Computer Science Technical Report

Synthesizing Self-Stabilization Through Superposition and Backtracking

Alex Klinkhamer and Ali Ebnenasir

Michigan Technological University
Computer Science Technical Report
CS-TR-14-01
May 2014

Department of Computer Science
Houghton, MI 49931-1295
www.cs.mtu.edu

Synthesizing Self-Stabilization Through Superposition and Backtracking*

Alex Klinkhamer and Ali Ebnenasir

May 2014

Abstract

This paper presents a sound and complete method for algorithmic design of self-stabilizing network protocols. While the design of self-stabilization is known to be a hard problem, several sound (but incomplete) heuristics exists for algorithmic design of self-stabilization. The essence of the proposed approach is based on variable superposition and backtracking search. We have validated the proposed method by creating both a sequential and a parallel implementation in the context of a software tool, called Protocon. Moreover, we have used Protocon to automatically design self-stabilizing protocols for problems which all existing heuristics fail to solve.

*__Superior__, a high performance computing cluster at Michigan Technological University, was used in obtaining results presented in this publication.

Contents

1 Introduction

Self-stabilization is an important property of today's distributed systems as it ensures *convergence* in the presence of transient faults (e.g., loss of coordination and bad initialization). That is, from *any* state/configuration, a Self-Stabilizing (SS) system recovers to a set of legitimate states (a.k.a. *invariant*) in a finite number of steps. Moreover, from its invariant, the executions of an SS system satisfy its specifications and remain in the invariant; i.e., *closure*. Design and verification of convergence are difficult tasks [12,19,29] in part due to the requirements of (i) recovery from *any* state; (ii) recovery under distribution constraints, where processes can read/write only the state of their neighboring processes (a.k.a. their *locality*), and (iii) the non-interference of convergence with closure. This paper presents a novel method for algorithmic design of self-stabilization by variable superposition [10] and a *complete* backtracking search.

Most existing methods for the design of self-stabilization are either manual [6, 8, 12, 19, 22, 37, 39] or heuristics [1, 3, 16, 17] that may fail to generate a solution for some systems. For example, Awerbuch *et al.* [8] present a method based on distributed snapshot and reset for locally correctable systems; systems in which the correction of the locality of each process results in global recovery to invariant. Gouda and Multari [22] divide the state space into a set of supersets of the invariant, called *convergence stairs*, where for each stair closure and convergence to a lower level stair is guaranteed. Stomp [37] provides a method based on ranking functions for design and verification of self-stabilization. Gouda [19] presents a theory for design and composition of self-stabilizing systems. Methods for algorithmic design of convergence [1, 3, 16, 17] are mainly based on sound heuristics that search through the state space of a non-stabilizing system in order to synthesize recovery actions while ensuring non-interference with closure. Specifically, Abujarad and Kulkarni [2] present a method for algorithmic design of self-stabilization in locally-correctable protocols. Farahat and Ebnenasir [15,17] present algorithms for the design of self-stabilization in non-locally correctable systems. They also provide a swarm method [16] to exploit computing clusters for the synthesis of self-stabilization. Nonetheless, the aforementioned methods may fail to find a solution while there exists one; i.e., they are sound but incomplete.

This paper proposes a sound and complete method (Figure 1) for the synthesis of SS systems. The essence of the proposed approach includes (1) systematic introduction of computational redundancy by introducing new variables, called *superposed variables*, to an existing protocol's variables, called *underlying variables*, and (2) an intelligent and parallel backtracking method. The backtracking search is conducted in a parallel fashion amongst a fixed number of threads that simultaneously search for an SS solution. When a thread finds a combination of design choices that would result in the failure of the search (a.k.a. *conflicts*), it shares this information with the rest of the threads, thereby improving resource utilization during synthesis.

The contributions of this work are multi-fold. First, the proposed synthesis algorithm is complete; i.e., if there is an SS solution, our algorithm will find it. Second, we relax the constraints of the problem of designing self-stabilization by allowing new superposed behaviors inside the invariant. This is in contrast to previous work where researchers require that during algorithmic design of self-stabilization no new behaviors are included in the invariant. Third, we provide three different implementations of the proposed method as a software toolset, called Protocon (`http://cs.mtu.edu/~apklinkh/protocon/`), where we provide a sequential implementation and two parallel implementations; one multi-threaded and the other an MPI-based implementation. Fourth, we demonstrate the power of the proposed method by synthesizing four challenging network protocols that all existing heuristics fail to synthesize. These case studies include the 3-bit (8-state) token passing protocol (due to Gouda and Haddix [21]), coloring on Kautz graphs [25] which can represent a P2P network topology, ring orientation and leader election on a ring.

Organization. Section 2 introduces the basic concepts of protocol, transient faults, closure and convergence. Section 3 formally states the problem of designing self-stabilization. Section 4 presents the proposed synthesis method. Section 5 presents the case studies. Section 6 summarizes experimental results. Section 7 discusses related work. Finally, Section 8 makes concluding remarks and presents future/ongoing work.

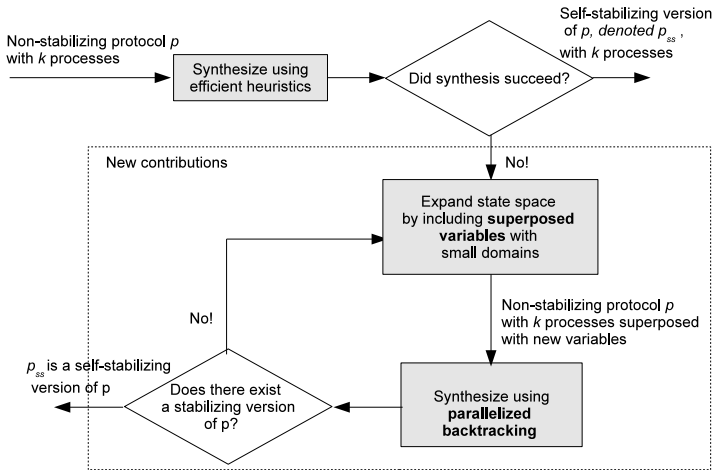

Figure 1: A complete backtracking method for synthesis of self-stabilization.

2 Preliminaries

In this section, we present the formal definitions of protocols and self-stabilization. Protocols are defined in terms of their set of variables, their actions and their processes. The definitions in this section are adapted from [6, 12, 19, 32]. For ease of presentation, we use a simplified version of Dijkstra's token ring protocol [12] as a running example.

Protocols. A *protocol* p comprises N processes $\{P_0, \cdots, P_{N-1}\}$ that communicate in a shared memory model under the constraints of an underlying network topology T_p. Each process P_i, where $i \in \mathbb{Z}_N$ and $\mathbb{Z}_N$ denotes values modulo N, has a set of local variables V_i that it can read and write, and a set of *actions* (a.k.a. *guarded commands* [13]). Thus, we have $V_p = \cup_{i=0}^{N-1} V_i$. The domain of variables in V_i is non-empty and finite. T_p specifies what P_i's neighboring processes are and which one of their variables P_i can read; i.e., *P_i's locality*. Each action of P_i has the form $grd \rightarrow stmt$, where grd is a Boolean expression specified over P_i's locality, and $stmt$ denotes an assignment statement that atomically updates the variables in V_i. A *local state* of P_i is a unique snapshot of its locality and a *global state* of the protocol p is a unique valuation of variables in V_p. The *state space* of p, denoted S_p, is the set of all global states of p, and $|S_p|$ denotes the size of S_p. A *state predicate* is any subset of S_p specified as a Boolean expression over V_p. We say a state predicate X *holds in a state* s (respectively, $s \in X$) *if and only if (iff)* X evaluates to true at s. A *transition* t is an ordered pair of global states, denoted (s_0, s_1), where s_0 is the source and s_1 is the target state of t. A *valid* transition of p must belong to some action of some process. The set of actions of P_i represent the set of all transitions of P_i, denoted δ_i. The set of transitions of the protocol p, denoted δ_p, is the union of the sets of transitions of its processes. A *deadlock state* is a state with no outgoing transitions. An action $grd \rightarrow stmt$ is *enabled* in a state s iff grd holds at s. A process P_i is *enabled* in s iff there exists an action of P_i that is enabled at s.

Example: Token Ring (TR). The Token Ring (TR) protocol (adapted from [12]) includes three processes $\{P_0, P_1, P_2\}$ each with an integer variable x_j, where $j \in \mathbb{Z}_3$, with a domain $\{0, 1, 2\}$. The process P_0 has the following action (addition and subtraction are in modulo 3):

$$A_0 : \quad (x_0 = x_2) \qquad \longrightarrow \qquad x_0 := x_2 + 1$$

When the values of x_0 and x_2 are equal, P_0 increments x_0 by one. We use the following parametric action to represent the actions of processes P_j for $1 \leq j \leq 2$:

$$A_j : \quad (x_j \neq x_{(j-1)}) \qquad \longrightarrow \qquad x_j := x_{(j-1)}$$

Each process P_j copies x_{j-1} only if $x_j \neq x_{j-1}$, where $j = 1, 2$. By definition, process P_j *has a token* iff $x_j \neq x_{j-1}$. Process P_0 *has a token* iff $x_0 = x_2$. We define a state predicate I_{TR} that captures the set of states in which only one token exists, where I_{TR} is

$$((x_0 = x_1) \wedge (x_1 = x_2)) \ \vee \ ((x_1 \neq x_0) \wedge (x_1 = x_2)) \ \vee \ ((x_0 = x_1) \wedge (x_1 \neq x_2))$$

Each process P_j is allowed to read variables x_{j-1} and x_j, but can write only x_j. Process P_0 is permitted to read x_2 and x_0 and can write only x_0. $\lhd$

Computations. Intuitively, a computation of a protocol p is an *interleaving* of its actions. Formally, a *computation* of p is a sequence $\sigma = \langle s_0, s_1, \cdots \rangle$ of states that satisfies the following conditions: (1) for each transition (s_i, s_{i+1}) in σ, where $i \geq 0$, there exists an action $grd \rightarrow stmt$ in some process such that grd holds at s_i and the execution of $stmt$ at s_i yields s_{i+1}, and (2) σ is *maximal* in that either σ is infinite or if it is finite, then σ reaches a state s_f where no action is enabled. A *computation prefix* of a protocol p is a *finite* sequence $\sigma = \langle s_0, s_1, \cdots, s_m \rangle$ of states, where $m > 0$, such that each transition (s_i, s_{i+1}) in σ (where $i \in \mathbb{Z}_m$) belongs to some action $grd \rightarrow stmt$ in some process. The *projection* of a protocol p on a non-empty state predicate X, denoted $\delta_p | X$, consists of transitions of p that start in X and end in X.

