A Lightweight Method for Automated Design of Convergence*

Ali Ebnenasir
Computer Science Department
Michigan Technological University
Houghton MI 49931, USA
Email: aebnenas@mtu.edu

Aly Farahat
Computer Science Department
Michigan Technological University
Houghton MI 49931, USA
Email: anfaraha@mtu.edu

***Abstract*—Design and verification of Self-Stabilizing (SS) network protocols are difficult tasks in part because of the requirement that a SS protocol must recover to a set of legitimate states from *any* state in its state space (when perturbed by transient faults). Moreover, distribution issues exacerbate the design complexity of SS protocols as processes should take local actions that result in global recovery/convergence of a network protocol. As such, most existing design techniques focus on protocols that are locally-correctable. To facilitate the design of finite-state SS protocols (that may not necessarily be locally-correctable), this paper presents a lightweight formal method supported by a software tool that automatically adds convergence to non-stabilizing protocols. We have used our method/tool to automatically generate several SS protocols with up to 40 processes (and 3^{40} states) in a few minutes on a regular PC. Surprisingly, our tool has automatically synthesized both protocols that are the same as their manually-designed versions as well as new solutions for well-known problems in the literature (e.g., Dijkstra's token ring [1]). Moreover, the proposed method has helped us reveal flaws in a manually designed SS protocol.**

***Keywords*-Fault Tolerance, Self-Stabilization, Convergence, Automated Design**

I. 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 [2], [3], [4] 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). Second, since a protocol includes a set of processes communicating via network channels, the convergence should be achieved with the coordination of multiple processes while each process is aware of only its locality – where *locality* of a process includes a set of processes whose state is readable by that process. Third, functionalities designed for convergence should not interfere with normal functionalities in the absence of faults (and vice versa).

Most existing methods for the design and verification of convergence are *manual*, which require ingenuity for the initial design (that may be incorrect) and need an extensive effort for proving the correctness of manual design [5]. For example, several techniques use layering and modularization [6], [7], [8], where a strictly decreasing *ranking function*, often designed manually, is used to ensure that the local actions of processes can only decrease the ranking function, thereby guaranteeing convergence. Constraint satisfaction methods [4], [9] first create a dependency graph of the local constraints of processes, and then illustrate how these constraints should be satisfied so global recovery is established. Local-checking/correction for global recovery [3], [4], [10], [9] is mainly used for the design of *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. It is unclear how one could directly use such methods for the design of convergence for protocols that are not locally-correctable (e.g., Dijkstra's token ring protocol [1]).

This paper proposes a lightweight formal method that automates the addition of convergence to non-locally correctable protocols. The approach is *lightweight* in that we start from instances of a protocol with small number of processes and add convergence automatically. Then, we inductively increase the number of processes as long as the available computational resources permit us to benefit from automation.[1] There are several advantages to this approach. First, we generate specific SS instances of non-stabilizing protocols that are correct-by-construction, thereby eliminating the need for proofs of correctness. Second, we facilitate the generation of an initial design of a SS protocol in a fully automatic way. Third, while for some protocols, the generated SS versions cannot easily be generalized for larger number of processes, the small instances of the protocol provide valuable insights for designers as to how convergence should be added/verified as a protocol scales up. To our knowledge, the proposed method is the first approach that automatically synthesizes SS protocols from their non-locally correctable non-stabilizing versions.

Contributions. The contributions of this paper are as follows. We present

* This work was partially sponsored by the NSF grant CCF-0950678 and a grant from Michigan Technological University.

[1] New processes often include new variables. Thus, increasing the number of processes often results in expanding the state space.

- a lightweight formal method (see Figure 1) supported by a software tool that facilitates the generation of initial designs of SS protocols;
- a sound heuristic that adds convergence to a non-stabilizing protocol p, for a specific number of processes k, a specified set of legitimate states I and a static topology. This heuristic first generates an approximation of convergence by identifying a sequence of sets of states $Rank[1], \cdots, Rank[M]$ ($M > 1$) such that the length of the shortest execution from any state in $Rank[i]$ to some state in I is equal to i; i.e., each $Rank[i]$ identifies a rank i for a subset of states. The significance of this ranking is that any scheme that augments p with convergence functionalities cannot decrease the rank of a state more than one unit in a single step (see Lemma IV.2 for proof). Thus, our ranks provide (i) a base set of recovery steps that should be included in any SS version of p, and (ii) a lower bound for all non-increasing ranking functions in terms of the number of recovery steps to I. We use this ranking of states to guide our heuristic as to how recovery should be added incrementally without creating interference with other functionalities. From a specific illegitimate state s_i, the success of convergence to some legitimate state s_l also depends on the order/sequence of processes that can execute from s_i to get the global state of the protocol to s_l, called a *recovery schedule*. From s_i, there may be several recovery schedules that result in executions that reach some legitimate state. Since during the execution of the heuristic the selected schedule remains unchanged, for each schedule, we can instantiate one instance of our heuristic on a separate machine (see Figure 1). If the proposed heuristic succeeds in finding a solution for a specific k and a specific schedule, then the resulting self-stabilizing protocol is correct-by-construction for k processes; otherwise, we declare failure in designing convergence for that instance of the protocol.
- a software tool called STabilization Synthesizer (STSyn), that implements the proposed heuristic. STSyn has synthesized several SS protocols (in a few minutes on a regular PC) similar to their manually-designed versions in addition to synthesizing new solutions. Thus far, STSyn has automatically generated instances of Dijkstra's token ring protocol [1] (3 different versions) with up to 5 processes, matching on a ring [11] with up to 11 processes, three coloring of a ring with up to 40 processes, and a two-ring self-stabilizing protocol with 8 processes. To the best of our knowledge, this is the first time that Dijkstra's SS token ring protocol is generated automatically.

Organization. Section II presents the preliminary concepts. Section III formulates the problem of adding convergence. Section IV discusses a method for generating an approximation of convergence. Section V presents a sound and efficient heuristic for automated addition of convergence. Section VI presents some case studies. Subsequently, Section VII demonstrates our experimental results, and Section VIII discusses related work, and some applications and limitations of the proposed approach. We make concluding remarks and discuss future work in Section IX.

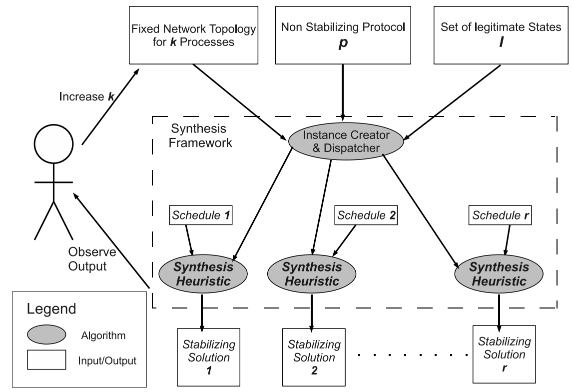

Figure 1. The proposed lightweight method for automated design of convergence.

II. Preliminaries

In this section, we present the formal definitions of protocols, our distribution model (adapted from [12]), 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 [1], [13], [5], [14]. For ease of presentation, we use a simplified version of Dijkstra's token ring protocol [1] as a running example.

