Towards an Extensible Framework for Automated Design of Self-Stabilization
Ali Ebnenasir and Aly Nour Farahat

Michigan Technological University
Computer Science Technical Report
CS-TR-10-03
May 2010

Department of Computer Science
Houghton, MI 49931-1295
www.cs.mtu.edu
Towards an Extensible Framework for Automated Design of Self-Stabilization

Ali Ebnenasir and Aly Nour Farahat

May 2010

Abstract

This paper proposes a framework for automatic design of (finite-state) self-stabilizing programs from their non-stabilizing versions. A method that automatically adds stabilization to non-stabilizing program is highly desirable as it generates self-stabilizing programs that are correct by construction, thereby eliminating the need for after-the-fact verification. Moreover, automated design methods enable separation of stabilization from functional concerns. In this paper, we first present a deterministically sound and complete algorithm that adds weak stabilization to programs in polynomial time (in the state space of the non-stabilizing program). We also present a sound method that automatically adds strong stabilization to non-stabilizing programs using the results of the proposed weak stabilization algorithm. These two methods constitute the first elements of the proposed extensible framework. To demonstrate the applicability of our algorithms in automated design of self-stabilization, we have implemented our algorithms in a software tool using which we have synthesized many self-stabilizing programs including Dijkstra’s token ring, graph coloring and maximal matching. While some of the automatically generated self-stabilizing programs are the same as their manually designed versions, our tool surprisingly has synthesized programs that represent alternative solutions for the same problem. Moreover, our algorithms have helped us reveal design flaws in manually designed self-stabilizing programs (claimed to be correct).
1 Introduction

This paper proposes an extensible repository of automated techniques for designing self-stabilization. The motivation behind such a repository is multi-fold. First, since manual design and verification of self-stabilization are known to be difficult tasks [1–4], it is desirable to facilitate these tasks by automated techniques and tools, thereby eliminating the need for after-the-fact verification. (Gouda [3] observes that verifying self-stabilization often takes more time than designing it almost by a factor of ten.) Second, such a repository provides a set of reusable design techniques/patterns for the research and engineering community. Third, the automated techniques can potentially be integrated in compilers for addition of self-stabilization for cases where such transformations are feasible [2]. Finally, since platform-specific constraints (such as fairness policy of the underlying scheduler, execution semantics, atomicity, etc.) impact the complexity of design, such a repository will enable designers to investigate self-stabilization under different assumptions.

Several general techniques for the design of self-stabilization exist [5–13], most of which provide problem-specific solutions [14–22] and lack the necessary tool support for automatic addition of self-stabilization to non-stabilizing systems. For example, Katz and Perry [7] 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 if necessary. Arora et al. [9, 23] provide a method based on constraint satisfaction, where they create a dependency graph of the local constraints whose satisfaction guarantees recovery of the entire system. Varghese [8] and Afek et al. [12] put forward a method based on checking local conditions towards providing global recovery. Nonetheless, this approach is only applicable to a family of systems whose set of legitimate states can be specified as the conjunction of a set of local conditions, known as locally checkable systems. Moreover, even if a program is locally checkable, it may not be locally correctable; i.e., even though a process can locally detect that the program is in an illegitimate global state, the correctness of its corrective actions depend on the state and the actions of its neighbors. (See the maximal matching program in Section 6.2 as an example of a locally checkable program that is not locally correctable.) To design non-locally checkable systems, Varghese [10] proposes a design technique based on counter flushing, where a leader node systematically increments and flushes the value of a counter throughout the network. Layering and modularization [5, 6, 11, 13] are also techniques that enable the design of self-stabilization by incremental construction of convergence using either strictly decreasing [1, 24, 25] or non-increasing ranking functions [26]. Nonetheless, these methods fail if convergence from an illegitimate state can only be achieved by oscillating for a finite number of times before converging to the set of legitimate states.

In this paper, we present an algorithmic approach, supported by a software tool called STabilization Synthesizer (STSyn), for systematic exploration of the computational structure of non-stabilizing programs towards generating their stabilizing versions. In particular, we start with a program $p$ and a state predicate $I$ representing a set of legitimate states from where computations of $p$ satisfy its specification, denoted spec. Starting from $I$, every computation of $p$ remains in $I$; i.e., strong closure [27].$^1$ Subsequently, we systematically include transitions in the non-stabilizing program to ensure two types of convergence, namely weak and strong convergences (defined in [3, 28]). Weak convergence to $I$ requires that from any state in the state space of $p$, there exists a program computation that reaches a state in $I$, whereas strong convergence stipulates that, from any state, every computation reaches a state in $I$. We present a deterministically sound and complete algorithm that incrementally builds reachability paths to $I$ from outside $I$, thereby ensuring weak convergence. Nonetheless, a weakly stabilizing system guarantees stabilization only on a strongly fair scheduler, where strong fairness guarantees that any process that is enabled infinitely often will be executed infinitely often. Previous work illustrates that, under the interleaving semantics, the design of maximally strong fair schedulers on distributed platforms is hard and in some cases rather impossible [29].$^2$ While there are self-stabilizing scheduling algorithms that provide strong fairness to applications [30], these methods are not maximal and have a high overhead. As such, it is desirable to design strongly stabilizing systems as they stabilize on any scheduler [28]; i.e., portability. However, designing strong convergence is known to be a hard problem [31] as one has to ensure progress in addition to reachability, where progress requires that every program computation has a state in $I$. A major challenge in the design of strong convergence is the resolution of non-progress cycles in $\neg I$, where a non-progress cycle comprises a sequence $\sigma = (s_i, s_{i+1}, \ldots, s_j, s_i)$ of

---

$^1$ Weak closure [28] requires that every computation starting in $I$ has a non-empty suffix that remains in $I$.

$^2$ A maximal scheduler generates all schedules that are permissible under its fairness policy.
states \((j \geq i)\) in which each state is reached from its predecessor by a program transition and no state in \(i\) is in \(I\). To design strong convergence, we present a sound heuristic that systematically explores the possibility of synthesizing reachability to \(I\) without creating non-progress cycles. The time complexity of the proposed heuristic is polynomial (in the state space of the non-stabilizing program). To evaluate the applicability of our heuristic, we have implemented it as the first tool of the STSyn framework. Using STSyn, we have automatically designed several strong self-stabilizing programs such as Dijkstra's token ring (3 different versions), matching on a ring [32], 3 coloring of a ring, equalizing over a ring and delay-insensitive self-stabilization (available at http://cs.mtu.edu/~anfaraha/CaseStudiesExamples). More importantly, STSyn has helped us uncover a non-progress cycle in a manually designed maximal matching program presented in [32].

**Organization.** Section 2 presents the preliminary concepts. Section 3 formulates a general problem of adding self-stabilization to non-stabilizing programs. Section 4 presents a sound and complete polynomial-time algorithm for automated design of weak stabilizing programs. We use the results of the algorithm for the addition of weak stabilization as an approximation for automated design of strong stabilization in Section 5. Section 6 demonstrates the addition of strong stabilization in the context of a token ring, a maximal matching and a three coloring program. Subsequently, Section 7 presents our experimental results, and then we make concluding remarks and discuss future work in Section 8.

## 2 Preliminaries

In this section, we present the formal definitions of programs, the read/write model and self-stabilization. The programs are defined in terms of their set of variables, their transitions and their processes/components. We adapt our read/write model from [33,34]. The definitions of self-stabilization is adapted from [1,3,27,28]. To simplify our presentation, we use a simplified version of Dijkstra's token ring program [1] as a running example.