Specifications. We follow [31] in defining a safety specification *sspec* as a set of bad transitions in $S_p \times S_p$ that should not be executed. A computation $\sigma = \langle s_0, s_1, \cdots \rangle$ satisfies *sspec* from s_0 iff no transition in σ is in *sspec*. A liveness specification *lspec* is a set of infinite sequences of states [5]. A computation $\sigma = \langle s_0, s_1, \cdots \rangle$ satisfies *lspec* from s_0 iff σ has a suffix in *lspec*. A computation σ of a protocol p satisfies the specifications *spec* of p from a state s_0 iff σ satisfies both safety and liveness of *spec* from s_0.

Closure and invariant. A state predicate X is *closed in an action* $grd \rightarrow stmt$ *iff* executing $stmt$ from any state $s \in (X \wedge grd)$ results in a state in X. We say a state predicate X is *closed in a protocol* p *iff* X is closed in every action of p. In other words, *closure* [19] requires that every computation of p starting in X remains in X. We say a state predicate I is an *invariant* of p *iff* I is closed in p and p satisfies its specifications from any state in I.

TR Example. Starting from a state in the predicate I_{TR}, the TR protocol generates an infinite sequence of states, where all reached states belong to I_{TR}. $\lhd$

Remark. In the problem of synthesizing self-stabilization (Problem 3.1), we start with a protocol that satisfies its liveness specifications from its invariant. Since during synthesis by superposition and backtracking we preserve liveness specifications in the invariant, we do not explicitly specify the nature of liveness specifications.

Convergence and self-stabilization. A protocol p *strongly converges* to I iff from any state in S_p, every computation of p reaches a state in I. A protocol p *weakly converges* to I iff from any state in S_p, there is a computation of p that reaches a state in I. We say a protocol p is strongly (respectively, weakly) self-stabilizing to I iff I is closed in p and p is strongly (respectively, weakly) converging to I. For ease of presentation, we drop the term "strongly" wherever we refer to strong stabilization.

3 Problem Statement

In this section, we state the problem of incorporating self-stabilization in non-stabilizing protocols using superposition. Let p be a non-stabilizing protocol and I be an invariant of p. As illustrated in Figure 1, when we fail to synthesize a self-stabilizing version of p, we manually expand the state space of p by including new variables. Such *superposed variables* provide computational redundancy in the hopes of giving the protocol sufficient information to detect and correct illegitimate states without forming livelocks. Let p' denote the self-stabilizing version of p that we would like to design and I' represent its invariant. $S_{p'}$ denotes the state space of p'; i.e., the expanded state space of p. Such an expansion can be captured by a function $\mathcal{H} : S_{p'} \rightarrow S_p$ that maps every state in $S_{p'}$ to a state in S_p. Moreover, we consider a one-to-many mapping $\mathcal{E} : S_p \rightarrow S_{p'}$ that maps each state $s \in S_p$ to a set of states $\{s' \mid s' \in S_{p'} \wedge \mathcal{H}(s') = s\}$. Observe that $\mathcal{H}$ and $\mathcal{E}$ can also be applied to transitions of p and p'. That is, the function $\mathcal{H}$ maps each transition

(s'_0, s'_1), where $s'_0, s'_1 \in S_{p'}$, to a transition (s_0, s_1), where $s_0, s_1 \in S_p$. Moreover, $\mathcal{E}((s_0, s_1)) = \{(s'_0, s'_1) \mid s'_0 \in S_{p'} \wedge s'_1 \in S_{p'} \wedge \mathcal{H}((s'_0, s'_1)) = (s_0, s_1)\}$. Furthermore, each computation (respectively, computation prefix) of p' in the new state space $S_{p'}$ can be mapped to a computation (respectively, computation prefix) in the old state space S_p using $\mathcal{H}$. Our objective is to design a protocol p' that is self-stabilizing to I' when transient faults occur. That is, from any state in $S_{p'}$, protocol p' must converge to I'. In the absence of faults, p' must behave similar to p. That is, each computation of p' that starts in I' must be mapped to a unique computation of p starting in I. Moreover, no new computations should be introduced in the computations of p in I. However, new computations are allowed in the new invariant I'. We use $\mathcal{H}$ and $\mathcal{E}$ to formally state the problem constraints[1] on I' and $\delta_{p'}$ with respect to I and δ_p as follows: (The function $\textsc{Pre}(\delta)$ takes a set of transitions δ and returns the set of source states of δ.)

Problem 3.1. Synthesizing Self-Stabilization.

- **Input**: A protocol p and its invariant I for specifications *spec*, the function $\mathcal{H}$ and the mapping $\mathcal{E}$ capturing the impact of superposed variables.
- **Output**: A protocol p' and its invariant I' in $S_{p'}$.
- **Constraints**:
  1. $I = \mathcal{H}(I')$
  2. $\forall s \in \textsc{Pre}(\delta_p) \cap I : \mathcal{E}(s) \subseteq \textsc{Pre}(\delta_{p'})$
  3. $\delta_p | I = \mathcal{H}(\{(s'_0, s'_1) \mid (s'_0, s'_1) \in (\delta_{p'} | I') \wedge \mathcal{H}(s'_0) \neq \mathcal{H}(s'_1)\})$
  4. $\forall s \in \textsc{Pre}(\delta_p) \cap I : \delta_{p'} | \mathcal{E}(s)$ is cycle-free
  5. p' strongly converges to I'

The first constraint requires that no states are added/removed to/from I; i.e., $I = \mathcal{H}(I')$. The second constraint requires that any non-deadlocked state in I should remain non-deadlocked. The third constraint requires that any transition in $\delta_p | I$ should correspond to some transitions in $\delta_{p'} | I'$, and each transition included in $\delta_{p'} | I'$ must be mapped to a transition (s_0, s_1) in $\delta_p | I$ while ensuring $s_0 \neq s_1$. Implicitly, this constraint requires that no transition in $\delta_{p'} | I'$ violates safety of *spec*. The fourth constraint stipulates that, for any non-deadlock state $s \in I$, the transitions included in the set of superposition states of s must not form a cycle; otherwise, liveness of *spec* may not be satisfied from I'. Notice that the combination of constraints 3 and 4 allows the inclusion of transitions $(s'_0, s'_1) \in \delta_{p'} | I'$ where $\mathcal{H}(s'_0) = \mathcal{H}(s'_1)$ under the constraint that such transitions do not form a cycle. Finally, p' must converge to I'. Notice that, the combination of Constraints 1 to 4 ensure that p' would satisfy *spec* from I'.

Example 3.2. *Token ring using two bits per process*

Consider the non-stabilizing token ring protocol p with N processes, where each process P_i owns a binary variable t_i and can read t_{i-1}. The first process P_0 is distinguished, in that it acts differently from the others. P_0 is said to have a token when $t_{N-1} = t_0$ and each other process P_i is said to have a token when $t_{i-1} \neq t_i$. P_0 and the other processes P_i (where $i > 0$) have the following actions:

$$t_{N-1} = t_0 \longrightarrow t_0 := 1 - t_0;$$
$$t_{i-1} \neq t_i \longrightarrow t_i := t_{i-1};$$

Let I denote the legitimate states, where exactly one process has a token, written $I \equiv \exists!\ i \in \mathbb{Z}_N : ((i = 0 \wedge t_{i-1} = t_i) \vee (i \neq 0 \wedge t_{i-1} \neq t_i))$, where the quantifier $\exists!$ means there exists a unique value of i. The above protocol is in fact similar to the Token Ring protocol presented in Section 2 except that $N > 3$ and t_i is a binary variable. Dijkstra [12] has shown that such a protocol is non-stabilizing. To transform this protocol to a self-stabilizing version thereof, we add a superposed variable x_i to each process P_i. Each process P_i can also read its predecessor's superposed variable x_{i-1}. Let $I' = \mathcal{E}(I)$ be the invariant of this transformed protocol. Let the new protocol have the following actions for P_0 and other processes P_i where $i > 0$:

[1]This problem statement is an adaptation of the problem of adding fault tolerance in [32].

$$
\begin{array}{lll}
P_0 : t_{N-1} = 0 \wedge & t_0 = 0 & \longrightarrow t_0 := 1; \\
P_0 : t_{N-1} = 1 \wedge x_{N-1} = 0 \wedge t_0 = 1 & & \longrightarrow t_0 := 0;\ x_0 := 1 - x_0; \\
P_0 : t_{N-1} = 1 \wedge x_{N-1} = 1 \wedge t_0 = 1 \wedge x_0 = 1 & & \longrightarrow t_0 := 0; \\
P_i : \ t_{i-1} = 0 \wedge & t_i = 1 & \longrightarrow t_i := 0;\ x_i := 1 - x_i; \\
P_i : \ t_{i-1} = 1 \wedge \ x_{i-1} = 1 \wedge t_i = 0 & & \longrightarrow t_i := 1; \\
P_i : \ t_{i-1} = 1 \wedge \ x_{i-1} = 0 \wedge (t_i = 0 \vee x_i = 1) & & \longrightarrow t_i := 1;\ x_i := 0;
\end{array}
$$

This protocol is stabilizing for rings of size $N = 2, \ldots, 7$ but contains a livelock when $N = 8$. We have found that given this topology and underlying protocol, no self-stabilizing token ring exists for $N \geq 8$. In Section 5, we expand the state space further towards synthesizing a self-stabilizing token rings with constant number of states for each process.