Protocols as (non-deterministic) finite-state machines. A *protocol* p is a tuple $\langle V_p, \delta_p, \Pi_p, T_p \rangle$ of a finite set V_p of variables, a set δ_p of transitions, a finite set Π_p of k processes, where $k \geq 1$, and a topology T_p. Each variable $v_i \in V_p$, for $1 \leq i \leq N$, has a finite non-empty domain D_i. A *state* s of p is a valuation $\langle d_1, d_2, \cdots, d_N \rangle$ of variables $\langle v_1, v_2, \cdots, v_N \rangle$, where $d_i \in D_i$. 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. For a variable v and a state s, $v(s)$ denotes the value of v in s. 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. A *state predicate* is any subset of S_p specified as a Boolean expression over V_p. We say a state predicate X *holds in a state* s (respectively, $s \in X$) if and only if (iff) X evaluates to true at s.

Distribution model (Topology). We adopt a shared memory model [15] since reasoning in a shared memory setting is easier, and several (correctness-preserving) transformations [16], [17] exist for the refinement of shared memory

SS protocols to their message-passing versions. We model the topological constraints (denoted T_p) of a protocol p by a set of read and write restrictions imposed on variables that identify the locality of each process. Specifically, we consider a subset of variables in V_p that a process P_j ($1 \leq j \leq k$) can write, denoted w_j, and a subset of variables that P_j is allowed to read, denoted r_j. We assume that for each process P_j, $w_j \subseteq r_j$; i.e., if a process can write a variable, then that variable is readable for that process. A process P_j is not allowed to update a variable $v \notin w_j$.

Every transition of a process P_j belongs to a *group* of transitions due to the inability of P_j in reading variables that are not in r_j. 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 a process P_j, denoted r_j, meet the following constraints: $\forall v : v \in r_j : (v(s_0) = v(s'_0)) \wedge (v(s_1) = v(s'_1))$ and $\forall v : v \notin r_j : (v(s_0) = v(s_1)) \wedge (v(s'_0) = v(s'_1))$.

Effect of distribution on protocol representation. Due to read/write restrictions, we represent a *process* P_j ($1 \leq j \leq k$) as a set of transition groups $P_j = \{g_{j1}, g_{j2}, \cdots, g_{jm}\}$ created due to read restrictions r_j, where $m \geq 1$. Due to write restrictions w_j, no transition group g_{ji} ($1 \leq i \leq m$) includes a transition (s_0, s_1) that updates a variable $v \notin w_j$. Thus, the set of transitions δ_p of a protocol p is equal to the union of the transition groups of its processes; i.e., $\delta_p = \cup_{j=1}^{k} P_j$. (It is known that the total number of groups is polynomial in $|S_p|$ [12]). We use p and δ_p interchangeably.

Example: Token Ring (TR). The Token Ring (TR) protocol (adapted from [1]) includes four processes $\{P_0, P_1, P_2, P_3\}$ each with an integer variable x_j, where $0 \leq j \leq 3$, with a domain $\{0, 1, 2\}$. We use Dijkstra's guarded commands language [18] as a shorthand for representing the set of protocol transitions. A guarded command (action) is of the form $grd \rightarrow stmt$, and includes a set of transitions (s_0, s_1) such that the predicate grd holds in s_0 and the atomic execution of the statement $stmt$ results in state s_1. An action $grd \rightarrow stmt$ is *enabled* in a state s iff grd holds at s. A process $P_j \in \Pi_p$ is *enabled* in s iff there exists an action of P_j that is enabled at s. The process P_0 has the following action (addition and subtraction are in modulo 3):

$$A_0 : \quad (x_0 = x_3) \quad \longrightarrow \quad x_0 := x_3 + 1$$

When the values of x_0 and x_3 are equal, P_0 increments x_0 by one. We use the following parametric action to represent the actions of processes P_j, for $1 \leq j \leq 3$:

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

Each process P_j increments x_j only if x_j is one unit less than x_{j-1}. By definition, process P_j, for $j = 1, 2, 3$, *has a token* iff $x_j + 1 = x_{j-1}$. Process P_0 *has a token* iff $x_0 = x_3$. We define a state predicate S_1 that captures the set of states in which only one token exists, where S_1 is

$$\begin{array}{l} ((x_0 = x_1) \wedge (x_1 = x_2) \wedge (x_2 = x_3)) \vee \\ ((x_1 + 1 = x_0) \wedge (x_1 = x_2) \wedge (x_2 = x_3)) \vee \\ ((x_0 = x_1) \wedge (x_2 + 1 = x_1) \wedge (x_2 = x_3)) \vee \\ ((x_0 = x_1) \wedge (x_1 = x_2) \wedge (x_3 + 1 = x_2)) \end{array}$$

Let $\langle x_0, x_1, x_2, x_3 \rangle$ denote a state of TR. Then, the state $s_1 = \langle 1, 0, 0, 0 \rangle$ belongs to S_1, where P_1 has a token.

Each process P_j ($1 \leq j \leq 3$) is allowed to read variables x_{j-1} and x_j, but can write only x_j. Process P_0 is permitted to read x_3 and x_0 and can write only x_0. Thus, since a process P_j is unable to read two variables (each with a domain of three values), each group associated with an action A_j includes nine transitions. For a TR protocol with n processes and with $n-1$ values in the domain of each variable x_j, each group includes $(n-1)^{n-2}$ transitions. $\lhd$

Computations. Intuitively, a computation of a protocol $p = \langle V_p, \delta_p, \Pi_p, T_p \rangle$ 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}) ($i \geq 0$) in σ, there exists an action $grd \rightarrow stmt$ in some process $P_j \in \Pi_p$ 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 $m \geq 0$, such that each transition (s_i, s_{i+1}) in σ ($0 \leq i < m$) belongs to some action $grd \rightarrow stmt$ in some process $P_j \in \Pi_p$. The projection of a protocol p on a non-empty state predicate X, denoted as $\delta_p|X$, is the protocol $\langle V_p, \{(s_0, s_1) : (s_0, s_1) \in \delta_p \wedge s_0, s_1 \in X\}, \Pi_p, T_p \rangle$. 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* [14] requires that every computation starting in I remains in I.

TR Example. Starting from a state in the state predicate S_1, the TR protocol generates an infinite sequence of states, where all reached states belong to S_1. $\lhd$

Convergence and self-stabilization. Let I be a state predicate. We say that a protocol $p = \langle V_p, \delta_p, \Pi_p, T_p \rangle$ *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 $\sigma = \langle s_i, s_{i+1}, \cdots, s_j, s_i \rangle$ 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 convergence involves the resolution of deadlocks and non-progress cycles, we restate the definition of strong convergence as follows:

Proposition II.1. A protocol p *strongly converges to* I iff there are no deadlock states in $\neg I$ and no non-progress cycles in $\delta_p \mid \neg I$.

TR Example. If the TR protocol starts from a state outside S_1, then it may reach a deadlock state; e.g., the state $\langle 0, 0, 1, 2 \rangle$ is a deadlock state. Thus, the TR protocol is neither weakly stabilizing nor strongly stabilizing to S_1. $\triangleleft$

III. Problem Statement

Consider a non-stabilizing protocol $p = \langle V_p, \delta_p, \Pi_p, T_p \rangle$ and a state predicate I, where I is closed in p. Our objective is to generate a (weakly/strongly) stabilizing version of p, denoted p_{ss}, by *adding* (weak/strong) convergence to I. To separate the convergence property from functional concerns, we do not change the behavior of p in the absence of transient faults during the addition of convergence. With this motivation, during the synthesis of p_{ss} from p, no states (respectively, transitions) are added to or removed from I (respectively, $\delta_p|I$). This way, if in the absence of faults p_{ss} starts in I, then p_{ss} will preserve the correctness of p; i.e., the added convergence does not interfere with normal functionalities of p in the absence of faults. Moreover, if p_{ss} starts in a state in $\neg I$, then only convergence to I will be provided by p_{ss}. This is a specific instance of a more general problem as follows: (Problem III.1 is an adaptation of the problem of adding fault tolerance in [12].)