### Programs. A program \(p = (V_p, \delta_p, \Pi_p)\) is a tuple of a finite set \(V_p\) of variables, a set of transitions \(\delta_p\) and a finite set \(\Pi_p\) of \(K\) processes, where \(K \geq 1\). 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 \((d_1, d_2, \cdots, d_N)\) of program variables \((v_1, v_2, \cdots, v_N)\), 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 state of \(t\). A process \(P_j\) \((1 \leq j \leq K)\) includes a set of transitions \(\delta_j\). The set \(\delta_p\) of program transitions is equal to the union of the transitions of its processes; i.e., \(\delta_p = \cup_{j=1}^{K} \delta_j\). For a variable \(v\) and a state \(s\), \(v(s)\) denotes the value of \(v\) in \(s\). The state space \(S_p\) is the set of all possible states of \(p\), and \(|S_p|\) denotes the size of the state space.

### Notation. When it is clear from the context, we use \(p\) and \(\delta_p\) interchangeably.

**Example: Token Ring (TR).** The Token Ring (TR) program (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 [35] as a shorthand for representing the set of program transitions. A guarded command (action) is of the form \(grd \rightarrow stmt\), where \(grd\) is a Boolean expression in terms of variables in \(V_p\) and \(stmt\) is a statement that may update program variables atomically. Formally, a guarded command \(grd \rightarrow stmt\) includes all program transitions \(\{s_0, s_1\} : grd\ holder at s_0\) and the atomic action of \(stmt\) at \(s_0\) only if it takes the program to state \(s_1\). We represent the new values of updated variables as primed values. For example, if an action updates the value of an integer variable \(v\) from 0 to 1, then we have \(v = 0\) and \(v' = 1\). The process \(P_0\) has the following action: (addition and subtraction are in modulo 3)

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

When the values of \(x_0\) and \(x_3\) are equal, \(P_0\) increments \(x_0\) by one. Since the actions of processes \(P_j\), for \(1 \leq j \leq 3\) are symmetric, we use the following parametric action to represent them.

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

Each process \(P_j\) \((1 \leq j \leq 3)\) increments \(x_j\) only if \(x_j\) is one unit less than \(x_{j-1}\).

### State and transition predicates. A state predicate of \(p\) is any subset of \(S_p\) specified as a Boolean expression over \(V_p\). We say a state predicate \(X\) holds at a state \(s\) (respectively, \(s \in X\)) if and only if \((X)\) \(X\) evaluates to true at \(s\). An unprimed state predicate is specified only in terms of unprimed variables.
Likewise, a primed state predicate includes only primed variables. We define a function $\text{Primed}$ (respectively, $\text{UnPrimed}$) that takes a state predicate $X$ (respectively, $X'$) and substitutes each variable in $X$ (respectively, $X'$) with its primed (respectively, unprimed) version, thereby returning a state predicate $X'$ (respectively, $X$). A transition predicate (adapted from [36,37]) is a subset of $S_p \times S_p$ represented as a Boolean expression over both unprimed and/or primed variables. The function $\text{getPrimed}$ (respectively, $\text{getUnPrimed}$) takes a transition predicate $T$ and returns a primed (respectively, an unprimed) state predicate representing the set of destination (respectively, source) states of all transitions in $T$. Note that, a state predicate $X$ also represents two transition predicates; one includes all transitions $(s, s')$, where $s \in X$ and $s'$ is an arbitrary state, and the other includes transitions $(s, s')$, where $s$ is an arbitrary state and $s'$ is in $\text{Primed}(X)$. An action $grd \rightarrow stmt$ is enabled at a state $s$ iff $grd$ holds at $s$. A process $P_j \in I_p$ is enabled at $s$ iff there exists an action of $P_j$ that is enabled at $s$.

**TR Example.** By definition, process $P_j$, $j = 1, 2, 3$, 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 in the TR program, where $S_1$ is

$$
((x_0 = x_1) \land (x_1 = x_2) \land (x_2 = x_3)) \lor ((x_0 + 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))
$$

Let $\langle x_0, x_1, x_2, x_3 \rangle$ denote a state of TR. Then, the states $s_1 = (0,1,1,1)$ and $s_2 = (1,1,1,2)$ belong to $S_1$, where $P_1$ has a token in $s_1$ and $P_3$ has a token in $s_2$.

**Computations and execution semantics.** We consider a nondeterministic interleaving of all program actions generating a sequence of states in the serial execution semantics. That is, enabled actions are executed one at a time. A computation of a program $p = \langle V_p, \delta_p, I_p \rangle$ 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 $\sigma$, there exists an action $grd \rightarrow stmt$ in some process $P_j \in I_p$ 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 no program action is enabled in its final state. A state that has no outgoing transitions is called a deadlock state. The final state of a deadlocked computation is a deadlock state. To distinguish between valid terminating computations and deadlocked computations, we stutter at the final state of valid terminating computations. A computation prefix of a program $p$ is a finite sequence $\sigma = \langle s_0, s_1, \cdots, s_k \rangle$ of states, where $k \geq 0$, such that each transition $(s_i, s_{i+1})$ in $\sigma$ ($0 \leq i < k$) belongs to some action $grd \rightarrow stmt$ in some process $P_j \in I_p$. The projection of a program $p$ on a non-empty state predicate $S$, denoted as $p|S$, is the program $\langle V_p, \{ (s_0, s_1) : (s_0, s_1) \in \delta_p \land s_0, s_1 \in S \}, I_p \rangle$. In other words, $p|S$ consists of transitions of $p$ that start in $S$ and end in $S$.

**Closure.** A state predicate $X$ is closed in an action $grd \rightarrow stmt$ iff executing $stmt$ from a state $s \in (X \land grd)$ results in a state in $X$. We say a state predicate $X$ is strongly closed in a program $p$ iff $X$ is closed in every action of $p$. A state predicate $X$ is weakly closed in a program $p$ if eventually $X$ becomes closed in every action of $p$. In other words, weak closure [28] requires that every computation starting in $I$ has a non-empty suffix that remains in $I$, whereas strong closure states that every computation that starts in $I$, remains in $I$.

**TR Example.** Starting from a state in the state predicate $S_1$, the TR program generates an infinite sequence of states (i.e., a computation), where all reached states belong to $S_1$.

**Read/Write model.** In order to capture distribution issues, for each process $P_j \in I_p$ of a program $p = \langle V_p, \delta_p, I_p \rangle$, we consider a subset of variables in $V_p$ that $P_j$ 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. No action in a process $P_j$ is allowed to update a variable $v \notin w_j$. Thus, the transition predicate $wRest_j \equiv \langle v : v \notin w_j \wedge v = v' \rangle$ captures the write restrictions of $P_j$; i.e., the value of all variables that action $P_j$ cannot write remain unchanged. Note that $w_j$ excludes any unreadable variable; i.e., the transition predicate $wRest_j$ captures the fact that a variable that cannot be read, cannot be written either.

A major impact of read restrictions is that every transition of a process $P_j$ is in fact a group of transitions due to the inability of that process 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 program. Now, if $P_1$ writes $x_1$ in a transition $\langle (0,0), (1,0) \rangle$, then since $P_1$ cannot read $x_2$, $P_1$ has to consider
the possibility of $x_2$ being 1 when it updates the value of $x_1$ 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: $\forall v : v \in r_j : (v(s_0) = v(s'_0)) \land (v(s_1) = v(s'_1))$ and $\forall v : v \notin r_j : (v(s_0) = v(s_1)) \land (v(s'_0) = v(s'_1))$.