Let us check that the expansion of state space preserves the behavior of the underlying two-bit token ring protocol for a ring of size $N = 3$. Figure 2 shows the transition systems of the underlying and transformed protocols, where each state is a node and each arc is a transition. Legitimate states are boxed and transitions within these states are black, while transitions from illegitimate states are gray. Using the conditions in the problem statement: (1) $I = \mathcal{H}(I')$ is true since $I' = \mathcal{E}(I)$, (2) $\forall s \in \textsc{Pre}(\delta_p) \cap I : \mathcal{E}(s) \subseteq \textsc{Pre}(\delta_{p'})$ holds since all invariant states of p' have outgoing transitions, (3) $\delta_p | I = \mathcal{H}(\{(s'_0, s'_1) \mid (s'_0, s'_1) \in (\delta_{p'} | I') \wedge \mathcal{H}(s'_0) \neq \mathcal{H}(s'_1)\})$ holds since there exists a transition in p' which matches each transition in the underlying protocol, (4) $\forall s \in \textsc{Pre}(\delta_p) \cap I : \delta_{p'} | \mathcal{E}(s)$ is cycle-free holds since there are no cycles in any of the boxed rows of p', and (5) convergence from $\neg I'$ to I' holds since there are not livelocks or deadlocks within $\neg I'$.

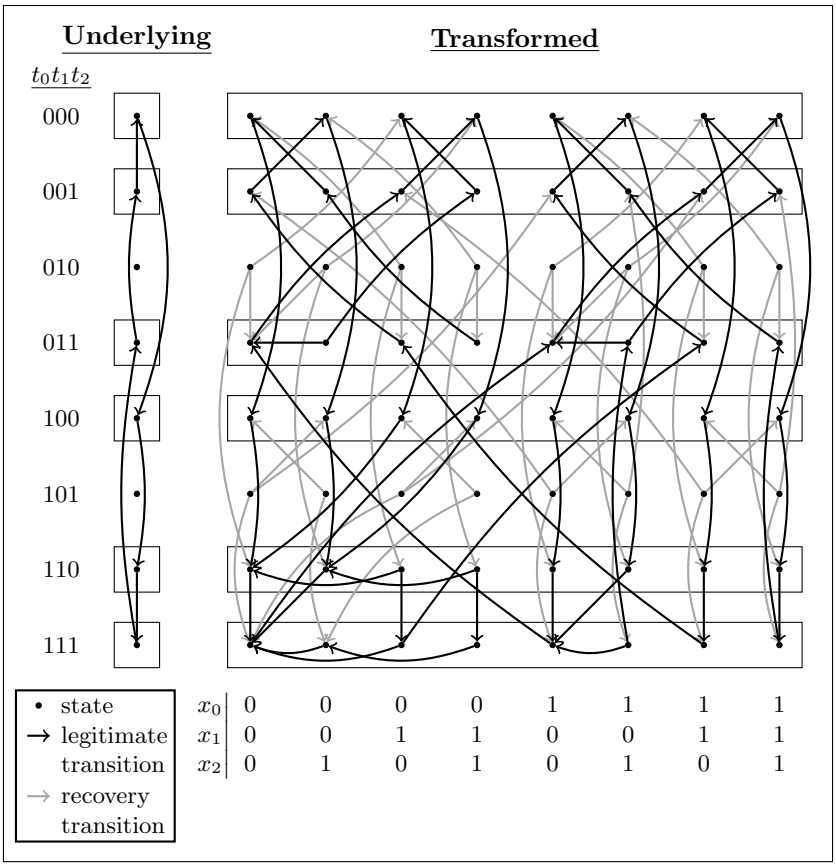

Figure 2: Underlying and transformed transition systems when $N = 3$

Deterministic, self-disabling processes. The following theorems show that the assumption of deterministic and self-disabling processes does not impact the completeness of any algorithm that solves Problem 3.1. In general, convergence is achieved by collaborative actions of all processes. That is, each process partially contributes to the correction of the global state of a protocol. As such, starting at a state $s_0 \in \neg I$, a single process may not be able to recover the entire system single-handedly. Thus, even if a process executes consecutive actions starting at s_0, it will reach a local deadlock from where other processes can continue their execution towards converging to I. The execution of consecutive actions of a process can be replaced by a single write action of the same process. As such, we assume that once a process executes an action it will be disabled until the actions of other processes enable it again. That is, processes are *self-disabling*.

Theorem 3.3. *Let p be a non-stabilizing protocol with invariant I. There is an SS version of p to I iff there is an SS version of p to I with self-disabling processes.*

Proof. The proof of right to left is straightforward, hence omitted. The proof of left to right is as follows. Let p_{ss} be an SS version of p to I, and P_j be a process of p_{ss}. Consider a computation prefix $\sigma = \langle s_0, s_1, \cdots, s_m \rangle$, where $m > 0$, $\forall i : 0 \leq i \leq m : s_i \notin I$, and all transitions in σ belong to P_j. Moreover, we assume that P_j becomes disabled at s_m. Now, we replace each transition $(s_i, s_{i+1}) \in \sigma$ $(0 \leq i < m)$ by a transition (s_i, s_m). Such a revision will not generate any deadlock states in $\neg I$. Moreover, if the inclusion of a transition (s_i, s_m), where $0 \leq i < m-1$, forms a non-progress cycle, then this cycle must have been already there in the protocol because a path from s_i to s_m already existed. Thus, this change does not introduce new livelocks in $\delta_p \mid \neg I$. □

Since transitions $(s_i, s_{i+1}) \in \sigma$, where $0 \leq i < m-1$, are removed without affecting the reachability of s_m from each $s_i \in \sigma$ $(0 \leq i < m)$, this change will not create any livelocks in $\delta_p \mid \neg I$. If the inclusion of a typical transition (s_i, s_m), where $0 \leq i < m-1$, forms a non-progress cycle, then this cycle must have been already there in the program because a path from s_i to s_m already existed.

Theorem 3.4. *Let p be a non-stabilizing protocol with invariant I. There is an SS version of p to I iff there is an SS version of p to I with deterministic processes.*

Proof. Any SS protocol with deterministic processes is an acceptable solution to Problem 3.1; hence the proof of right to left. Let p_{ss} be an SS version of p to I with non-deterministic but self-disabling processes. Moreover, let P_j be a self-disabling process of p_{ss} with two non-deterministic actions A and B originated at a global state $s_0 \notin I$, where A takes the state of p_{ss} to s_1 and B puts p_{ss} in a different state s_2. The following argument does not depend on s_1 and s_2 because by the self-disablement assumption, if s_1 and s_2 are in $\neg I$, then there must be a different process other than P_j that executes from there. Otherwise, the transitions (s_0, s_1) and (s_0, s_2) recover p_{ss} to I.

The state s_0 identifies an equivalence class of global states. Let s'_0 be a state in the equivalence class. The local state of P_j is identical in s_0 and s'_0, and the part of s_0 (and s'_0) that is unreadable to P_j could vary among all possible valuations. As such, corresponding to (s_0, s_1) and (s_0, s_2), we have transitions (s'_0, s'_1) and (s'_0, s'_2). To enforce determinism, we remove the action B from P_j. Such removal of B will not make s_0 and s'_0 deadlocked. Since s'_0 is an arbitrary state in the equivalence class, it follows that no deadlocks are created in $\neg I$. Moreover, removal of transitions cannot create livelocks. Therefore, the self-stabilization property of p_{ss} is preserved in the resulting deterministic protocol after the removal of B. □

4 Backtracking Search

We present an efficient and complete backtracking search algorithm to solve Problem 3.1. Backtracking search is a well-studied technique [36] which is easy to implement and can give very good results. We employ some optimizations to improve runtime over a naive approach, yet leave plenty of room for improvement.

4.1 Overview of the Search Algorithm

Like any other backtracking search, our algorithm incrementally builds upon a guess, or a partial solution, until it either finds a complete solution or finds that the guess is inconsistent. We decompose the *partial solution* into two parts: (1) an *under-approximation* formed by making well-defined decisions about the form of a solution, and (2) an *over-approximation* which is the set of remaining possible solutions (given the current under-approximation). In a standard constraint satisfaction problem, a backtracking search builds upon a partial assignment of variables, and it is inconsistent when the constraints upon those assigned variables are broken (i.e., the under-approximation causes a conflict) or the constraints cannot be satisfied by the remaining variable assignments (i.e., the over-approximation cannot contain a solution). In our context, a backtracking search builds a set of actions to form a protocol, and it is inconsistent when a livelock is formed by the chosen "delegate" actions (i.e., the under-approximation causes a conflict) or some deadlock cannot be resolved by the remaining "candidate" actions (i.e., the over-approximation cannot contain a solution). Each time a choice is made to build upon the under-approximation, the current under/over-approximations are saved at decision level j and a copy along with the new choice are placed at level $j+1$. If the guess at level $j+1$ is found to be inconsistent, we move back to level j and discard the choice which brought us to level $j+1$. If the guess at level 0 is found to be inconsistent, then enough guesses have been tested to determine that no solution exists.

Our algorithm starts in the AddStabilization function (Algorithm 1) which takes a non-stabilizing protocol p, its invariant I, its safety specifications *badTrans*, its required actions `delegates` and a mapping $\mathcal{E} : S_p \rightarrow S_{p'}$ as input and generates a self-stabilizing version of p in the variable `delegates`. The algorithm AddStabilization constructs a list of candidate actions that can be included in a solution and an empty list of actions which form the under-approximation. These lists are respectively denoted by the `candidates`

and `delegates` variables. To begin at decision level 0, we call the recursive search function AddStabilizationRec. This function continues to move actions from `candidates` to the under-approximation `delegates` while no inconsistencies are found (that is, the protocol formed by `delegates` contains no livelocks, and the actions from `candidates` can potentially resolve all remaining deadlocks). Each time an action is added to the under-approximation, the function AddStabilizationRec recurses, moving to the next decision level. When the partial solution is found to be inconsistent, AddStabilizationRec returns **false**. If a complete solution is found, AddStabilizationRec returns **true**, and the solution itself is contained in the variable passed to AddStabilizationRec as the `delegates` parameter (i.e., a return argument). AddStabilization follows the same convention for returning success and failure.