Problem III.1: Adding Convergence

- **Input**: (1) a protocol $p = \langle V_p, \delta_p, \Pi_p, T_p \rangle$; (2) a state predicate I such that I is closed in p; (3) a property of L_s converging, where $L_s \in$ {weakly, strongly}, and (4) topological constraints captured by read/write restrictions.
- **Output**: A protocol $p_{ss} = \langle V_p, \delta_{p_{ss}}, \Pi_p, T_p \rangle$ such that the following constraints are met: (1) I is unchanged; (2) $\delta_{p_{ss}}|I = \delta_p|I$, and (3) p_{ss} is L_s converging to I. ■

IV. Approximating Strong Convergence

Two intertwined problems complicate the design and verification of strong convergence, namely deadlock and non-progress cycle resolution (see Proposition II.1). Let p be a protocol that fails to converge to a state predicate I from states in $\neg I$. That is, transient faults may perturb p to either a deadlock state $s_d \in \neg I$ or a state $s_c \in \neg I$ from where a non-progress cycle is reachable. To resolve s_d, designers should include recovery/convergence actions [13], [4] in processes of p and ensure that any computation prefix that starts in s_d will eventually reach a state in I. However, due to the incomplete knowledge of processes with respect to the global state of the protocol, the local recovery transitions (represented as groups of transitions in our model) may create non-progress cycles. For example, process P_1 is deadlocked if the TR protocol (introduced in Section II) reaches the state $\langle 1, 2, 2, 0 \rangle$ by the occurrence of a transient fault. To resolve this deadlock state and ensure convergence to S_1, we include the recovery action $x_1 = x_0 + 1 \rightarrow x_1 := x_0 - 1$ in P_1. However, this recovery action creates a non-progress cycle starting from the state $\langle 1, 2, 1, 0 \rangle$ with the schedule (P_3, P_2, P_1, P_0) repeated three times. To resolve non-progress cycles, one has to break the cycle by removing a transition in the cycle with its group-mates, which in turn may result in creating new deadlock states and so on. As such, it appears that for deadlock and cycle resolution, one has to consider an exponential number of combinations of the groups of transitions that should be included in a non-converging protocol to make it strongly converging. While Problem III.1 is known to be in NP [12], [19], a polynomial-time algorithm (in $|S_p|$) for the addition of convergence is unknown, nor do we know whether the addition of convergence is NP-complete. Therefore, we resort to designing efficient heuristics that may fail to add convergence to some protocols; nonetheless, if they succeed to add convergence, the resulting SS protocol is correct by construction.

In order to mitigate the complexity of algorithmic design of strong convergence, we first design an approximation of strong convergence that includes all *potentially useful* transitions. Specifically, our approximation includes two steps.

1) First, we calculate an intermediate protocol p_{im} that includes the transition groups of a non-stabilizing protocol p and the weakest set of transitions that start in $\neg I$ and adhere to the read/write restrictions of the processes of p. Hence, we guarantee that $p_{im}|I$ remains the same as $p|I$; i.e., the *closure* of I in p is preserved. Step 1 in Figure 2 computes p_{im}.
2) Second, we compute a sequence of state predicates $Rank[1], \cdots, Rank[M]$, where $Rank[i] \subseteq \neg I$ and $Rank[i]$ includes the set of states s from where the length of the shortest computation prefix of p_{im} from s to I, called the *rank* of s, is equal to i, for $1 \leq i \leq M$. That is, $Rank[i]$ includes all states with rank i. Note that, for any state $s \in I$, the rank of s is zero. Figure 2 illustrates the algorithm ComputeRanks that

takes a protocol p and a state predicate I that is closed in p, and returns an array of state predicates *Rank*[]. The repeat-until loop in Figure 2 computes the set of backward reachable states from I, denoted *explored*, using the transitions of p_{im}. In each iteration i, Line 3 calculates a set of states *Rank*[i] outside *explored* from where some state in *explored* can be reached by a single transition of p_{im}. The repeat-until loop terminates when no more states can be added to *explored*. The rank of a state in $\neg I$ that does not belong to any rank is infinity, denoted ∞. That is, if rank of s is ∞, then there is no computation prefix of p_{im} from s that includes a state in I.

Theorem IV.1 Upon termination of the above two steps, if $\{s|$ rank of s is $\infty\} = \emptyset$, then p_{im} is a weakly stabilizing version of p. Otherwise, no stabilizing version of p exists. That is, ComputeRanks is sound and complete for finding a weakly stabilizing version of p. (Proof in [20]) ■

ComputeRanks(p: protocol, I: state predicate) {
/* *Rank* is an array of state predicates. */
- $p_{im} := \delta_p \cup \{g \mid \exists P_j \in \Pi_p : g \in P_j : (\forall (s_0, s_1) : (s_0, s_1) \in g : s_0 \notin I)\}$ (1)
- $explored := I$; $Rank[0] := I$; $i := 1$; (2)
- repeat {
  - $Rank[i] := \{s_0 \mid (s_0 \notin explored) \wedge (\exists s_1, g : (s_1 \in explored) \wedge (g \in p_{im}) : (s_0, s_1) \in g\}$; (3)
  - $explored := explored \cup Rank[i]$; (4)
  - $i := i + 1$; } until $(Rank[i-1] = \emptyset)$; (5)
- return *Rank*; }

Figure 2. Compute ranks of each state $s \in S_p$, where rank of s is the length of the shortest computation prefix of p from s to some state in I.

Let p_{ss} be a strongly converging version of p that meets the requirements of Problem III.1 for a predicate I that is closed in p.

Lemma IV.2 The protocol p_{ss} excludes any transition (s_0, s_1), where $s_0 \in$ *Rank*[i] and $s_1 \in$ *Rank*[j] such that $j + 1 < i$, for $1 \leq i \leq M$.

Proof. By contradiction, let p_{ss} include a transition (s_0, s_1) such that $s_0 \in$ *Rank*[i] and $s_1 \in$ *Rank*[j], and $j+1 < i$. Then, two cases could have happened: either (1) ComputeRanks() has missed s_0 as a state that is backward reachable from s_1 in a single step, which contradicts with the completeness of ComputeRanks(), or (2) ComputeRanks() has assigned a rank to s_0 which is greater than one unit from the rank of s_1 even though s_0 is backward reachable from s_1 in a single step, which is a contradiction with the soundness of ComputeRanks(). ■

Definition. We call a transition $t = (s_0, s_1)$ *rank decreasing iff* $s_0 \in$ *Rank*[i] and $s_1 \in$ *Rank*[j], where $j = i - 1$ $(0 < i \leq M)$.

Theorem IV.3 Every computation of p_{ss} that starts in a state $s_0 \in$ *Rank*[i] $(i > 0)$ includes a *rank decreasing* transition starting in *Rank*[j] for every j where $0 < j \leq i$.