We consider a function $\text{Group}(r_j, \text{transPred})$ that returns the groups of transitions of $P_j$ corresponding to each transition in the predicate $\text{transPred}$. We formally specify an action $\text{grd} \rightarrow \text{stmt}$ of $P_j$ as a transition predicate $\text{Group}(r_j, \text{grd} \land \text{stmtPred} \land w\text{Rest}_j)$, where $\text{stmtPred}$ is a transition predicate generated from the assignment $\text{stmt}$. For example, an assignment $x := y + 1$ can be specified as the transition predicate $x' = y + 1$, where $x$ and $y$ are two variables and the predicate $x' = y + 1$ includes all transitions $(s_0, s_1)$ that meet the constraint $x(s_1) = y(s_0) + 1$.

**TR Example.** 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 $P_i$ is unable to read two $x$ values (each with a domain of three values), each group associated with an action $A_j$ includes nine transitions. Notice that, for a program with $n$ processes, each group includes $3^{n-2}$ transitions.

**Convergence and self-stabilization.** Let $I$ be a state predicate. We say that a program $p = \langle \text{V}_p, \delta_p, \Pi_p \rangle$ strongly converges to $I$ iff from any state, every computation of $p$ reaches a state in $I$. A program $p$ weakly converges to $I$ if from any state, there exists a computation of $p$ that reaches a state in $I$. A program $p$ is strongly (respectively, weakly) self-stabilizing to a state predicate $I$ iff (1) $I$ is strongly closed in $p$ and (2) $p$ strongly (respectively, weakly) converges to $I$.

**TR Example.** If the TR program 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 program is neither weakly stabilizing nor strongly stabilizing to $I$.

**Fairness/Scheduling Policy.** The way processes of a program are given a chance of execution determines the fairness assumption, which affects the correctness of self-stabilization [4]. Consider a transition $(s, s')$ and a computation $\sigma(s_0, s_1, \ldots)$ of a program $p$. The computation $\sigma$ is strongly fair iff for every transition $(s, s')$ of $p$, if $s$ occurs infinitely often in $\sigma$, then $(s, s')$ will infinitely often be executed in $\sigma$ (definition adapted from [28]). For an algorithm that adds stabilization to non-stabilizing programs, the fairness assumption would make a significant difference [28] in its complexity. For instance, fairness determines which cycles should be resolved. Consider a program $p$ that should converge to a closed predicate $I$. Moreover, consider that $p$ is perturbed to a cycle outside $I$ that consists of two states $s_0$ and $s_1$, where a process $P_j$ executes the transition $(s_0, s_1)$ and $P_k$ executes $(s_1, s_0)$. If there is a third process $P_m$ that includes a transition $(s_0, s_2)$, then $P_m$ is infinitely often enabled. As such, if $s_2$ is different from $s_0$ and $s_1$, then a strongly fair scheduler will make it possible to recover from the cycle. By contrast, with no fairness assumptions, the cycle should be resolved so convergence to $I$ can be achieved.

### 3 Problem Statement

Consider a non-stabilizing program $p = \langle \text{V}_p, \delta_p, \Pi_p \rangle$ and a state predicate $I$, where $I$ is strongly closed in $p$. Our objective is to generate a (weakly/strongly) stabilizing version of $p$, denoted $p_s$. To separate stabilization property from functional concerns, we require that no states (respectively, transitions) are added to or removed from $I$ (respectively, $p(I)$). The motivation behind this separation of concerns is to ensure that if in the absence of transient faults, $p$ meets its specification, then $p_s$ will preserve the correctness of $p$ in the absence of faults, and only self-stabilization is added to $p$ to ensure convergence to $I$ in the presence of faults. This is a specific instance of a more general problem, namely Problem 3.1, that we are currently investigating. (Problem 3.1 is an adaptation of the problem of adding fault tolerance in [33].)

**Problem 3.1: Adding Self-Stabilization**

- **Input:**
  - A program $p = \langle \text{V}_p, \delta_p, \Pi_p \rangle$,
  - A state predicate $I$ such that $I$ is $L_c$ closed in $p$, where $L_c \in \{\text{weakly, strongly}\}$,
– A property of $L_s$ stabilizing, where $L_s \in \{\text{weakly, strongly}\}$.
– An atomicity model captured by read/write restrictions (with respect to $V_p$) representing the underlying communication topology of the processes in $\Pi_p$, and
– An execution semantics $E \in \{\text{interleaving, concurrent}\}$.

- **Output:** A program $p_s = (V_p, \delta_{p_s}, \Pi_p)$
- **Constraints:**

  1. $I$ is unchanged.
  2. $\delta_{p_s}|I = \delta_p|I$.
  3. $p_s$ is $L_s$ stabilizing to $I$ with the execution semantics $E$.

In this paper, we shall investigate Problem 3.1 for two cases where $L_c = \text{strongly}$ and $E = \text{interleaving}$. Specifically, Section 4 investigates the case where $L_s = \text{weakly}$, and Section 5 presents a solution for the case where $L_s = \text{strongly}$.

**Comment.** While in this paper we focus on the case where the state space of $p_s$ is the same as that of $p$, new variables can be introduced manually to the non-stabilizing program to generate an input instance of Problem 3.1. We are currently investigating automated techniques [38] where, if necessary, new variables will be introduced automatically during algorithmic addition of self-stabilization.

### 4 Algorithmic Design of Weak Stabilization

In this section, we present a (deterministically) sound and complete algorithm for adding weak stabilization to programs. In Section 5, we illustrate how we use the synthesized weakly stabilizing programs as approximations that guide the design of strong stabilization. Figure 1 demonstrates the algorithm $\text{Add Weak Stabilization}$ that takes a program $p$, a state predicate $I$ that is strongly closed in $p$ and the read/write restrictions of the processes of $p$. $\text{Add Weak Stabilization}$ generates a program $p_s$ that is weakly self-stabilizing to $I$ while preserving the computations of $p$ starting in $I$. Specifically, in the for loop (Line 2) in Figure 1, $\text{Add Weak Stabilization}$ computes the program $p_s$ as the disjunction of all possible transitions starting in $\neg I$ and adhering to the read/write restrictions of each process. As such, we guarantee that $p|I$ remains unchanged and the closure of $I$ in $p$ is preserved. The purpose of the repeat-until loop (Lines 3-6) is to compute the set $\text{explored}$ of backward reachable states from $I$ using the transitions of $p_s$. In Line 3, the transition predicate $\text{convergTransPred}$ captures the transitions of $p_s$ whose destination states are in $\text{explored}$. Line 4 computes $\text{Rank}[i]$ ($i$ is the loop index) as the set of states in $\neg \text{explored}$ from where $\text{convergTransPred}$ reaches $\text{explored}$. In other words, $\text{Rank}[i]$ denotes the set of states from where $I$ is reachable in $i$ steps using the transitions of $p_s$. The repeat-until loop terminates when there are no more states to be added to $\text{explored}$; i.e., $\text{Rank}[i - 1] = \emptyset$. Line 7 checks whether $\text{explored}$ contains the entire state space ($\text{true}$). If so, then $p_s$ is returned as a weakly self stabilizing version of $p$. Otherwise, $\text{Add Weak Stabilization}$ declares the non-existence of a solution.

**Theorem 4.1** The algorithm $\text{Add Weak Stabilization}$ is sound.

**Proof.** We illustrate that, if $\text{Add Weak Stabilization}$ returns a program $p_s$, then $p_s$ satisfies the constraints of Problem 3.1. In the for loop in Line 2, $p_s$ includes only the transitions that start outside ($\neg I$). Hence, $I$ remains unchanged and $p|I = p_s|I$, which ensures the closure of $I$ in $p_s$.

At each iteration $i$ of the repeat-until loop (Lines 3-6), $\text{explored}$ includes the set of states from where $I$ is reachable in at most $i$ steps using the transitions of $p_s$. In Line 5, $\text{explored}$ is updated such that in the subsequent iteration, $\text{Rank}[i]$ is excluded from the set of states to explore. Since $S_p$ has a finite size, the repeat-until loop (Lines 3-6) eventually terminates. By assumption, $\text{Add Weak Stabilization}$ returns a program. Thus, $\text{explored}$ is equal to $\text{true}$. Hence, $p_s$ has a computation prefix to $I$ from every state in $S_p$. Therefore, $p_s$ is weakly self-stabilizing.

