This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2536023, IEEE Transactions on Parallel and Distributed Systems

Shadow/Puppet Synthesis: A Stepwise Method for the Design of Self-Stabilization

Alex Klinkhamer and Ali Ebnenasir

Abstract—This paper presents a novel two-step method for automated design of self-stabilization. The first step enables the specification of legitimate states and an intuitive (but imprecise) specification of the desired functional behaviors in the set of legitimate states (hence the term "shadow"). After creating the shadow specifications, we systematically introduce the main variables and the topology of the desired self-stabilizing system. Subsequently, we devise a parallel and complete backtracking search towards finding a self-stabilizing solution that implements a precise version of the shadow behaviors, and guarantees recovery to legitimate states from any state. To the best of our knowledge, the shadow/puppet synthesis is the first sound and complete method that exploits parallelism and randomization along with the expansion of the state space towards generating self-stabilizing systems that cannot be synthesized with existing methods. 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 three new self-stabilizing protocols that we conjecture to require the minimal number of states per process to achieve stabilization (when processes are deterministic): 2-state maximal matching on bidirectional rings, 5-state token passing on unidirectional rings, and 3-state token passing on bidirectional chains.

Index Terms—Self-Stabilization, Distributed computing, Program synthesis

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, starting from its invariant, the executions of an SS system remain in the invariant; i.e., *closure*. Design and verification of convergence are difficult tasks [1], [2], [3] in part due to the requirements of (i) recovery from *any* state; (ii) recovery under distribution constraints, where processes can read only the state of their neighboring processes (a.k.a. their *locality*), and (iii) the non-interference of convergence with closure. One approach to facilitate the development of SS systems is to separate the concerns of closure and convergence for the designer; i.e., separate functional concerns from self-stabilization. Moreover, automating the design steps could potentially decrease development costs. However, there are two important impediments before such a two-step design method. First, for some protocols (e.g., token passing systems like Dijkstra's 4-state chain [1] and Gouda's 3-bit ring [4]) specifying the functional behaviors (in the absence of faults) amounts to specifying their self-stabilizing version! That is, the actions enabled in the invariant (a.k.a. closure actions) also provide convergence from illegitimate states. Second, existing automated methods are either incomplete [5], [6], [7], [8] or complete [9], [10] but require exact specification of legitimate states using formal logics (e.g., temporal logic), which are difficult to use by mainstream engineers. This paper presents a novel method where we address these challenges.

Most existing methods for the design of self-stabilization are either manual [1], [11], [12], [13], [14], [15], [2] or rely on heuristics [5], [6], [7], [8] that may fail to generate a solution for some systems. For example, Awerbuch *et al.* [11] 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 [12] divide the state space into a set of supersets of the legitimate states, called *convergence stairs*, where for each stair closure and convergence to a lower-level stair are guaranteed. Stomp [13] provides a method based on ranking functions for design and verification of self-stabilization. Gouda [2] presents a theory for design and composition of self-stabilizing systems. Demirbas and Arora [16] present a method for stabilization-preserving refinement of abstract designs using wrappers at the specification and implementation levels. Nesterenkol and Tixeuil [17] define the notion of ideal stabilization where all states are considered legitimate, and show how ideal stabilization can facilitate the design and composition of some stabilizing systems. Methods for algorithmic design of convergence [5], [6], [7], [8] are mainly based on sound heuristics that

• *The authors are with the Department of Computer Science, Michigan Technological University, Houghton, MI 49931, U.S.A. E-mail: {apklinkh,aebnenas}@mtu.edu*

This work was sponsored by the NSF grant CCF-1116546.
***Superior**, a high performance computing cluster at Michigan Technological University, was used in obtaining the experimental results presented in this paper.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2536023, IEEE Transactions on Parallel and Distributed Systems

search through the state space of a non-stabilizing system in order to synthesize recovery actions while ensuring non-interference with closure. However, these heuristics may fail to find a solution while there exists one; i.e., they are sound but incomplete. The approach in [9], [10] is sound and complete, but it employs constraint solvers as a blackbox. By contrast, the shadow/puppet synthesis relies on a backtracking algorithm that is customized for the synthesis of self-stabilization and can be micromanaged towards providing better efficiency (evident by our experimental results). Moreover, to increase the likelihood of finding a solution, we exploit parallelism and randomization in our backtracking algorithm. Bounded synthesis [18] allows for a systematic state space expansion, but it does not consider all states as potential initial states.

This paper proposes a two-step method for automated design of self-stabilization where we enable designers to deal with the concerns of closure and convergence separately, and provide a method for state space expansion. The underlying philosophy behind our work is that *the constraints that the synthesis algorithm must follow (e.g., read restrictions due to distribution) should not be imposed on designers*. Specifically, the proposed approach includes two layers (see Figure 1): (1) intuitive specification of an invariant and functional behaviors in the absence of faults (i.e., closure behaviors/actions), called the *shadow* specification, and (2) automated synthesis of a self-stabilizing system that implements a precise characterization of the shadow behavior in the invariant and enables convergence to the specified invariant in terms of some superposed variables, called the *puppet* system. This *shadow/puppet synthesis* is inspired by the way a puppeteer wishes to tell a story via shadows of puppets on a screen. A puppet itself can be more intricate than the shadow it casts. To make our desired shadow (functional behavior), we therefore rely on a clever puppeteer (the computer) to construct a puppet (synthesized protocol).

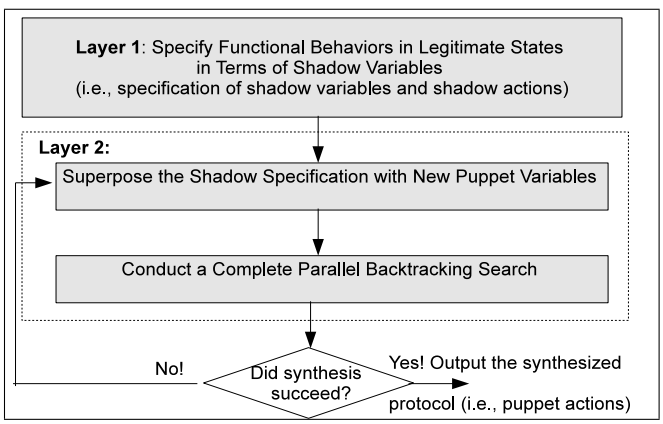

Fig. 1: The proposed shadow/puppet synthesis method.

The algorithmic design of the puppet actions is based on a backtracking search. The backtracking search is conducted in a parallel fashion amongst a fixed number of threads that simultaneously search for an SS solution in the space of all acceptable actions. When a thread finds a combination of design choices (i.e., actions) that would result in the failure of the search (a.k.a. *a conflict*), it shares this information with other threads, thereby improving resource utilization.

Contributions. The contributions of this work are multi-fold. First, we devise a two-step design method that separates the concerns of closure and convergence for the designer, and enables designers to intuitively specify functional behaviors and systematically include computational redundancy. Second, we propose a *parallel* and *complete* backtracking search that finds an SS solution if one exists. If a solution does not exist in the current state space of the program, then designers can include additional puppet variables or alternatively increase the domain size of the existing puppet variables and rerun the backtracking search. Third, we present three different implementations of the proposed method as a software toolset, called Protocon (http://asd.cs.mtu.edu/projects/protocon/), where we provide a sequential implementation and two parallel implementations; one multi-threaded and the other an MPI-based implementation. We also demonstrate the power of the proposed method by synthesizing several new network protocols that all existing heuristics fail to synthesize. These case studies include 2-state maximal matching on bidirectional rings, 5-state token passing on unidirectional rings, and 3-state token passing on a bidirectional chains. In [19], [20], we perform detailed serial and parallel benchmarks for simpler systems such as coloring on Kautz graphs [21] which can represent a P2P network topology, the 3-bit token ring of Gouda and Haddix [4], ring orientation, and leader election on a ring.