Proof. p_{ss} strongly converges to I. Hence, every computation of p_{ss} that starts in *Rank*[i] has a prefix $\sigma = \langle s_0, s_1, \cdots, s_f \rangle$ where $s_f \in I$. Based on Lemma IV.2, a transition can at most decrease the rank of a *state* by 1. Hence, to change the rank from i to 0 (i.e. to reach I), σ should include a transition from *Rank*[i] to *Rank*[$i-1$], a transition from *Rank*[$i-1$] to *Rank*[$i-2$], $\cdots$, and a transition from *Rank*[1] to I. ■

V. Algorithmic Design of Strong Convergence

In this section, we present a sound heuristic for adding strong convergence. This heuristic uses the approximation method presented in Section IV as a preprocessing phase. Then, the heuristic incrementally includes recovery transitions from deadlock states towards synthesizing convergence from $\neg I$ to I under the following constraints:

(C1) a recovery transition must not have a groupmate transition that originates in I. (Recall that, due to read restrictions, all transitions in a group must be either included or excluded.);

(C2) recovery transitions are added from each *Rank*[i] to *Rank*[$i-1$], for $1 \leq i \leq M$;

(C3) the groupmates of added recovery transitions must not form a cycle outside I, and

(C4) no transition grouped with a recovery transition reaches a deadlock state.

The inclusion of recovery transitions is performed in three passes, where in the second and third passes we respectively ignore constraints (C4) and (C2). For ease of presentation, we first informally explain the proposed heuristic as follows (p is the non-stabilizing protocol, p_{ss} is the protocol being synthesized, which is initially equal to p):

1) *Preprocessing*:
   - If the transitions of p form any non-progress cycle in $\neg I$ and the transitions participating in the cycle have groupmates in $p|I$, then *exit*. The reasoning behind this step is that the elimination of cycle transitions violates the second constraint in the *output* of Problem III.1.
   - Perform the approximation proposed in Section IV. The results include an array of state predicates *Rank*[$0..M$], where *Rank*[i] denotes a state predicate containing states with rank i.
   - Compute the deadlock states in $\neg I$, denoted by the state predicate *deadlockStates*.
2) *Pass 1*: For each i, where $1 \leq i \leq M$, include the following recovery transitions in p_{ss}: any transition that starts from a deadlock state in *Rank*[i] and ends in a state in *Rank*[$i-1$] without having a groupmate transition that either starts in I or reaches a deadlock state. If no more deadlock states remain then return p_{ss}.
3) *Pass 2*: For each i, where $1 \leq i \leq M$, include the following recovery transitions in p_{ss}: any transition

that starts from a deadlock state in *Rank*$[i]$ and ends in a state in *Rank*$[i-1]$ without having a groupmate transition that starts in I. If no more deadlock states remain then return p_{ss}.

4) *Pass 3*: include (in p_{ss}) any transition that starts from a remaining deadlock state and ends in any state without having a groupmate transition that starts in I. If no more deadlock states remain then return p_{ss}.
5) If there are any remaining unresolved deadlock states, then declare failure.

Adding convergence from a state predicate to another. Before we discuss the details of each pass, we describe the Add_Convergence routine (see Figure 3) that adds recovery transitions from a state predicate *From* to another state predicate *To*. Such an inclusion of recovery transitions in the protocol p_{ss} is performed (i) under the read/write restrictions of processes, (ii) without creating cycles in $\neg I$, (iii) without including any transition group that is ruled out by the constraints of that pass, denoted *ruledOutTrans*, and (iv) based on the recovery schedule given in the array $sch[]$. An example recovery schedule for the TR program is $\{P_1, P_2, P_3, P_0\}$; i.e., $sch[1] = 1, sch[2] = 2, sch[3] = 3$ and $sch[4] = 0$. That is, when adding recovery from a deadlock state s_d, we first check the ability of P_1 in including a recovery transition from s_d, then the ability of P_2 and so on. We shall invoke Add_Convergence in Passes 1-3 with different input parameters.

In each iteration of the for loop in Line 1 of Add_Convergence, we use the routine Add_Recovery to check whether $P_{sch[j]}$ can add recovery transitions from $From$ to To using the transitions of p_{ss}. This addition of recovery transitions is performed while adhering to read/write restrictions of $P_{sch[j]}$ and excluding any transition in the set of transition groups *ruledOutTrans* (see Line 1 of Add_Recovery in Figure 3). Once a recovery transition is added, we need to make sure that its groupmate transitions do not create cycles with the groupmates of the transitions of p_{ss}. For this reason, we use the Identify_Resolve_Cycles (see Figure 3) routine in Line 2 of Add_Recovery.

The Identify_ Resolve_Cycles (see Figure 3) routine identifies any Strongly Connected Components (SCCs) that are created in $\neg I$ due to the inclusion of new recovery transitions in p_{ss}. A SCC is a state transition graph in which every state is reachable from any other state. Thus, a SCC may include multiple cycles. For the detection of SCCs, we implement an existing algorithm due to Gentilini *et al.* [21] (see Detect_SCC in Line 2 of Identify_ Resolve_Cycles). Detect_SCC returns an array of state predicates, denoted *SCCs*, where each array cell contains the states of a SCC in $p_{ss}|(\neg I)$. Detect_SCC also returns the number of SCCs. The for-loop in Line 3 of Identify_ Resolve_Cycles determines a set of groups of transitions *badTrans* that include at least a transition (s_0, s_1) that starts and ends in a SCC; i.e., (s_0, s_1) participates in at least one cycle. Step 3 in Add_Recovery excludes such groups of transitions from the set of groups of transitions added for recovery. As such, the remaining groups add recovery without creating any cycles.

```
Add_Convergence(From, To, I: state predicate; P_1, ···, P_K: set
                    of transition groups; sch[1..K]: integer array;
   p_ss, ruledOutTrans: set of transition groups, passNo: integer)
 /* sch is an array representing a preferred schedule based on
 /* which processes are used in the design of convergence. */
 /* The input p_ss is the union of P_1, ···, P_K. */
{ - for j := 1 to K {
   // use the schedule in array sch for adding recovery
    - p_ss := Add_Recovery(From, To, I, P_sch[j],
                              p_ss, ruledOutTrans);        (1)
    - deadlockStates := { s0 | s0 ∉ I ∧
                    (∀s_1, g: (s_0, s_1) ∈ g: g ∉ p_ss)};   (2)
    - if (deadlockStates = ∅) then
                         return deadlockStates, p_ss;      (3)
    - if (passNo = 1) then
         ruledOutTrans := {(s_0, s_1) | (s_0 ∈ I)∨
                        (s_1 ∈ deadlockStates)};           (4)
       } // for loop
  - return deadlockStates, p_ss;                           (5)
}

Add_Recovery(From, To, I: state predicate;
          P_j, p_ss, ruledOutTrans: set of transition groups)
{ - addedRecovery_j := { g | (g ∈ P_j) ∧
   (∃ (s0,s1) : (s0,s1) ∈ g ∧ s0 ∈ From ∧ s1 ∈ To ∧
                         g ∉ ruledOutTrans) }              (1)
  - badTrans := Identify_Resolve_Cycles(p_ss,
                         addedRecovery_j, ¬I);             (2)
  - return (p_ss∪ (addedRecovery_j − badTrans));           (3)
}

Identify_Resolve_Cycles(p_ss, addedTrans: set of transition groups;
                                          X: state predicate)
{ - badTrans := ∅; // transitions to be removed from cycles.  (1)
  - SCCs, numOfSCCs := Detect_SCC(p_ss∪ addedTrans, X);  (2)
     // SCCs is an array of state predicates in which
     // each array cell includes the states in an SCC.
  - for i := 1 to numOfSCCs {
      - groupsInSCC := { g | g ∈ addedTrans ∧
            (∃ (s0,s1) ∈ g :: s0 ∈ SCCs[i] ∧ s1 ∈ SCCs[i])};  (3)
      - badTrans := badTrans ∪ groupsInSCC;      }         (4)
  - return badTrans;                                       (5)
}
```

Figure 3. Add convergence. (The Detect_SCC routine is an implementation of the SCC detection algorithm due to Gentilini *et al.* [21].)

