Computer Science Technical Report

Swarm Synthesis of Convergence for Symmetric Protocols

Ali Ebnenasir and Aly Farahat

Michigan Technological University
Computer Science Technical Report
CS-TR-11-02
May 2011

MichiganTech

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

Swarm Synthesis of Convergence for Symmetric Protocols

Ali Ebnenasir and Aly Farahat

May 2011

Abstract

Due to their increasing complexity, today's distributed systems are subject to a variety of transient faults (e.g., loss of coordination, soft errors, bad initialization), thereby making self-stabilization a highly important property of such systems. However, designing Self-Stabilizing (SS) network protocols is a complex task in part because a SS protocol should recover to a set of legitimate states from *any* state in its state space; i.e., *convergence*. Once converged, a SS protocol should remain in its set of legitimate states as long as no faults occur; i.e., *closure*. The verification of SS protocols is even harder as developers have to prove the interference-freedom of closure and convergence. To facilitate the design and verification of SS protocols, previous work proposes techniques that take a non-stabilizing protocol and automatically *add convergence* while guaranteeing interference-freedom. Nonetheless, such algorithmic methods must search an exponential space of candidate sequences of transitions that could be included in a SS protocol. This paper presents a novel method for exploiting the computational resources of computer clusters and search diversification towards increasing the chance of success in automated design of finite-state self-stabilizing symmetric protocols. We have implemented our approach in a software tool that enables an embarrassingly parallel platform for the addition of convergence. Our tool has automatically synthesized several SS protocols that cannot be designed by extant automated techniques.

1 Introduction

Self-Stabilizing (SS) network protocols have increasingly become important as today's complex systems are subject to different kinds of transient faults (e.g., soft errors, loss of coordination, bad initialization). Nonetheless, design and verification of SS protocols are difficult tasks [1–3] mainly for the following reasons. First, a SS protocol must *converge* (i.e., recover) to a set of legitimate states from *any* state in its state space (when perturbed by transient faults). Once converged, a SS protocol should remain in its set of legitimate states as long as no faults occur; i.e., *closure*. Second, since a protocol includes a set of processes communicating via network channels, the convergence should be achieved with the coordination of several processes while each process is aware of only its locality; *locality* of a process includes a set of processes whose state is readable by that process. Third, proving the interference-freedom of closure and convergence is a hard problem. An automated method for augmenting an existing non-stabilizing protocol with convergence (i.e., *adding convergence*) facilitates the generation of SS protocols that are correct by construction, thereby eliminating the need for their proof of correctness. However, while the problem of adding convergence is known to be in NP [4], we are not aware of any algorithm that adds convergence in polynomial time (in protocol state space).

Existing automated techniques either present algorithms that add convergence to specific families of non-stabilizing protocols [2, 5–7] or focus on sound heuristics that add convergence (possibly to a wider range of protocols) at the expense of completeness [8]. That is, if the heuristics succeed, then the generated protocol is SS; otherwise, the heuristics declare failure while a SS version of the non-stabilizing protocol might exist. For instance, Awerbuch-Varghese [2, 5] present compilers that generate SS versions of non-interactive protocols where correctness criteria are specified as a relation between the input and the output of the protocol (e.g., given a graph, compute its spanning tree). Furthermore, the input to their compilers is a synchronous and deterministic non-stabilizing protocol. Abujarad and Kulkarni [6, 7] present an algorithmic method for the addition of convergence to *locally-correctable* protocols where processes can correct the global state of the protocol by correcting their local states to a legitimate state without corrupting the state of their neighbors. By contrast, our previous work [8] provides a family of sound heuristics that add convergence to non-stabilizing protocols that could be non-locally correctable, interactive and non-deterministic. Each heuristic is a deterministic strategy for searching an exponential problem space for a SS solution. For example, in a unidirectional token passing network protocol, there is a unique token in the network that gets circulated by each process copying it from its predecessor (if the predecessor has the token). Since a process is aware of only its own state and the state of its predecessor, copying a token may seem like a legitimate local action. However, in a perturbed global state where multiple tokens exist, a subset of processes may create a non-terminating sequence of token passing actions amongst themselves (called a *non-progress* cycle), thereby preventing global recovery to legitimate states where there is at most one token. We currently have several heuristics to deal with such cases. For instance, a heuristic randomly prohibits one of the participating actions in a cycle while another one prohibits all of them and then activates them one by one to check their impact on other recovery actions taken by other processes. Our experience [8] shows that a heuristic that succeeds in adding convergence to a protocol may fail to do so for another protocol. Thus, one has to use a diverse set of heuristics. Moreover, applying the steps of a heuristic in different orders sometimes generates different results. Thus, the success and failure of adding convergence depends on two major factors: search strategy and randomization.

This report presents a novel method that adds convergence to *finite-state* protocols by exploiting search diversification and parallelism. The proposed method includes three phases. Starting with a non-stabilizing protocol p and its set of legitimate states I (see Figure 1), the proposed method first partitions $\neg I$ to disjoint sets of states based on the least number of recovery actions that can recover the state of the protocol from any state $s \in \neg I$ to I, called the *rank* of s. Thus, Rank j includes all states whose rank is equal to $j > 0$ (see Figure 1). States in I have rank 0, and there is no recovery path from states with rank infinity. If all states in $\neg I$ get a finite rank, then including the corresponding recovery paths in p would generate a protocol p_{ws} that is weakly stabilizing; i.e., from every state in $\neg I$ there exists an execution path that reaches a state in I [9]. In other words, p_{ws} provides an approximation for a SS version of p. If p_{ws} does not exist, then p does not have a SS version. In Phase 2, the proposed method *orders* each possible recovery action based on the smallest rank from which that action can be executed, called the *rank of the recovery action*. For example,

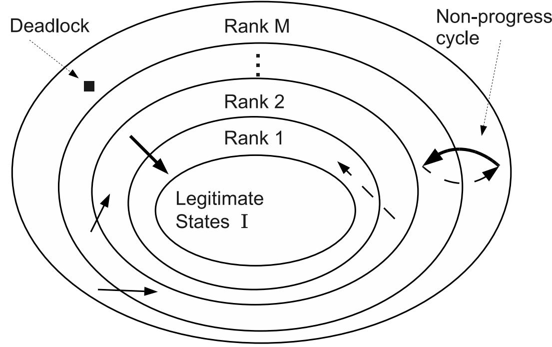


Figure 1: Ranking and approximating stabilization.

the rank of the **bold** transitions in Figure 1 is 2. The output of Phase 2 is an array *Groups* whose size is equal to the number of ranks/partitions of $\neg I$. Each array element *Groups*$[i]$ is an *ordered* list of candidate recovery actions whose rank is i.