Organization. Section 2 provides a motivating example for shadow/puppet synthesis. Section 3 introduces the basic concepts of protocols, transient faults, closure and convergence. Section 4 formally states the problem of designing self-stabilization. Section 5 presents the backtracking algorithm. Section 6 provides an overview of our case studies. Section 7 discusses related work. Finally, Section 8 makes concluding remarks and presents future/ongoing work.

2 Motivating Example

In order to illustrate the proposed approach, we discuss the design of the well-known token passing protocol in a unidirectional ring (proposed by Dijkstra [1]) using shadow/puppet synthesis. Token passing in a unidirectional ring includes the following requirements: (1) in the absence of faults, there is exactly one token in the ring (i.e., legitimate states or invariant) and the token is circulated along the ring, and (2) if there are multiple tokens in the ring (due to the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2536023, IEEE Transactions on Parallel and Distributed Systems

occurrence of faults), then the protocol will eventually reach a configuration where exactly one token exists thereafter. Dijkstra [1] formulated this protocol as follows: each process has a variable x with a finite domain of at least N values, where N is the number of processes in the ring. Each process P_i $(0 \leq i < N)$ can read and write its own variable x_i and can also read x_{i-1}, where addition and subtraction are modulo N. The following actions guarantee both closure and convergence: $(x_{N-1} = x_0 \longrightarrow x_0 := x_0 + 1)$ of P_0 and $(x_{i-1} \neq x_i \longrightarrow x_i := x_{i-1})$ of each other P_i $(1 \leq i < N)$. Each action has a guard condition (placed before $\longrightarrow$) and a statement (placed after $\longrightarrow$) that is executed *atomically* if the guard evaluates to true. Process P_0 *has the token* iff (if and only if) $x_{N-1} = x_0$, and any other P_i $(1 \leq i < N)$ *has the token* iff $x_{i-1} \neq x_i$.

Research Problem. Now, imagine that we did not have Dijkstra's solution and we would like to design a self-stabilizing protocol that meets the aforementioned requirements. We are faced with two options: (1) do what Dijkstra did; that is, in a single step design a protocol that circulates the unique token in legitimate states and ensures convergence to legitimate states if there are multiple tokens, or (2) assume no faults occur and design a system that ensures the existence of a unique token in the absence of faults, and then devise actions that ensure convergence without interfering with closure actions. It is not hard to see that the first option seems more difficult since we have to simultaneously tackle two problems. However, the challenge of the second option is that specifying the behaviors in the absence of faults in terms of protocol variables could be a daunting task for some protocols. For example, these actions capture the closure actions of Dijkstra's token passing protocol: $(x_{N-1} = x_0 \longrightarrow x_0 := x_0 + 1)$ of P_0 and $(x_{i-1} = x_i + 1 \longrightarrow x_i := x_{i-1})$ of each other P_i $(1 \leq i < N)$. Observe that the design of these actions is not straightforward. For some protocols (e.g., Gouda's constant space 3-bit token passing [4]), it is even impossible to separate the closure actions from convergence actions (since closure actions provide convergence). Next, we illustrate the proposed approach (Figure 1) in the context of Dijkstra's token passing protocol.

Layer 1 - Step 1: Specify legitimate states. Based on the first layer of the proposed approach in Figure 1, designers should first specify functional behaviors. This layer includes two steps: the specification of (i) legitimate states, and (ii) the actions that may be executed in legitimate states. First, we consider a *shadow* Boolean variable tok_i for each process P_i indicating whether P_i has the token. Thus, the set of legitimate states is simply specified as: $I = (\exists! i : i \in \mathbb{Z}_N : (tok_i = 1))$ capturing the configurations where there is a unique token in the ring. The quantifier $\exists!$ means there exists a unique value of i.

Layer 1 - Step 2: Specify shadow actions. To express that each P_i should pass the token to P_{i+1}, designers simply write the intuitive action $(tok_i = 1 \longrightarrow tok_i := 0;\ tok_{i+1} := 1;)$. Notice that P_i explicitly changes the state of P_{i+1}, thereby violating the read/write restrictions of the locality of P_i specified on x variables in Dijkstra's protocol. The fundamental idea behind shadow/puppet synthesis is that *such restrictions must not tie the hands of designers in expressing what they want*.

Layer 2 - Step 1: Superpose puppet variables. After specifying the shadow variables and actions, designers should augment the shadow specification with the puppet variables (which in the case of token ring includes the x variables) along with the corresponding read/write restrictions for processes. The domain of puppet variables and their read/write restrictions can respectively be obtained from the system requirement and topology.

Layer 2 - Step 2: Automated design using a parallel backtracking search algorithm. This step is fully automated where designers benefit from the proposed parallel backtracking algorithm (called the *synthesis algorithm*) to find a solution that is self-stabilizing to the specified legitimate states. We have developed three implementations of this algorithm, namely a centralized version, a multi-threaded version and an MPI-based distributed version. The synthesis algorithm is not allowed to read shadow variables since the shadow variables/actions provide only an outline of what is required to guide the synthesis algorithm. As such, the guards of the actions of the synthesized protocol will only be specified in terms of puppet variables.

Layer 2 - Step 3: Existence of a solution. If the backtracking algorithm fails to find a solution, then designers should expand the state space by either increasing the domain of puppet variables or introducing additional puppet variables. This can be achieved by jumping to Step 1 of Layer 2 and repeating the steps of Layer 2.

3 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 [1], [14], [22], [2]. For ease of presentation, we use a simplified version of Dijkstra's token ring protocol [1] 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* [23]) as defined in Section 2. 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*. A *local*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2536023, IEEE Transactions on Parallel and Distributed Systems

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, $s \in X$) 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 $grd \longrightarrow stmt$ 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). For simplicity, consider a 3-process (i.e., $N = 3$) version of Dijkstra's Token Ring (TR) protocol represented in Section 2 (adapted from [1]). Each process P_i $(0 \leq i \leq 2)$ includes an integer variable x_i with a domain $\{0, 1, 2\}$. Each P_i can read and write x_i and can read x_{i-1} (where $x_{0-1} = x_2$ when $i = 0$). The actions of processes are as specified in Section 2. The state predicate I_{TR} captures the set of states in which only one token exists, where I_{TR} is:

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

Minimal Actions. Notice that the guard of an action $A : grd \longrightarrow stmt$ of a process P_i can be specified in terms of a proper subset of V_i. In such cases, the action A is the union of a set of $k > 1$ minimal actions $grd_1 \longrightarrow stmt_1, \cdots, grd_k \longrightarrow stmt_k$, where $grd = (grd_1 \vee \cdots \vee grd_k)$, and each grd_j $(1 \leq j \leq k)$ is specified in terms of the values of all variables in V_i (where $i \in \mathbb{Z}_N$). More precisely, a *minimal* action of a process P_i includes a single valuation of all readable variables for P_i in its guard and a single valuation of all writable variables for P_i in its assignment statement. For example, consider an action $(x_2 = x_0 \longrightarrow x_0 := x_0 + 1)$ in the TR protocol. This action is the union of the minimal actions $(x_2 = 0 \wedge x_0 = 0 \longrightarrow x_0 := 1)$, $(x_2 = 1 \wedge x_0 = 1 \longrightarrow x_0 := 2)$, and $(x_2 = 2 \wedge x_0 = 2 \longrightarrow x_0 := 0)$. The proposed backtracking algorithm in Section 5 explores the space of all minimal actions.

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 $(grd \longrightarrow stmt)$ 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 $grd \longrightarrow stmt$ 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.

Closure and Invariant. A state predicate X is *closed in an action* $grd \longrightarrow stmt$ 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* [2] 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.