Pass 1: Adding recovery from *Rank*$[i]$ **to** *Rank*$[i-1]$ **excluding transitions that reach deadlocks.** In the first pass, we invoke Add_Convergence as follows while ensuring that the constraints (C1) to (C4) are met.

for $i := 1$ to M // Go through each rank
- *From* $:= \{s \mid s \in$ *Rank*$[i] \ \wedge\ s \in$ *deadlockStates*$\}$;
- *To* $:= \{s \mid s \in$ *Rank*$[i-1]\}$; *passNo* $:= 1$;
- *ruledOutTrans* $:= \{(s_0, s_1) \mid (s_0 \in I) \vee (s_1 \in$ *deadlockStates*$)\}$;
- *deadlockStates*, p_{ss} := Add_Convergence(*From, To*, $I, P_1, \cdots, P_K, sch[1..K], p_{ss}$, *ruledOutTrans*, *passNo*);
- If (*deadlockStates* $= \emptyset$), then return p_{ss};

In this pass, we iterate through each rank i ($0 < i \leq M$) and explore the possibility of adding recovery to the rank below. To enforce the constraints (C1)-(C4) in Pass 1, we do not include any recovery transition that has a groupmate that either starts in I or reaches a deadlock state (see *ruledOutTrans*). Then, in each iteration, we invoke Add_Convergence to add recovery from states in predicate *From* to states of *To*. If all deadlock states are resolved in some iteration, then p_{ss} is a strongly stabilizing protocol that converges to I. Otherwise, we move to the next pass.

TR Example. For the TR example introduced in Section II, the state predicate I is equal to S_1 (defined in Section II). ComputeRanks calculates two ranks ($M = 2$) that cover the entire predicate $\neg I$. The non-stabilizing TR protocol does not have any non-progress cycles in $\neg S_1$. The recovery schedule is P_1, P_2, P_3, P_0. We could not add any recovery transitions in the first phase as the groups that do not terminate in deadlock states cause cycles. $\triangleleft$

Pass 2: Adding recovery from ***Rank***$[i]$ **to** ***Rank***$[i-1]$ **including transitions that reach deadlocks.** If there are still some remaining deadlock states, then we proceed to Pass 2. In this pass, we execute the same for-loop as in Pass 1. The predicates *From* and *To* are computed in the same way as in Pass 1, nonetheless, we have *ruledOutTrans*= $\{(s_0, s_1) \mid (s_0 \in I)\}$. That is, we permit the inclusion of recovery groups that include transitions reaching deadlock states (i.e., we relax constraint C4).

TR Example. In the second phase, we add the recovery action $x_j = x_{j-1}+1 \rightarrow x_j := x_{j-1}$, for $1 \leq j \leq 3$, without introducing any cycles. No new transitions are included in P_0. The union of the added recovery action and the action A_j in the non-stabilizing TR protocol results in the action $x_j \neq x_{j-1} \rightarrow x_j := x_{j-1}$ for the domain {0, 1, 2}. Notice that, the synthesized TR protocol is the same as Dijkstra's token ring protocol in [1]. $\triangleleft$

Pass 3: Adding recovery from any remaining deadlock states to wherever possible. If Pass 2 does not generate a SS protocol, we explore the feasibility of adding recovery transitions from remaining deadlock states to any state without adhering to the ranking constraint (i.e., relaxing constraint C2). As such, we invoke Add_Convergence only once with *From* = *deadlockStates*, *To* = $true$ and *ruledOutTrans*= $\{(s_0, s_1) \mid (s_0 \in I)\}$.

Check for failure. After the three passes of adding recovery, if there are still some remaining deadlock states, then we declare failure in designing strong stabilization.

Theorem V.2 The heuristic presented in this section is sound, and has a polynomial time complexity in $|S_p|$. (Proof in [20]) ■

Comment on completeness. The proposed heuristic is incomplete in that for some protocols it may fail to add convergence while there exist SS versions of the input non-stabilizing protocol that meet the constraints of Problem III.1. One reason behind such incompleteness lies in the way we currently resolve cycles, which is a conservative removal of transitions (and their associated groups) that participate in non-progress cycles (see Lines 3-4 of Identify_Resolve_Cycles in Figure 3). Another cause of incompleteness is the conservative way we include recovery transitions. Our heuristic adds recovery transitions without backtracking to previously added transitions, thereby disregarding possible alternative solutions. We are currently investigating more intelligent methods of cycle resolution that synthesize a SS protocol in cases where the proposed heuristic fails. This way, STSyn will enable the addition of convergence for a wider class of protocols.

VI. Case Studies

In this section, we present some of our case studies for the addition of strong convergence. We presented a 4-process non-stabilizing Token Ring (TR) protocol in Section II and we demonstrated the synthesis of its strongly stabilizing version in Section V. STSyn also has synthesized alternative strongly stabilizing versions of the TR protocol available in [20]. Section VI-A discusses the synthesis of a strongly stabilizing maximal matching protocol, Section VI-B presents a stabilizing three coloring protocol, and Section VI-C presents our synthesis of a two-ring token ring protocol.

A. Maximal Matching on a Bidirectional Ring

The Maximal Matching (MM) protocol (presented in [11]) has K processes $\{P_0, \cdots, P_{K-1}\}$ located in 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 P_i points to its left neighbor, right neighbor or itself. Intuitively, two neighbor processes are *matched* iff they point to each other. More precisely, 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*). Process P_i points to itself iff m_i = self. 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 I_{MM} in that after stabilizing to I_{MM}, the actions of the synthesized MM protocol should no longer be enabled. We have automatically synthesized stabilizing MM protocols for $K = 5$ to 11 in at most 65 seconds. Due to space constraints, we present only the actions of P_0 in a synthesized protocol for $K = 5$ (see [20] for the actions of all processes).

$$m_4 = \text{left} \wedge m_0 \neq \text{self} \wedge m_1 = \text{right} \longrightarrow m_0 := \text{self}$$

$$(m_0 = \text{self} \wedge m_4 = \text{right}) \vee (m_0 \neq \text{left} \wedge m_1 \neq \text{self} \wedge m_4 = \text{right}) \longrightarrow m_0 := \text{left}$$

$$(m_0 = \text{self} \wedge m_1 = \text{left}) \vee (m_0 \neq \text{right} \wedge m_1 = \text{left} \wedge m_4 = \text{left}) \longrightarrow m_0 := \text{right}$$

If the left neighbor of P_0 (i.e., P_4) points to its left and its right neighbor (i.e., P_1) points to its right and P_0 does not point to itself, then it should point to itself. P_0 should point to its left neighbor in two cases: (1) P_0 points to itself and its left neighbor points to right, or (2) P_0 does not point to its left, its right neighbor does not point to itself, and its left neighbor points to right. Likewise, P_0 should point to its right neighbor in two cases: (1) P_0 points to itself and its right neighbor points to left, or (2) P_0 does not point to its right, its right neighbor points to left and its left neighbor points to left. These actions are different from the actions in the manually design MM protocol presented by Gouda and Acharya [11] as follows ($1 \leq i \leq K$):

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

Observe that the actions of processes in Gouda and Acharya's protocol are symmetric, whereas in our synthesized protocol they are not. This difference motivated us to investigate the causes of such differences. Surprisingly, while analyzing Gouda and Acharya's protocol, we found out that their protocol includes a non-progress cycle starting from the state $\langle left, self, left, self, left\rangle$ with a schedule P_0, P_1, P_2, P_3, P_4 repeated twice, where the tuple $\langle m_0, m_1, m_2, m_3, m_4\rangle$ denotes a state of the MM protocol. This experiment illustrates how difficult the design of strongly convergent protocols is and how automated design can facilitate the design and verification of convergence.