In the third phase, for each i from 1 to the total number of ranks/partitions of $\neg I$, the proposed method traverses the ordered list *Groups*$[i]$ and includes an action A in the stabilizing protocol if and only if A resolves some deadlock states in $\neg I$ without forming cycles with previously included recovery actions. A deadlock state has no outgoing transitions (see Figure 1). If cycles are formed, then A is excluded from the stabilizing protocol and the subsequent actions in the list *Groups*$[i]$ are similarly considered. If at the end of the third phase there are still some deadlock states, then we permute the candidate recovery actions in each *Groups*$[i]$ and re-do the third phase. This will increase the likelihood of finding a SS version of p by exploiting randomization. Moreover, we create several parallel instances of the proposed method. Each instance permutes the actions in each rank until either a solution is found or an upper bound (specified for the time/space of synthesis) is reached.

We have designed and implemented the proposed approach in a software tool, called the parallel STabilization Synthesizer (pSTSyn). Using pSTSyn, we have generated the SS versions of several symmetric SS protocols including maximal matching, graph coloring, agreement and leader election on a ring. pSTSyn has generated new SS protocols that we could not synthesize with extant heuristics (see Section 5).

Organization. Section 2 provides preliminary concepts. Section 3 formulates the problem of adding convergence. Section 4 present a new method that enables the addition of convergence in an embarrassingly parallel fashion, called the *swarm synthesis* of convergence. Section 5 demonstrates some case studies synthesized by pSTSyn and provides the experimental results on time/space efficiency of swarm synthesis. Section 6 makes concluding remarks and discusses future work.

2 Preliminaries

In this section, we present the formal definitions of finite-state symmetric protocols, our distribution model (adapted from [4]), convergence and self-stabilization. Protocols are defined in terms of their set of variables, their transitions and their processes. The definitions of convergence and self-stabilization is adapted from [9–12]. To simplify our presentation, we use a protocol for 3-coloring on a ring (adapted from [13]) as a running example.

Symmetric protocols as (non-deterministic) finite-state machines. A *symmetric protocol* p includes a finite set of $K > 1$ similar processes $\{P_0, \cdots, P_{K-1}\}$ such that for every pair of processes P_i and P_j, where $0 \leq i, j \leq K-1$, the code of P_j can be obtained from P_i by a simple renaming (re-indexing) of variables, and vice versa [14]. As such, the protocol p can be identified by a representative process P_r that captures the functionalities of the symmetric processes. The representative process P_r is a tuple $\langle V_r, R_r, W_r, \delta_r \rangle$, where V_r is a finite set $\{v_0, \cdots, v_{N-1}\}$ of N variables. The set of variables of p, denoted V_p, is the union of V_r, for

$0 \leq r \leq K-1$. All variables in V_r (also denoted R_r) are readable for P_r, and W_r is a subset of V_r that P_r is allowed to write. Each variable $v_i \in V_r$, for $0 \leq i \leq N-1$, has a finite non-empty domain D_i. The *local state* of P_r is a unique valuation of the variables in V_r. The *global state* of p is a snapshot of the local states of P_r for $0 \leq r \leq K-1$. For brevity, we use the terms state and global state interchangeably throughout the paper. The *state space* of p, denoted S_p, is the set of all possible states of p, and $|S_p|$ denotes the size of S_p. The *locality/neighborhood* of P_r is determined by R_r, which also defines the underlying communication topology of p. For a variable v and a state s, $v(s)$ denotes the value of v in s. A *transition* t is an ordered pair of states, denoted (s_0, s_1), where s_0 is the source and s_1 is the target/destination state of t. δ_r represents the set of transitions of P_r $(0 \leq r \leq K-1)$. Moreover, δ_p is the union of δ_r for $0 \leq r \leq K-1$. We use p and δ_p interchangeably. A *state predicate* is any subset of S_p specified as a Boolean expression over variables of 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.

We use Dijkstra's guarded commands language [15] as a shorthand for representing the set of transitions of P_r (i.e., δ_r). A guarded command (i.e., *action*) is of the form $L : grd \rightarrow stmt$, where L is a label, grd is a Boolean expression in terms of variables in V_r and $stmt$ is a statement that updates variables of W_r *atomically*. Formally, an action $grd \rightarrow stmt$ includes a set of transitions (s_0, s_1) such that grd holds in s_0 and the atomic execution of $stmt$ results in state s_1. An action $grd \rightarrow stmt$ is *enabled* in a state s iff grd holds at s. The process P_r is *enabled* in s iff there exists an action of P_r that is enabled at s.

Example: Three Coloring (TC). The Three Coloring (TC) protocol (adapted from [13]) has K processes located on a ring. The representative process P_r has a variable c_r with a domain $D_r = \{0, 1, 2\}$ representing three distinct colors that can be assigned to c_r. Thus, we have $V_{TC} = \{c_0, \cdots, c_{K-1}\}$ and the state space of TC, denoted S_{TC}, has 3^K states. Process P_r can read and write c_r, but it can only read the colors of its left and right neighbors. That is, $R_r = \{c_{r-1}, c_r, c_{r+1}\}$ and $W_r = \{c_r\}$. The representative process includes the following action (addition and subtraction are performed in modulo K):

$$A_r : \quad (c_r = c_{r-1}) \vee (c_r = c_{r+1}) \ \rightarrow \ c_r := other(c_{r-1}, c_{r+1})$$

If the color of P_r is equal to any of its neighbors, then P_r sets c_r to a color different from both of its neighbors. The function $other(x, y)$ non-deterministically returns either $(x+1)$ *mod* 3 or $(x+2)$ *mod* 3 if $x = y$. Otherwise, $other(x, y)$ returns the third remaining value. $\triangleleft$

Effect of distribution on protocol representation. Every transition of P_r belongs to a *group* of transitions due to the inability of P_r in reading variables that are not in R_r $(0 \leq r \leq K-1)$. Consider two processes P_1 and P_2 each having a Boolean variable that is not readable for the other process. That is, P_1 (respectively, P_2) can read and write x_1 (respectively, x_2), but cannot read x_2 (respectively, x_1). Let $\langle x_1, x_2 \rangle$ denote a state of this protocol. Now, if P_1 writes x_1 in a transition $(\langle 0, 0 \rangle, \langle 1, 0 \rangle)$, then P_1 has to consider the possibility of x_2 being 1 when it updates x_1 from 0 to 1. As such, executing an action in which the value of x_1 is changed from 0 to 1 is captured by the fact that a group of two transitions $(\langle 0, 0 \rangle, \langle 1, 0 \rangle)$ and $(\langle 0, 1 \rangle, \langle 1, 1 \rangle)$ is included in P_1. In general, a transition is included in the set of transitions of a process *if and only if* its associated group of transitions is included. Formally, any two transitions (s_0, s_1) and (s'_0, s'_1) in a group of transitions formed due to the read restrictions of P_r, meet the following constraints: $\forall v : v \in R_r : (v(s_0) = v(s'_0)) \wedge (v(s_1) = v(s'_1))$ and $\forall v : v \notin R_r : (v(s_0) = v(s_1)) \wedge (v(s'_0) = v(s'_1))$.