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. Some researchers [24], [22] have a stronger definition for the notion of invariant, where in addition to closure a program must meet its specifications from its invariant. Since in the problem of synthesizing self-stabilization (Problem 4.1) we have a constraint of preserving specifications in invariant, we have a more relaxed definition of invariant where only closure is required. We also note that a program may have multiple invariants, however, ideally we would like to use the weakest possible invariant.

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.

4 Problem Statement

In this section, we formally state the problem of creating a self-stabilizing puppet protocol from a specified shadow protocol. Let p be a non-stabilizing protocol and I be an invariant of p. We *manually* expand the state space of p by including new variables. Such *puppet variables* provide computational redundancy in the hopes of giving the protocol sufficient information to detect and correct illegitimate states without forming livelocks. Our experience shows that better performance can be achieved if variables with small domains are included initially. If the synthesis fails, then designers can incrementally increase variable domains or include additional puppet variables. This way designers can manage the growth of the state space and keep the synthesis time/space costs under control.

Now, let p' denote the self-stabilizing version of p that we would like to design and I' represent an invariant. $S_{p'}$ denotes the state space of p'; i.e.,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2536023, IEEE Transactions on Parallel and Distributed Systems

the state space of p expanded by adding puppet variables. Such an expansion can be reversed by a function $\mathcal{H} : S_{p'} \rightarrow S_p$ that maps every state in $S_{p'}$ to a state in S_p by removing the puppet variables. The expansion itself is expressed as 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. Thus, each computation of p' that starts in I' must be mapped to a unique computation of p starting in I. We state the problem as follows[1]: (The function $\mathsf{Pre}(\delta)$ takes a set of transitions δ and returns the source states of δ.)

Problem 4.1: **Synthesizing Self-Stabilization**.

- **Input**: A protocol p and an invariant I, the function $\mathcal{H}$ and the mapping $\mathcal{E}$ capturing the impact of puppet variables.
- **Output**: A protocol p' and an invariant I' in $S_{p'}$.
- **Constraints**:
  1) Preserve the invariant: $I = \mathcal{H}(I')$
  2) Preserve transitions in the invariant: $\big(\delta_p | I = \mathcal{H}(\delta_{p'} | I') \setminus \{(s, s) \mid s \in I\}\big)$ $\wedge \forall (s'_0, s'_1) \in \delta_{p'} : (s'_0 \in I' \implies s'_1 \in I')$
  3) Preserve deadlock freedom in the invariant: $\forall s \in \mathsf{Pre}(\delta_p | I) : (\mathcal{E}(s) \cap I') \subseteq \mathsf{Pre}(\delta_{p'} | I')$
  4) Preserve progress in the invariant: $\forall s \in \mathsf{Pre}(\delta_p | I) : (\delta_{p'} | I') | \mathcal{E}(s)$ 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 each transition of the shadow protocol $\delta_p | I$ is represented by some transitions in $\delta_{p'} | I'$, and all other transitions in $\delta_{p'}$ originating from I' remain in I' and do not change shadow values. The third and fourth constraints require that non-silent states (i.e., where a process is enabled) of p in I should remain deadlock-free and livelock-free in p' in order to ensure progress to a state with different shadow values. Finally, p' must be livelock-free and deadlock-free in $\neg I'$.

Example 4.2: 4-State Token Ring

Keeping with the idea of superposition from [20] for the sake of example, we can specify a token ring using a non-stabilizing 2-state token ring. Each process P_i owns a binary variable t_i and can read t_{i-1}. The first process P_0 is distinguished as *Bot*, in that it acts differently from the others. *Bot* has a token when $t_{N-1} = t_0$ and each other process $P_{i>0}$ is said to have a token when $t_{i-1} \neq t_i$. *Bot* has action $(t_{N-1} = t_0 \longrightarrow t_0 := 1 - t_0;)$, and each other process $P_{i>0}$ has action $(t_{i-1} \neq t_i \longrightarrow t_i := t_{i-1};)$. Let I denote the legitimate states where exactly one process has a token:

$$I = \exists!\ i \in \mathbb{Z}_N : ((i{=}0 \wedge t_{i-1}{=}t_i) \vee (i{\neq}0 \wedge t_{i-1}{\neq}t_i))$$

To transform this protocol to a self-stabilizing version thereof, we add a binary puppet variable x_i to each process P_i. Each process P_i can also read its predecessor's variable x_{i-1}. Let $I' = \mathcal{E}(I)$ be the invariant of this transformed protocol. Let the new protocol p' have the following actions for *Bot* and the other processes $P_{i>0}$:

$$\begin{aligned}
&\textit{Bot} : (x_{N-1} = x_0) \wedge (t_{N-1} \neq t_0) \longrightarrow x_0{:=}1 - x_0;\\
&\textit{Bot} : (x_{N-1} = x_0) \wedge (t_{N-1} = t_0) \longrightarrow x_0{:=}1 - x_0;\ t_0{:=}x_{N-1};\\
&P_i :\ (x_{i-1} \neq x_i) \wedge\ (t_{i-1} = t_i) \longrightarrow x_i{:=}1 - x_i;\\
&P_i :\ (x_{i-1} \neq x_i) \wedge\ (t_{i-1} \neq t_i) \longrightarrow x_i{:=}1 - x_i;\ t_i{:=}x_{i-1};
\end{aligned}$$

This protocol is stabilizing for all rings of size $N \in \{2, \ldots, 7\}$ but contains a livelock when $N = 8$. In [20], we found that given this topology and shadow protocol, no self-stabilizing protocol exists for all $N \in \{2, \ldots, 8\}$. Gouda and Haddix [4] give a similar token ring protocol that stabilizes for all ring sizes. They introduce another binary variable $ready_i$ to each process.

Let us check that the superposition preserves the 2-state token ring protocol for a ring of size $N = 3$. Figure 2 shows the transition systems of the non-stabilizing 2-state protocol p within I and stabilizing 4-state protocol p', where each state is a node and each arc is a transition. Legitimate states are boxed and reoccurring transitions within these states are black. Recovery transitions are drawn with dashed gray lines. Solid gray lines denote transitions within our maximal choice of I' but would serve as recovery transitions for smaller choices of I'. Verifying the conditions from Problem 4.1, we find: (1) $I = \mathcal{H}(I')$ is true since $I' = \mathcal{E}(I)$, (2) the shadow protocol is preserved since there exists a transition of p' that changes rows iff a similar transition exists in the 2-state protocol p, (3) no deadlocks are introduced in the invariant since all states in the boxed rows have outgoing transitions, (4) progress is preserved in the invariant since 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 no livelocks or deadlocks within $\neg I'$.

Shadow/Puppet vs Superposition. Compare the shadow/puppet method (Section 2) with the superposition method (Example 4.2) when used to specify a token ring's invariant and behavior. In the shadow

1. This problem statement is an adaptation of the problem of adding fault tolerance in [22].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2536023, IEEE Transactions on Parallel and Distributed Systems

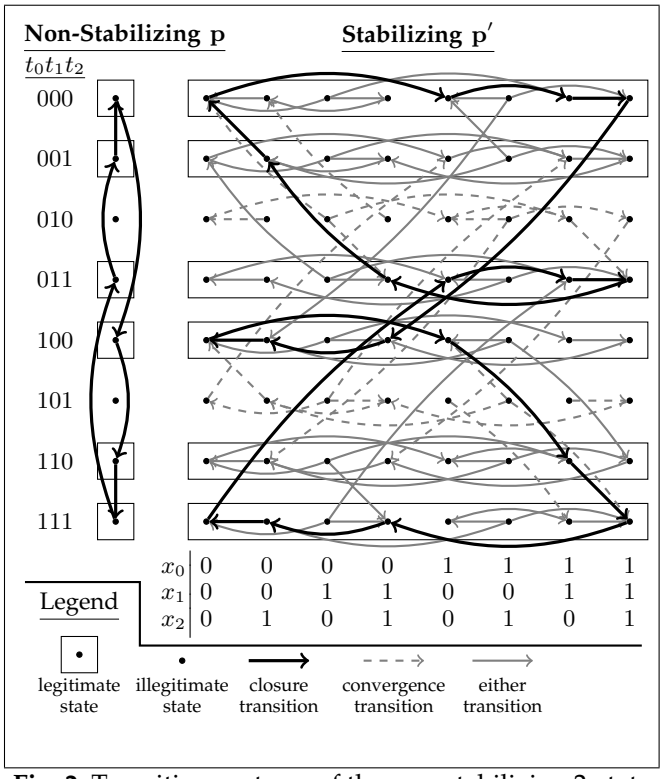

Fig. 2: Transition systems of the non-stabilizing 2-state and stabilizing 4-state token rings of size $N = 3$.

