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

1New 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 \( \text{Rank}[1], \ldots, \text{Rank}[M] (M > 1) \) such that the length of the shortest execution from any state in \( \text{Rank}[i] \) to some state in \( I \) is equal to \( i \); i.e., each \( \text{Rank}[i] \) 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 correctly-constructed 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.

**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 \( \delta_p \) of transitions, a finite set \( \Pi_p \) 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 \( \langle d_1, d_2, \ldots, d_N \rangle \) of variables \( \langle v_1, v_2, \ldots, v_N \rangle \), 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 \( \delta \) 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 $P_i$ 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 $(x_1, x_2)$ denote a state of this protocol. Now, if $P_1$ writes $x_1$ in a transition $(0, 0), (1, 0)$, 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 $(0, 0), (1, 0)$ and $(0, 1), (1, 1)$ 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: \( \forall v: v \in r_j : (v(s_0) = v(s'_0)) \lor (v(s_1) = v(s'_1)) \) and \( \forall v: v \notin r_j : (v(s_0) = v(s_1)) \lor (v(s'_0) = v(s'_1)) \).

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 $P_j = \{g_{j1}, g_{j2}, \ldots, g_{jm}\}$ 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 $\delta_p$ 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 $\delta_p$ 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):

\[
A_0: \quad (x_0 = x_3) \quad \rightarrow \quad x_0 := x_3 + 1
\]

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 $1 \leq j \leq 3$:

\[
A_j: \quad (x_j + 1 = x_{(j-1)}) \quad \rightarrow \quad x_j := x_{(j-1)}
\]

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

\[
\begin{align*}
&((x_0 = x_1) \land (x_1 = x_2) \land (x_2 = x_3)) \lor \\
&((x_1 + 1 = x_0) \land (x_1 = x_2) \land (x_2 = x_3)) \lor \\
&((x_0 = x_1) \land (x_2 + 1 = x_1) \land (x_2 = x_3)) \lor \\
&((x_0 = x_1) \land (x_1 = x_2) \land (x_3 + 1 = x_2))
\end{align*}
\]

Let $(x_0, x_1, x_2, x_3)$ denote a state of TR. Then, the state $s_1 = (1, 0, 0, 0)$ 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. < Computation. Intuitively, a computation of a protocol $p = (V_p, \delta_p, \Pi_p, T_p)$ is an interleaving of its actions. Formally, a computation of $p$ is a sequence $\sigma = \langle s_0, s_1, \ldots \rangle$ of states that satisfies the following conditions: (1) for each transition $(s_i, s_{i+1})$ (i \geq 0) in $\sigma$, 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) $\sigma$ is maximal in that either $\sigma$ is infinite or if it is finite, then $\sigma$ reaches a state $s_f$ where no action is enabled. A computation prefix of a protocol $p$ is a finite sequence $\sigma = \langle s_0, s_1, \ldots, s_m \rangle$ of states, where $m \geq 0$, such that each transition $(s_i, s_{i+1})$ in $\sigma$ (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 $(V_p, \{\langle s_0, s_1 \rangle : (s_0, s_1) \in \delta_p \land s_0, s_1 \in X\}, \Pi_p, T_p)$. 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 $s \in (X \land grd)$ 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 $S_1$. < Convergence and self-stabilization. Let $I$ be a state predicate. We say that a protocol $p = (V_p, \delta_p, \Pi_p, T_p)$ 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 $\sigma = \langle s_i, s_{i+1}, \ldots, s_j, s_i \rangle$ be a sequence of states outside $I$, where $j \geq i$ and each state is reached from its predecessor by the transitions in $\delta_p$. The sequence $\sigma$ 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 $S_1$. <

### 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, $\delta_p \mid I$). 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 $L_s \in \{\text{weakly, strongly}\}$, 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) $\delta_{p_{ss}} \mid I = \delta_p \mid I$, and (3) $p_{ss}$ is $L_s$ converging to $I$. [1]