Due to read restrictions R_r, we represent P_r as a set of transition groups $\{g_{r1}, g_{r2}, \cdots, g_{rl}\}$, where $l \geq 1$. Due to write restrictions W_r, no transition group g_{ri} $(1 \leq i \leq l)$ includes a transition (s_0, s_1) that updates a variable $v \notin W_r$ (It is known that the total number of groups is polynomial in $|S_p|$ [4]). $W(P_r)$ denotes the set of transition groups that adhere to the read/write restrictions of P_r.

TC Example. Consider a transition $t = (s_0, s_1)$ of P_r such that $c_{r-1}(s_0) = 0$, $c_r(s_0) = 0$, $c_{r+1}(s_0) = 1$ and $c_{r-1}(s_1) = 0$, $c_r(s_1) = 2$, $c_{r+1}(s_1) = 1$. That is, the transition t changes the value of c_r from 0 to 2. Transition t is associated with a group of 3^{K-3} transitions because P_r cannot read the values of $K-3$ variables, where each could take three values. $\triangleleft$

Computations and execution semantics. A *computation* of a protocol p is a sequence $\sigma = \ll s_0, s_1, \cdots \gg$ of states that satisfies the following conditions: (1) for each transition (s_i, s_{i+1}) $(i \geq 0)$ in σ, there exists an action $grd \rightarrow stmt$ in some process P_j $(0 \leq j \leq K-1)$ 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. In other words, a computation is generated by a nondeterministic interleaving of actions. A *computation prefix* of a protocol p is a *finite* sequence $\sigma = \ll s_0, s_1, \cdots, s_m \gg$ of states, where $m \geq 0$, such that each transition (s_i, s_{i+1}) in σ $(0 \leq i < m)$ belongs to some action $grd \rightarrow stmt$ in P_r for some $0 \leq r \leq K-1$. The projection of a protocol p on a non-empty state predicate X, denoted as $\delta_p|X$, is a protocol with the set of transitions $\{(s_0, s_1) : (s_0, s_1) \in \delta_p \ \wedge \ s_0, s_1 \in X\}$. In other words, $\delta_p|X$ consists of transitions of p that start in X and end in X.

Closure. 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* [9] requires that every computation that starts in I remains in I.

TC Example. The state predicate I_{color} captures the set of states in which any two neighboring processes have different colors. Formally, I_{color} is equal to $\forall r : 0 \leq r \leq K-1 : (c_r \neq c_{r+1})$, which is an abbreviation of the state predicate $\{s \mid (s \in S_{TC}) \wedge (\forall r : 0 \leq r \leq K-1 : (c_r(s) \neq c_{r+1}(s)))\}$. The predicate I_{color} is closed in the protocol TC since no action is enabled in I_{color}. As such, we call TC a *silent* protocol in I_{color}. $\lhd$

Convergence and self-stabilization. Let I be a state predicate. We say that a protocol p *strongly converges to* I iff from any state, *every* computation of p reaches a state in I. A protocol p *weakly converges* to I iff from any state, *there exists* a computation of p that reaches a state in I. A protocol p is *strongly (respectively, weakly) self-stabilizing* to a state predicate I iff (1) I is closed in p and (2) p strongly (respectively, weakly) converges to I.

Let $s_d \in \neg I$ be a state with no outgoing transitions; i.e., a *deadlock* state. Moreover, let $\sigma = \ll s_i, s_{i+1}, \cdots, s_j, s_i \gg$ be a sequence of states outside I, where $j \geq i$ and each state is reached from its predecessor by the transitions in δ_p. The sequence σ denotes a *non-progress* cycle. Since adding strong convergence involves the resolution of deadlocks and non-progress cycles, we restate the definition of strong convergence as follows:

Proposition 2.1. A protocol p *strongly converges to* I iff there are no deadlock states in $\neg I$ and no non-progress cycles in $\delta_p \mid \neg I$.

TC Example. In any state outside I_{color} there must be two processes that have the same colors. Thus, there is at least one enabled action in any state in $\neg I_{color}$. That is, there are no deadlock states in $\neg I_{color}$. Moreover, since action A_r assigns a color different from both neighbors, correcting the state of P_r does not corrupt the state of its neighbors. As such, no cycles are formed in $\neg I_{color}$. Thus, TC is strongly stabilizing to I_{color}. $\lhd$

3 Problem Statement

Consider a non-stabilizing symmetric protocol p with a representative process $P_r = \langle V_r, R_r, W_r, \delta_r \rangle$ and a state predicate I closed in p. Our objective is to generate a strongly stabilizing version of p, denoted p_{ss}, by *adding* convergence to I while preserving the symmetry. We assume that p is correct as far as its original specification is concerned. Accordingly, we require that, in the absence of transient faults, the behaviors of p_{ss} from any state in I remain the same as p. With this motivation, during the addition of convergence to p, no states (respectively, transitions) are added to or removed from I (respectively, $\delta_p|I$). This way, the behaviors of p_{ss} are exactly the same as p's starting from any state inside I. Moreover, if p_{ss} starts in a state outside I, p_{ss} will provide strong convergence to I.

Problem 3.1: Adding Convergence

- **Input**: (1) a symmetric protocol p with a representative process $P_r = \langle V_r, R_r, W_r, \delta_r \rangle$ and K processes, and (2) a *non-empty* state predicate I such that I is closed in p.
- **Output**: A symmetric protocol p_{ss} with a representative process $P_{r_{ss}} = \langle V_r, R_r, W_r, \delta_{r_{ss}} \rangle$ and K processes such that the following constraints are met: (1) I is unchanged; (2) $(\delta_{r_{ss}} \neq \emptyset) \wedge (\delta_{p_{ss}}|I = \delta_p|I)$, and (3) p_{ss} is strongly self-stabilizing to I. $\square$

Comment. While in this report we focus on cases where the state space of p_{ss} is the same as that of p, for the expansion of state space new variables can be manually introduced in the non-stabilizing protocol to generate an input instance of Problem 3.1. Moreover, we assume that the transition groups of p exclude transitions that form non-progress cycles in $\neg I$. Otherwise, resolving such cycles would violate the constraint

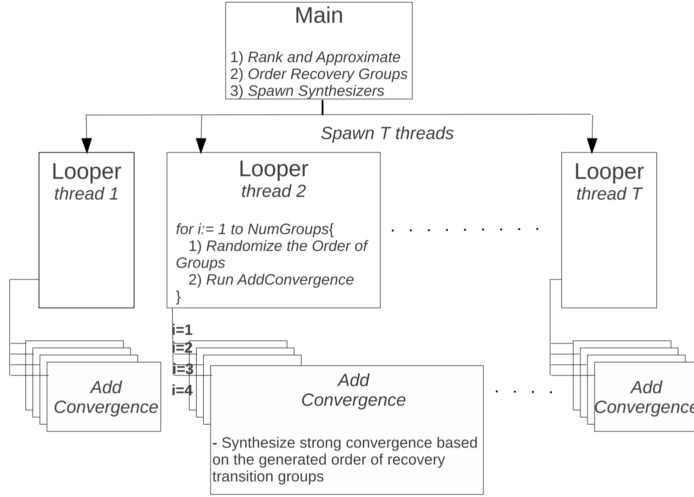


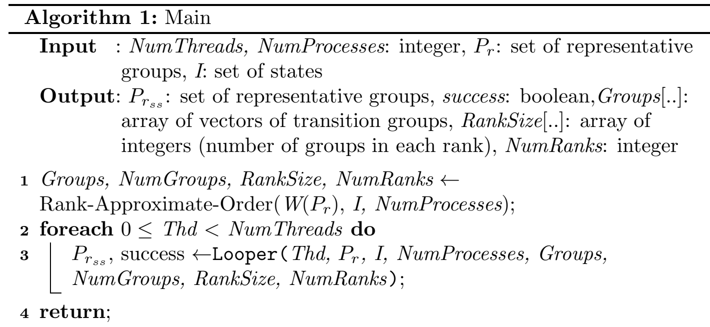
Figure 2: Overview of swarm synthesis of convergence.