B. Three Coloring

In this section, we present a strongly stabilizing three-coloring protocol in a ring (adapted from [11]). The Three Coloring (TC) protocol has $K > 1$ processes located in a ring, where each process P_i has the left neighbor $P_{(i-1)}$ and the right neighbor $P_{(i+1)}$, where addition and subtraction are modulo K. Each process P_i has a variable c_i with a domain of three distinct values representing three colors. Each process P_i is allowed to read $c_{(i-1)}, c_i$ and $c_{(i+1)}$ and write only c_i. The non-stabilizing protocol has no transitions initially. The synthesized protocol must strongly stabilize to the predicate $I_{coloring} = \forall i : 0 \leq i \leq K-1 : c_{(i-1)} \neq c_i$ representing the set of states where every adjacent pair of processes get different colors (i.e., *proper coloring*). STSyn synthesized a stabilizing protocol with 40 processes with the following actions labeled by process index i ($1 < i < 40$).

$$P_1: \ (c_1 = c_0) \vee (c_1 = c_2) \longrightarrow c_1 := \text{other}(c_0, c_2)$$

$$P_i: \ (c_{(i-1)} \neq c_i) \wedge (c_i = c_{(i+1)}) \longrightarrow c_i := \text{other}(c_{(i-1)}, c_{(i+1)})$$

Notice that P_0 has no actions. The nondeterministic function *other*(x,y) returns a color different from x and y. This protocol is different from the TC protocol presented in [11]; i.e., STSyn generated an alternative solution.

C. Two-Ring Token Ring

In order to illustrate that our approach is applicable for more complicated topologies, in this section, we demonstrate how we added convergence to an extended version of Dijkstra's token ring.

The non-stabilizing Two-Ring Token Ring (TR^2) protocol. The TR^2 protocol includes 8 processes located in two rings A and B (see Figure 4). In Figure 4, the arrows show the direction of token passing. Process *PA*$_i$ (respectively, *PB*$_i$), $0 \leq i \leq 2$, is the predecessor of *PA*$_{i+1}$ (respectively, *PB*$_{i+1}$). Process *PA*$_3$ (respectively, *PB*$_3$) is the predecessor of *PA*$_0$ (respectively, *PB*$_0$). Each process *PA*$_i$ (respectively, *PB*$_i$), $0 \leq i \leq 3$, has an integer variable a_i (respectively, b_i) with the domain {0, 1, 2, 3}.

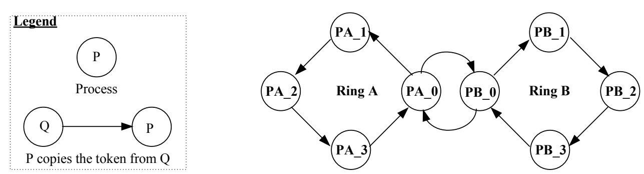

Figure 4. The Two-Ring Token Ring (TR^2) protocol.

Process *PA*$_i$, for $1 \leq i \leq 3$, *has the token* iff $(a_{i-1} = a_i \oplus 1)$, where $\oplus$ denotes addition modulo 4. Intuitively, *PA*$_i$ has the token iff a_i is one unit less a_{i-1}. Process *PA*$_0$ has the token iff $(a_0 = a_3) \wedge (b_0 = b_3) \wedge (a_0 = b_0)$; i.e., *PA*$_0$ has the same value as its predecessor and that value is equal to the values held by *PB*$_0$ and *PB*$_3$. Process *PB*$_0$ has the token iff $(b_0 = b_3) \wedge (a_0 = a_3) \wedge ((b_0 \oplus 1) = a_0)$. That is, *PB*$_0$ has the same value as its predecessor and that value is one unit less than the values held by *PA*$_0$ and *PA*$_3$. Process *PB*$_i$ $(1 \leq i \leq 3)$ *has the token* iff $(b_{i-1} = b_i \oplus 1)$. The TR^2 protocol also has a Boolean variable $turn$; ring A executes only if $turn = true$, and if ring B executes then $turn = false$.

In the absence of transient faults, there is at most one token in both rings. However, transient faults may set the variables to arbitrary values and create multiple tokens in the rings. We have synthesized a strongly self-stabilizing version of this protocol that ensures recovery to states where only one token exists in both rings. Due to space constraints, we omit the details of this example (available in [20]).

Figure 5 illustrates a summary of the case studies we have conducted in terms of being locally-correctable.

Table 1: Local Correctability of Case Studies

Case Study	Locally Correctable
3-Coloring	Yes
Matching	No
Token Ring (TR)	No
Two-Ring TR	No

Figure 5. Summary of case studies.

VII. Experimental Results

While the significance of our work is in enabling the automated design of convergence, we would like to report the potential bottlenecks of our work in terms of tool development. With this motivation, in this section, we first present the platform on which we conducted our experiments. Then we discuss our experimental results. We conducted our experiments on a Linux Fedora 10 distribution personal computer, with a 3GHz dual core Intel processor and 1GB of RAM. We have used C++ and the CUDD/GLU [22] library version 2.1 for Binary Decision Diagram (BDD) [23] manipulation in the implementation of STSyn.

Effect of increasing the number of processes. Figures 6 and 7 respectively represent how time and space complexity of synthesis grow as we increase the number of processes in the matching protocol. We measure the space complexity in terms of the number of BDD nodes rather than in kBytes for two reasons: (1) in a platform-independent fashion, the number of BDD nodes reflects how space requirements of our heuristic grow during synthesis, and (2) measuring the exact amount of allocated memory is often inaccurate. Observe that, for maximal matching, increasing the number of processes significantly increases the time and space complexity of synthesis. Nonetheless, since the domain size is constant, we were able to scale up the synthesis and generate a strongly stabilizing protocol with 11 processes in almost 65 seconds.

Figures 8 and 9 respectively demonstrate time/space complexity of adding convergence to the TC protocol. We have added convergence to the coloring protocol for 8 versions from 5 to 40 processes with a step of 5. Since the added recovery transitions for the coloring protocol do not create any SCCs outside $I_{coloring}$, we have been able to scale up the synthesis and generate a stabilizing protocol with 40 processes.

While variables have a domain of three values in both the coloring and the matching protocols, we observe that the synthesis of the coloring protocol is more scalable. This

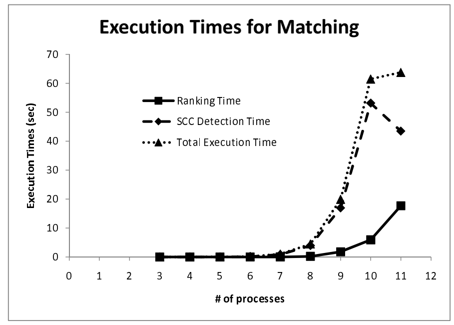

Figure 6. Time spent for adding convergence to matching versus the number of processes.