specification of Section 2, each tok_i variable denotes whether a process has the token, and a token is passed by simply moving a 1 value forward to tok_{i+1}. In the non-stabilizing protocol of Example 4.2, each t_i variable holds token information in a clever way resulting from a comparison with t_{i-1}, and a token is passed by flipping the t_i bit. We see that in the case of superposition, a designer must have some foresight about how the t_i variables can be manipulated in a stabilizing protocol. By contrast, the shadow/puppet method gives a designer freedom to define token passing with the tok_i shadow variables without constraining the specific actions of the synthesized protocol p'. It is therefore not surprising that shadow/puppet synthesis can find protocols that cannot be reasonably expressed with superposition (Section 6).

In Protocon, the two methods differ only by whether the variables used to specify the invariant are marked with the *shadow* keyword or not. In the case that shadow variables exist, our synthesis method treats them as write-only when constructing p'. When the shadow protocol is silent (i.e., has no actions in its invariant), processes should know the final values of their writable shadow variables. Self-loops are a trivial consequence of this and must be ignored for actions that only assign shadow variables (e.g., Section 6.1). When the shadow protocol is non-silent, processes should know when they are performing shadow actions, but they should also be able to act without affecting the shadow state (i.e. shadow self-loops). Thus, the minimal actions we use must allow shadow variables to not be assigned (e.g., Section 6.3).

Deterministic, Self-Disabling Processes. Theorems 4.3 and 4.4 below show that the assumptions of deterministic and self-disabling processes do not impact the completeness of any algorithm that solves Problem 4.1. 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 4.3: 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 $\delta_p|\neg I$. □

Theorem 4.4: 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 4.1; hence the proof of right to left. Let p_{ss} be an SS version of p to I with nondeterministic but self-disabling processes. Moreover, let P_j be a self-disabling process of p_{ss} with two nondeterministic 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2536023, IEEE Transactions on Parallel and Distributed Systems

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. □

5 Synthesis Using Backtracking

In [3], we have shown that Problem 4.1 is an NP-complete problem in the size of the expanded state space. In this section, we present an efficient and complete backtracking search algorithm to solve Problem 4.1. Backtracking search is a well-studied technique [25] that is easy to implement and can give very good results. Throughout this section, we use "actions" and "minimal actions" interchangeably (unless otherwise stated). Section 5.1 provides a high-level description of the algorithm, and Section 5.2 presents the details of the algorithm. Section 5.3 presents an intelligent method for the inclusion of new actions in candidate solutions. Finally, Section 5.4 discusses some issues related to the optimization of the proposed algorithm.

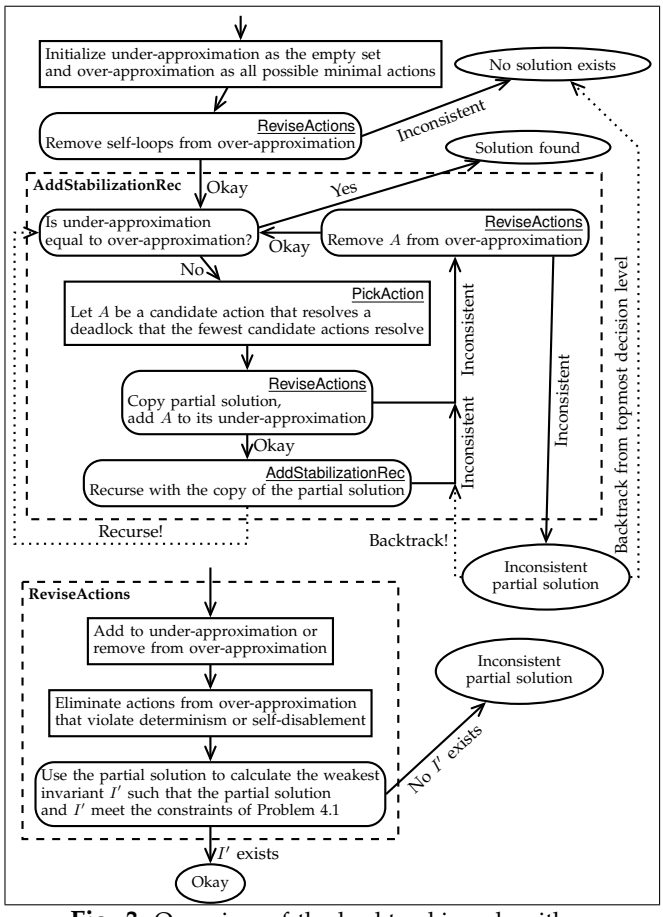

Fig. 3: Overview of the backtracking algorithm.

5.1 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 pieces: (1) an *under-approximation* formed by making well-defined decisions about the form of a solution, and (2) an *over-approximation* that 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 to problem variables. The partial assignment is inconsistent in two cases: (i) the constraints upon assigned variables are broken (i.e., the under-approximation causes a conflict), and/or (ii) the constraints cannot be satisfied by the remaining variable assignments (i.e., the over-approximation cannot contain a solution). Each time a choice is made to build upon the under-approximation, the current partial solution is saved at decision level j and a copy that incorporates the new choice is placed at level $j+1$. If the guess at level $j+1$ is inconsistent, we move back to level j and discard the choice that 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.

In the context of our work, we apply a backtracking search in the space of all valid minimal actions that can be included in a solution. Specifically, we use a set of actions, called `delegates`, that plays the role of the under-approximation, and another set of actions, called `candidates`, that contains the remaining actions to potentially include in `delegates`. Thus, the set (`delegates`∪`candidates`) constitutes the over-approximation.

Figure 3 illustrates an abstract flowchart of the proposed backtracking algorithm. We start with the non-stabilizing protocol p, its invariant I (which is closed in p), the topology of p', and the mappings to ($\mathcal{E}$) and from ($\mathcal{H}$) its expanded state space. The algorithm in Figure 3 starts by computing all *valid* candidate actions (in the expanded state space) that adhere to the read/write permissions of all processes. The initial value of `delegates` is often the empty set unless there are specific actions that must be in the solution (e.g., to ensure the reachability of particular states). The algorithm in Figure 3 then calls ReviseActions to remove self-loops from `candidates` (since they violate convergence), and checks for inconsistencies in the partial solution. The designer may give additional constraints that forbid certain actions.

In general, ReviseActions (see the bottom dashed box in Figure 3) is invoked whenever we strengthen the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2536023, IEEE Transactions on Parallel and Distributed Systems

partial solution by adding to the under-approximation or removing from the over-approximation. It may further remove from the over-approximation by enforcing the determinism and self-disablement constraints (see Theorem 4.3). Then ReviseActions computes the largest possible invariant I' that could be used by the current partial solution. That is, it finds the weakest predicate I' for which the constraints of Problem 4.1 can be satisfied using some set of transitions $\delta_{p'}$ permissible by the partial solution. The partial solution requires $\delta_{p'}$ to include all transitions corresponding to actions in `delegates`. Additionally, $\delta_{p'}$ can include any subset of transitions corresponding to actions in `candidates`. For example, Constraint 5 of Problem 4.1 stipulates that the transitions of `delegates` are cycle-free outside of I' and that the transitions of `delegates` $\cup$ `candidates` provide weak convergence to I'. If such an I' does not exist, then the partial solution is inconsistent.

