary initial state reaches a *legitimate state* in a finite number of steps, and without outside intervention. Such an arbitrary state may be reached due to wrong initialization, or occurrence of transient faults. Upon reaching a legitimate state, the system is guaranteed to remain in such states thereafter in the absence of faults. These two conditions are known as *convergence* and *closure*.

Definition 4: A distributed program $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ is *self-stabilizing* for a set $LS \subseteq S$ of *legitimate states* iff the following two conditions hold:

- *Strong convergence:* In any computation $\sigma = s_0 s_1 \cdots$ of $\mathcal{D}$, where s_0 is an arbitrary state in S, there exists $i \geq 0$, such that $s_i \in LS$.
- *Closure:* For all transitions $(s, s') \in T_{\mathcal{D}}$, if $s \in LS$, then $s' \in LS$ as well. ■

Convergence, as defined in Definition 4, is a strong property since it should be satisfied in all computations. This property is weakened in *weak-stabilizing* distributed programs [5], where the existence of a converging computation or *possibility* of convergence suffices.

Definition 5: We say that a distributed program $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ is *weak-stabilizing* for a set $LS \subseteq S$ of *legitimate states* iff the following two conditions hold:

- *Weak convergence:* For each state $s_0 \in S$, there *exists* a computation $\sigma = s_0 s_1 \cdots$ of $\mathcal{D}$, and $i \geq 0$, such that $s_i \in LS$.
- *Closure:* As defined in Definition 4. ■

The computation existence condition in weak convergence, unlike strong convergence, allows for execution cycles outside the set of legitimate states. In the rest of this paper, we use 'convergence' in place of 'strong convergence'. We refer to both classes, either weak-stabilizing or self-stabilizing as 'stabilizing'.

B. Average Recovery Time of Stabilizing Programs

Let us call the number of steps that a stabilizing program takes to reach a legitimate state the *recovery time* (or *convergence time*).

Definition 6: Let $\sigma = s_0 s_1 \cdots$ be a computation of a stabilizing program that starts from initial state s_0 and reaches a legitimate state in LS. The *recovery time* of σ is the following:

$$RT(\sigma) = \min\{j \mid s_j \in LS\},$$

where s_j is the j^{th} state in σ. ■

Since this metric depends both on the initial state of the program and the computation that reaches a legitimate state, we are interested in calculating the *average* recovery time of stabilizing programs. Following the techniques introduced in [13], [14], such average can be computed through statistical expected value of recovery time from different initial states and through various computations. Thus, we need to calculate the expected recovery time defined in Definition 6 for all initial states $s_0 \in S$ and take the probabilistic average of them accounting for the impact of different recovery computations and different starting points.

Calculating the expected value of a discrete random variable requires valid probabilities for the occurrence of the elements in its domain. Here, we take *recovery time*, RT, as a discrete random variable with domain $D_{RT} = [0, \infty]$. Our final goal is to find the probability of RT having a specific value i which requires probability values for transitions. Yet, the transition relation of a distributed program as defined in Definition 2 lacks probability distribution. Without loss of generality, we assume a *uniform distribution* over the set of outgoing transitions of each state in the state space. Such assumption, basically, makes the transition system of distributed programs a *Markov chain*. On that account, one can define the *probability of a computation* in a distributed program as follows.

Given a distributed program $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ with state space S, the probability of a finite computation $\sigma = s_0 s_1 \cdots s_n$ is computed by the following formula:

$$\mu(\sigma) = \prod_{i=0}^{n} \mathbf{P}(s_i, s_{i+1}) \tag{1}$$

where $\mathbf{P} : S \times S \to [0, 1]$ is the transition probability function such that for all $s \in S$:

- $$\sum_{s' \in S} \mathbf{P}(s, s') = 1, \tag{2}$$
- Given $T(s) = \{(s, s') \mid \exists s' \in S : (s, s') \in T_{\mathcal{D}}\}$, the set of outgoing transitions of state s, a *uniform* probability distribution over the transitions can be obtained by:

$$\mathbf{P}(s, s') = \frac{1}{|T(s)|} \tag{3}$$

Equation 3 assigns equal probability to all transitions originating from a state. We emphasize that this uniform probability distribution was introduced only to permit computing weighted average for recovery time. If, however, the distribution over the transition relation was known (e.g., for probabilistic programs or particular schedulers in which processes work in a random fashion with specific probability distribution), Equation 3 could be modified trivially while satisfying Equation 2.