$\delta_{p_{ss}}|I = \delta_p|I$. Our ongoing work investigates the case where constraints (1) and (2) on the output can be relaxed towards finding a SS version of p (which is outside the scope of this report).

4 A Method for Swarm Synthesis

In this section, we present a method for adding convergence to symmetric protocol by exploiting search diversification and parallelism. The proposed approach enables the addition of convergence in an embarrassingly parallel fashion, called *swarm synthesis*. To solve Problem 3.1, the proposed method (see Figure 2) includes three phases, namely *Rank and Approximate, Order Recovery Groups* and *Spawn Synthesizers*. These three phases are initiated by the Main component. Figure 2 illustrates an overview of the proposed method.

Phase 1: Rank and Approximate. In the rank-and-approximation phase (see Figures 2 and 3), we compute the rank of every state s in $\neg I$, where $Rank(s)$ is the *length of the shortest computation prefix from s to some state in I* (see Figure 1). The computation prefixes are formed only with the transition groups of WP_r (see Line 1 of Figure 3). Note that $Rank(s) = 0$ iff $s \in I$. Moreover, if $Rank(s) = \infty$, then there is no computation prefix from s that reaches a state in I. If each state $s \in \neg I$ has a finite rank, then including the computation prefixes originated at s would result in a weakly stabilizing version of p. (Please see [8] for a formal proof of correctness.) *NumRanks* denotes the total number of ranks.


Algorithm 1: Main

Input : *NumThreads, NumProcesses*: integer, P_r: set of representative groups, I: set of states

Output: $P_{r_{ss}}$: set of representative groups, *success*: boolean, *Groups*[..]: array of vectors of transition groups, *RankSize*[..]: array of integers (number of groups in each rank), *NumRanks*: integer

1 *Groups, NumGroups, RankSize, NumRanks* $\leftarrow$ Rank-Approximate-Order($W(P_r)$, *I, NumProcesses*);
2 **foreach** $0 \leq$ *Thd* $<$ *NumThreads* **do**
3 $P_{r_{ss}}$, success $\leftarrow$Looper(*Thd*, P_r, *I, NumProcesses, Groups, NumGroups, RankSize, NumRanks*);
4 **return**;

Figure 3: The 'Main' routine.

Phase 2: Order Recovery Groups. This phase of the proposed method takes the ranks generated by

Phase 1 and computes a partial order of all *candidate Recovery Transition Groups* (RTGs), where a candidate RTG is a transition group that excludes any transition starting in I. We say that the *rank of an RTG g* is $i > 0$ iff i is the smallest rank from where g includes a *rank decreasing transition* (s_0, s_1) such that $Rank(s_1) < Rank(s_0)$. Using such a ranking of RTGs, we generate a partial order of RTGs as an array of vectors, denoted *Groups*[], where *Groups*[i] is an ordered list of all RTGs whose rank is i (see Line 1 of Figure 3). The array *RankSize* has *NumRanks* elements, where *RankSize*[i] contains the number of RTGs whose rank is i; i.e., size of *Groups*[i] (see Line 1 of Figure 3).

Phase 3: Spawn Synthesizers. After creating a partial order of RTGs based on the ranks, the Main routine spawns a fixed number of Looper threads (see Lines 2-3 of Figure 3). Each Looper thread randomly reorders the RTGs of each rank (using the ShuffleGroups routine in Figure 4) and invokes the AddConvergence routine in an iterative fashion (see the for-loop in Figure 4). The ShuffleGroups routine generates permutations that depend upon the *ThreadIndex* of that Looper and the iteration i of the for-loop in Figure 4, thereby ensuring that different Loopers explore different permutations. Once a Looper thread succeeds in synthesizing a SS protocol, a termination signal is sent to all the Loopers.

Algorithm 2: Looper

Input : *ThreadIndex, NumProcesses, NumRanks, NumGroups*: integer,
P_r: set of representative groups, I: set of states,
Groups[*NumRanks*]: array of vectors, *RankSize*: array of integers
Output: $P_{r_{ss}}$: set of representative groups, *success*: boolean

```
1 success ← false; P_rss ← P_r;
2 for i ← 0 to NumGroups-1 do
3     Groups_interm ← ShuffleGroups(Groups, RankSize[..],
      NumRanks,ThreadIndex, i);
4     success_interm, P_interm ←AddConvergence(P_r, I, NumProcesses,
      Groups_interm);
5     if (success_interm = true) then
6         P_rss ← P_interm;
7         success ← true;
8         return;
9 return;
```

Figure 4: The Looper routine.

The AddConvergence (see Figure 5) routine takes a representative process P_r, a state predicate I that is closed in the symmetric protocol represented by P_r, the number of processes, and the partial order of RTGs in the array *Groups*[]. The objective of AddConvergence is to check whether or not convergence can be designed by incremental inclusion of RTGs of *Groups*[i] in order, for $1 \leq i \leq NumRanks$ (see the for-loops in Lines 4-5 of Figure 5). Initially, we assign P_r to a representative process $P_{r_{ss}}$ that is updated during the inclusion of RTGs (Line 1 in Figure 5). The Unfold routine instantiates $P_{r_{ss}}$ for all $0 \leq r \leq NumProcess$ to generate the transition system of an intermediate synthesized protocol p_{interm}. Starting from *Groups*[1], an RTG g is included iff g resolves some deadlock states in $\neg I$ and the inclusion of g preserves the cycle-freedom of transitions starting in $\neg I$ (Lines 6-9 in Figure 5). We reuse a symbolic cycle detection algorithm due to Gentilini *et al.* [16] that we have implemented in the DetectCycles routine (see Line 8 in Figure 5). If an RTG creates a cycle, then we skip its inclusion and check the feasibility of including the next RTG in the list *Groups*[1]. Upon the inclusion of an RTG in $P_{r_{ss}}$ (Line 9), we unfold the structure of $P_{r_{ss}}$ to update the intermediate protocol p_{interm}, which will be used to recalculate the deadlock states (Lines 10-11 in Figure 5).