Throughout the search, the ReviseActions function is used to add delegate actions, remove candidate actions, and check for inconsistencies. Like the other functions, if it finds the partial solution to be inconsistent, it returns **false**. Otherwise, it returns **true**, and its modified `delegates` and `candidates` parameters are accessible to the calling function. During the updates, ReviseActions can add actions to delegates and remove actions from candidates if it does not affect completeness. For example, if only one action remains which can resolve a certain deadlock, then that action can and will be added to the under-approximation without the need to recurse to a new decision level.

To choose the next action to add to the partial solution, AddStabilizationRec calls PickAction. A simple implementation of PickAction can return any candidate action without sacrificing correctness. Thus, for reasoning about correctness, the reader may assume an arbitrary candidate is chosen. Nonetheless, to improve the performance of the search, our implementation resolves the deadlocks that the fewest candidate actions can resolve. By picking one of the n actions which resolve one of these deadlocks, we guarantee that if the partial solution is found to be inconsistent, we only need to try $n - 1$ more candidate actions to resolve the deadlock before backtracking. This is analogous to the *minimum remaining values* heuristic [36] which is commonly used in backtracking search for constraint satisfaction problems.

4.2 Details of the Search Algorithm

Misusing C/C++ notation, we prefix a function parameter with an ampersand (`&`) if modifications to it will affect its value in the caller's scope (i.e., it is a return parameter).

AddStabilization. Algorithm 1 is the entry point of our backtracking algorithm. This function returns **true** *iff* a self-stabilizing protocol is found which will then be formed by the actions in `delegates`. Initially, the function determines all possible candidate actions, where actions are minimal in that they cannot be represented equivalently by multiple actions. That is, each candidate action of a process (or set of symmetric processes) is formed by a single valuation of all readable variables in its guard and a single valuation of all writable variables in its assignment statement. Next, the function determines which actions are explicitly required (Line 2) or disallowed by the safety specification (Line 4). As an optimization which is not shown in the algorithm, we can disallow the invariant from changing (i.e., enforce $I' = \mathcal{E}(I)$), allowing us to include actions which break closure in the `dels` set on Line 4. We invoke ReviseActions to include `adds` in the under-approximation and remove `dels` from the over-approximation on Line 6. If the resulting partial solution is consistent, then the recursive version of this function (AddStabilizationRec) is called. Otherwise, a solution does not exist.

Algorithm 1 Backtracking algorithm for solving Problem 3.1.

AddStabilization(p: protocol, I: state predicate, $\mathcal{E}$: mapping $S_p \to S_{p'}$, `&delegates`: protocol actions, *badTrans*: safety specifications)

Output: Return **true** when a solution `delegates` can be found. Otherwise, **false**.

```
1: let candidates be the set of all possible actions from E(δ_p)
2: let adds := delegates
3: delegates := ∅
4: let dels be a set of actions from candidates such that dels ∈ badTrans
5: let I' := ∅
6: if ReviseActions(p, I, E, &delegates, &candidates, &I', adds, dels) then
7:   return AddStabilizationRec(p, I, E, &delegates, candidates, I')
8: else
9:   return false
10: end if
```

AddStabilizationRec. Algorithm 2 defines the main recursive search. Like AddStabilization, it returns **true** if and only if a self-stabilizing protocol is found which is formed by the actions in `delegates`. This function continuously adds candidate actions to the under-approximation `delegates` as long as candidate actions exist. If no candidates remain, then `delegates` and over-approximation `delegates` $\cup$ `candidates` of the protocol are identical. If ReviseActions did not find anything wrong, then `delegates` is self-stabilizing already, hence the successful return on Line 16.

On Line 2 of AddStabilizationRec, a candidate action A is chosen by calling PickAction (Algorithm 4). Any candidate action may be picked without affecting the search algorithm's correctness, but the next section explains a heuristic we use to pick certain candidate actions over others to improve search efficiency. After picking an action, we copy the current partial solution into `next_delegates` and `next_candidates`, and add the action A on Line 6. If the resulting partial solution is consistent, then we recurse by calling AddStabilizationRec. If that recursive call finds a self-stabilizing protocol, then it will store its actions in `delegates` and return successfully. Otherwise, if action A does not yield a solution, we will remove it from the candidates on Line 12. If this removal makes the partial solution unable to form a self-stabilizing protocol, return in failure; otherwise, continue the loop.

Algorithm 2 Recursive backtracking function to add stabilization.

AddStabilizationRec(p, I, $\mathcal{E}$, `&delegates`, `candidates`, I')

Output: Return **true** if `delegates` contains the solution. Otherwise, return **false**.

```
1: while candidates ≠ ∅ do
2:   let A := PickAction(p, E, delegates, candidates, I')
3:   let next_delegates := delegates
4:   let next_candidates := candidates
5:   let I'' := ∅
6:   if ReviseActions(p, I, E, &next_delegates, &next_candidates, &I'', {A}, ∅) then
7:     if AddStabilizationRec(p, I, E, &next_delegates, next_candidates, I'') then
8:       delegates := next_delegates {Assign the actions to be returned.}
9:       return true
10:    end if
11:  end if
12:  if not ReviseActions(p, I, E, &delegates, &candidates, &I', ∅, {A}) then
13:    return false
14:  end if
15: end while
16: return true
```

ReviseActions. Algorithm 3 is a key component of the backtracking search. ReviseActions performs five tasks: it (1) adds actions to the under-approximated protocol by moving the `adds` set from `candidates` to `delegates`; (2) removes safety-violating actions from the over-approximated protocol by removing the `dels` set from `candidates`; (3) enforces self-disablement and determinism (see Theorem 3.3) which results in removing more actions from the over-approximated protocol; (4) computes the maximal invariant I' and transitions $(\delta_{p'}|I')$ in the expanded state space given the current under/over-approximations, and (5) verifies constraints 1 to 4 of Problem 3.1, livelock freedom of the under-approximation in $\neg I'$, and weak convergence of the over-approximation to I'. If the check fails, then ReviseActions returns **false**. Finally, ReviseActions invokes the function CheckForward to infer actions which must be added to the under-approximation or removed from the over-approximation. (A simple implementation of CheckForward can always return **true** without inferring any new actions.)

A good ReviseActions implementation should provide early detection for when `delegates` and `candidates` cannot be used to form a self-stabilizing protocol. At the same time, since the function is called whenever converting candidates to delegates or removing candidates, it cannot have a high cost. Therefore, we do checks that make sense, such as ensuring that actions in `delegates` do not form a livelock and that actions in `delegates` $\cup$ `candidates` provide weak stabilization.

Line 12 calls CheckForward which infers new actions to add and remove, and it may return **false** only when it determines that the partial solution cannot be used to form a self-stabilizing protocol. If some new actions are inferred to be added or removed, simply call ReviseActions again to add or remove them. A very simple implementation of CheckForward can always return **true** without inferring any new actions.

Algorithm 3 Add `adds` to under-approximation and removing `dels` from over-approximation.

ReviseActions(p, I, $\mathcal{E}$, `&delegates`, `&candidates`, `&`I', `adds`, `dels`)

Output: Return **true** if `adds` can be added to `delegates` and `dels` can be removed from `candidates`, and I' can be revised accordingly. Otherwise, return **false**.

1: `delegates` := `delegates` $\cup$ `adds`
2: `candidates` := `candidates` $\setminus$ `adds`
3: **for** $A \in$ `adds` **do**
4: Add each candidate action B to `dels` if it belongs to the same process as A and satisfies one of the following conditions:
- A enables B (enforce self-disabling process)
- B enables A (enforce self-disabling process)
- A and B are enabled at the same time (enforce determinism)

{This adds actions to `dels` from `candidates` which are now trivially unnecessary for stabilization}
5: **end for**
6: `candidates` := `candidates` $\setminus$ `dels`
7: Compute the maximal I' and $(\delta_{p'}|I')$ such that:
- $I' \subseteq \mathcal{E}(I)$
- $(\delta_{p'}|I')$ transitions can be formed by actions in `delegates` $\cup$ `candidates`
- Constraints 2, 3, and 4 of Problem 3.1 hold

8: Perform the following checks to see if:
- $I = \mathcal{H}(I')$
- All transitions of `delegates` beginning in I' are included in $(\delta_{p'}|I')$
- The protocol formed by `delegates` is livelock-free in $\neg I'$
- Every state in $\neg I'$ has a computation prefix by transitions of `delegates`$\cup$`candidates` that reaches some state in I'

9: **if** all checks pass **then**
10: `adds` := $\emptyset$
11: `dels` := $\emptyset$
12: **if** CheckForward(p, I, $\mathcal{E}$, `delegates`, `candidates`, I', `&adds`, `&dels`) **then**
13: **if** `adds` $\neq \emptyset$ **or** `dels` $\neq \emptyset$ **then**
14: **return** ReviseActions(p, I, $\mathcal{E}$, `&delegates`, `&candidates`, `&`I', `adds`, `dels`)
15: **end if**
16: **return true**
17: **end if**
18: **end if**
19: **return false**

Theorem 4.1 (Completeness)**.** *The* AddStabilization *algorithm is complete.*

Proof. We want to show that if AddStabilization returns **false**, then no solution exists. Since each candidate action is minimal in that it cannot be broken into separate actions, and we begin by considering all such actions as candidates, a subset of the candidate actions must form a self-stabilizing protocol if and only if such a protocol exists. Observe that AddStabilizationRec follows the standard backtracking [30] procedure where (1) a new decision level to add a candidate action to the under-approximation, and (2) we backtrack and remove that action from the candidates when an inconsistency (which cannot be fixed by adding to the under-approximation) is discovered by ReviseActions at that new decision level. Note also that, even though ReviseActions removes candidate actions in order to enforce deterministic and self-disabling processes, but we know by Theorems 3.4 and 3.3 that this will not affect the existence of a self-stabilizing protocol. Thus, the search algorithm will test every subset of the initial set of candidate actions where processes are deterministic and self-disabling and ReviseActions did not find an inconsistency in a smaller subset. □

Theorem 4.2 (Soundness)**.** *The* AddStabilization *algorithm is sound.*

Proof. We want to show that if AddStabilization returns **true**, then it has found a self-stabilizing protocol formed by the actions in `delegates`. Notice that when AddStabilizationRec returns **true**, the AddStabilization (Line 7) or AddStabilizationRec (Line 8) function which called it simply returns **true** with the same `delegates` set. The only other case when **true** is returned is on `candidates` is empty in AddStabilizationRec (Line 16). Notice that to get to this point, ReviseActions must have been called and must have returned **true** after emptying the `candidates` set and performing the checks on Line 8. When `candidates` is empty, ReviseActions verifies that the protocol formed by `delegates` satisfies the constraints of Problem 3.1. Thus, a return value of **true** corresponds with the actions of `delegates` forming a self-stabilizing protocol, and those actions are subsequently the solution found by AddStabilization. □

4.3 Picking Actions via the Minimum Remaining Values Heuristic

The worst-case complexity of a depth-first backtracking search is determined by the branching factor b and depth d of its decision tree, evaluating to $O(b^d)$. We can therefore tackle this complexity by reducing the branching factor. To do this, we use a minimum remaining values (MRV) heuristic in `PickActions`. MRV is classically applied to constraint satisfaction problems by assigning a value to a variable which has the minimal remaining candidate values. In our setting, we pick an action which resolves a deadlock with the minimal number of remaining actions which can resolve it.