**Theorem 4.2** The output of $\text{Add Weak Stabilization}$ is maximal.
Weaken stabilization

Theorem 4.3 The algorithm Add\_WeakStabilization is complete.

Proof. Consider a program \( p \), its given read/write restrictions and a state predicate \( I \) that is closed in \( p \). Assume that Add\_WeakStabilization fails to generate a weakly stabilizing version of \( p \), denoted \( p^* \). Moreover, assume that a weakly stabilizing version of \( p \), denoted \( p'_s \), exists, where \( p'_s \) meets the constraints of Problem 3.1. Thus, there must exist a set of states in \( \sim I \) from where \( I \) is reachable by transitions that adhere to read/write restrictions of \( p \), but Add\_WeakStabilization failed to find such transitions. This is a contradiction with Theorem 4.2. Therefore, Add\_WeakStabilization would have found a weakly stabilizing program.

Theorem 4.4 The time complexity of Add\_WeakStabilization is polynomial in \(|S_p|\).

Proof. The for loop in Line 2 iterates at most \( K \) times (\( K \) is a constant equal to the number of processes). The number of iterations of the for loop in Lines 3-6 is at most equal to the number of states in \( \sim I \). Therefore, the time complexity of Add\_WeakStabilization is polynomial in \(|S_p|\).

5 Algorithmic Design of Strong Stabilization

In this section, we present a sound heuristic for adding strong stabilization in polynomial time (in the state space of the non-stabilizing program). Consider a program \( p = (V_p, \delta_p, \Pi_p) \), a state predicate \( I \) that is closed in \( p \) and the read/write restrictions of the processes \( \{P_1, \ldots, P_K\} \) of \( p \). First, we invoke the Add\_WeakStabilization algorithm presented in Section 4 to check whether a weakly stabilizing version of \( p \) exists. If so, then the Add\_StrongStabilization heuristic (see Figure 2) uses the ranks generated by Add\_WeakStabilization to systematically synthesize strong stabilization. We first consider some temporary variables for representing \( p \) and its intermediate version during synthesis, denoted intermProgs (see Line 1 in Figure 2). Then, we check whether or not the transitions of \( p \) form any cycles outside \( I \), i.e., non-progress cycles (see Line 2 in Figure 2). If so, Add\_StrongStabilization declares failure in designing a stabilizing program and exits. This is because resolving those cycles would result in changing \( p/I \), which violates the second constraint of Problem 3.1. Since we implement Add\_StrongStabilization using symbolic data structures (such as Binary Decision Diagrams [39]), we reuse a symbolic cycle detection algorithm with linear time complexity due to Gentilini et al. [40]. The Detect\_SCC routine in Line 2 of Figure 2 (implementing Gentilini et al.’s algorithm) returns an array of state predicates, denoted SCCs, where each array cell contains the states of a Strongly Connected Component (SCC) in the state transition graph of \( p \) in \( \sim I \). Detect\_SCC also returns the number of SCCs, which if it is non-zero, then Add\_StrongStabilization terminates.

Add\_StrongStabilization includes three rounds (see Line 5 in Figure 2) for designing strong convergence, where four constraints of adding recovery transitions are relaxed in each round. Specifically, Add\_StrongStabilization explores the possibility of adding recovery transitions under these constraints: (1) no transition that is grouped with a transition originating in \( I \) can be used for recovery (Line 6 in Figure 2).
Add_StrongStabilization($\{P_1, \ldots, P_k\}$: transition predicate; $I$: state predicate, $sch[1..K]$: integer array; $Rank[1..M]$: array of state predicates, \{wRest[1], \ldots, wRest[K]\}: write restrictions; \{r_1, \ldots, r_K\}: read restrictions)

\{ // sch is an array representing a preferred schedule based on which */
/* processes are in the design of convergence. */
- $p := \bigvee_{j=1}^{K} P_j$; \hspace{2cm} (1)
- intermProg := $p$;
- SCCs, numOfSCCs := Detect_SCC($p$, \neg $I$); // SCCs is an array of SCCs (2)
- if (numOfSCCs $> 0$) then declare that
  failed to add strong self-stabilization to $p$; exit; \hspace{2cm} (3)
- deadlockStates := \neg getUnPrimed($p$) \land \neg $I$; \hspace{2cm} (4)
- for $Round := 1$ to 3 do \{ // go through 3 rounds \hspace{2cm} (5)
  - if ($Round = 1$) then ruledOutTrans := $I$ \lor Primed(deadlockStates); \hspace{2cm} (6)
  - else ruledOutTrans := $I$; \hspace{2cm} (7)
  - for $i := 1$ to $M$ do \{ // go through $M$ ranks \hspace{2cm} (8)
    - if ($Round \neq 3$) then From := $Rank[i]$ \land deadlockStates; To := $Rank[i] - 1$; \hspace{2cm} (9)
    - else From := deadlockStates; To := true;
    - for $j := 1$ to $K$ do \{ // use the schedule in array $sch$ for recovery \hspace{2cm} (10)
      - intermProg := Add_Recovery(From, To, wRest,sch[j]; r,sch[j]; $P_{sch[j]}$); \hspace{2cm} (11)
      - deadlockStates := \neg getUnPrimed(intermProg) \land \neg $I$; \hspace{2cm} (12)
      - if (deadlockStates = $\emptyset$) then return intermProg;
    - } \hspace{2cm} (13)
  - if ($Round = 3$) then break; \hspace{2cm} (14)
- declare that failed to add strong self-stabilization to $p$; \hspace{2cm} (15)
\}

Figure 2: Adding strong stabilization.

(Recall that, due to read restrictions, all transitions in a group must be either included or excluded.); (2) recovery transitions are added from each $Rank[i]$ to $Rank[i - 1]$, where $1 \leq i \leq M$ and $Rank[0] = I$ (see the for loop in Line 7; see also Line 8 in Figure 2); (3) no two transitions grouped with two recovery transitions form a cycle outside $I$, and (4) no transition grouped with a recovery transition reaches a deadlock state (Line 6 in Figure 2).

In the first round, Add_StrongStabilization explores the possibility of adding recovery from deadlock states in $\neg I$ to non-deadlock states under the aforementioned constraints. The routine Add_Recovery (see Line 10 of Figure 2 and Figure 3) investigates whether or not a process $P_j$ can include new transitions that add recovery from the states in a state predicate From to another state predicate To. No recovery transition should have a groupmate ruled out by the transition predicate ruledOutTrans. The transition predicate resolvedDeadlocks$_j$ denotes such transitions (Line 1 in Figure 3). We use an array $sch[]$ to specify the order based on which the ability of processes in adding recovery is investigated in the for loop in Line 9 of Figure 2. We call this order the recovery schedule.

The Add_Recovery routine verifies whether or not the new recovery transitions form non-progress cycles with the transitions of the intermediate program and/or their groupmates. Towards this end, Add_Recovery uses the Identify_Resolving_Cycles routine (Line 2 in Figure 3 and Figure 4), which first invokes Detect_SCC to determine if there are cycles in $\neg I$. The for loop in Step 3 of Figure 4 removes the newly added recovery transitions (along with their group mates) that have a groupmate starting and ending in a SCC. This way, we ensure that the remaining recovery transitions do not create non-progress cycles in $\neg I$.