After all RTGs in *Groups*[1] are checked for inclusion, then we respectively perform the same analysis on the RTGs in *Groups*[2], *Groups*[3], $\cdots$, *Groups*[*NumRanks*]. The success of adding convergence depends upon the order based on which RTGs of each *Groups*[i] are included. As such, a different order of traversing the RTGs of *Groups*[i] might provide a different result. To increase the likelihood of generating a solution, the Looper threads generate a new random permutation of the RTGs in *Groups*[] before invoking AddConvergence.

Theorem 4.1 AddConvergence is sound and has a polynomial-time complexity in $|S_p|$.

Proof. If AddConvergence declares success by returning a solution, then its exit point is Line 14 and this is only possible if *Deadlocks*=$\emptyset$. In Line 11, *Deadlocks* is assigned the set of deadlocks of p_{interm}. Lines

Algorithm 3: AddConvergence

Input : P_r: set of transition groups, I: set of states, *NumProcesses*: integer, *Groups*[*NumRanks*]: array of vectors of groups
Output: $P_{r_{ss}}$: set of transition groups, *success*: boolean

1 $P_{r_{ss}} \leftarrow P_r$;
2 $p_{interm} \leftarrow$ `Unfold(`$P_{r_{ss}}$, *NumProcesses*`)` ; `/* generates the whole transition system by instantiating` $P_{r_{ss}}$ `for all processes */`
3 *Deadlocks* $\leftarrow$ $\{s : (s \in \neg I) \wedge (\forall g, s_1, s_2 : g \in p_{interm} \wedge (s_1, s_2) \in g \wedge (s_1 \neq s)\}$;
4 **for** $i \leftarrow 1$ **to** *NumRanks* **do**
5   **for** $j \leftarrow 1$ **to** *RankSize[i]* **do**
6   *Pre* $\leftarrow$ {s: $\exists (s_1, s_2) : (s_1, s_2) \in$ *Groups*[i][j] $\wedge (s_1 = s)\}$;
7   **if** *Pre* $\cap$ *Deadlocks* $\neq \emptyset$ **then**
8   **if** `DetectCycles(`p_{interm},*Groups[i][j]*`)` =*false* **then**
                `/* Detects cycles created due to including the transition group` *Groups*`[i][j] */`
9                 $P_{r_{ss}} \leftarrow P_{r_{ss}} \cup$ *Groups*[i][j];
10                $p_{interm} \leftarrow$ `Unfold(`$P_{r_{ss}}$, *NumProcesses*`)`;
11   *Deadlocks* $\leftarrow \{s : (s \in \neg I) \wedge (\forall g, s_1, s_2 : g \in p_{interm} \wedge (s_1, s_2) \in g \wedge (s_1 \neq s)\}$;
12   **if** *Deadlocks* $= \emptyset$ **then**
13   success $\leftarrow$ *true*;
14   **return**;

15 success $\leftarrow$ *false*;
16 **return**;

Figure 5: The AddConvergence routine.

9-11 are executed *iff* the inclusion of the last candidate transition group does not cause non-progress cycles. Consequently, p_{interm} has no cycles and no deadlocks. Thus, $P_{r_{ss}}$ represents a symmetric protocol that has no deadlocks and is cycle-free in $\neg I$; i.e., $P_{r_{ss}}$ represents a self-stabilizing symmetric protocol (see Proposition 2.1).

The nested for-loop in AddConvergence runs for at most the number of all possible groups in a process. Kulkarni and Arora show that the total number of groups is polynomial in $|S_p|$. Moreover, we use Gentilini *et al.*'s [16] algorithm for symbolic cycle detection. The time-complexity of this algorithm is linear in $|S_p|$ too. □

5 Case Studies

In this section, we present some of the case studies that we have conducted with pSTSyn, which is a software tool that implements the proposed swarm synthesis method. The implementation of pSTSyn is in C++ and we use Binary Decision Diagrams (BDDs) [17] to represent state and transition predicates in memory. We have deployed pSTSyn on a computer cluster with 24 nodes, where each node is an Intel(R) Xeon(R) CPU 5120 @ 1.86GHz (4 cores) with 4GB RAM and the Linux operating system (kernel 2.6.9-42.ELsmp). The Looper threads are created using the MPICH2-1.3.2p1 run-time system. Section 5.1 discusses how swarm synthesis simultaneously generates multiple solutions of a Maximal Matching protocol that would have been impossible to generate with existing approaches. Section 5.2 presents a SS agreement protocol, and Section 5.3 provides an alternative solution for leader election on a ring. The 3-cloring SS protocol presented in Section 2 is different from the manually-designed 3-coloring protocol in [13].

5.1 Maximal Matching

The Maximal Matching (MM) protocol (adapted from [13]) has $K > 3$ processes $\{P_0, \cdots, P_{K-1}\}$ located on a ring, where $P_{(i-1)}$ and $P_{(i+1)}$ are respectively the left and right neighbors of P_i, and addition and subtraction are in modulo K ($1 \leq i < K$). The left neighbor of P_0 is P_{K-1} and the right neighbor of P_{K-1} is P_0. Each process P_i has a variable m_i with a domain of three values {left, right, self} representing whether

or not P_i points to its left neighbor, right neighbor or itself. Process P_i is *matched* with its left neighbor $P_{(i-1)}$ (respectively, right neighbor $P_{(i+1)}$) iff m_i = left and $m_{(i-1)}$ = right (respectively, m_i = right and $m_{(i+1)}$ = left). When P_i is matched with its left (respectively, right) neighbor, we also say that P_i *has a left match* (respectively, *has a right match*). Each process P_i can read the variables of its left and right neighbors. P_i is also allowed to read and write its own variable m_i. The non-stabilizing protocol is empty; i.e., does not include any transitions. Our objective is to automatically generate a strongly stabilizing protocol that converges to a state in $I_{MM} = \forall i : 0 \leq i \leq K-1 : LC_i$, where LC_i is a local state predicate of process P_i as follows

$$LC_i \equiv (m_i = \text{left} \Rightarrow m_{(i-1)} = \text{right}) \wedge (m_i = \text{right} \Rightarrow m_{(i+1)} = \text{left}) \wedge$$
$$(m_i = \text{self} \Rightarrow (m_{(i-1)} = \text{left} \ \wedge m_{(i+1)} = \text{right}\ ))$$

In a state in I_{MM}, each process is in one of these states: (i) matched with its right neighbor, (ii) matched with left neighbor or (iii) points to itself, and its right neighbor points to right and its left neighbor points to left. The MM protocol is silent in that after stabilizing to I_{MM}, the actions of the synthesized MM protocol should no longer be enabled. Figure 6 illustrates a new solution to the MM problem synthesized by pSTSyn. For example, action $B_{self,1}$ means that a process sets m_i to self if it is not pointing to itself, its left neighbor points to left and its right neighbor points to right. Other actions can be interpreted similarly.