Algorithm 4 shows the details of PickAction that keeps an array `deadlock_sets`, where each element `deadlock_sets[`i`]` contains all the deadlocks that are resolved by exactly i candidate actions. We initially start with array size |`deadlock_sets`| = 1 and with `deadlock_sets[0]` containing all unresolved deadlocks. We then loop through all candidate actions, shifting deadlocks to the next highest element in the array (bubbling up) for each candidate action which resolves them. After building the array, we find the lowest index i for which the deadlock set `deadlock_sets[`i`]` is nonempty, and then return an action which can resolve some deadlock in that set. Line 21 can only be reached if either the remaining deadlocks cannot be resolved (but ReviseActions catches this earlier) or all deadlocks are resolved.

Algorithm 4 Pick an action using the most-constrained variable heuristic.

PickAction(p, $\mathcal{E}$, `delegates`, `candidates`, I')

Output: Next candidate action to pick.

1: **let** `deadlock_sets` be a single-element array, where `deadlock_sets[0]` holds a set of deadlocks in $\neg I' \cup \text{Pre}(\mathcal{E}(\delta_p))$ which actions in `delegates` do not resolve.
2: **for all** `action` $\in$ `candidates` **do**
3: **let** $i :=$ |`deadlock_sets`|
4: **while** $i > 0$ **do**
5: $i := i - 1$
6: **let** `resolved` := `deadlock_sets`$[i] \cap \text{Pre}($`action`$)$
7: **if** `resolved` $\neq \emptyset$ **then**
8: **if** $i =$ |`deadlock_sets`| $- 1$ **then**
9: **let** `deadlock_sets`$[i+1] := \emptyset$ {Grow array by one element}
10: **end if**
11: `deadlock_sets`$[i]$:= `deadlock_sets`$[i] \setminus$ `resolved`
12: `deadlock_sets`$[i+1]$:= `deadlock_sets`$[i+1] \cup$ `resolved`
13: **end if**
14: **end while**
15: **end for**
16: **for** $i = 1, \ldots,$ |`deadlock_sets`| $- 1$ **do**
17: **if** `deadlock_sets`$[i] \neq \emptyset$ **then**
18: **return** An action from `candidates` which resolves a deadlock in `deadlock_sets[i]`.
19: **end if**
20: **end for**
21: **return** An action from `candidates`. {This line may never execute!}

4.4 Optimizing the Decision Tree

This section presents the technique that we use to improve the efficiency of our backtracking algorithm.

Forward Checking. Forward checking can be used to prune candidate actions from the over-approximation and infer actions to add to the under-approximation. A classical implementation of CheckForward would try to add each candidate action to the `delegates` list, using the approach in ReviseActions to check for validity. We have tried this, and it is expectedly slow due to the large number of candidate actions combined with the high cost of cycle detection and reachability analysis.

Conflicts. Our approach instead uses sets of actions which conflict with each other. During the search, whenever ReviseActions detects an inconsistency, we record a minimal set of actions from the under-approximation which form a cycle or cause the over-approximation to be insufficient to provide weak stabilization. For larger systems where cycle detection is costly, the minimization phase should be skipped. Even without minimization, checking against conflict sets is much cheaper than the classical forward checking approach and allows us to avoid duplicate work.

Randomization and Restarts. When using the standard control flow of a depth-first search, a bad choice near the top of the decision tree can lead to infeasible runtime. This is the case since the bad decision exists in the partial solution until the search backtracks up the tree enough to change the decision. To limit the search time in these branches, we employ a method outlined by Gomes et al. [18] which combines randomization with restarts. In short, we limit the amount of backtracking to a certain height (we use 3). If the search backtracks past the height limit, it forgets the current decision tree and restarts from the root. To avoid trying the same unfruitful decisions after a restart, PickAction is implemented to randomly select a candidate action after the MRV heuristic is applied.

Parallel Search. When a protocol is difficult to find, many AddStabilization tasks can run in parallel to increase the chance of finding a solution. The tasks avoid overlapping computations naturally due to the randomization used in PickAction. Further, the search tasks share conflicts with each other, allowing

CheckForward to leverage knowledge obtained by other tasks. In our MPI implementation, conflict dissemination occurs between tasks using a virtual network topology formed by a generalized Kautz graph [25] of degree 4. This topology has a diameter logarithmic in the number of nodes and is fault-tolerant in that multiple paths between two nodes ensure message delivery. That is, even if some nodes are performing costly cycle detection and do not check for incoming messages, they will not necessarily slow the dissemination of new conflicts.

4.5 Exploiting Symmetry

Parameterized Systems. For protocols which scale to many processes, each process behaves as one of a finite number of template processes. For example, consider a bidirectional ring topology of N processes $P_0, \ldots, P_{N-1}$ where each P_i owns a variable x_i. We might want to enforce that all processes have the same code. That is, P_i has action $x_{i-1} = a \wedge x_i = b \wedge x_{i+1} = c \longrightarrow x_i := d$ if and only if each other process P_j has action $x_{j-1} = a \wedge x_j = b \wedge x_{j+1} = c \longrightarrow x_j := d$. In this case, all actions of this form would be treated as one indivisible candidate action during synthesis.

This has the advantage that the set of candidate actions does not vary with the ring size N as long as the domain of x_i does not change. Since we would like to find a protocol which is self-stabilizing for arbitrary N, we can consider multiple systems at once, perhaps those of size 3, 4, 5, 6, and 7. That is, the checks in ReviseActions and the deadlock ranking in PickAction would apply to all systems. When a solution is found, we can further check if it is self-stabilizing for $N = 8$ and beyond (up to some feasible limit). If the check fails due to a livelock, we record a conflict and continue searching. If the check fails due to a deadlock, it is not appropriate to add a conflict since it may be possible to add actions to resolve the deadlock. Our implementation simply restarts if a deadlock is found during this verification phase since we feel this issue is rare (our case studies never encounter it).

Symmetric Links. In some cases, we find it useful to enforce symmetry in how a process can modify variables. For example, consider a bidirectional ring topology of N processes $P_0, \ldots, P_{N-1}$ where each P_i owns a variable x_i. We might want to enforce that each P_i has action $x_{i-1} = a \wedge x_i = b \wedge x_{i+1} = c \longrightarrow x_i := d$ if and only if P_i also has the action $x_{i-1} = c \wedge x_i = b \wedge x_{i+1} = a \longrightarrow x_i := d$. In this case, both actions would be treated as one indivisible candidate action during synthesis.

Intuitively, this models a scenario where a process P_i can communicate with its neighbors but nondeterministically confuses the x_{i-1} and x_{i+1} values. When carefully applied, this feature allows us to model the topology needed for solving ring orientation, where each process P_i has two neighbors P_j and P_k where $\{P_j, P_k\} = \{P_{i-1}, P_{i+1}\}$. Process P_i does not confuse which *values* come from P_j and P_k, but it does not have any sense which is P_{i-1} or P_{i+1}.

5 Case Studies

We introduce four distributed system problems: leader election, coloring, token circulation, and ring orientation. In the next section, we discuss how our search algorithm performs on each problem and options we must consider to make them feasible.

5.1 Leader Election

We use a bidirectional ring topology defined by Huang [24].

Non-Stabilizing Protocol. The non-stabilizing protocol is empty since once a leader is found, nothing should change. For rings of size N, each process P_i owns a variable $x_i \in \mathbb{Z}_N$ and can read both x_{i-1} and x_{i+1}.

Invariant. The invariant is defined as $\forall i \in \mathbb{Z}_N : ((x_{i-1} - x_i) \bmod N) = ((x_i - x_{i+1}) \bmod N)$. When N is prime, the processes hold unique values in legitimate states (i.e., $\{x_0, \ldots, x_{N-1}\} = \{0, \ldots, N-1\}$). A process P_i can therefore consider itself the leader when its value is $x_i = 0$. When N is composite, the invariant is still useful in that it guarantees that each process P_i agrees on the same value $((x_{i-1} - x_i) \bmod N)$.

Superposed Variables. No variables are superposed, therefore $I' = I$ for synthesis.

5.2 Four-Coloring on Kautz Graphs

Kautz graphs have many applications in peer-to-peer networks due to their constant degree, optimal diameter, and low congestion [33]. Recall that in our parallel search algorithm, we use a generalized Kautz graph of degree 4 to disseminate conflicting sets of actions.

Non-Stabilizing Protocol. The non-stabilizing protocol is empty since once a valid coloring is found, nothing should change. This case study uses generalized Kautz graphs of degree 2. Each process P_i owns a variable $x_i \in \mathbb{Z}_4$ which denotes its color. From [25], an arc exists from vertex i to vertex j if and only if $j = -(2i + q + 1) \bmod N$ for some $q \in \{0, 1\}$. We interpret this as a topology, letting P_j read the color of P_i when an arc exists from i to j in the graph. Some generalized Kautz graphs contain self-loops which would make a coloring protocol impossible. In such a case, where vertex j has a self-loop an arc from vertex i, we simply remove the self loop and add another arc from vertex i.