If our initialized `delegates` and `candidates` give a consistent partial solution, then we invoke the AddStabilizationRec routine. The objective of AddStabilizationRec (see the top dashed box in Figure 3) is to go through all actions in `candidates` and check their eligibility for inclusion in the self-stabilizing solution. In particular, AddStabilizationRec has a loop that iterates through all actions of `candidates` until it becomes empty or an inconsistency is found. In each iteration, AddStabilizationRec picks a candidate action to resolve some remaining deadlock at the next decision level. In general, the candidate action can be randomly selected. However, to limit the possible choices, we use an intelligent method for picking candidate actions described in Section 5.3. After picking a new action A, we invoke ReviseActions to add action A to a copy of the current partial solution by including A in the copy of `delegates` and removing it from the copy of `candidates`. If the copied partial solution is consistent, then AddStabilizationRec makes a recursive call to itself, using the copied partial solution for the next decision level. If the copied partial solution is found to be inconsistent (either by a call to ReviseActions or by the exhaustive search in the call to AddStabilizationRec), then we remove action A from `candidates` using ReviseActions. If after removal of A the partial solution is consistent, then we continue in the loop. Otherwise, we backtrack since no stabilizing protocol exists with the current under-approximation.

5.2 Details of the Algorithm

This section presents the details of the proposed backtracking method. Notice that we assume that the input to this algorithm is a non-stabilizing shadow protocol already superposed with some new finite-domain puppet variables. 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).

Algorithm 1 Entry point of the backtracking algorithm for solving Problem 4.1.

AddStabilization(p: protocol, I: state predicate,
$\mathcal{E}$: mapping $S_p \rightarrow S_{p'}$,
`&delegates`: protocol actions,
`forbidden`: forbidden actions)

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

1: **let** `candidates` be the set of all possible actions that have transitions in $\mathcal{E}\big((\delta_p|I) \cup \{(s,s) \mid s \in S_p\}\big)$.
2: **let** `adds` := `delegates` {Forced actions, if any}
3: `delegates` := $\emptyset$
4: **let** `dels` := `candidates` $\cap$ `forbidden`
5: **let** $I' := \emptyset$
6: **if not** ReviseActions(p, I, $\mathcal{E}$, `&delegates`, `&candidates`, `&`I', `adds`, `dels`) **then**
7: **return false**
8: **end if**
9: **return** AddStabilizationRec(p, I, $\mathcal{E}$, `&delegates`, `candidates`, I')

AddStabilization. Algorithm 1 is the entry point of our backtracking algorithm. The AddStabilization 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 minimal actions. Next, the function determines which actions are explicitly required (Line 2) or disallowed by additional constraints (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.

AddStabilizationRec. Algorithm 2 defines the main recursive search. Like AddStabilization, it returns **true** iff a self-stabilizing protocol is found that 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 the over-approximation `delegatesUcandidates` of the protocol are identical. If ReviseActions does not find anything wrong, then `delegates` is self-stabilizing, 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2536023, IEEE Transactions on Parallel and Distributed Systems

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 $\neq \emptyset$ **do**
2: **let** $A :=$ PickAction(p, $\mathcal{E}$, delegates, candidates, I')
3: **let** next_delegates := delegates
4: **let** next_candidates := candidates
5: **let** $I'' := \emptyset$
6: **if** ReviseActions(p, I, $\mathcal{E}$, &next_delegates, &next_candidates, &I'', $\{A\}$, $\emptyset$) **then**
7: **if** AddStabilizationRec(p, I, $\mathcal{E}$, &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, $\mathcal{E}$, &delegates, &candidates, &I', $\emptyset$, $\{A\}$) **then**
13: **return false**
14: **end if**
15: **end while**
16: **return true**

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 creates a non-stabilizing protocol, then return in failure; otherwise, continue the loop.

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 forbidden actions from the over-approximated protocol by removing the dels set from candidates; (3) enforces self-disablement (Theorem 4.3) and determinism (Theorem 4.4) 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 to satisfy Constraints 2–4 of Problem 4.1 given the current under/over-approximations, and (5) verifies Constraints 1 and 5 of Problem 4.1 by ensuring the invariant I' captures all of I, the under-approximation is livelock-free in $\neg I'$, and the over-approximation weakly converges to I'. If the check fails, then ReviseActions returns **false**. Finally, ReviseActions invokes the CheckForward function to infer actions that must be added to the under-approximation or removed from the over-approximation, and will return **false** only if it infers that the current partial solution cannot be used to form a self-stabilizing protocol. A trivial version of CheckForward can just return **true**.

A good ReviseActions implementation should pro-

Algorithm 3 Add adds to the under-approximation and remove dels from the 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 action $B \in$ candidates 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)

{Find candidate actions that 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
- All transitions of delegates beginning in I' are included in $(\delta_{p'}|I')$
- Constraints 2, 3, and 4 of Problem 4.1 hold

8: Check Constraints 1 and 5 of Problem 4.1:
- $I = \mathcal{H}(I')$
- 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**

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2536023, IEEE Transactions on Parallel and Distributed Systems

vide 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. Thus, we ensure that actions in `delegates` do not form a livelock and that actions in `delegates` $\cup$ `candidates` provide weak stabilization.

A good CheckForward implementation should at least remove candidate actions that are not needed to resolve deadlocks. This can be performed quickly and allows the AddStabilizationRec function to immediately return a solution when all deadlocks are resolved.

Theorem 5.1 (Completeness): The AddStabilization algorithm is complete.

Proof: We show that if AddStabilization returns **false**, then no solution exists. Since each candidate action is minimal and we consider all such actions as candidates, a subset of the candidate actions would form a self-stabilizing protocol iff such a protocol exists. Observe that AddStabilizationRec follows the standard backtracking [26] procedure where we (1) add a candidate action to the under-approximation in a new decision level, and (2) backtrack and remove that action from the candidates if an inconsistency (which cannot be fixed by adding to the under-approximation) is discovered by ReviseActions at that new decision level. Even though ReviseActions removes candidate actions in order to enforce deterministic and self-disabling processes, we know by Theorems 4.4 and 4.3 that this will not affect the existence of a self-stabilizing protocol. Thus, since we follow the general template of backtracking [26], the search will test every consistent subset of the initial set of candidate actions where processes are deterministic and self-disabling. Therefore, if our search fails, then no solution exists. □

Theorem 5.2 (Soundness): The AddStabilization algorithm is sound.

Proof: We 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 or AddStabilizationRec function that called it simply returns **true** with the same `delegates` set. The only other case where AddStabilization returns **true** is when `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 verifying the Constraints of Problem 4.1 on Line 8. Therefore, when AddStabilization returns **true** the actions of `delegates` form a self-stabilizing protocol. □

5.3 Picking Actions via the Minimum Remaining Values Method

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 tackle this complexity by reducing the branching factor. To do this, we use a minimum remaining values (MRV) method in PickAction. MRV is classically applied to constraint satisfaction problems [25] by assigning a value to a variable that has the minimal remaining candidate values. In our setting, we pick an action that resolves a deadlock with the minimal number of remaining actions available to resolve it.

Algorithm 4 Pick an action using the minimum remaining values (MRV) method.

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 $\neg I' \cup \mathcal{E}(\mathsf{Pre}(\delta_p))$ that 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`$[i]$ $\cap$ 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`$[i]$ \ `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` that resolves a deadlock in `deadlock_sets[i]`.
19: **end if**
20: **end for**
21: **return** An action from `candidates`. {Edge case}

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 shift deadlocks to the next highest element in the array (bubbling up) for each candidate action that 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 that can resolve some deadlock in that set. Line

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2536023, IEEE Transactions on Parallel and Distributed Systems

21 can only be reached if either the remaining deadlocks cannot be resolved (but ReviseActions catches this earlier) or all deadlocks are resolved (but CheckForward can catch this earlier).