Notation 2: Let $\eth^c_{s,n}$ denote all *converging computations* originating from state s with recovery time n. That is,

$$\eth^c_{s,n} = \{\sigma_s \mid RT(\sigma_s) = n\}.$$

For a computation σ_s starting from state $s \in S$, the probability of the event "σ_s having recovery time equal to n" can be calculated as follows:

$$\mathcal{P}(RT(\sigma_s) = n) = \sum_{\sigma_s \in \eth^c_{s,n}} \mu(\sigma_s) \tag{4}$$

In Equation 4, the probability of the computations with recovery time n starting from s are added together because they are *disjoint events*. In other words, two distinct computations cannot happen at the same time, so the probability of their union is the sum of their individual probabilities. We have now built the necessary background to demonstrate how to compute the expected recovery time of a stabilizing program.

The *expected recovery time* of a state $s \in S$ is the following:

$$ERT(s) = E[RT(\sigma_s)] = \sum_{i=0}^{\infty} (i \times \mathcal{P}(RT(\sigma_s) = i))$$

Finally, we take the *average* of the expected recovery time values computed for all initial states in the state space. The reason behind this is to consider the fact that stabilizing programs may start executing from any arbitrary state. The *average recovery time* of a stabilizing program $\mathcal{D}$ with state space S and initial state distribution ι_{init} is obtained by the following formula:

$$AvgRT(\mathcal{D}, S) = \sum_{s \in S} \iota_{init}(s).ERT(s) \tag{5}$$

where $\iota_{init} : S \to [0,1]$ is a probability distribution function such that

$$\sum_{s \in S} \iota_{init}(s) = 1 \qquad (6)$$

Recall that a stabilizing program can reach any state due to the occurrence of transient faults. The appearance of this state (as the initial state) can follow a specific probability distribution. In cases where this distribution is not given, we decidedly assume uniform distribution; i.e., $\iota_{init}(s) = \frac{1}{|S|}$ for all $s \in S$. Otherwise, $\iota_{init}(s)$ can be any *real* value in $[0,1]$ provided that it meets the condition in Equation 6. We stress that all definitions and computations presented in this section are valid for weak-stabilizing programs as well as self-stabilizing programs, as it should be clear from our use of the term *stabilizing program* all along.

IV. Problem Statement

In this section, we formally state the problem of synthesizing and repairing weak-stabilizing programs whose average recovery time is expected to be below a certain value.

A. 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 $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$, and a real number ert.

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

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

B. 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 $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$, where $T_{\mathcal{D}} = \emptyset$, a state predicate LS, and a real number ert.

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

- $AvgRT(\mathcal{D}') \leq ert$.

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

V. 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 IV-A is NP-complete both when the output is required to be weak-stabilizing (cf. Subsection V-A).

A. Weak-stabilizing Repair

We present a polynomial-time reduction from the *3-Dimensional Matching (3DM)* [15] problem to our repair problem.

Theorem 1: The repair decision problem in Section IV-A to obtain a weak-stabilizing programs is NP-complete.

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

Proof of membership to NP

Given a program $\mathcal{D}' = \langle \Pi_{\mathcal{D}}, T'_{\mathcal{D}} \rangle$ as a solution, we should verify the following two conditions:

1) It is weak-stabilizing.
2) $AvgRT(\mathcal{D}') \leq ert$

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 III-B) can be solved in polynomial time as well [16].

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 $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$, a set of legitimate states LS, and an upper bound of average recovery time ert. Figure 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. $V_{\mathcal{D}} = \{v_1, v_2\}$, 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 1. The rest of the states are in LS. As shown in Figure 1, we include $3q+3M$ states in LS denoted by LSA_1 and $LSB_0^{m_t}$, 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 3: In the sequel, set $U = X \cup Y \cup Z$ is the universal set. We denote by $\mathcal{C}(i)$ the set of tuples that contain element $i \in U$. For example, in Figure 1, we have $\mathcal{C}(1) = \{m_1, m_3\}$. 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, $i_0^{\mathcal{C}(i)} = \{i_0^{m_t} \mid (i \in U) \wedge (m_t \in \mathcal{C}(i))\}$ denotes $|\mathcal{C}(i)|$ 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:

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

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:

$$\mathcal{G}(\{(i_0, i_0^{m_t}), (j_0, j_0^{m_t}), (k_0, k_0^{m_t}), (i_1, i_1^{m_t}), (j_1, j_1^{m_t}), (k_1, k_1^{m_t}) \mid m_t = (i,j,k) \in m\}) \qquad (7)$$

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).