P_i: $B_{\text{self},1}$: $(m_{i-1}=\text{left}) \wedge (m_i \neq \text{self}) \wedge (m_{i+1}=\text{right}) \longrightarrow m_i:=\text{self}$

$B_{\text{left},1}$: $(m_{i-1} \neq \text{left}) \wedge (m_i=\text{self}) \wedge (m_{i+1} =\text{self}) \longrightarrow m_i:=\text{left}$
$B_{\text{left},2}$: $(m_{i-1}=\text{right}) \wedge (m_i \neq \text{left}) \wedge (m_{i+1}=\text{right}) \longrightarrow m_i:=\text{left}$
$B_{\text{left},3}$: $(m_{i-1}=\text{right}) \wedge (m_i=\text{right}) \wedge (m_{i+1} \neq \text{right}) \longrightarrow m_i:=\text{left}$

$B_{\text{right},1}$: $(m_i=\text{self}) \wedge (m_{i+1}=\text{left}) \longrightarrow m_i:=\text{right}$
$B_{\text{right},2}$: $(m_{i-1} \neq \text{right}) \wedge (m_i=\text{left}) \wedge (m_{i+1} \neq \text{right}) \longrightarrow m_i:=\text{right}$
$B_{\text{right},3}$: $(m_{i-1}=\text{left}) \wedge (m_i \neq \text{right}) \wedge (m_{i+1} \neq \text{right}) \longrightarrow m_i:=\text{right}$

Figure 6: Synthesized self-stabilizing Maximal Matching

The Appendix includes three more solutions of the MM problem (generated by pSTSyn). The diversity of solutions we have generated demonstrates the effectiveness of exploiting search diversification and parallelism in automating the design of self-stabilization. We have synthesized the protocol in Figure 6 for $5 \leq K \leq 14$. Figures 7 and 8 respectively represent the time and space costs of synthesis, where memory costs are represented in terms of BDD nodes. Notice that pSTSyn synthesizes the SS version of MM in less than 10 minutes. While the ranking time is significant, it is performed only once as a preprocessing phase. Figure 8 demonstrates that as we scale up the number of processes the space cost of cycle detection becomes a bottleneck due to the large size of BDDs. We are currently working on more efficient cycle detection methods.

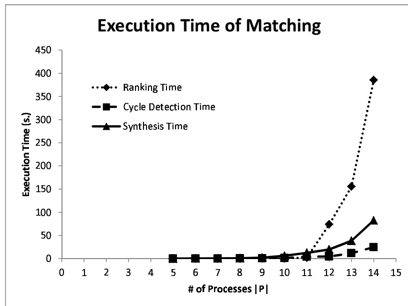


Figure 7: Time spent for adding convergence to matching versus the number of processes

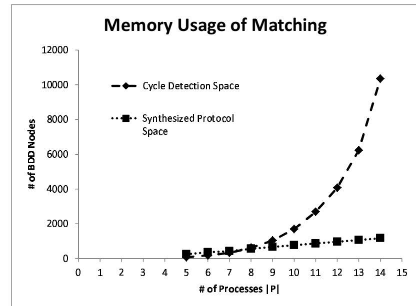


Figure 8: Space usage for adding convergence to matching versus the number of processes

5.2 Agreement

We present a symmetric protocol on a bidirectional ring where the processes need to agree on a specific value: from an initial arbitrary state, all the variables should eventually be equal to one another. The ring

has K processes P_i $(0 \leq i \leq K-1)$. Each process P_i can write its local variable a_i where $a_i \in \{0, \cdots, L-1\}$. Each process P_i can read its left a_{i-1}, right a_{i+1} and its own variable a_i (operations on process and variable indices are modulo K). The set of legitimate states is $I_{agreement} = \bigwedge_{i=1}^{i=K-1}(a_{i-1} = a_i)$. The protocol is not locally correctable: the establishment of $a_{i-1} = a_i$ by an action of P_i can invalidate $a_i = a_{i+1}$. This fact complicates the search for a solution with similar processes. Nonetheless, pSTSyn generates the following protocol with 6 processes from an empty protocol.

$$P_i: \ A_{\mathrm{amt},1}: \quad (a_i > a_{i-1}) \vee (a_i > a_{i+1}) \longrightarrow \quad a_i := \mathsf{min}(a_{i-1}, a_{i+1})$$

Figure 9: Self-Stabilizing agreement protocol.

Figures 10 and 11 respectively illustrate the impact of the domain size of a_i values (denoted L) on time/space efficiency of synthesis. ($|P|$ denotes the number of processes.) Notice that synthesis time grows slowly (note the scale of the y axe in Figure 10), whereas memory costs increase exponentially as we increase the domain of a_i. The reason behind this is that the size of transition groups (respectively, the size of BDDs representing them) increases, thereby raising the number of cycles and the time needed for cycle detection exponentially.

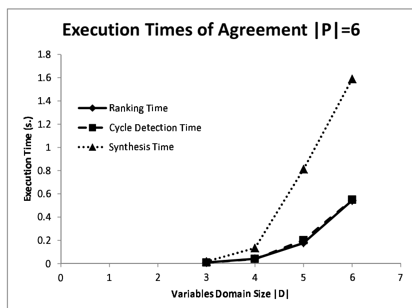


Figure 10: Time spent for adding convergence to agreement versus the size of the variable domain

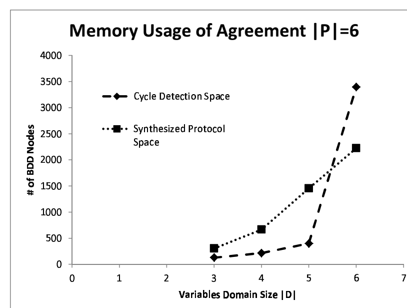


Figure 11: Space usage for adding convergence to agreement versus the size of the variable domain

Keeping the domain size constant (equal to 3), we can scale up the synthesis up to 22 processes (see Figures 12 and 13). We observe that the super linear jump in the synthesis time is due to the thrashing phenomenon when the BDD sizes go beyond a threshold and secondary memory has to be used. ($|D|$ denotes the domain size of a_i.)