5.4 Optimizing the Decision Tree

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

Conflicts. Every time a new candidate action is included in `delegates`, ReviseActions checks for inconsistencies, which involves cycle detection and reachability analysis. These procedures become very costly as the complexity of the transition system grows. To mitigate this problem, whenever an inconsistency is found, we record a minimal set of decisions (subset of `delegates`) that causes it. We reference these *conflict sets* [25] in CheckForward to remove candidate actions that would cause an inconsistency.

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 sufficiently to change the decision. To limit the search time in these branches, we employ a method outlined by Gomes et al. [27] that 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 randomly selects a candidate action permissible by the MRV method. This approach remains complete since a conflict is recorded for any set of decisions that causes a restart.

Parallel Search. In order to increase the chance of finding a solution, we instantiate several parallel executions of the algorithm in Figure 3; i.e., *search diversification*. As noted in [27], parallel tasks will generally avoid overlapping computations due to the randomization used in PickAction. We have observed linear speedup when the search tasks are expected to restart several times before finding a solution [19]. The parallel tasks share conflicts with each other to prevent the re-exploration of branches that contain no solutions. In our MPI implementation, conflict dissemination occurs between tasks using a virtual network topology formed by a generalized Kautz graph [21] 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 slow the dissemination of new conflicts.

Protocol	Procs	Shadow Self-Loops	MPI Procs	Compute Time
1-Bit Maximal Matching	2–7	Forbidden	1	0.54 secs
4-State Token Ring*	2–8	Forbidden	4	1.50 hrs
5-State Token Ring	2–9	Forbidden	4	23.75 mins
3-State Token Chain*	2–5	Forbidden	4	5.29 secs
3-State Token Chain	2–5	Allowed	4	48.93 secs
4-State Token Chain [1]	2–4	Forbidden	4	10.11 secs
4-State Token Chain [1]	2–4	Allowed	4	1.14 mins
3-State Token Ring [1]	2–5	Forbidden	4	1.02 mins
3-State Token Ring [1]	2–5	Allowed	4	5.22 mins

* No protocol is found to stabilize for all system sizes under consideration.

Fig. 4: Synthesis runtimes for case studies.

6 Case Studies

In this section, we look for exact lower bounds for constant-space maximal matching and token passing protocols using shadow/puppet synthesis. Section 6.1 presents an intuitive way to synthesize maximal matching with shadow variables. Section 6.2 explores the unidirectional token ring with a distinguished process. Section 6.3 explores bidirectional token passing protocols. Each section gives a new self-stabilizing protocol that we conjecture uses the minimal number of states per process to achieve stabilization.

Figure 4 provides the synthesis runtimes of all case studies. The *Procs* column indicates the system sizes (numbers of processes) that are simultaneously considered during synthesis. For example, the maximal matching case shows 2–7, which means that we are resolving deadlocks without introducing livelocks for 5 different systems of various sizes. These ranges are chosen to increase the chance of synthesizing a generalizable protocol. The *Shadow Self-Loops* column indicates whether actions within the I' invariant may leave shadow variables unchanged. This is achieved by modifying Line 7 of Algorithm 3 to enforce that $(\delta'_p|I') \subseteq \mathcal{E}(\delta_p)$; Forbidding these actions makes every action of a token passing protocol actually pass a token. The *MPI Procs* column indicates the number of MPI processes used for synthesis.

6.1 2-State Maximal Matching on a Ring

A matching for a graph is a set of edges that do not share any common vertices. A matching is *maximal* iff adding any new edge would violate the matching property. In a ring, a set of edges is a matching as long as at least 1 of every 2 consecutive edges is excluded from the set. For the set to be a maximal matching, at least 1 out of every 3 consecutive edges must be included. To see that the matching is maximal, consider selecting another edge to create a new matching. The edge itself cannot be in the current matching nor can either of the two adjacent edges, but we have already enforced that one of those three is selected, therefore the new edge cannot be added!

To specify this problem, use a binary shadow variable e_i to denote whether the edge between adjacent

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2536023, IEEE Transactions on Parallel and Distributed Systems

processes P_{i-1} and P_i is in the matching ($e_i = 1$ means to include the edge, otherwise exclude the edge). The invariant is therefore the states where at least 1 of every 3 consecutive e values equals 1, but at least 1 of every 2 consecutive e values equals 0.

$$I = \forall i \in \mathbb{Z}_N : (e_{i-1}{=}1 \vee e_i{=}1 \vee e_{i+1}{=}1) \wedge (e_i{=}0 \vee e_{i+1}{=}0)$$

We would like processes to determine whether their neighboring links are included in a matching, therefore we give each P_i write access to e_i and e_{i+1}. Each e_i is a shadow variable since it exists only for specification, therefore processes have write-only access. Since a matching should not change, there are no shadow actions. For the system's implementation, we give each process P_i a binary puppet variable x_i to read and write along with read access to the x_{i-1} and x_{i+1} variables of its neighbors in the ring. We only are guessing that an x_i domain size of 2 is large enough to achieve stabilization and represent (e_i, e_{i+1}) values with (x_{i-1}, x_i, x_{i+1}) values in some way. Synthesis gives the following protocol, which we believe to be generalizable after verification of rings up to size $N = 100$. Notice that the e values are fully determined based on the x values.

$$\begin{array}{lll}
P_i : x_{i-1}{=}1 \wedge x_i{=}1 \wedge x_{i+1}{=}1 \longrightarrow & e_i{:=}1; & x_i{:=}0;\ e_{i+1}{:=}0; \\
P_i : x_{i-1}{=}0 \wedge x_i{=}1 \wedge x_{i+1}{=}1 \longrightarrow & e_i{:=}0; & x_i{:=}0;\ e_{i+1}{:=}0; \\
P_i : x_{i-1}{=}0 \wedge x_i{=}0 \wedge x_{i+1}{=}0 \longrightarrow & e_i{:=}0; & x_i{:=}1;\ e_{i+1}{:=}1; \\
P_i : x_{i-1}{=}1 \wedge x_i{=}0 \longrightarrow & e_i{:=}1; & e_{i+1}{:=}0; \\
P_i : x_{i-1}{=}0 \wedge x_i{=}0 \wedge x_{i+1}{=}1 \longrightarrow & e_i{:=}0; & e_{i+1}{:=}0; \\
P_i : x_{i-1}{=}0 \wedge x_i{=}1 \wedge x_{i+1}{=}0 \longrightarrow & e_i{:=}0; & e_{i+1}{:=}1;
\end{array}$$

An implementation of this protocol consists only of puppet variables, and therefore discards the e variables. Of the 6 actions above, only the first 3 modify x values. From these 3, we discard the e variables and combine the first 2 actions (for simplification). This leaves us with the puppet protocol that would be used for implementation, where each P_i has the following actions:

$$\begin{array}{ll}
P_i : & x_i{=}1 \wedge x_{i+1}{=}1 \longrightarrow x_i{:=}0; \\
P_i : x_{i-1}{=}0 \wedge x_i{=}0 \wedge x_{i+1}{=}0 & \longrightarrow x_i{:=}1;
\end{array}$$