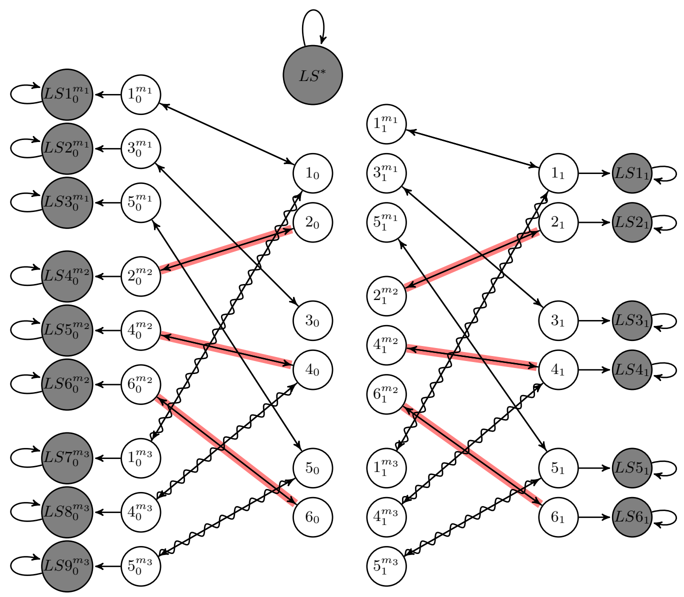

Fig. 1: Weak repair instance mapped from $3DM$ instance $X = \{1,2\}$, $Y = \{3,4\}$, $Z = \{5,6\}$ and $m = \{m_1 = (1,3,5), m_2 = (2,4,6), m_3 = (1,4,5)\}$.

Process π_2 has the following transitions:

$$T_{\pi_2} = \{(i_0^{m_t}, LSA_0^{m_t}), \; (j_0^{m_t}, LSB_0^{m_t}), \; (k_0^{m_t}, LSC_0^{m_t}) \mid m_t = (i,j,k) \in m\} \; \cup \; \{(i_1, LSD_1) \mid i \in U\} \cup \; \{(s,s) \mid s \in LS\}$$

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 $|\mathcal{C}(i)| - 1$ 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 1. In the repaired graph, we will have two types of connected components. One type can be seen among the

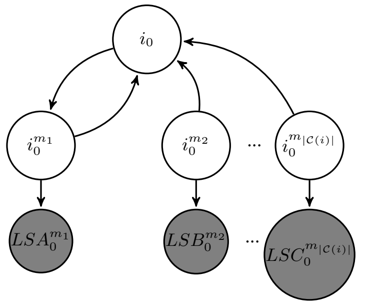

Fig. 2: Connected blocks with subscript 0.

states with subscript 0 (see Figure 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 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). 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 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 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.

■

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

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

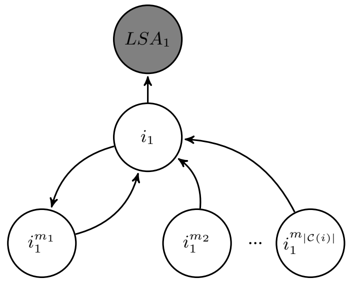

Fig. 3: Connected blocks with subscript 1.

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 7: A weak-stabilizing program $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ is *maximal* in an asynchronous setting if

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

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 $\mathcal{D}_{max}$ 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 $\mathcal{D}'_{max}$. The transition relation of π_1 and π_2 in the new maximal program are as follows:

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

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 1.

Proof: 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 $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}}\rangle$, 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. $V_{\mathcal{D}} = \{v_1, v_2, v_3, v_4\}$, 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:

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

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 1.

First, we determine LS and transition relations of π_1, π_2. Similar to Figure 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 7) 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 4). As we will discuss shortly, this problem is solved by including variable v_3 and process π_3 to the system. Figure 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 $\{i_{0/1}, i_{0/1}^{\mathcal{C}(i)}\}$ 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: $i_{0,v}$, $i_{0,v}^{\mathcal{C}(i)}$. Similarly, $\forall v \in [3q+1, 6q]$ we have additional states $i_{1,v}$, $i_{1,v}^{\mathcal{C}(i)}$ in LS (see Figure 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 6). In Figure 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).