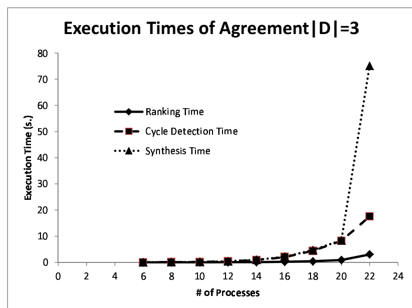


Figure 12: Time spent for adding convergence to agreement versus the number of processes

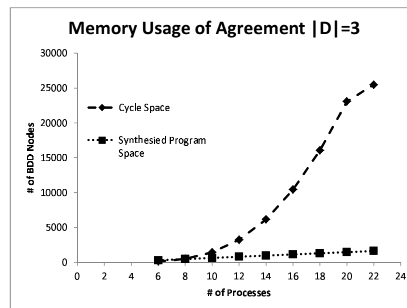


Figure 13: Space usage for adding convergence to agreement versus the number of processes

5.3 Leader Election

We synthesize a Leader Election (LE) protocol adopted from Huang *et. al.* [18]. LE is defined over a bidirectional ring with K processes. Each process P_i has a variable $x_i \in \{0, 1, ..., K-1\}$ $(0 \leq i \leq K-1)$ where x_i is an identifier for P_i. A stable state is such that x_i uniquely identifies P_i. The set of legitimate states for LE is defined as $I_{leader} = \forall i : 0 \leq i \leq K-1 : ((x_i - x_{i-1}) = (x_{i+1} - x_i))$, where K is a prime value and additions/subtractions of variables and indices are modulo K. For a composite K, Huang *et.*

al. demonstrate the impossibility of having a SS protocol. Given I_{leader} and an empty input protocol p, pSTSyn synthesizes the solution in [18] up to 5 processes. pSTSyn also synthesizes an alternative solution demonstrated in Figure 14 in almost 27 seconds; however, synthesis for $K = 7$ failed due to space complexity.

$$
\begin{array}{lll}
P_i: & A_{\text{neq},1}: & (x_{i-1} \neq x_{i+1}) \wedge \neg((x_{i-1}=0 \wedge x_{i+1}=1) \\
& & \quad \vee (x_{i-1}=4 \wedge x_i=0 \wedge x_{i+1}=2) \\
& & \quad \vee (x_{i-1}=2 \wedge x_i=3 \wedge x_{i+1}=0) \\
& & \quad \vee (x_{i-1}=0 \wedge (x_i=2 \vee x_i=3) \wedge x_{i+1}=2) \\
& & \quad \vee (x_{i-1}=4 \wedge (x_i=1 \vee x_i=3) \wedge x_{i+1}=0) \\
& & \quad \vee (x_{i-1}=0 \wedge (x_i=1 \vee x_i=4) \wedge x_{i+1}=4) \\
& & \quad \vee (x_{i-1}=3 \wedge x_i=4 \wedge x_{i+1}=1) \\
& & \quad \vee (x_{i-1}=1 \wedge (x_i=1 \vee x_i=3 \vee x_i=4) \wedge x_{i+1}=3)) \\
& & \qquad \longrightarrow \quad x_i := \frac{x_{i-1}+x_{i+1}}{2} \\
& A_{\text{eq},0}: & (x_{i-1}=x_{i+1}) \wedge ((x_{i-1}=3 \wedge x_i=3) \\
& & \vee (x_{i-1}=4 \wedge (x_i=1 \vee x_i=3 \vee x_i=4)) \\
& & \qquad \longrightarrow \quad x_i:=0 \\
& A_{\text{eq},1}: & (x_{i-1}=x_{i+1}) \wedge ((x_{i-1}=0 \wedge ((x_i=0 \vee x_i=4) \\
& & \vee (x_{i-1}=2 \wedge x_i=2))) \\
& & \qquad \longrightarrow \quad x_i:=1 \\
& A_{\text{eq},2}: & (x_{i-1}=x_{i+1}) \wedge ((x_{i-1}=1 \wedge (x_i=0 \vee x_i=1)) \\
& & \qquad \longrightarrow \quad x_i:=2 \\
& A_{\text{eq},3}: & (x_{i-1}=x_{i+1}) \wedge ((x_{i-1}=3 \wedge x_i=2) \\
& & \vee (x_{i-1}=2 \wedge (x_i=1 \vee x_i=4)) \\
& & \qquad \longrightarrow \quad x_i:=0 \\
& A_{\text{eq},4}: & (x_{i-1}=x_{i+1}) \wedge ((x_{i-1}=1 \wedge x_i=3) \\
& & \vee (x_{i-1}=3 \wedge x_i=1)) \\
& & \qquad \longrightarrow \quad x_i:=4
\end{array}
$$

Figure 14: A new self-stabilizing protocol for Leader Election on a ring.

6 Conclusions and Future Work

We presented a swarm synthesis method that exploits search diversification and parallelism to add convergence to non-stabilizing symmetric protocols. A symmetric protocol includes a set of processes with similar functionalities where the code of a process can be obtained from the code of another by a simple re-indexing of variables [14]. While the problem of adding convergence to non-stabilizing protocols is known to be in NP, we are not aware of any algorithm that adds convergence in polynomial-time (in protocol state space). We conjecture that adding convergence is most likely a hard problem due to the exponential number of the combinations of recovery actions that could resolve deadlocks without creating non-progress cycles in the set of illegitimate states of the protocol. Existing methods [7,8] for adding convergence perform a sequential search in the state space of non-stabilizing protocols to synthesize necessary convergence actions that result in a Self-Stabilizing (SS) version of the non-stabilizing protocol. However, such techniques often search only a part of the problem space due to their sequential nature, thereby resulting in premature failures in finding a SS protocol. The main contribution of the proposed approach is to increase the chance of success in automated design of self-stabilization by exploiting search diversification (through randomization) and parallelism. Specifically, we first make an approximation of stabilization by assigning a rank to any state s based on the least number of actions that can recover the protocol to a legitimate state from s. Using such ranking, we create a partial order of recovery actions, thereby assigning a rank to each candidate recovery action. Then, we make an ordered list of all recovery actions in the same rank. Subsequently, we instantiate several threads in an embarrassingly parallel fashion, where each thread investigates the addition of convergence based on a distinct order of recovery actions in each rank. This way, each thread searches a portion of problem space for SS protocol. We have designed and implemented a software tool, called the parallel STabilization Synthesizer (pSTSyn), that has automatically generated new solutions for several well-known protocols such as maximal matching, graph coloring, agreement and leader election on a ring. To the best

of our knowledge, pSTSyn is the first tool that enables swarm synthesis of convergence.

There are several extensions to this work that we would like to investigate. First, we plan to devise a method for swarm synthesis of convergence for asymmetric protocols. Second, we will investigate how pSTSyn can be used for adding convergence to protocols with dynamic topologies (e.g., overlay networks). The design of protocols with dynamic topologies is especially challenging as the locality of each process may change, thereby changing the transition groups that are created due to different scope of readability for processes. As a result, in a dynamic network, for each state of the network topology we have a distinct set of transition groups that form the transition system of the protocol. Parallelism can be especially beneficial in tackling this problem. Finally, we are currently investigating how we can use alternative solutions for a protocol to synthesize a generic solution for arbitrary number of processes.

References

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

[2] G. Varghese. *Self-stabilization by local checking and correction.* PhD thesis, MIT/LCS/TR-583, 1993.

[3] A. Arora, M. Gouda, and G. Varghese. Constraint satisfaction as a basis for designing nonmasking fault-tolerant systems. *Journal of High Speed Networks*, 5(3):293–306, 1996. A preliminary version appeared at ICDCS'94.

[4] 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.

[5] 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.

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

[7] Fuad Abujarad and Sandeep S. Kulkarni. Automated constraint-based addition of nonmasking and stabilizing fault-tolerance. *Journal of Theoretical Computer Science*, 258(2):3–15, 2011. In Press.