Further, we can derive the meaning of the puppet variables by observing how the value of (e_i, e_{i+1}) is assigned for each particular value of (x_{i-1}, x_i, x_{i+1}). We could do this by observing all 6 actions, but we only need to observe the first 3 since they subsume the last 3. The last 3 actions respectively assign $(1, 0)$, $(0, 0)$, and $(0, 1)$ to (e_i, e_{i+1}) to denote that P_i is matched with (i) P_{i-1}, (ii) itself, and (iii) P_{i+1}. We use (x_{i-1}, x_i, x_{i+1}) values to represent these cases:

$$\begin{array}{ll}
x_{i-1}{=}1 \wedge x_i{=}0 & (P_i \text{ matched with } P_{i-1}) \\
x_{i-1}{=}0 \wedge x_i{=}0 \wedge x_{i+1}{=}1 & (P_i \text{ not matched}) \\
x_{i-1}{=}0 \wedge x_i{=}1 \wedge x_{i+1}{=}0 & (P_i \text{ matched with } P_{i+1})
\end{array}$$

6.2 5-State Unidirectional Token Ring

In the token ring shadow specification in Section 2, each process P_i is given a binary shadow variable tok_i that denotes whether the process has a token. The invariant is all states where exactly one token exists ($\exists! i : tok_i = 1$), and each process should eventually pass the token within the invariant ($tok_i = 1 \longrightarrow tok_i := 0;\ tok_{i+1} := 1;$). For synthesis, we give each P_i a puppet variable x_i and also let it read x_{i-1}. Like in Dijkstra's token ring, we distinguish P_0 as *Bot* to allow its actions to differ from the other processes. We also force each action in the invariant to pass a token by forbidding shadow self-loops. With this restriction, we found that no protocol using 4 states per process is stabilizing for all rings of size $N \in \{2, \ldots, 8\}$.

Using 5 states per process (i.e., each x_i has domain $\mathbb{Z}_5$), the synthesized protocol is not always generalizable, even when we synthesize for all rings of size $N \in \{2, \ldots, 9\}$. However, we can increase our chances of finding a generalizable version by allowing the search to record many solutions and verify correctness for larger ring sizes. After sufficient synthesize-and-verify, we are left with the following protocol, which we think is generalizable after verification of rings up to size $N = 30$.

$$\begin{array}{lll}
Bot : x_{N-1} = 0 \wedge x_0 = 0 \longrightarrow & x_0{:=}1; & tok_i{:=}0;\ tok_{i+1}{:=}1; \\
Bot : x_{N-1} = 1 \wedge x_0 \leq 1 \longrightarrow & x_0{:=}2; & tok_i{:=}0;\ tok_{i+1}{:=}1; \\
Bot : x_{N-1} > 1 \wedge x_0 > 1 \longrightarrow & x_0{:=}0; & tok_i{:=}0;\ tok_{i+1}{:=}1; \\
P_i : x_{i-1} = 0 \ \wedge x_i > 1 \longrightarrow & x_i{:=}\lfloor x_i/4 \rfloor; & tok_i{:=}0;\ tok_{i+1}{:=}1; \\
P_i : x_{i-1} = 1 \ \wedge x_i \neq 1 \longrightarrow & x_i{:=}1; & tok_i{:=}0;\ tok_{i+1}{:=}1; \\
P_i : x_{i-1} = 2 \ \wedge x_i \leq 1 \longrightarrow & x_i{:=}2 + x_i; & tok_i{:=}0;\ tok_{i+1}{:=}1; \\
P_i : x_{i-1} \geq 3 \ \wedge x_i \leq 1 \longrightarrow & x_i{:=}4; & tok_i{:=}0;\ tok_{i+1}{:=}1;
\end{array}$$

Our previous work [20] used the approach of Example 4.2 to synthesize a 6-state token ring and a 3-bit token ring similar to that of Gouda and Haddix [4]. In doing so, we observe an interesting efficiency trade-off between process memory, convergence speed, and adherence to the shadow protocol. For example, a process may act several times in the 3-bit token ring of Gouda and Haddix [4] without passing a token, but this protocol converges in fewer steps (faster) than our 5-state token ring. Dijkstra's N-state token ring requires much more process memory, yet it converges in fewer steps than the 3-bit protocol while also ensuring that the token will be passed with every action.

6.3 3-State Token Chain

Rather than around a ring, we can also pass a token back-and-forth along a linear (chain) topology. Formally, the token passing behavior can be described by guarded commands. As with the ring, let there be N processes, where each process P_i has a binary shadow variable tok_i that denotes whether it has a token. Each process P_i can read and write tok_{i-1} when $i > 0$, and each P_i can read and write tok_{i+1} when $i < N-1$. Additionally, all processes can read a binary shadow variable *fwd* that denotes the direction the token is moving (1 means up, 0 means down). That is, a process $P_{0<i<N-1}$ passes the token to P_{i+1} when $fwd = 1$ and passes it to P_{i-1} when $fwd = 0$. The two

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2536023, IEEE Transactions on Parallel and Distributed Systems

end processes P_0 and P_{N-1} must behave differently than the others, therefore they are distinguished as *Bot* and *Top* respectively. Both *Bot* and *Top* can write *fwd*, and do assign it when they pass the token in order to change the direction of passing. The full shadow protocol is given by the following actions.

$$\begin{aligned}
&\textit{Bot} : \textit{tok}_0{=}1 \longrightarrow \quad \textit{fwd}{:=}1;\ \textit{tok}_0{:=}0; \quad \textit{tok}_1{:=}1;\\
&P_i : \textit{tok}_i{=}1 \wedge \textit{fwd}{=}1 \longrightarrow \quad \textit{tok}_i{:=}0; \quad \textit{tok}_{i+1}{:=}1;\\
&P_i : \textit{tok}_i{=}1 \wedge \textit{fwd}{=}0 \longrightarrow \quad \textit{tok}_{i-1}{:=}1;\ \textit{tok}_i{:=}0;\\
&\textit{Top} : \textit{tok}_{N-1}{=}1 \longrightarrow \textit{fwd}{:=}0;\ \textit{tok}_{N-2}{:=}1;\ \textit{tok}_{N-1}{:=}0;
\end{aligned}$$

For synthesis, give each process a ternary puppet variable x_i. Each $P_{0<i<N-1}$ can also read x_{i-1} and x_{i+1}. Likewise, *Bot* can read x_1 and *Top* can read x_{N-2}. One of the synthesized protocols is as follows, which we conjecture to be generalizable after verification of chains up to size $N = 30$.

$$\begin{aligned}
&\textit{Bot} : x_i{\neq}1 \wedge x_{i+1}{=}2 \longrightarrow x_0{:=}1; \quad \textit{fwd}{:=}1;\ \textit{tok}_i{:=}0; \quad \textit{tok}_{i+1}{:=}1;\\
&\textit{Bot} : x_i{\neq}0 \wedge x_{i+1}{\neq}2 \longrightarrow x_i{:=}0;\\
&P_i : x_{i-1}{=}1 \wedge x_i{\neq}1 \quad \longrightarrow x_i{:=}1;\ \textit{tok}_i{:=}0; \quad \textit{tok}_{i+1}{:=}1;\\
&P_i : x_{i-1}{=}0 \wedge x_i{=}1 \wedge x_{i+1}{=}1 \longrightarrow x_i{:=}0;\\
&P_i : x_{i-1}{=}0 \wedge x_i{=}0 \wedge x_{i+1}{=}2 \longrightarrow x_i{:=}2;\ \textit{tok}_{i-1}{:=}1;\ \textit{tok}_i{:=}0;\\
&\textit{Top} : x_{i-1}{=}1 \wedge x_i{\neq}1 \longrightarrow x_i{:=}1;\\
&\textit{Top} : x_{i-1}{\neq}1 \wedge x_i{\neq}2 \longrightarrow x_{N-1}{:=}2;\ \textit{fwd}{:=}0;\ \textit{tok}_{N-2}{:=}1;\ \textit{tok}_{N-1}{:=}0;
\end{aligned}$$

This protocol does not always pass the token within the invariant, however we found that no such protocol exists. As shown by Dijkstra [1], we can obtain a stabilizing protocol with this behavior by either using 4 states per process or allowing *Bot* and *Top* to communicate. Using the same shadow specification and the puppet topologies from [1], we synthesized 4-state token chains and 3-state token rings. Both cases appear to give generalizable protocols (verified up to $N = 15$), regardless of whether we force each action in the invariant to pass a token.