In $Gadget_0$, everything remains the same, as described before. However, in $Gadget_1$, states $i_{0,0,1}^{LS}$ and $i_{1,0,1}^{m_t,LS}$ are the only states in LS (see Figure 7). All other gadgets $gadget_v$, $v \in [1, 6q+1]$ in $Gadget_1$ have the same structure as in Figure 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 $(i_{0,0,1}, i_{0,v,1}^{LS}), v \in [1, 3q]$ and $(i_{1,0,1}^{m_t}, i_{1,v,1}^{m_t,LS}), v \in [3q+1, 6q]$ 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 $i_{1,0,0}^{m_t}$ to directly connect to LS is through transitions $(i_{0,0,0}, i_{0,0,1}^{LS})$ and $(i_{1,0,0}^{m_t}, i_{1,0,1}^{m_t,LS})$ of π_4. That is not possible since their group transitions $(i_{0,v,0}^{LS}, i_{0,v,1})$ and $(i_{1,v,0}^{m_t,LS}, i_{1,v,1})$, $(v \in [1, 6q])$

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 $(i^{LS}_{0,0,1}, i^{\mathcal{C}(i)}_{0,0,1})$ and $(i^{m_t,LS}_{1,0,1}, i_{1,0,1})$ in $Gadget_1$.

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

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

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 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:

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

■

VII. 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.

A. 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 1 takes as input a stabilizing program $\mathcal{D}$, and a set LS of legitimate states. In the case of synthesis, $\mathcal{D}$ must be maximal. Algorithm 1 works as follows. We consider $Diam + 1$ classes of states, where $Diam$ is the diameter of the underlying graph of $\mathcal{D}$. 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 1 Synthesis/Repair Heuristic

```
1: Input: $\mathcal{D}$, $LS$
2: $d[0..Diam(\mathcal{D})] \leftarrow BFS(\mathcal{D}, LS)$
3: for $i = 1$ to $Diam(\mathcal{D})$ do
4:     $pq \leftarrow PriorityQueue(\{(s,t) \mid s \in d[i],\ (s,t) \in T_{\mathcal{D}}\})$  ▷ Keep transitions based on effect on avg recovery time in priority queue.
5:     while $\neg pq.empty$ do
6:         $edge \leftarrow pq.get()$  ▷ edge[0]=s, edge[1]=t
7:         if $edge[1] \notin LS \wedge \neg Visited(edge) \wedge edge[1] \notin d[i-1]$ then
8:             DeleteTransitionGroup($\mathcal{D}$, $edge$)
9:             if $\neg Converge(\mathcal{D})$ then
10:                InsertTransitionGroup($\mathcal{D}$, $edge$)
11:                VisitedTransitionGroup($edge$)
12:            end if
13:        end if
14:    end while
15: end for
```

We now analyze the complexity of our algorithm for an input program with $|S| = n$ states (exponential in the number of variables/processes), $|T| = m$ transitions and g groups. Classification can be done in $O(n(n + m))$ by performing BFS several times, each time starting from a state in LS. Since every edge is either removed or marked as visited so it will not be considered for removal again, the three nested loops in Algorithm 1 will visit each edge twice all together. Once, for storing them in a priority queue and once for removal. Convergence is verified once per each group of edges. Convergence can be verified in the same way as classification. Furthermore, each calculation of average recovery time imposes an overhead of $O(n^{2.3})$ computational steps. Hence, the total running time of our algorithm is $O(n(n+m) + 2m + gn(n+m) + gn^{2.3})$. In the worst case $g = O(m)$, while in an asynchronous setting $m = O(nlogn)$. To conclude, Algorithm 1 has running time $O(n^3log^2n + n^{3.3}logn)$.

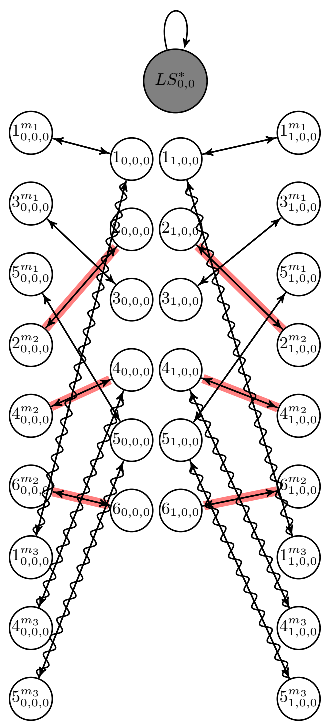

Fig. 4: $v_3 = v_4 = 0$ in $gadget_0 \in Gadget_0$.