TR Example. For the TR example introduced in Section 2, the state predicate $I$ is equal to $S_1$ (defined in Section 2). Add_WeakStabilization computes two ranks ($M = 2$). That is, in the weakly stabilizing version of TR, $S_1$ is reachable from any state in at most 2 steps. The non-stabilizing TR program does not have any non-progress cycles in $\neg S_1$. The recovery schedule to Add_StrongStabilization is $P_1$, $P_2$, $P_3$, $P_0$. That is, in the for loop in Line 9 of Figure 2, we first investigate the actions of $P_1$ for adding recovery, and subsequently use the actions of $P_2$, $P_3$ and $P_0$ in order. We have observed that the recovery schedule has an impact on the success of synthesis as a schedule may create less number of non-progress cycles with respect to other schedules. For the TR program, Add_StrongStabilization could not add any recovery transitions in the first round as the groups that do not terminate in deadlock states cause cycles. For example, the recovery action $x_3 = x_0 + 1 \rightarrow x_0 := x_3$ for $P_0$ and the recovery action $x_j = x_{j-1} + 1 \rightarrow x_j := x_{j-1} - 1$ for process $P_j$
participate in non-progress cycles, where $1 \leq j \leq 3$ and addition and subtraction are performed in modulo 3. For instance, the recovery action added for $P_1$ has transitions in a 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. Therefore, no new transitions are included in the TR program during the first round.

In the second round, AddStrongStabilization explores the possibility of adding recovery from deadlock states to any state while adhering to the ranking constraints. That is, the rank of the source state of each recovery transition must be one unit more than the rank of its destination state; i.e., constructing a strictly decreasing ranking. In the third round, we remove the ranking constraint as well. Nonetheless, the first and the third constraints must be met in all three rounds. Round 3 is an important contribution of our work as it enables AddStrongStabilization to explore other possibilities for self-stabilization than just designing strictly decreasing computation prefixes that reach $I$.

**TR Example.** In the second round, AddStrongStabilization adds 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. Notice that no new transitions are included in $P_0$. The union of the added recovery action and the action $A_j$ in the non-stabilizing program results in the action $x_j \neq x_{j-1} \rightarrow x_j := x_{j-1}$ for the stabilizing TR program. The synthesized TR program is exactly the same as Dijkstra’s token ring program in [1].

**Theorem 5.1** AddStrongStabilization is sound.

**Proof.** In Step 6, we ensure that no transition originating in $I$ will be included. Moreover, AddStrongStabilization adds new recovery transitions; it does not remove any transitions. Thus, throughout the execution of AddStrongStabilization, $I$ remains unchanged and $\text{intermProg}|I = p|I$. Hence, the first two constraints of Problem 3.1 are met by $p_x$. The only step where AddStrongStabilization exits successfully is Step 12, where it returns $\text{intermProg}$ when no more deadlock states exist. Now, by contradiction, consider a computation $\sigma = \langle s_0, s_1, \cdots \rangle$, where $\forall j : j \geq 0 : s_j \in \neg I$. Since the state space of $p$ is finite, there must be some state $s_i$ that is revisited in $\sigma$; i.e., a non-progress cycle in $\neg I$. Nonetheless, the IdentifyResolveCycles routine ensures that no cycles are formed in $\neg I$ every time a recovery action is added by AddRecovery in Line 10 of Figure 2. Thus, the computation $\sigma$ must include a state in $I$. Therefore, the returned program $\text{intermProg}$ is strongly stabilizing to $I$. \hfill \Box

**Theorem 5.2** The time complexity of AddStrongStabilization is polynomial in $|S_p|$.  

**Proof.** AddStrongStabilization comprises three nested for loops. The outer loop in Line 5 iterates three times. The first inner loop in Line 7 iterates $M$ times, where $M$ is the number of ranks. The second inner loop iterates $K$ times, where $K$ is the number of processes. The AddRecovery routine takes at most a linear time in the size of $S_p$ [40]. In the worst case, each rank would include a single state, and hence $M$ would be in the order of $|S_p|$. Therefore, the time complexity of AddStrongStabilization is at most quadratic in $|S_p|$. \hfill \Box

### 6 Case Studies

In this section, we present some of our case studies for the addition of strong stabilization. The strongly stabilizing programs presented in this section have been automatically generated by STSyn.
6.1 Token Ring: Alternative Ways for Stabilization

We presented a 4-process non-stabilizing Token Ring (TR) program in Section 2 and we demonstrated the synthesis of its strongly stabilizing version in Section 5. In this section, we present some alternative strongly stabilizing versions of the TR program that STSyn generated automatically. These alternative solutions illustrate how useful a tool for automated addition of stabilization could be in helping designers understand the impact of different factors (such as variable domains, the scheduling policy of processes, etc.) on the stabilization property.

Let $K$ denote the number of processes and $L$ denote the size of the domain of each variable $x_i$, for $0 \leq i \leq K-1$. Moreover, let $sch[K]$ be an array of $K$ integers, where $sch[i]$ contains the index of the process that is used for the addition of recovery in iteration $j$ of the for loop in Line 9 of Figure 2.

**Case 1:** $K = 4, L = 3$ and $sch[] = \{1, 2, 3, 0\}$. In this case, STSyn synthesized Dijkstra’s token ring program as in [1].

**Case 2:** $K = 4, L = 4$ and $sch[] = \{0, 1, 2, 3\}$. In this case, STSyn synthesized the following program from the non-stabilizing TR program. This program stabilizes for cases where $L > K - 1$. Moreover, observe that only processes $P_1$ and $P_2$ are symmetric. That is, instead of one distinguished process (as in Dijkstra’s program), we have two distinguished processes.

- $P_0$: $x_3 = x_0$  $\rightarrow$  $x_0 := x_0$  $x_0 \neq x_3 + 1 \land x_0 \neq x_3$  $\rightarrow$  $x_0 := x_0$  $x_0 = x_1 - 1$  $\rightarrow$  $x_1 := x_0 - 1$  $x_0 \neq x_1 - 1 \land x_0 \neq x_1$  $\rightarrow$  $x_1 := x_0$  $x_1 = x_2 - 1$  $\rightarrow$  $x_2 := x_1 - 1$  $x_1 \neq x_2 - 1 \land x_1 \neq x_2$  $\rightarrow$  $x_2 := x_1$  $x_2 = x_3 + 1$  $\rightarrow$  $x_3 := x_2$

**Case 3:** $K = 5, L = 4$ and $sch[] = \{1, 2, 3, 4, 0\}$. Interestingly, STSyn generated a program for this case that stabilizes for $L \geq K - 1$, but is somewhat different from Dijkstra’s program in [1]. In particular, process $P_0$ is still a distinguished process and is no different than the first process in non-stabilizing program $p$. Yet, the other processes are different than their counterparts in Dijkstra’s solution! (The action of $P_i$, for $1 \leq i \leq 4$, in Dijkstra’s program is $x_i \neq x_{i-1} \rightarrow x_i := x_{i-1}$.) Each process $P_i$ ($1 \leq i \leq 4$) has two actions with mutually exclusive guards. The second action performs an assignment similar to the action in Dijkstra’s program, but it has a strengthened guard. Notice that the actions of $P_i$ are a sequential decomposition of $P_i$ action in Dijkstra’s program ($1 \leq i \leq 4$). Thus, the convergence of the synthesized process takes more steps compared to Dijkstra’s solution (addition and subtraction are performed in modulo 4).