### 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 $s_d \notin I$ 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 moving 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 $p_{im} \mid I$ remains the same as $p \mid I$; 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 $\text{Rank}[1], \cdots, \text{Rank}[M]$, where $\text{Rank}[i] \subseteq \neg I$ and $\text{Rank}[i]$ 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, $\text{Rank}[i]$ 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 \( \text{Rank}[i] \). The repeat-unti loop in Figure 2 computes the set of backward reachable states from \( I \), denoted \( \text{explored} \), using the transitions of \( p_{im} \). In each iteration \( i \), Line 3 calculates a set of states \( \text{Rank}[i] \) outside \( \text{explored} \) from where some state in \( \text{explored} \) can be reached by a single transition of \( p_{im} \). The repeat-unti loop terminates when no more states can be added to \( \text{explored} \). The rank of a state in \( \neg I \) that does not belong to any rank is infinity, denoted \( \infty \). That is, if rank of \( s \) is \( \infty \), 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 \( \{ s \mid \text{rank of } s \text{ is } \infty \} = \emptyset \), then \( p_{im} \) is a weakly stabilizing version of \( p \). Otherwise, no stabilizing version of \( p \) exists. That is, \( \text{ComputeRanks} \) is sound and complete for finding a weakly stabilizing version of \( p \). (Proof in [20])

**Proof.** \( p_{ss} \) strongly converges to \( I \). Hence, every computation of \( p_{ss} \) that starts in \( \text{Rank}[i] \) has a prefix \( \sigma = (s_0, s_1, \ldots, s_f) \) 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 \( I \)), \( \sigma \) should include a transition from \( \text{Rank}[i] \) to \( \text{Rank}[i-1] \), a transition from \( \text{Rank}[i-1] \) to \( \text{Rank}[i-2] \), \ldots, and a transition from \( \text{Rank}[1] \) 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:

1. 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.);
2. Recovery transitions are added from each \( \text{Rank}[i] \) to \( \text{Rank}[i-1] \), for \( 1 \leq i \leq M \);
3. The groupmates of added recovery transitions must not form a cycle outside \( I \), and
4. 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 \( \text{Rank}[0..M] \), where \( \text{Rank}[i] \) denotes a state predicate containing states with rank \( i \).
   - Compute the deadlock states in \( \neg I \), denoted by the state predicate \( \text{deadlockStates} \).

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 \( \text{Rank}[i] \) and ends in a state in \( \text{Rank}[i-1] \) 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 \( \text{Rank}[i] \) and ends in a state in \( \text{Rank}[i - 1] \) 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 \( F \) from another state predicate \( T \). 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 \( I \), (iii) without including any transition group that is ruled out by the constraints of that pass, denoted \( \text{ruledOutTrans} \), and (iv) based on the recovery schedule given in the array \( \text{sch}[] \). An example recovery schedule for the TR program is \( \{ P_1, P_2, P_3, P_0 \} \); i.e., \( \text{sch}[1] = 1, \text{sch}[2] = 2, \text{sch}[3] = 3 \) and \( \text{sch}[4] = 0 \). That is, when adding recovery from a deadlock state \( s_d \), we first check the ability of \( P_i \) in including a recovery transition from \( s_d \) and \( P_2 \) 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 \( P_{\text{sch}[j]} \) can add recovery transitions from \( F \) to \( T \) using the transitions of \( p_{ss} \). This addition of recovery transitions is performed while adhering to read/write restrictions of \( P_{\text{sch}[j]} \) and excluding any transition in the set of transition groups \( \text{ruledOutTrans} \) (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 groupmante of the transitions of \( p_{ss} \). For this reason, we use the **Identify.Resolve.Cycles** routine (see Figure 3) in Line 2 of **Add_Recovery**.

The **Identify.Resolve.Cycles** 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 \( \text{badTrans} \) 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.

### Pass 1: Adding recovery from \( \text{Rank}[i] \) to \( \text{Rank}[i - 1] \) 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 := \( \{ s \mid s \in \text{Rank}[i] \land s \in \text{deadlockStates} \} \);
- To := \( \{ s \mid s \in \text{Rank}[i - 1] \} \);
- \( \text{passNo} := 1 \);
- \( \text{ruledOutTrans} := \{ (s_0, s_1) \mid (s_0, s_1) = (s_0, s_1) \land (s_1 \in \text{deadlockStates}) \} \);
- \( \text{deadlockStates}, p_{ss} := \text{Add_Convergence}(\text{From}, \text{To}, I, P_1, \ldots, P_K, \text{sch}[1..K], p_{ss}, \text{ruledOutTrans}, \text{passNo}) \);
- If \( (\text{deadlockStates} = \emptyset) \), 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.