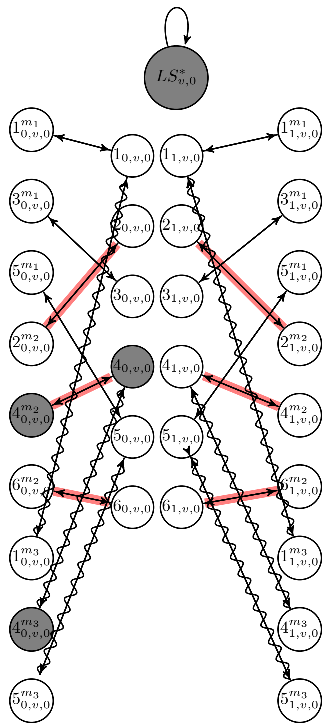

Fig. 5: $v_3 \in [1, 6q], v_4 = 0$ in $gadget_v \in Gadget_0$.

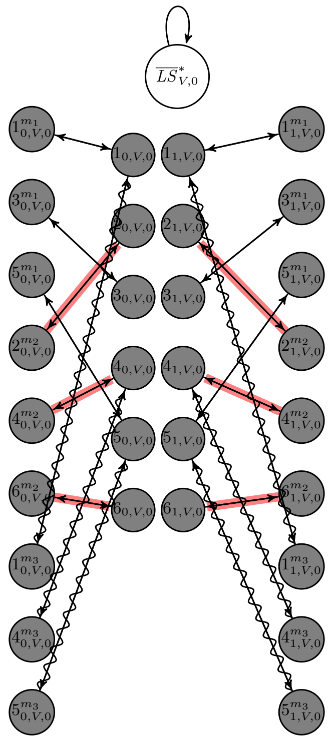

Fig. 6: $V = v_3 = 6q + 1, v_4 = 0$ in $gadget_{6q+1} \in Gadget_0$.

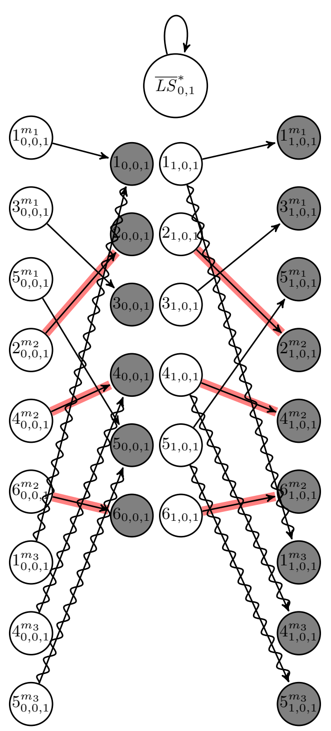

Fig. 7: $= v_3 = 0, v_4 = 1$ in $gadget_0 \in Gadget_1$.

Gadgets obtained in our mapping. Variables v_1 and v_2 have the same valuation as in Figure 1.

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 I, and II come from an implementation without a priority queue.

B. Case Studies and Experimental Results

We ran Algorithm 1 on a system with 8GB RAM and Intel Core i5 2.60GHz CPU for two problems: dining philosophers and token circulation in rings. 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 two problems in the literature using the technique in [13]. 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.

1) Case Study 1: Dining Philosophers: A solution to the *dining philosophers* 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. [17] (see Table I). 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.

2) Case Study 2: Token circulation in anonymous rings: A solution to this problem ensures that only one process holds the token and each process holds the token infinitely often. Table II compares the average recovery time of our synthesized solutions to that of the algorithm in [4], 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 [4]. We note that since the second condition requires the existence of a cycle in LS, we slightly modified Algorithm 1 to preserve a cycle in LS.

Table II shows a trend that for rings with odd length, Algorithm 1 and [4] have the same average recovery time. This is because they both generate the maximal program. Algorithm 1 could not remove any group of transitions due to violation of convergence suggesting that the maximal program is the only stabilizing program.

Algorithm 1 synthesizes a strong-stabilizing program for a ring consisting of 4 processes which means the program can be used on non-anonymous networks [4]. It was not feasible to analyze the output for even-length networks of size 6 and higher due to very long execution time.

VIII. Related Work

In [9], the authors show that adding strong convergence is NP-complete in the size of the state space, which itself is exponential in the size of variables of the protocol. In their formulation, the authors assume that the behavior of the protocol inside the set of legitimate states is given as an input.