- $P_0$: $x_4 = x_0$  $\rightarrow$  $x_0 := x_4 + 1$  $P_1$: $x_{i-1} = x_i - 1$  $\rightarrow$  $x_i := x_{i-1} - 1$  $x_{i-1} \neq x_i - 1 \land x_{i-1} \neq x_i$  $\rightarrow$  $x_i := x_{i-1}$
6.2 Maximal Matching on a Bidirectional Ring

The Maximal Matching (MM) program (presented in [32]) has \( K \) processes \( \{P_0, \ldots, P_{K-1}\} \) located on a ring, where \( P_{i-1} \) and \( P_{i+1} \) are respectively the left and right neighbors of \( P_i \), for \( 1 \leq i < K \). The left neighbor of \( P_0 \) is \( P_{K-1} \) and the right neighbor of \( P_{K-1} \) is \( P_0 \). Each process \( P_i \) has a variable \( m_i \) with a domain of three values \{left, right, self\} representing whether or not \( P_i \) points to its left neighbor, right neighbor or itself. Intuitively, two neighbor processes are *matched* if 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 \( m_{i+1} = \text{left} \)), where addition and subtraction are in modulo \( K \). 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 \( P_i \) is *not matched* with any of its neighbors 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 program is empty; i.e., does not include any transitions. Our objective is to automatically generate a strongly stabilizing program 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

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

The state predicate \( I_{MM} \) is a silent predicate in that when the program stabilizes to \( I_{MM} \), no program action is enabled. In a state in \( I_{MM} \), each process is matched with either its right neighbor or its left neighbor. If \( P_i \) is unmatched, then its right neighbor points to right and its left neighbor points to left. The following parameterized actions represent a process \( P_i \) in a manually designed MM program presented by Gouda and Acharya’s [32].

\[
\begin{align*}
  m_i = \text{left} \land m_{i-1} = \text{left} & \quad \Longrightarrow \quad m_i := \text{self} \\
  m_i = \text{right} \land m_{i+1} = \text{right} & \quad \Longrightarrow \quad m_i := \text{self} \\
  m_i = \text{self} \land m_{i-1} \neq \text{left} & \quad \Longrightarrow \quad m_i := \text{left} \\
  m_i = \text{self} \land m_{i+1} \neq \text{right} & \quad \Longrightarrow \quad m_i := \text{right}
\end{align*}
\]

We have automatically synthesized stabilizing MM programs for \( K = 5, 6, 8 \) and 10 in at most 69 seconds. The following actions represent the synthesized MM program for \( K = 5 \).

**Actions of \( P_0 \):**
\[
\begin{align*}
  m_0 = \text{left} \land m_0 \neq \text{self} \land m_1 = \text{right} & \quad \Longrightarrow \quad m_0 := \text{left} \\
  (m_0 \neq \text{left} \land m_4 = \text{right}) \land (m_0 \neq \text{right} \lor m_1 \neq \text{self}) & \quad \Longrightarrow \quad m_0 := \text{left} \\
  m_0 \neq \text{right} \land m_1 = \text{left} \land (m_0 \neq \text{left} \lor m_4 = \text{left}) & \quad \Longrightarrow \quad m_0 := \text{right}
\end{align*}
\]

**Actions of \( P_1 \):**
\[
\begin{align*}
  m_0 = \text{left} \land m_1 \neq \text{self} \land m_2 = \text{right} & \quad \Longrightarrow \quad m_1 := \text{self} \\
  m_1 \neq \text{left} \land m_0 \neq \text{left} \land (m_0 \neq \text{right} \land m_2 = \text{self}) & \quad \Longrightarrow \quad m_1 := \text{left} \\
  m_1 \neq \text{right} \land m_2 \neq \text{right} \land ((m_0 \neq \text{right} \land (m_0 \neq \text{left} \land m_1 \neq \text{left})) \lor (m_2 \neq \text{self})) \lor (m_1 \neq \text{left} \land m_2 \neq \text{self})) & \quad \Longrightarrow \quad m_1 := \text{right}
\end{align*}
\]

**Actions of \( P_2 \):**
\[
\begin{align*}
  m_1 = \text{left} \land m_2 \neq \text{self} \land m_3 = \text{right} & \quad \Longrightarrow \quad m_2 := \text{self} \\
  m_2 \neq \text{left} \land m_1 \neq \text{right} \land ((m_1 \neq \text{self} \land (m_2 \neq \text{right} \lor m_4 \neq \text{left})) \lor (m_2 \neq \text{self} \land m_3 = \text{right} \lor (m_4 \neq \text{right} \land m_3 = \text{self}))) & \quad \Longrightarrow \quad m_2 := \text{left} \\
  m_2 \neq \text{right} \land ((m_1 = \text{self} \land m_2 \neq \text{left}) \lor (m_1 = \text{left} \land m_3 = \text{left}) \lor (m_2 \neq \text{left} \land m_3 = \text{left})) & \quad \Longrightarrow \quad m_2 := \text{right}
\end{align*}
\]
Actions of $P_3$:

\[
\begin{align*}
    m_2 &= \text{left} \land m_3 \neq \text{self} \land m_4 = \text{right} & \rightarrow m_4 := \text{self} \\
    m_3 &\neq \text{left} \land m_3 \neq \text{left} \land (m_2 \neq \text{self} \land (m_3 \neq \text{right} \lor m_4 = \text{right})) & \rightarrow m_4 := \text{left} \\
    \neg (m_3 \neq \text{right} \land m_4 = \text{self}) & \quad \rightarrow m_3 := \text{right} \\
    m_3 &\neq \text{right} \land (m_3 \neq \text{left} \land m_2 = \text{self}) \lor (m_2 \neq \text{self} \land m_4 = \text{left}) & \rightarrow m_4 := \text{right}
\end{align*}
\]

Actions of $P_4$:

\[
\begin{align*}
    m_3 &= \text{left} \land m_4 \neq \text{self} \land m_0 = \text{right} & \rightarrow m_4 := \text{self} \\
    m_4 &\neq \text{left} \land (m_3 = \text{right} \lor (m_4 \neq \text{right} \land m_0 = \text{self})) & \rightarrow m_4 := \text{left} \\
    m_4 &\neq \text{right} \land ((m_0 = \text{left} \land (m_3 = \text{left} \lor m_4 \neq \text{left})) & \rightarrow m_4 := \text{right}
\end{align*}
\]

Interestingly, our automatically generated MM program is different from that of Gouda and Acharya. This difference motivated us to investigate the causes of such differences. Surprisingly, while analyzing Gouda and Acharya’s program, we found out that their program includes a non-progress cycle starting from the state ⟨left, self, left, self, 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 program. This experiment illustrates how difficult the design and verification of strongly stabilizing programs is and how automated design can facilitate the development of stabilizing systems by generating programs that are correct by construction. To gain more confidence in the implementation of STSyn, we have model checked the MM program we have synthesized (available at http://cs.mtu.edu/~anfaraha/CaseStudiesExamples).

6.3 Three Coloring

In this section, we present a strongly stabilizing three-coloring program in a ring (adapted from [32]). The Three Coloring (TC) program 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 $i - 1$ and $i + 1$ 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 program has no transitions initially. The synthesized program 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 program with 40 processes with the following actions labeled by process numbers ($1 \leq i \leq 40$), where other($x,y$) is a nondeterministic function that returns a color different from $x$ and $y$ if $x \neq y$; other($x,x$) non-deterministically returns one of the two remaining colors.

\[
\begin{align*}
    P_1: & \quad (c_1 = c_0) \lor (c_1 = c_2) & \rightarrow c_1 := \text{other}(c_0, c_2) \\
    P_2: & \quad (c_{(i-1)} \neq c_i) \land (c_i = c_{(i+1)}) & \rightarrow c_i := \text{other}(c_{(i-1)}, c_{(i+1)})
\end{align*}
\]

We would like to mention that this program is different from the TC program presented in [32]; i.e., STSyn generated an alternative solution for the TC problem.