Specifically, P_i reads $x_{j'}$ and $x_{k'}$ which are computed from the following indices j and k. When N is even, these are computed as $j = \lfloor \frac{N-1-i}{2} \rfloor \bmod N$ and $k = (j + \frac{N}{2}) \bmod N$. When N is odd, they are $j = \frac{N-1}{2}(i + 1) \bmod N$ and $k = \frac{N-1}{2}(i + 2) \bmod N$. The indices j' and k' are equal to j and k respectively if $j \neq i$ and $k \neq i$. Otherwise, $j = i$ or $k = i$, then $j' = k' = k$ or $k' = j' = j$ respectively.

Invariant. We desire a silent coloring protocol, where no two adjacent processes have the same color. As such, the invariant is defined as $I \equiv \forall i \in \mathbb{Z}_N, q \in \mathbb{Z}_2 : ((i = -(2 * i + q + 1) \bmod N) \vee (x_i \neq x_{-(2i+q+1)}))$.

Superposed Variables. No variables are superposed, therefore $I' = I$ for synthesis.

5.3 Token Rings of Constant Space

Using the four-state token ring of Example 3.2, we consider small modifications to obtain a six-state version and the well-known three-bit version from Gouda and Haddix [21].

Non-Stabilizing Protocol. The non-stabilizing protocol uses a single bit to circulate a token around a unidirectional ring. Each process P_i can write a binary variable t_i and read t_{i-1}. The distinguished process P_0 behaves differently from each other process P_i where $i > 0$.

$$P_0 : t_{N-1} = t_0 \longrightarrow t_0 := 1 - t_0;$$
$$P_i : t_{i-1} \neq t_i \quad \longrightarrow t_i := t_{i-1};$$

Invariant. The invariant is defined as exactly one process in the non-stabilizing protocol being able to act. Formally this is written $I \equiv \exists!\, i \in \mathbb{Z}_N : ((i = 0 \wedge t_{i-1} = t_i) \vee (i \neq 0 \wedge t_{i-1} \neq t_i))$. Even though we superpose variables, we choose to enforce $I' = I$ during synthesis since the token ring of Gouda and Haddix [21] is self-stabilizing for this invariant.

Superposed Variables. In the six-state token ring, we give each process P_i a superposed variable $x_i \in \mathbb{Z}_3$ to write and allow P_i to read x_{i-1}. In the three-bit token ring, we give each process P_i two superposed binary variables e_i and $ready_i$ to write and allow P_i to read e_{i-1}. As is the case with the non-stabilizing protocol, we distinguish process P_0 but enforce symmetry on $P_1, \ldots, P_{N-1}$.

5.4 Ring Orientation

If anonymous processes have some method of finding each other, it is easy for them to form a ring topology. However, a problem arises when they must decide on a common direction around the ring. Israeli and Jalfon [26] give general solution for this protocol, though we focus on a version for odd-sized rings from Hoepman [23]. Hoepman's protocol uses token circulation to determine a common direction around the ring. By only considering rings of odd size, the number of tokens can be forced to be odd. Eventually, tokens of opposing directions will cancel and leave at least one token circulating.

Non-Stabilizing Protocol. The non-stabilizing protocol is empty since once the ring is oriented, nothing should change. Each process P_i is given two binary variables way_{2i} and way_{2i+1} to write and cannot read variables of other processes.

Invariant. The invariant states that all processes must decide on a common direction. We say that P_i has chosen a direction pointing to P_{i-1} (anticlockwise) if $way_{2i} = 1 \wedge way_{2i+1} = 0$, and P_i has chosen a direction pointing to P_{i+1} (clockwise) if $way_{2i} = 0 \wedge way_{2i+1} = 1$. Simply we can write the invariant as $I \equiv \forall j \in \mathbb{Z}_{2N} : way_{j-1} \neq way_j$.

Superposed Variables. We use a topology adapted from [23]. Each process P_i in the ring of size N is given two superposed binary variables $color_i$ and $phase_i$. Process P_i can read $color_{i-1}$ and $phase_{i-1}$ from P_{i-1} along with $color_{i+1}$ and $phase_{i+1}$ from P_{i+1}.

To reduce the number of candidate actions, we forbid P_i from acting when it has not chosen a direction (i.e., when $way_{2i} = way_{2i+1}$). In cases where P_i has not chosen a direction, we model it making an arbitrary choice by introducing new processes to make that arbitrary choice (since P_i is not allowed to act in this case). This allows a synthesis procedure to focus only on how each P_i will change its chosen direction, rather than how P_i will choose a direction when it has not yet chosen (which should be a trivial matter).

To model an unoriented ring, we enforce symmetry on the links between processes. That is, a process P_i has an action ($color_{i-1} = \alpha_0 \wedge color_{i+1} = \beta_0 \wedge phase_{i-1} = \alpha_1 \wedge phase_{i+1} = \beta_1 \wedge way_{2i} = \alpha_2 \wedge way_{2i+1} = \beta_2 \wedge color_i = \gamma_0 \wedge phase_i = \gamma_1 \longrightarrow way_{2i} := \alpha_3;\ way_{2i+1} := \beta_3;\ color_i := \gamma_2;\ phase_i := \gamma_3;$) if and only if it also has an action where the α and β values are respectively swapped ($color_{i-1} = \beta_0 \wedge color_{i+1} = \alpha_0 \wedge phase_{i-1} = \beta_1 \wedge phase_{i+1} = \alpha_1 \wedge way_{2i} = \alpha_2 \wedge way_{2i+1} = \alpha_2 \wedge color_i = \gamma_0 \wedge phase_i = \gamma_1 \longrightarrow way_{2i} := \beta_3;\ way_{2i+1} := \alpha_3;\ color_i := \gamma_2;\ phase_i := \gamma_3;$). In this way, process P_i cannot use any implicit knowledge from the input topology to determine direction. Rather, it must treat $color_{i-1}$, $phase_{i-1}$, and way_{2i} exactly the same as $color_{i+1}$, $phase_{i+1}$, and way_{2i+1}, only differentiating them by their values. Effectively those links are differentiated by the directional information specified by way_{2i} and way_{2i+1}.

Alternative Version. The general ring orientation protocol from Israeli and Jalfon [26] can be posed with some slight modifications. First increase the domain of each $color_i$ and $phase_i$ variable by one, giving $color_i, phase_i \in \mathbb{Z}_3$. Next, each process needs to know the direction of its neighbors, therefore we allow each process P_i to read way_{2i-1} and way_{2*i-2} from P_{i-1} and read way_{2*i+2} and way_{2*i+3} from P_{i+1}. These new readable variables should be bundled with the other variables in links to P_{i-1} and P_{i+1} respectively. We omit this version from our case studies due to a high number of candidate actions.

6 Experimental Results

Leader Election / Agreement. Huang's leader election protocol gives an agreement protocol for rings of composite size. We consider a ring of size 6 for synthesis as it is somewhat difficult for our randomized backtracking search to find a stabilizing protocol. Conversely, the ring of size 5 is usually solved within 8 restarts (using a backtracking height limit of 3). For the ring of size 6, we ran 25 trials of the parallel search using different numbers of MPI processes to measure the effect of parallelism on runtime. Averaging the times for taken by 8, 16, 32, and 64 processes, we get 2468.36, 1120.76, 659.96, and 371.96 seconds respectively. Figure 3 shows all time measurements and how they vary.

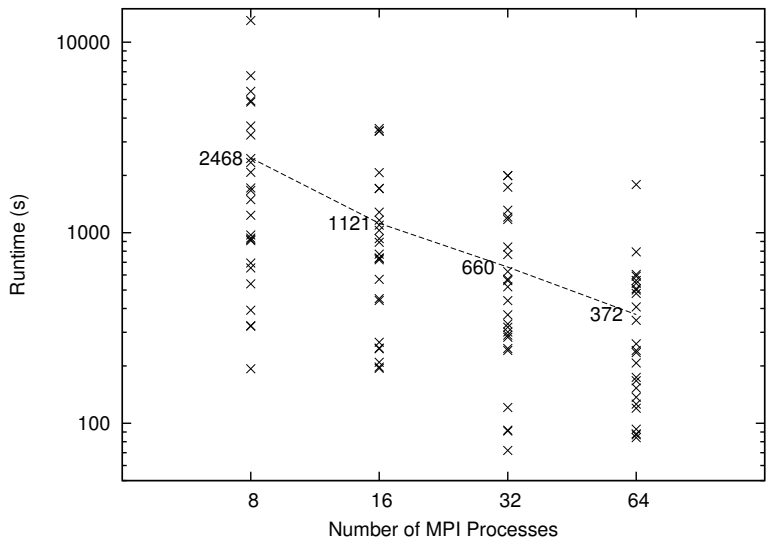

Figure 3: Runtimes of leader election synthesis, varying the MPI process count.

Four-Coloring on Kautz Graphs. On generalized Kautz graphs, we found that substantial knowledge can be gained when the readable variables of a process have a certain ordering. When P_i reads $x_{j'}$ and $x_{k'}$ in order as defined in Section 5.2, we are able to find a four-coloring protocol which is self-stabilizing for $N = 2, \ldots, 13$. However, if P_i must treat the $x_{j'}$ and $x_{k'}$ as symmetric links or without order (as with ring orientation), then a four-coloring protocol exists for $N = 2, \ldots, 7$ but not $N = 8$. The result holds if we allow P_i to also read the variables of processes which can read x_i, yet we preserve the symmetric link restriction.
Token Rings of Constant Space. As stated of the four-state token ring in Example 3.2, we found that no version exists which stabilizes for all rings of size $N = 2, \ldots, 8$. This trial was completed in 7451 seconds without restarts. Using the modified token ring topologies in Section 5.3, we were able to find a stabilizing protocols. Figures 4 and 5 show the search time for the six-state three-bit and token ring topologies. Each trial considers rings from size 2 up to some maximum size. To simplify analysis, restarts were not used in the search. Further, due to high cost, we do not consider rings of sizes 8 and 9 during the conflict minimization phase.