### 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 = $\{ (s_0, s_1) | (s_0 \in I) \}$. 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].

### 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 = $\{ (s_0, s_1) | (s_0 \in I) \}$.

**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 synthesizes 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 $\{\text{left, right, self}\}$ 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 $m_i = \text{left}$ and $m_{(i-1)} = \text{right}$ (respectively, $m_i = \text{right}$ and $m_{(i+1)} = \text{left}$). 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 $m_i = \text{self}$. 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

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

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).

$$m_4 = \text{left} \land m_0 \neq \text{self} \land m_1 = \text{right} \quad \rightarrow \quad m_0 := \text{self}
$$

$$(m_0 = \text{self} \land m_4 = \text{right}) \lor
(m_0 \neq \text{left} \land m_1 \neq \text{self} \land m_4 = \text{right})
\quad \rightarrow \quad m_0 := \text{left}
$$

$$(m_0 = \text{self} \land m_1 = \text{left}) \lor
(m_0 \neq \text{right} \land m_1 = \text{left} \land m_4 = \text{left})
\quad \rightarrow \quad m_0 := \text{right}
$$

If the left neighbor of $P_0$ (i.e., $P_0$) 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 ($1 \leq i \leq K$):

$$m_i = \text{left} \land m_{i-1} = \text{left} \quad \rightarrow \quad m_i := \text{self}
$$

$$m_i = \text{right} \land m_{i+1} = \text{right} \quad \rightarrow \quad m_i := \text{self}
$$

$$m_i = \text{self} \land m_{i-1} \neq \text{left} \quad \rightarrow \quad m_i := \text{left}
$$

$$m_i = \text{self} \land m_{i+1} \neq \text{right} \quad \rightarrow \quad m_i := \text{right}
$$

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 $(\text{left}, \text{self}, \text{left}, \text{self}, \text{left})$ with a schedule $P_0, P_1, P_2, P_3, P_4$ repeated twice, where the tuple $(m_0, m_1, m_2, m_3, m_4)$ 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 $I_{\text{coloring}} = \forall i : 0 \leq i \leq K - 1 : c_{i-1} \neq c_i$ 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 $i$ ($1 < i < 40$).

$$P_1: (c_1 = c_0) \lor (c_1 = c_2) \quad \rightarrow \quad c_1 := \text{other}(c_0, c_2)
$$

$$P_i: (c_{i-1} \neq c_i) \land (c_i = c_{i+1}) \quad \rightarrow \quad c_i := \text{other}(c_{i-1}, c_{i+1})
$$

Notice that $P_0$ has no actions. The nondeterministic function $\text{other}(x, y)$ 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 (TR²) protocol. The TR² 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 value $a_i$ (respectively, $b_i$) with the domain ${0, 1, 2, 3}$.

![Figure 4. The Two-Ring Token Ring (TR²) protocol.](image)

Process $PA_i$, for $1 \leq i \leq 3$, has the token iff $(a_{i-1} = a_i + 1)$, 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 $(a_0 = a_3) \land (b_0 = b_3) \land (a_0 = b_0)$; 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 $(b_0 = b_3) \land (a_0 = a_3) \land ((b_0 \oplus 1) = a_0)$. 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 $(b_{i-1} = b_i + 1)$. The TR² protocol also has a Boolean variable $\text{turn}$; ring A executes only if $\text{turn} = \text{true}$, and if ring B executes then $\text{turn} = \text{false}$.

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>
<thead>
<tr>
<th>Case Study</th>
<th>Locally Correctable</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-Coloring</td>
<td>Yes</td>
</tr>
<tr>
<td>Matching</td>
<td>No</td>
</tr>
<tr>
<td>Token Ring (TR)</td>
<td>No</td>
</tr>
<tr>
<td>Two-Ring TR</td>
<td>No</td>
</tr>
</tbody>
</table>

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 develop-
ment. 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 com-
plexity 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 $I_{\text{coloring}}$, 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
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,
$m_i = \text{left}$ and $m_{i+1} \neq \text{right}$. 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 $m_{i+1} \neq \text{left}$. The last option for $P_i$ would be to set $m_i$ to right,
which may not make the second conjunct true if $m_{i+1} \neq \text{left}$. 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 $I_{\text{MM}}$,
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 com-
plexity 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-
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 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**