[8] Ali Ebnenasir and Aly Farahat. A lightweight method for automated design of convergence. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 219–230, 2011.

[9] 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.

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

[11] 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.

[12] M. Gouda. The triumph and tribulation of system stabilization. In Jean-Michel Helary and Michel Raynal, editors, *Distributed Algorithms, (9th WDAG'95)*, volume 972 of *Lecture Notes in Computer Science (LNCS)*, pages 1–18. Springer-Verlag, Le Mont-Saint-Michel, France, September 1995.

[13] Mohamed G. Gouda and Hrishikesh B. Acharya. Nash equilibria in stabilizing systems. In *11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 311–324, 2009.

[14] E.A. Emerson and A.P. Sistla. Symmetry and model checking. *Formal methods in system design*, 9(1):105–131, 1996.

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

[16] R. Gentilini, C. Piazza, and A. Policriti. Computing strongly connected components in a linear number of symbolic steps. In *the 14th Annual ACM-SIAM symposium on Discrete algorithms*, pages 573–582, 2003.

[17] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions On Computers*, 35(8):677–691, 1986.

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

7 Appendix: Alternative Self-Stabilizing Solutions for Maximal Matching

In addition to the SS maximal matching protocol presented in Section 5.1, pSTSyn also synthesized three other SS protocols as follows:

$$
\begin{array}{lll}
P_i:\ A_{\text{self},1}: & (\mathrm{m}_{i-1}{=}\mathsf{left}) \wedge (\mathrm{m}_{i+1}{=}\mathsf{right}) \wedge (\mathrm{m}_i \neq \mathsf{self}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{self} \\
\quad A_{\text{self},2}: & (\mathrm{m}_{i-1}{=}\mathsf{left}) \wedge (\mathrm{m}_{i+1}{=}\mathsf{self}) \wedge (\mathrm{m}_i \neq \mathsf{self}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{self} \\
\quad A_{\text{self},3}: & (\mathrm{m}_{i-1} \neq \mathsf{right}) \wedge (\mathrm{m}_{i+1}{=}\mathsf{right}) \wedge (\mathrm{m}_i{=}\mathsf{right}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{self} \\
\\
\quad A_{\text{right},1}: & (\mathrm{m}_{i-1} \neq \mathsf{right}) \wedge (\mathrm{m}_i \neq \mathsf{right}) \wedge (\mathrm{m}_{i+1}{=}\mathsf{left}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{right} \\
\\
\quad A_{\text{left},1}: & (\mathrm{m}_{i-1} \neq \mathsf{left}) \wedge (\mathrm{m}_i \neq \mathsf{left}) \wedge (\mathrm{m}_{i+1} \neq \mathsf{left}) & \\
 & \wedge \neg((\mathrm{m}_{i-1}{=}\mathsf{left}) \wedge (\mathrm{m}_i{=}\mathsf{right}) \wedge (\mathrm{m}_{i+1}{=}\mathsf{right})) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{left}
\end{array}
$$

Figure 15: An alternative solution to Maximal Matching

$$
\begin{array}{lll}
P_i:\ C_{\text{self},1}: & (\mathrm{m}_{i-1}{=}\mathsf{left}) \wedge (\mathrm{m}_i \neq \mathsf{self}) \wedge (\mathrm{m}_{i+1}{=}\mathsf{right}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{self} \\
\quad C_{\text{self},2}: & (\mathrm{m}_{i-1}{=}\mathsf{left}) \wedge (\mathrm{m}_i{=}\mathsf{left}) \wedge (\mathrm{m}_{i+1} \neq \mathsf{left}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{self} \\
\quad C_{\text{self},3}: & (\mathrm{m}_{i-1} \neq \mathsf{right}) \wedge (\mathrm{m}_i{=}\mathsf{left}) \wedge (\mathrm{m}_{i+1}{=}\mathsf{right}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{self} \\
\\
\quad C_{\text{left},1}: & (\mathrm{m}_{i-1}{=}\mathsf{right}) \wedge (\mathrm{m}_i \neq \mathsf{left}) \wedge (\mathrm{m}_{i+1} \neq \mathsf{left}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{left} \\
\quad C_{\text{left},2}: & (\mathrm{m}_{i-1}{=}\mathsf{right}) \wedge (\mathrm{m}_i{=}\mathsf{self}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{left} \\
\quad C_{\text{left},3}: & (\mathrm{m}_{i-1} \neq \mathsf{left}) \wedge (\mathrm{m}_i{=}\mathsf{right}) \wedge (\mathrm{m}_{i+1}{=}\mathsf{right}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{left} \\
\\
\quad C_{\text{right},1}: & (\mathrm{m}_{i-1} \neq \mathsf{right}) \wedge (\mathrm{m}_i \neq \mathsf{right}) \wedge (\mathrm{m}_{i+1}{=}\mathsf{left}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{right} \\
\quad C_{\text{right},2}: & (\mathrm{m}_{i-1} \neq \mathsf{right}) \wedge (\mathrm{m}_i{=}\mathsf{self}) \wedge (\mathrm{m}_{i+1} \neq \mathsf{right}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{right}
\end{array}
$$

Figure 16: An alternative solution to Maximal Matching

$$
\begin{array}{lll}
P_i:\ D_{\text{self},1}: & (\mathrm{m}_{i-1}{=}\mathsf{left}) \wedge (\mathrm{m}_i \neq \mathsf{self}) \wedge (\mathrm{m}_{i+1} \neq \mathsf{left}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{self} \\
\\
\quad D_{\text{left},1}: & (\mathrm{m}_{i-1}{=}\mathsf{right}) \wedge (\mathrm{m}_i \neq \mathsf{left}) \wedge (\mathrm{m}_{i+1} \neq \mathsf{left}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{left} \\
\quad D_{\text{left},2}: & (\mathrm{m}_{i-1}{=}\mathsf{right}) \wedge (\mathrm{m}_i{=}\mathsf{right}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{left} \\
\quad D_{\text{left},3}: & (\mathrm{m}_{i-1} \neq \mathsf{left}) \wedge (\mathrm{m}_i{=}\mathsf{self}) \wedge (\mathrm{m}_{i+1}{=}\mathsf{right}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{left} \\
\\
\quad D_{\text{right},1}: & (\mathrm{m}_{i-1}{=}\mathsf{self}) \wedge (\mathrm{m}_i{=}\mathsf{self}) \wedge (\mathrm{m}_{i+1} \neq \mathsf{right}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{right} \\
\quad D_{\text{right},2}: & (\mathrm{m}_{i-1} \neq \mathsf{right}) \wedge (\mathrm{m}_i \neq \mathsf{right}) \wedge (\mathrm{m}_{i+1}{=}\mathsf{left}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{right} \\
\quad D_{\text{right},3}: & (\mathrm{m}_i{=}\mathsf{self}) \wedge (\mathrm{m}_{i+1}{=}\mathsf{left}) & \longrightarrow \quad \mathrm{m}_i{:=}\mathsf{right}
\end{array}
$$

Figure 17: An alternative solution to Maximal Matching