6.4 Two-Ring Token Passing

In this section, we demonstrate the addition of strong stabilization to a token ring program with a two-ring topology. The Two-Ring Token Passing (TRTP) program includes 8 processes located in two rings A and B (see Figure 5). In Figure 5, 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 $\{0, 1, 2, 3\}$. Process $PA_i$ (respectively, $PB_i$), $1 \leq i \leq 3$, is allowed to read its own state and the state of its predecessor and write only $a_i$ (respectively, $b_i$). Process $PA_0$ (respectively, $PB_0$) can read its own state and the state of its predecessor $PA_3$, $PB_0$ and $PB_3$ (respectively, $PB_3$, $PA_0$ and $PA_3$) and turn. Process $PA_0$ (respectively, $PB_0$) is permitted to write only $a_0$ (respectively, $b_0$) and turn.

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 =
of actions AC

in ring B are as follows (\(i\) is true in

2

\(PB\)). Process \(PB\) has the token iff \((b_0 = b_3) \land (a_0 = a_3) \land ((b_0 \oplus 1) = a_0)\). That is, \(PB\) 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 \(PB\) \((1 \leq i \leq 3)\) has the token iff \((b_{i-1} = b_i \oplus 1)\). The TRTP program also has a Boolean variable \(turn\): ring A executes only if \(turn = true\), and if ring B executes then \(turn = false\). Using the following actions, the processes circulate the token in rings A and B \((i = 1, 2, 3)\):

\[
AC_0 : \quad (a_0 = a_3) \land turn \quad \rightarrow \quad \text{if } (a_0 = b_0) \quad a_0 := a_3 \oplus 1;
\]
\[
\text{else } \quad turn := false;
\]
\[
AC_1 : \quad (a_{i-1} = a_i \oplus 1) \quad \rightarrow \quad a_i := a_{i-1};
\]
\[
BC_0 : \quad (b_0 = b_3) \land \neg turn \quad \rightarrow \quad \text{if } (a_0 \neq b_0) \quad b_0 := b_3 \oplus 1;
\]
\[
\text{else } \quad turn := true;
\]
\[
BC_1 : \quad (b_{i-1} = b_i \oplus 1) \quad \rightarrow \quad b_i := b_{i-1};
\]

Notice that the action \(AC_i\) is a parameterized action for processes \(PA_1\), \(PA_2\) and \(PA_3\). The actions of the processes in ring B are as follows \((i = 1, 2, 3)\):

A closed predicate, denoted \(I_{TRTP}\). Consider a state \(s_0\) where \((\forall i : 0 \leq i \leq 3 : (a_i = 0) \land (b_i = 0))\) and

\(turn\) is true in \(s_0\). The predicate \(I_{TRTP}\) contains all the states that are reached from \(s_0\) by the execution of actions \(AC_i\) and \(BC_i\), for \(0 \leq i \leq 3\). Starting from a state \(s_0\)

where \((turn(s_0) = true) \land (\forall i : 0 \leq i \leq 3 : (a_i(s_0) = 0) \land (b_i(s_0) = 0))\), process \(PA_0\) has the token and starts circulating the token until the program reaches the state \(s_1\), where \((turn(s_1) = false) \land (\forall i : 0 \leq i \leq 3 : (a_i(s_1) = 1) \land (b_i(s_1) = 0))\).

\(PB\) has the token. Process \(PB\) circulates the token until the program reaches a state \(s_2\), where \((turn(s_2) = true) \land (\forall i : 0 \leq i \leq 3 : (a_i(s_2) = 1) \land (b_i(s_2) = 1))\), process \(PA_0\) again has the token. This way the token circulation continues in both rings, where there is exactly one token in both rings at any time. The predicate \(I_{TRTP} = I_A \land I_B\), where

\[
I_A = \{ s | (\forall i : 0 \leq i \leq 3 : a_i(s) = a_{i+1}(s)) \lor
\quad ((turn(s) = true) \land (\exists j : 1 \leq j \leq 3 : (a_{j-1}(s) = a_j(s) + 1) \land
\quad (\forall k : 0 \leq k < j - 1 : a_k(s) = a_{k+1}(s)) \land
\quad (\forall k : j \leq k < 3 : a_k(s) = a_{k+1}(s)) )\}
\]
\[
I_B = \{ s | (\forall i : 0 \leq i \leq 3 : b_i(s) = b_{i+1}(s)) \lor
\quad ((turn(s) = false) \land (\exists j : 1 \leq j \leq 3 : (b_{j-1}(s) = b_j(s) + 1) \land
\quad (\forall k : 0 \leq k < j - 1 : b_k(s) = b_{k+1}(s)) \land
\quad (\forall k : j \leq k < 3 : b_k(s) = b_{k+1}(s)) )\}
\]

The state predicate \(I_A\) (respectively, \(I_B\)) includes the states in which either all \(a\) (respectively, \(b\)) values are equal or it is the turn of ring A (respectively, B) and there is only one token in ring A (respectively, B). Intuitively, in any state of \(I_{TRTP}\) at most one token exists.

Transient faults. Transient faults may non-deterministically assign a value between 0 and 3 to any variable, which may generate multiple tokens.

\[
\text{FNA: } true \quad \rightarrow \quad a_0 := 0|1|2|3, \quad a_1 := 0|1|2|3, \quad a_2 := 0|1|2|3, \quad a_3 := 0|1|2|3;
\]
\[
\text{FNB: } true \quad \rightarrow \quad b_0 := 0|1|2|3, \quad b_1 := 0|1|2|3, \quad b_2 := 0|1|2|3, \quad b_3 := 0|1|2|3;
\]

Figure 5: The two-ring token passing program.
The self-stabilizing program includes the recovery actions $AC_i$ and $BC_i$, for $1 \leq i \leq 3$, that enable recover to states from where at most one token exists and every process will receive the token.

$$AC_i : (a_{i-1} \neq a_i + 1) \land (a_{i-1} \neq a_i) \rightarrow a_i := a_{i-1};$$

$$BC_i : (b_{i-1} \neq b_i + 1) \land (b_{i-1} \neq b_i) \rightarrow b_i := b_{i-1};$$

**Remark.** While we have presented the TRTP program in the context of 4 processes in each ring, the example can be generalized for any fixed number of processes. Moreover, observe that the number of rings can also be increased, where one process from each ring participates in a higher level ring of processes in which token circulation determines which ring is active.

## 7 Experimental Results

While the significance of our work is in enabling the automated design of self-stabilization, we would like to identify the practical bottlenecks of our work in terms of tool development. With this motivation, in this section, we introduce a set of metrics that we have used in our case studies to evaluate the time/space complexity of automated design of self-stabilization using STSyn. First, we present the platform on which we conducted our experiments. Then we introduce a set of criteria based on which we analyze the performance of STSyn. Finally, we discuss our results for programs presented in Section 6.

**Platform of experiments.** We conducted our experiments on a Linux Fedora 10 distribution personal computer, with 3GHz dual core Intel processor and 1GB of RAM. We have used C++ and the CUDD/GLU library version 2.1 for BDD manipulation [41] in the implementation of STSyn.

**Metrics.** We consider the following metrics in our case studies:

- **Domain Size** denotes the number of distinct values that can be assigned to a program variable.
- **Initial SCC Detection Time** captures the time required to detect if there are SCCs in $\neg I$ of the non-stabilizing program (see Line 2 in Figure 2).
- **Average SCC Detection Time** is the total time for detecting SCCs divided by the number of SCCs in $\neg I$.
- **Ranking Time** is the time required to compute the ranks in the AddWeakStabilization algorithm for the non-stabilizing program.
- **Average SCC Size** is the average number of BDD nodes per SCC, which gives an estimate of the size of SCCs. We believe that the number of BDD nodes is a better measure of space complexity for two reasons: (1) the number of BDD nodes reflects how large the internal data structures created by our algorithms become during synthesis in a platform-independent fashion, and (2) measuring the exact memory space used by an application is often inaccurate because memory chunks might be allocated/freed by the operating system and the heap manager of the CUDD/GLU package on which we have little control.
- **Number of non-trivial SCCs**, where a non-trivial SCC contains multiple states, whereas a trivial SCC is a single state.
- **Total SCC Detection Time** is the time required to detect both non-trivial and trivial SCCs.
- **Synthesized program size** is the number of BDD nodes representing the transition predicate of the synthesized stabilizing program.