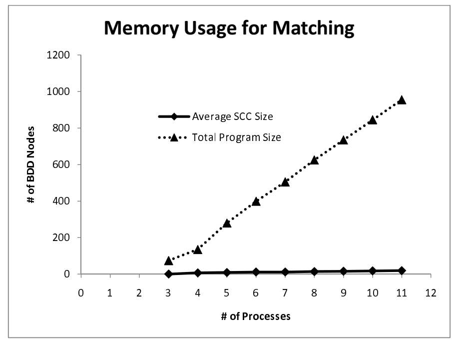

Figure 7. Space usage for adding convergence to matching versus the number of processes.

is in part due to the fact that the MM protocol is non-locally correctable whereas the coloring protocol is locally-correctable. More specifically, consider a case where the first conjunct of the local predicate LC_i is false for P_i. That is, m_i = left and $m_{i-1} \neq$ right. If P_i makes an attempt to satisfy its local predicate LC_i by setting m_i to self, then the third conjunct of its invariant may become invalid if $m_{i-1} \neq$ left. The last option for P_i would be to set m_i to right, which may not make the second conjunct true if $m_{i+1} \neq$ left. Thus, the success of P_i in correcting its local predicate depends on the actions of its neighbors as well. Likewise, corrective actions of P_i can change the truth value of LC_{i-1} and LC_{i+1}. Such dependencies cause cycles outside I_{MM}, which complicate the design of convergence. By contrast, in the coloring protocol, each process can easily establish its local predicate $c_{(i-1)} \neq c_i$ (without invalidating the recovery of its neighbors) by selecting a color that is different from its left and right neighbors.

Figures 10 and 11 respectively illustrate how time/space complexity of synthesis increases for the token ring protocol as we keep size of the domain of x variables constant (i.e., $|D| = 4$) and increase the number of processes.

We have conducted similar investigation (available at http://www.cs.mtu.edu/~anfaraha/CaseStudy) on the effect of the size of variable domains and the recovery schedule on the time/space complexity of synthesis, which we omit due to space constraint.

Comment. We have observed that the cause of irregulari-

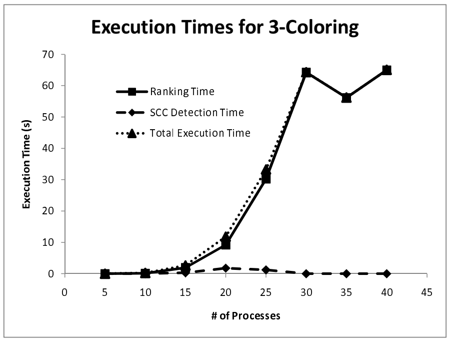

Figure 8. Time spent for adding convergence to 3-Coloring versus the number of processes.

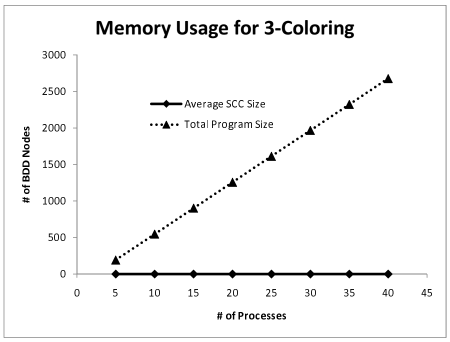

Figure 9. Space usage for adding convergence to 3-Coloring versus the number of processes.

ties in the time complexity of the coloring and matching protocols are due to two factors: (1) the number of processes causes asymmetry in the way non-progress cycles are formed, and (2) the underlying BDD library used for symbolic representation of protocols behaves irregularly when BDDs are not effectively optimized.

VIII. Discussion and Related Work

In this section, we discuss issues related to the applications, strengths and some limitations of our lightweight method.

Applications. There are several applications for the proposed lightweight method. First, STSyn can actually be integrated in model-driven development environments (such as Unified Modeling Language [24] and Motorola

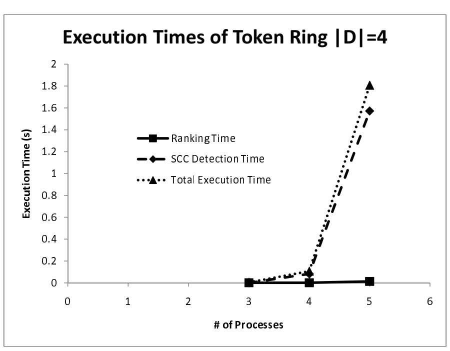

Figure 10. Time spent for adding convergence to Token Ring versus the number of processes.

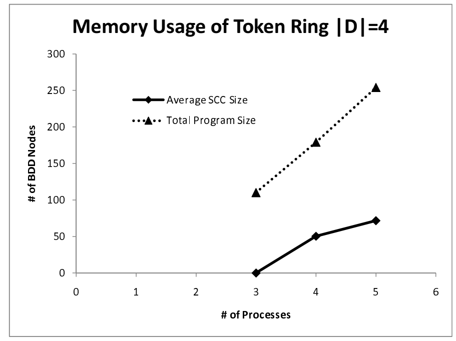

Figure 11. Space usage for adding convergence to Token Ring versus the number of processes.

WEAVER [25]) for protocol design and visualization. While model checkers generate a scenario as to how a protocol fails to self-stabilize, the burden of revising the protocol in such a way that it becomes self-stabilizing remains on the shoulders of designers. Our heuristics revise a protocol towards generating a SS version thereof. As such, an integration of our heuristics with model checkers can greatly benefit the designers of SS protocols.

Manual design. Several techniques for the design of self-stabilization exist [26], [6], [27], [3], [28], [7], [29], [30], [4], [10], [31], [8], most of which provide problem-specific solutions and lack the necessary tool support for automatic addition of convergence to non-stabilizing systems. For example, Katz and Perry [27] present a general (but expensive) method for transforming a non-stabilizing system to a stabilizing one by taking global snapshots and resetting the global state of the system. Varghese [3] and Afek *et al.* [10] put forward a method based on local checking for global recovery of locally correctable protocols. (See the maximal matching protocol in Section VI-A as an example of a protocol that is not locally correctable.) To design non-locally correctable systems, Varghese [28] proposes a counter flushing technique, where a leader node systematically increments and flushes the value of a counter throughout the network. Control-theoretic methods [32] mainly focus on synchronous systems and consider a centralized supervisor that enforces corrective actions.

Automated design. Awerbuch-Varghese [33], [3] 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). By contrast, our approach can also be applied to interactive protocols where correctness criteria are specified as a set of (possibly non-terminating) computations (e.g., Dijkstra's token ring). Furthermore, the input to their compilers is a synchronous and deterministic non-stabilizing protocol, whereas our heuristic adds convergence to asynchronous non-deterministic protocols as well. In our previous work [19], [34], we investigate the automated addition of recovery to distributed protocols for types of faults other than transient faults. In this approach, if recovery cannot be added from a deadlock state outside the set of legitimate states, then we verify whether or not that state can be made unreachable. This is not an option in the addition of self-stabilization; recovery should be added from any state in protocol state space. Bonakdarpour and Kulkarni [35] investigate the problem of adding progress properties to distributed protocols, where they allow the exclusion of reachable deadlock states from protocol computations towards ensuring progress. Abujarad and Kulkarni [36] present a heuristic for automatic addition of convergence to locally-correctable systems. By contrast, the proposed approach in this paper is more general in that we add convergence to non-locally correctable protocols (see Section VII).