P	**Algorithm 1**	**Adamek et al [17]**	**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 I: Dining Philosophers (tree)

P	**Algorithm 1**	**Devismes et al [4]**	**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 II: Token circulation (ring)

In this paper, we do not make such an assumption. Ebnenasir and Farahat [10] also propose a heuristic to synthesize self-stabilizing algorithms. A heuristic is naturally an incomplete method, meaning that it may not be able to synthesize a solution, even if there exists one. To remedy this shortcoming, [11] and [12] propose complete synthesis algorithms based on SMT solving and backtracking, respectively. Note that these techniques focus on synthesizing any solution, but do not consider any performance requirement for the synthesized program. On the contrary, in this paper, we take the desirable average recovery time of the solution as an input to the synthesis problem. To the best of our knowledge, this is the first work on synthesis of self-stabilizing systems under recovery time constraints.

The other line of work related to the synthesis of self-stabilizing algorithms is the area of synthesizing fault-tolerant systems. The proposed algorithm in [18] synthesizes a fault-tolerant distributed algorithm from its fault-intolerant version. While the focus of the work in [18] is on efficient development of synthesis algorithms, this paper concentrates on complexity analysis of synthesis of self-stabilization subject to recovery time constraints.

Finally, in [19], the authors study the problem of model repair for probabilistic systems. Their goal is in fact the opposite of the objective of this paper. That is, they attempt to preserve the structure of the model and change the probability distribution of executions.

IX. Conclusion

In this paper, we studied the problem of synthesizing and repairing stabilizing protocols, such that its average recovery time satisfies a given upper bound. We showed that the problem is NP-complete for weak stabilization in the size of the state space of the protocol. This result is, in particular, counter-intuitive in the context of weak-stabilizing protocols, as their synthesis without recovery time constraints can be achieved in linear time in the state space of the protocol. We purposefully chose *average* recovery time following the work in [13], [14], where the authors show that the traditional asymptotic complexity metrics do not characterize the performance of stabilizing protocols accurately and fairly. To cope with the exponential complexity (unless P = NP), we proposed an efficient polynomial-time heuristic. Our case studies demonstrated the effectiveness of our approach.

For future work, there are several interesting open questions. We conjecture that to solve our synthesis and repair problems, there exists no heuristic whose approximation ratio is constant. One can study the problem of synthesizing snap-stabilizing and ideal-stabilizing algorithms. A more challenging problem is to synthesize parameterized stabilizing protocols under recovery time constraints.

X. Acknowledgment

This work was partially sponsored by Canada NSERC Discovery Grant 418396-2012 and NSERC Strategic Project Grant 430575-2012.

References

[1] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, vol. 17, no. 11, pp. 643–644, 1974.

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

[3] 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)*, 2000, pp. 55–70.

[4] S. Devismes, S. Tixeuil, and M. Yamashita, "Weak vs. self vs. probabilistic stabilization," in *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2008, pp. 681–688.

[5] M. G. Gouda, "The theory of weak stabilization," in *International Workshop on Self-Stabilizing Systems (SSS)*, 2001, pp. 114–123.

[6] S. Dolev and E. Schiller, "Self-stabilizing group communication in directed networks," *Acta Informatica*, vol. 40, no. 9, pp. 609–636, 2004.

[7] T. Herman, "Models of self-stabilization and sensor networks," in *Proceedings of the International Workshop on Distributed Computing (SSS)*, 2003, pp. 205–214.

[8] F. Ooshita and S. Tixeuil, "On the self-stabilization of mobile oblivious robots in uniform rings," in *Proceedings of the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2012, pp. 49–63.

[9] A. Klinkhamer and A. Ebnenasir, "On the complexity of adding convergence," in *Proceedings of the 5th International Conference Fundamentals of Software Engineering*, 2013, pp. 17–33.

[10] A. Ebnenasir and A. Farahat, "A lightweight method for automated design of convergence," in *Proceedings of the 25th International Parallel and Distributed Processing Symposium (IPDPS)*, 2011, pp. 219–230.

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

[12] 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*, 2014, pp. 252–267.

[13] 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.

[14] 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.

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

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

[17] J. Adamek, M. Nesterenko, and S. Tixeuil, "Comparing self-stabilizing dining philosophers through simulation," Kent State University, Tech. Rep. TR-KSU-CS-2013-01, 2013.

[18] B. Bonakdarpour, S. S. Kulkarni, and F. Abujarad, "Symbolic synthesis of masking fault-tolerant programs," *Springer Journal on Distributed Computing*, vol. 25, no. 1, pp. 83–108, March 2012.

[19] E. Bartocci, R. Grosu, P. Katsaros, C. R. Ramakrishnan, and S. A. Smolka, "Model repair for probabilistic systems," in *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2011, pp. 326–340.