Some of the protocols from our experiments are (or appear to be) stabilizing for all ring sizes. We have verified these cases up to rings of size 12 using asynchronous execution and up to size 25 using synchronous execution, as cycle detection is faster. These generalizable cases were found from all searches, including trials involving rings of sizes 2 and 3. Some of these protocols converge to states with exactly one enabled process as in Dijkstra's token ring [12] but unlike three-bit token ring of Gouda and Haddix [21]. For these, we can redefine a token as the ability to act, but convergence time may suffer.

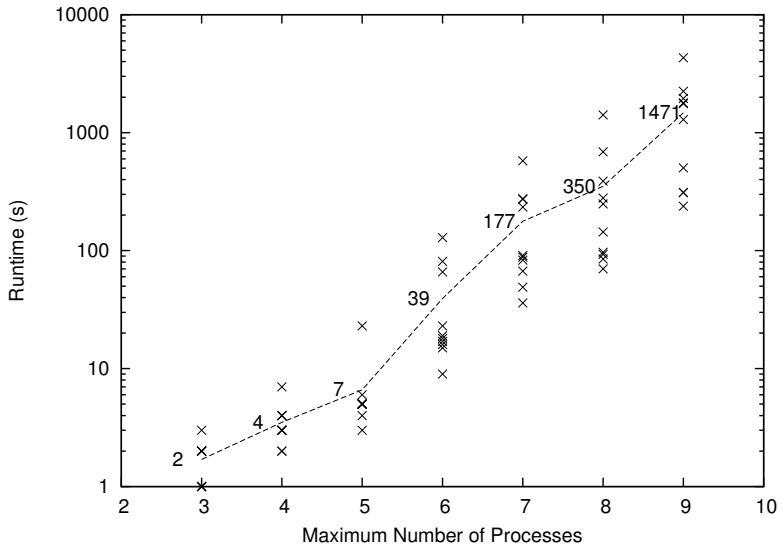

Figure 4: Runtimes of six-state token ring synthesis, varying the maximum ring size.

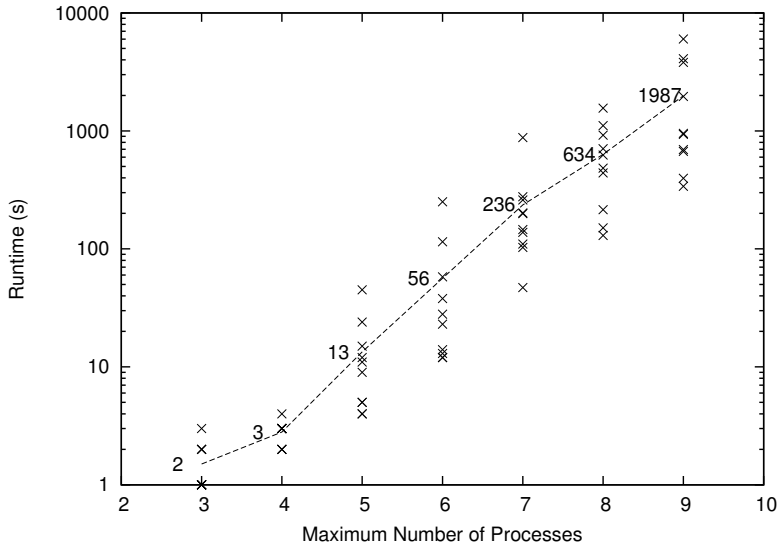

Figure 5: Runtimes of three-bit token ring synthesis, varying the maximum ring size.

For completeness, we give the actions of a stabilizing six-state token ring protocol. This protocol converges to having one process enabled, works under asynchronous and synchronous schedulers, and has one of the fastest convergence times of the six-state token rings that we synthesized. The distinguished process has 20 local transitions, while the other processes have 23 local transitions. These are represented below in a somewhat condensed form using 12 and 13 actions respectively.

$$
\begin{aligned}
&P_0 : t_{N-1} = 0 \wedge x_{N-1} \neq 2 \wedge t_0 = 0 \wedge x_0 = 0 \longrightarrow t_0 := 0;\ x_0 := 2;\\
&P_0 : t_{N-1} = 0 \wedge x_{N-1} = 0 \wedge t_0 = 1 \wedge x_0 \neq 2 \longrightarrow t_0 := 1;\ x_0 := 2;\\
&P_0 : t_{N-1} = 0 \wedge x_{N-1} = 1 \wedge \quad x_0 = 1 \longrightarrow t_0 := 1;\ x_0 := 2;\\
&P_0 : t_{N-1} = 0 \wedge x_{N-1} = 2 \wedge t_0 = 0 \wedge x_0 = 0 \longrightarrow t_0 := 1;\ x_0 := 0;\\
&P_0 : t_{N-1} = 0 \wedge x_{N-1} = 2 \wedge t_0 = 0 \wedge x_0 = 1 \longrightarrow t_0 := 1;\ x_0 := 1;\\
&P_0 : t_{N-1} = 0 \wedge x_{N-1} = 2 \wedge \quad x_0 = 2 \longrightarrow t_0 := 1;\ x_0 := 0;\\
&P_0 : t_{N-1} = 1 \wedge x_{N-1} \neq 2 \wedge t_0 = 0 \wedge x_0 = 0 \longrightarrow t_0 := 0;\ x_0 := 1;\\
&P_0 : t_{N-1} = 1 \wedge x_{N-1} \neq 2 \wedge t_0 = 1 \wedge x_0 = 1 \longrightarrow t_0 := 0;\ x_0 := 1;\\
&P_0 : t_{N-1} = 1 \wedge x_{N-1} = 0 \wedge t_0 = 1 \wedge x_0 = 0 \longrightarrow t_0 := 0;\ x_0 := 2;\\
&P_0 : t_{N-1} = 1 \wedge x_{N-1} = 2 \wedge t_0 = 0 \wedge x_0 \neq 2 \longrightarrow t_0 := 0;\ x_0 := 2;\\
&P_0 : t_{N-1} = 1 \wedge x_{N-1} = 2 \wedge t_0 = 1 \wedge x_0 = 2 \longrightarrow t_0 := 0;\ x_0 := 2;\\
&P_0 : \quad x_{N-1} = 1 \wedge t_0 = 1 \wedge x_0 = 0 \longrightarrow t_0 := 1;\ x_0 := 2;
\end{aligned}
$$

$$
\begin{aligned}
&P_i :\ t_{i-1} = 0 \wedge x_{i-1} \neq 2 \wedge t_i = 0 \wedge x_i \neq 1 \longrightarrow t_i := 0;\ x_i := 1;\\
&P_i :\ t_{i-1} = 0 \wedge x_{i-1} = 0 \wedge t_i = 1 \wedge x_i = 2 \longrightarrow t_i := 1;\ x_i := 0;\\
&P_i :\ t_{i-1} = 0 \wedge x_{i-1} = 1 \wedge t_i = 1 \quad \longrightarrow t_i := 0;\ x_i := 1;\\
&P_i :\ t_{i-1} = 0 \wedge x_{i-1} = 2 \wedge t_i = 1 \quad \longrightarrow t_i := 0;\ x_i := 2;\\
&P_i :\ t_{i-1} = 0 \wedge x_{i-1} = 2 \wedge \quad x_i = 0 \longrightarrow t_i := 0;\ x_i := 2;\\
&P_i :\ t_{i-1} = 1 \wedge x_{i-1} = 0 \wedge t_i = 0 \wedge x_i = 0 \longrightarrow t_i := 1;\ x_i := 0;\\
&P_i :\ t_{i-1} = 1 \wedge x_{i-1} = 0 \wedge t_i = 0 \wedge x_i = 1 \longrightarrow t_i := 1;\ x_i := 1;\\
&P_i :\ t_{i-1} = 1 \wedge x_{i-1} = 0 \wedge \quad x_i = 2 \longrightarrow t_i := 1;\ x_i := 1;\\
&P_i :\ t_{i-1} = 1 \wedge x_{i-1} = 1 \wedge t_i = 0 \wedge x_i = 2 \longrightarrow t_i := 0;\ x_i := 0;\\
&P_i :\ t_{i-1} = 1 \wedge x_{i-1} = 1 \wedge t_i = 1 \wedge x_i = 2 \longrightarrow t_i := 1;\ x_i := 0;\\
&P_i :\ t_{i-1} = 1 \wedge x_{i-1} = 2 \wedge t_i = 1 \wedge x_i = 1 \longrightarrow t_i := 1;\ x_i := 2;\\
&P_i :\ t_{i-1} = 1 \wedge x_{i-1} = 2 \wedge \quad x_i = 0 \longrightarrow t_i := 1;\ x_i := 2;\\
&P_i : x_{i-1} = 2 \wedge \quad t_i = 0 \wedge x_i = 1 \longrightarrow t_i := 0;\ x_i := 2;
\end{aligned}
$$

Ring Orientation. This search used 32 MPI processes and considered rings of size 3, 5, and 7 simultaneously. As with large token rings, we did not perform conflict minimization for the ring of size 5. For the ring of size 7, we automatically ran verifications after finding a stabilizing protocol for the smaller rings. If this verification failed, then the search continued as if the ReviseActions function returned **false**. A stabilizing protocol was found in 281 seconds (0.078 hours), and most MPI search tasks restarted 4 times after exceeding the backtracking limit of 3. After obtaining the solution, we verified that is stabilizing for rings of sizes 9 and 11 in 509 seconds (0.14 hours) and 26880 seconds (7.47 hours) respectively.