Scalability. The objectives of this research place scalability at a low degree of priority as the philosophy behind our lightweight method is to benefit from automation as long as available computational resources permit. Nonetheless, we have analyzed the behavior of STSyn regarding time/space complexity (see Section VII). The extent to which we can currently scale up a protocol depends on many factors including the number of processes, variable domains and topology. For example, while our tool is able to synthesize a SS protocol with up to 40 processes for the 3-coloring problem, it is only able to find solutions for Dijkstra's token ring with up to 5 processes, each with a variable domain size of 5. One of the major factors affecting the scalability of our heuristic is the cycle resolution problem. The number of cycles mainly depends on the size of the variable domains and the size of the transition groups (which is also determined by the number of unreadable variables and their domains). Our experience shows that the larger the size of the groups and the variable domains, the more cycles we get. We believe that scaling-up our heuristic is strongly dependent on our ability to scale-up cycle resolution, which is the focus of one of our current investigations. Although the proposed heuristic does not scale-up systematically for all input protocols, our lightweight approach allows designers to have some concrete examples of a possibly general SS version of a non-stabilizing protocol.

Symmetry. We have synthesized several protocols (e.g., Dijkstra's token ring, 3-coloring, the two-ring token ring) in which processes have a similar structure (in terms of their locality); i.e., *symmetric* processes. However, for some protocols (e.g. maximal matching in Section VI-A), STSyn has generated asymmetric SS versions. We are currently investigating several factors that affect the symmetry of the resulting protocol including the recovery schedule, variable domains and order of adding recovery transitions. We are also investigating the design of heuristics that enforce symmetry on the asymmetric protocols generated by the heuristic presented in Section V.

IX. Conclusions and Future Work

We presented a lightweight method for automated addition of convergence to non-stabilizing network protocols to make them Self-Stabilizing (SS), where a SS protocol recovers/converges to a set of legitimate states from *any* state in its state space (reached due to the occurrence of transient faults). The addition of convergence is a problem for which no polynomial-time algorithm is known yet, nor is there a proof of NP-completeness for it (though it is in NP). As a building block of our lightweight method, we presented a heuristic that automatically adds strong convergence to non-stabilizing protocols in polynomial time (in the state space of the non-stabilizing protocol). We also presented a sound and complete method for automated design of weak convergence (Theorem IV.1). While most existing manual/automatic methods for the addition of convergence mainly focus on locally-correctable protocols, our method automates the addition of convergence to non-locally-correctable protocols. We have implemented our method in a software tool, called STabilization Synthesizer (STSyn), using which we have automatically generated many stabilizing protocols including several versions of Dijkstra's token ring protocol [1], maximal matching, three coloring in a ring and a two-ring protocol. STSyn has generated alternative solutions and has facilitated the detection of a design flaw in a manually designed self-stabilizing protocol (i.e., the maximal matching protocol in [11]).

We are currently investigating extensions of our work in addition to what we mentioned in Section VIII. We will focus on identifying sufficient conditions (e.g., locally correctable protocols) for efficient addition of strong stabilization. We also will investigate the parallelization of our algorithms towards exploiting the computational resources of computer clusters for automated design of self-stabilization.

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

[3] G. Varghese, "Self-stabilization by local checking and correction," Ph.D. dissertation, MIT/LCS/TR-583, 1993.

[4] A. Arora, M. Gouda, and G. Varghese, "Constraint satisfaction as a basis for designing nonmasking fault-tolerant systems," *Journal of High Speed Networks*, vol. 5, no. 3, pp. 293–306, 1996, a preliminary version appeared at ICDCS'94.

[5] M. Gouda, "The triumph and tribulation of system stabilization," in *Distributed Algorithms, (9th WDAG'95)*, ser. Lecture Notes in Computer Science (LNCS), J.-M. Helary and M. Raynal, Eds. Le Mont-Saint-Michel, France: Springer-Verlag, Sep. 1995, vol. 972, pp. 1–18.

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

[7] A. Arora and M. G. Gouda, “Distributed reset,” *IEEE Transactions on Computers*, vol. 43, no. 9, pp. 1026–1038, 1994.

[8] M. Gouda, “Multiphase stabilization,” *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 201–208, 2002.

[9] W. Leal and A. Arora, “Scalable self-stabilization via composition,” in *IEEE International Conference on Distributed Computing Systems*, 2004, pp. 12–21.

[10] 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.

[11] M. G. Gouda and H. B. Acharya, “Nash equilibria in stabilizing systems,” in *11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, 2009, pp. 311–324.

[12] 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.

[13] 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.

[14] 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.

[15] L. Lamport and N. Lynch, *Handbook of Theoretical Computer Science: Chapter 18, Distributed Computing: Models and Methods*. Elsevier Science Publishers B. V., 1990.

[16] M. Nesterenko and A. Arora, “Stabilization-preserving atomicity refinement,” *Journal of Parallel and Distributed Computing*, vol. 62, no. 5, pp. 766–791, 2002.

[17] M. Demirbas and A. Arora, “Convergence refinement,” in *Proceedings of the 22nd International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, July 2002, pp. 589–597.

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

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

[20] A. Ebnenasir and A. Farahat, “Towards an extensible framework for automated design of self-stabilization,” Michigan Technological University, Tech. Rep. CS-TR-10-03, May 2010, http://www.cs.mtu.edu/html/tr/10/10-03.pdf.

[21] 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*, 2003, pp. 573–582.

[22] F. Somenzi, “CUDD: CU decision diagram package release 2.3. 0,” 1998.

[23] R. Bryant, “Graph-based algorithms for boolean function manipulation.” *IEEE Transactions On Computers*, vol. 35, no. 8, pp. 677–691, 1986.

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

[25] T. Cottenier, A. van den Berg, and T. Elrad, “Motorola weavr: Aspect and model-driven engineering,” *Journal of Object Technology*, vol. 6, no. 7, pp. 51–88, 2007.

[26] G. Brown, M. Gouda, and C.-L. Wu, “Token systems that self-stabilize,” *IEEE Transactions on Computers*, 1989.

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

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

[29] I. ling Yen and F. B. Bastani, “A highly safe self-stabilizing mutual exclusion algorithm,” in *Proceedings of the Second Workshop on Self-Stabilizing Systems*, 1995, pp. 301–305.

[30] S. Dolev and J. L. Welch, “Self-stabilizing clock synchronization in the presence of byzantine faults (abstract),” in *ACM symposium on Principles of Distributed Computing*, 1995, p. 256.

[31] J. Beauquier, S. Tixeuil, and A. K. Datta, “Self-stabilizing census with cut-through constraint,” *International Conference on Distributed Computing Systems*, vol. 0, pp. 70–77, 1999.

[32] S. Young and V. K. Garg, “Self-stabilizing machines: An approach to design of fault-tolerant systems,” in *32nd Conference on Decision and Control*, 1993, pp. 1200–1205.

[33] 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.

[34] A. Ebnenasir, S. S. Kulkarni, and A. Arora, “FTSyn: A framework for automatic synthesis of fault-tolerance,” *International Journal on Software Tools for Technology Transfer*, vol. 10, no. 5, pp. 455–471, 2008.

[35] B. Bonakdarpour and S. S. Kulkarni, “Revising distributed UNITY programs is np-complete,” in *12th International Conference on Principles of Distributed Systems (OPODIS)*, 2008, pp. 408–427.

[36] 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.