7 Related Work and Discussion

This section discusses related work on manual and automated 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 [28] provide a method for augmenting fault-intolerant systems with a set of new actions that implement fault tolerance functionalities. Katz and Perry [29] present a general (but expensive) method for global snapshot and reset towards adding convergence to non-stabilizing systems. Varghese [30] and Afek *et al.* [31] provide a method based on local checking for global recovery of locally correctable protocols. Varghese [15] also proposes a counter flushing method for detection and correction of global predicates. Nesterenkol and Tixeuil [17] employ a mapping to define all system states as legitimate state of an abstract specification. This effectively removes the need for convergence, but it is not always possible or may require human ingenuity in the specification. Chandy and Misra introduce *variable superposition* [32], where the functional concerns are specified on a set of variables, and extra implementation details are carried out using a set of superposed variables. Our work further decouples these two sets but uses their superposition as an implicit mapping.

Since it is unlikely that an efficient method exists for algorithmic design of self-stabilization [3], most existing techniques [5], [6], [7], [33], [8] are based on sound heuristics. For instance, Abujarad and Kulkarni [5], [6] present a heuristic for adding convergence to locally-correctable systems. Zhu and Kulkarni [33] 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 [7] provide a lightweight method for designing self-stabilization even for non-locally correctable protocols. They also devise [8] 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. Methods for automated design of fault tolerance [22], [34] have the option to make deadlock states unreachable. This is not an option in the addition of self-stabilization; recovery should be provided from any state in protocol state space.

8 Conclusions and Future Work

We presented a two-step method for automated design of self-stabilizing systems, called *shadow/puppet* synthesis. In the first step, designers provide a *shadow* specification consisting of a set of legitimate states, system topology and the expected system behaviors in the absence of faults. In the second step, we algorithmically generate a self-stabilizing system that accurately implements the behaviors in legitimate states and provides convergence to legitimate states in terms of some superposed variables, called the *puppet* system. Then, we use a parallel backtracking search to intelligently look for a self-stabilizing solution. We have implemented our approach and have automatically generated self-stabilizing protocols that none of the existing heuristics can generate (to the best of our knowledge). These protocols include token passing protocols on the topologies given by Dijkstra in [1], 2-state maximal matching, 5-state token ring, 3-state token chain, coloring on Kautz graphs, ring orientation, and leader election on a ring [19].

We are currently investigating several extensions of this work. First, we use 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. Another extension is to leverage the techniques used in SMT solvers and apply them in our backtracking search.

ACKNOWLEDGMENT

The authors would like to thank the anonymous referees for their valuable comments and suggestions.

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] M. Gouda, "The theory of weak stabilization," in *5th International Workshop on Self-Stabilizing Systems*, ser. Lecture Notes in Computer Science, vol. 2194, 2001, pp. 114–123.

[3] A. Klinkhamer and A. Ebnenasir, "On the hardness of adding nonmasking fault tolerance," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 3, pp. 338–350, May 2015.

[4] M. G. Gouda and F. F. Haddix, "The stabilizing token ring in three bits," *Journal of Parallel and Distributed Computing*, vol. 35, no. 1, pp. 43–48, May 1996.

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

[6] F. Abujarad and S. S. Kulkarni, "Automated constraint-based addition of nonmasking and stabilizing fault-tolerance," *Theoretical Computer Science*, vol. 412, no. 33, pp. 4228–4246, 2011.

[7] A. Farahat and A. Ebnenasir, "A lightweight method for automated design of convergence in network protocols," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 7, no. 4, pp. 38:1–38:36, Dec. 2012.

[8] A. Ebnenasir and A. Farahat, "Swarm synthesis of convergence for symmetric protocols," in *Proceedings of the Ninth European Dependable Computing Conference*, 2012, pp. 13–24.

[9] F. Faghih and B. Bonakdarpour, "SMT-based synthesis of distributed self-stabilizing systems," in *16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Springer, 2014.

[10] F. Faghih and B. Bonakdarpour, "SMT-based synthesis of distributed self-stabilizing systems," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2015, to appear.

[11] 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*, 1991, pp. 268–277.

[12] M. G. Gouda and N. J. Multari, "Stabilizing communication protocols," *IEEE Transactions on Computers*, vol. 40, no. 4, pp. 448–458, 1991.

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

[14] A. Arora and M. G. Gouda, "Closure and convergence: A foundation of fault-tolerant computing," *IEEE Transactions on Software Engineering*, vol. 19, no. 11, pp. 1015–1027, 1993.

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

[16] M. Demirbas and A. Arora, "Specification-based design of self-stabilization," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 27, no. 1, pp. 263–270, Jan 2016.

[17] M. Nesterenko and S. Tixeuil, "Ideal Stabilization," *International Journal of Grid and Utility Computing*, vol. 4, no. 4, pp. 219–230, Oct. 2013.

[18] B. Finkbeiner and S. Schewe, "Bounded synthesis," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5-6, pp. 519–539, 2013.

[19] A. Klinkhamer and A. Ebnenasir, "Synthesizing self-stabilization through superposition and backtracking," Michigan Technological University, Tech. Rep. CS-TR-14-01, May 2014. [Online]. Available: http://mtu.edu/cs/research/papers/pdfs/CS-TR-14-01.pdf

[20] A. Klinkhamer and A. Ebnenasir, "Synthesizing self-stabilization through superposition and backtracking," in *16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Springer, 2014, pp. 252–267.

[21] M. Imase and M. Itoh, "A design for directed graphs with minimum diameter," *IEEE Trans. Computers*, vol. 32, no. 8, pp. 782–784, 1983.

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

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

[24] A. Arora and S. S. Kulkarni, "Detectors and correctors: A theory of fault-tolerance components," *International Conference on Distributed Computing Systems*, pp. 436–443, May 1998.

[25] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, 2009.

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

[27] C. P. Gomes, B. Selman, H. Kautz *et al.*, "Boosting combinatorial search through randomization," *AAAI/IAAI*, vol. 98, pp. 431–437, 1998.

[28] Z. Liu and M. Joseph, "Transformation of programs for fault-tolerance," *Formal Aspects of Computing*, vol. 4, no. 5, pp. 442–469, 1992.

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

[30] G. Varghese, "Self-stabilization by local checking and correction," Ph.D. dissertation, MIT, 1993.

[31] Y. Afek, S. Kutten, and M. Yung, "The local detection paradigm and its application to self-stabilization," *Theoretical Computer Science*, vol. 186, no. 1-2, pp. 199–229, 1997.

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

[33] L. Zhu and S. Kulkarni, "Synthesizing round based fault-tolerant programs using genetic programming," in *Stabilization, Safety, and Security of Distributed Systems*. Springer, 2013, pp. 370–372.

[34] A. Ebnenasir, "Automatic synthesis of fault-tolerance," Ph.D. dissertation, Michigan State University, May 2005.

Alex Klinkhamer Alex Klinkhamer is a PhD student in the Department of Computer Science at Michigan Technological University. Alex received his bachelor's and master's degrees in 2010 and 2013, both from Michigan Tech. His research interests include self-stabilization, distributed systems and parallel algorithms.

Ali Ebnenasir Ali Ebnenasir is an Associate Professor of Computer Science at Michigan Technological University and a senior member of the ACM. Ali received his bachelor's and master's degrees in 1994 and 1998 respectively from the University of Isfahan and Iran University of Science and Technology. In 2005, he received his PhD from the Computer Science and Engineering Department at Michigan State University (MSU). After finishing his postdoctoral fellowship at MSU in 2006, he joined the Department of Computer Science at Michigan Tech. His research interests include Software Dependability, Formal Methods and Parallel and Distributed Computing.