7 Related Work and Discussion

This section discusses related work on manual and auotmated design of fault tolerance in general and self-stabilization in particular. Manual methods are mainly based on the approach of *design and verify*, where one designs a fault-tolerant system and then verifies the correctness of (1) functional requirements in the absence of faults, and (2) fault tolerance requirements in the presence of faults. For example, Liu and Joseph [34] provide a method for augmenting fault-intolerant systems with a set of new actions that implement fault

tolerance functionalities. Katz and Perry [27] present a general (but expensive) method for global snapshot and reset towards adding convergence to non-stabilizing systems. Varghese [38] and Afek *et al.* [4] provide a method based on local checking for global recovery of locally correctable protocols. Varghese [39] also proposes a counter flushing method for detection and correction of global predicates.

Methods for automated design of fault tolerance can be classified into specification-based and incremental addition techniques. In specification-based methods [7] the inter-process synchronization mechanisms of programs are derived from formal specifications often specified in some variant of temporal logic. By contrast, in addition methods one incorporates fault tolerance functionalities in an existing system while ensuring that the resulting program would still satisfy its specifications in the absence of faults. For example, Kulkarni and Arora [32] study the addition of three different levels of fault tolerance, namely failsafe, nonmasking and masking fault tolerance. A *failsafe* protocol meets its safety specifications under all circumstances (i.e., in the absence and in the presence of faults), whereas a *nonmasking* protocol ensures recovery to invariant from the set of states that are reachable in the presence of faults (but not necessarily equal to the entire state space). A *masking* fault-tolerant program is both failsafe and nonmasking. Ebnenasir [14] has investigated the automated addition of recovery to distributed protocols for types of faults other than transient faults. Nonetheless, their method has the option to remove deadlock states by making them unreachable. A similar choice is utilized in Bonakdarpour and Kulkarni's work [9] on adding progress properties to distributed protocols. This is not an option in the addition of self-stabilization; recovery should be provided from any state in protocol state space.

Since it is unlikely that an efficient method exists for algorithmic design of self-stabilization [28,29], most existing techniques [1,3,16,17,40] are based on sound heuristics. For instance, Abujarad and Kulkarni [1,3] present a heuristic for adding convergence to locally-correctable systems. Zhu and Kulkarni [40] give a genetic programming approach for the design of fault tolerance, using a fitness function to quantify how close a randomly-generated protocol is to being fault-tolerant. Farahat and Ebnenasir [17] provide a lightweight method for designing self-stabilization even for non-locally correctable protocols. They also devise [16] a swarm method for exploiting the computational power of computer clusters towards automated design of self-stabilization. While the swarm synthesis method inspires the proposed work in this paper, it has two limitations: it is incomplete and forbids any change in the invariant.

Applications. This work has applications in any area where convergence or equilibrium is of interest. As an example application in economics, Gouda and Acharya [20] investigate self-stabilizing systems that converge to legitimate states that represent Nash equilibrium. In this context, our proposed algorithm and tool can enable automated design of agents in an economic system where achieving Nash equilibrium is a global objective. Moreover, Protocon can be integrated in model-driven development environments (such as Unified Modeling Language [35] and Motorola WEAVER [11]) for protocol design and visualization.

Limitations. We are currently investigating several challenges. First, due to scalability issues, Protocon can generate self-stabilizing protocols with a small number of processes (depending on the underlying communication topology). To tackle this limitation, we are investigating the use of theorem proving in generalizing the small protocols synthesized by Protocon. Thus, the combination of Protocon (i.e., synthesizer) and a theorem prover will provide a framework for automated design of parameterized self-stabilizing protocols, where a protocol comprises several families of symmetric processes. Second, to mitigate the scalability problem, we will devise more efficient cycle detection methods since our experimental results indicate that significant resources are spent for cycle detection. Third, randomization

8 Conclusions and Future Work

This paper presents a method for algorithmic design of self-stabilization based on variable superposition and backtracking. Unlike existing algorithmic methods [1, 3, 16, 17] the proposed approach is sound and complete; i.e., if there is an SS solution, our algorithm will find it. We have devised sequential and parallel implementations of the proposed method in a software tool, called Protocon. Variable superposition allows us to systematically introduce computational redundancy where existing heuristics fail to generate a solution. Afterwards, we use the backtracking search to intelligently look for a self-stabilizing solution. The

novelty of our backtracking method lies in finding and sharing design conflicts amongst parallel threads to improve the efficiency of search. We have used Protocon to automatically generate self-stabilizing protocols that none of the existing heuristics can generate (to the best of our knowledge). For example, we have automatically designed an 8-state self-stabilizing token ring protocol for the same topology as the protocol manually designed by Gouda and Haddix [21]. We have even improved this protocol further by designing a 6-state version thereof available at `http://cs.mtu.edu/~apklinkh/protocon/`. Besides token rings, we have synthesized other protocols such as coloring on Kautz graphs, ring orientation and leader election on a ring.

We are currently investigating several extensions of this work. First, we would like to synthesize protocols such as Dijkstra's 4-state token chain and 3-state token ring [12], where the invariant and legitimate behavior cannot be expressed using the protocol's variables without essentially writing the self-stabilizing version. Second, we are using theorem proving techniques to figure out why a synthesized protocol may not be generalizable. Then, we plan to incorporate the feedback received from theorem provers in our backtracking method. A third extension is to leverage the techniques used in SAT solvers and apply them in our backtracking search.

References

[1] F. Abujarad and S. S. Kulkarni. Multicore constraint-based automated stabilization. In *11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 47–61, 2009.

[2] F. Abujarad and S. S. Kulkarni. Weakest invariant generation for automated addition of fault-tolerance. *Electronic Notes on Theoretical Computer Science*, 258(2):3–15, 2009.

[3] F. Abujarad and S. S. Kulkarni. Automated constraint-based addition of nonmasking and stabilizing fault-tolerance. *Theoretical Computer Science*, 412(33):4228–4246, 2011.

[4] Y. Afek, S. Kutten, and M. Yung. The local detection paradigm and its application to self-stabilization. *Theoretical Computer Science*, 186(1-2):199–229, 1997.

[5] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[6] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[7] P. C. Attie, anish Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(1):125–185, 2004.

[8] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.

[9] B. Bonakdarpour and S. S. Kulkarni. Revising distributed UNITY programs is NP-complete. In *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS)*, pages 408–427, 2008.

[10] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[11] T. Cottenier, A. van den Berg, and T. Elrad. Motorola weavr: Aspect and model-driven engineering. *Journal of Object Technology*, 6(7):51–88, 2007.

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

[13] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1990.

[14] A. Ebnenasir. *Automatic Synthesis of Fault-Tolerance*. PhD thesis, Michigan State University, May 2005.

[15] A. Ebnenasir and A. Farahat. A lightweight method for automated design of convergence. In *IEEE International Symposium on Parallel and Distributed Processing*, pages 219–230, 2011.

[16] A. Ebnenasir and A. Farahat. Swarm synthesis of convergence for symmetric protocols. In *Proceedings of the Ninth European Dependable Computing Conference*, pages 13–24, 2012.

[17] A. Farahat and A. Ebnenasir. A lightweight method for automated design of convergence in network protocols. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7(4):38:1–38:36, Dec. 2012.

[18] C. P. Gomes, B. Selman, and H. A. Kautz. Boosting combinatorial search through randomization. In J. Mostow and C. Rich, editors, *AAAI/IAAI*, pages 431–437. AAAI Press / The MIT Press, 1998.

[19] M. Gouda. The theory of weak stabilization. In *5th International Workshop on Self-Stabilizing Systems*, volume 2194 of *Lecture Notes in Computer Science*, pages 114–123, 2001.

[20] M. G. Gouda and H. B. Acharya. Nash equilibria in stabilizing systems. *Theor. Comput. Sci.*, 412(33):4325–4335, 2011.

[21] M. G. Gouda and F. F. Haddix. The stabilizing token ring in three bits. *Journal of Parallel and Distributed Computing*, 35(1):43–48, May 1996.

[22] M. G. Gouda and N. J. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.

[23] J.-H. Hoepman. Uniform deterministic self-stabilizing ring-orientation on odd-length rings. In G. Tel and P. M. B. Vitányi, editors, *WDAG*, volume 857 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 1994.

[24] S.-T. Huang. Leader election in uniform rings. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15:563–573, July 1993.

[25] M. Imase and M. Itoh. A design for directed graphs with minimum diameter. *IEEE Trans. Computers*, 32(8):782–784, 1983.

[26] A. Israeli and M. Jalfon. Uniform self-stabilizing ring orientation. *Information and Computation*, 104(2):175–196, 1993.

[27] S. Katz and K. Perry. Self-stabilizing extensions for message passing systems. *Distributed Computing*, 7:17–26, 1993.

[28] A. P. Klinkhamer and A. Ebnenasir. On the complexity of adding convergence. In *Proceedings of the 5th International Symposium on Fundamentals of Software Engineering (FSEN)*, pages 17–33, 2013.

[29] A. P. Klinkhamer and A. Ebnenasir. On the hardness of adding nonmasking fault tolerance. *IEEE Transactions on Dependable and Secure Computing*, 2014. In Press.

[30] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

[31] S. S. Kulkarni. *Component-based design of fault-tolerance*. PhD thesis, Ohio State University, 1999.

[32] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 82–93, London, UK, 2000. Springer-Verlag.

[33] D. Li, X. Lu, and J. Wu. Fissione: a scalable constant degree and low congestion dht scheme based on kautz graphs. In *INFOCOM*, pages 1677–1688. IEEE, 2005.

[34] Z. Liu and M. Joseph. Transformation of programs for fault-tolerance. *Formal Aspects of Computing*, 4(5):442–469, 1992.

[35] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[36] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[37] F. Stomp. Structured design of self-stabilizing programs. In *Proceedings of the 2nd Israel Symposium on Theory and Computing Systems*, pages 167–176, 1993.

[38] G. Varghese. *Self-stabilization by local checking and correction*. PhD thesis, MIT, Oct. 1992.

[39] G. Varghese. Self-stabilization by counter flushing. In *The 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 244–253, 1994.

[40] L. Zhu and S. Kulkarni. Synthesizing round based fault-tolerant programs using genetic programming. In T. Higashino, Y. Katayama, T. Masuzawa, M. Potop-Butucaru, and M. Yamashita, editors, *SSS*, volume 8255 of *Lecture Notes in Computer Science*, pages 370–372. Springer, 2013.