Table 1 illustrates the results of our experiments on the three cases of the token ring program presented in Section 6. While the total synthesis time is less than 2 seconds for Case 3 (with 5 processes), the SCC detection appears as the major bottleneck as it comprises the bulk of the total synthesis time and the total number of BDD nodes. We make two observations on the impact of increasing the domain size and the
number of processes on time/space complexity of synthesis. First, moving from Case 1 to Case 2, we keep the number of processes unchanged, but increase the domain size by one. Notice that this change significantly affects the average SCC size and the total SCC detection time. Second, increasing the number of processes from Case 2 to Case 3 has a similar effect in addition to increasing the number of ranks and a drastic increase in the number of SCCs. For these reasons, STSyn failed to synthesize token ring programs with more than 5 processes and domain size of greater than 5 as the number of processes and the domain size are correlated in the token ring program.

<table>
<thead>
<tr>
<th>Table 1: Token Ring Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric</td>
</tr>
<tr>
<td>Domain Size</td>
</tr>
<tr>
<td># of Processes</td>
</tr>
<tr>
<td># of Ranks</td>
</tr>
<tr>
<td>Initial SCC Detection Time (sec.)</td>
</tr>
<tr>
<td>Average SCC Detection Time (sec.)</td>
</tr>
<tr>
<td>Ranking Time (sec.)</td>
</tr>
<tr>
<td>Average SCC Size (BDD nodes)</td>
</tr>
<tr>
<td># of non trivial detected SCC(s)</td>
</tr>
<tr>
<td>Total SCC Detection Time (sec.)</td>
</tr>
<tr>
<td>Total Execution Time (sec.)</td>
</tr>
<tr>
<td>Synthesized Program Size (BDD nodes)</td>
</tr>
</tbody>
</table>

Table 2 presents the results of synthesizing three versions of the maximal matching program for 6, 8 and 10 processes in a ring. Observe that, 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 program with 10 processes in almost 69 seconds.

<table>
<thead>
<tr>
<th>Table 2: Matching Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric</td>
</tr>
<tr>
<td>Domain Size</td>
</tr>
<tr>
<td># of Processes</td>
</tr>
<tr>
<td># of Ranks</td>
</tr>
<tr>
<td>Initial SCC Detection Time (sec.)</td>
</tr>
<tr>
<td>Average SCC Detection Time (sec.)</td>
</tr>
<tr>
<td>Ranking Time (sec.)</td>
</tr>
<tr>
<td>Average Detected SCC Size (BDD nodes)</td>
</tr>
<tr>
<td># of non trivial detected SCC(s)</td>
</tr>
<tr>
<td>Total SCC Detection Time (sec.)</td>
</tr>
<tr>
<td>Total Execution Time (sec.)</td>
</tr>
<tr>
<td>Synthesized Program Size (BDD nodes)</td>
</tr>
</tbody>
</table>

Table 3 demonstrates the values of our metrics for four cases of the coloring program with 10, 20, 30 and 40 processes respectively. Since the coloring program does not include any SCCs outside I\textcolor{red}{\textit{coloring}}, we have been able to scale up the synthesis and generate a stabilizing program with 40 processes. While both I\textcolor{red}{\textit{coloring}} and I\textcolor{red}{\textit{MM}} in the matching program are locally checkable for each process P_i, we note that the complexity of synthesizing a maximal matching program is due to the fact that the correction of the local predicate of P_i cannot be easily achieved. We call such systems non-locally correctable versus locally correctable programs such as coloring. More specifically, consider a case where the first conjunct of the local predicate LC_i (in I\textcolor{red}{\textit{MM}}) 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
Thus, the success of $P_i$ in correcting its local predicate depends on the actions of its neighbors as well. Such dependencies cause cycles outside $I_{MM}$, which complicate the design of self-stabilization. By contrast, in the coloring program, each process can easily establish its local predicate $x_{i-1} \neq x_i$ by selecting a color that is different from its left and right neighbors.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 3</th>
<th>Case 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain Size</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td># of Processes</td>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
</tr>
<tr>
<td># of Ranks</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>Initial SCC Detection Time (sec.)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Average SCC Detection Time (sec.)</td>
<td>0.007</td>
<td>0.009</td>
<td>0.076</td>
<td>0.121</td>
</tr>
<tr>
<td>Ranking Time (sec.)</td>
<td>0.130</td>
<td>9.342</td>
<td>74.805</td>
<td>313.776</td>
</tr>
<tr>
<td>Average Detected SCC Size (BDD nodes)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td># of non trivial detected SCC(s)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total SCC Detection Time (sec.)</td>
<td>0.066</td>
<td>1.772</td>
<td>2.200</td>
<td>4.703</td>
</tr>
<tr>
<td>Total Execution Time (sec.)</td>
<td>0.247</td>
<td>11.888</td>
<td>80.207</td>
<td>328.147</td>
</tr>
<tr>
<td>Synthesized Program Size (BDD nodes)</td>
<td>547</td>
<td>1257</td>
<td>1967</td>
<td>2677</td>
</tr>
</tbody>
</table>

In summary, our experience demonstrates that SCC detection and resolution constitute major bottlenecks that we will have to address in our future work. Moreover, it is interesting to identify sufficient conditions that enable the design of locally correctable programs, which mitigates SCC detection and resolution.

8 Conclusions and Future Work

In this paper, we proposed an extensible repository for automated design of self-stabilization. In particular, we presented a deterministically sound and complete algorithm for automatic design of weak stabilizing programs from their non-stabilizing version. We illustrated that the complexity of such synthesis of weak stabilization is polynomial in the state space of the non-stabilizing program. We then presented a polynomial-time heuristic that uses the synthesized weakly stabilizing programs as an approximation for automatic design of strong stabilization. We have developed a software tool, called STabilization Synthesized (STSyn), using which we have automatically generated many (strongly) stabilizing programs including several versions of Dijkstra’s token ring program, maximal matching, three coloring in a ring and delay insensitive stabilization (available at http://cs.mtu.edu/~anfaraha/CaseStudiesExamples). While the current version of STSyn facilitates the design of many self-stabilizing systems, it fails to generate self-stabilizing versions of some programs due to the exponential complexity of automated design. As such, our vision for STSyn is an extensible framework that plays the role of a repository of ready-to-use heuristics that facilitate the design of self-stabilization. Moreover, STSyn has generated alternative solutions (see Section 6) and has facilitated the detection of design errors in manually designed self-stabilizing programs (e.g., in the maximal matching program [32]).

The proposed approach in this paper differs from previous work in several directions. In our previous work [36,42,43], we investigate the automated addition of nonmasking fault tolerance, where we identify a superset of the set of legitimate states reached in the presence of faults, called the fault-span, which may exclude some states. Then we add recovery from the fault-span to a (possibly proper) subset of the set of legitimate states. Nonetheless, automatic addition of self-stabilization is more challenging as recovery should be added from any state in program state space. Bonakdarpour and Kulkarni [31] investigate the problem of adding progress properties under read restrictions, where they might exclude reachable states from program computations towards ensuring progress. Abujarad and Kulkarni [44] present a heuristic for the addition of stabilization to locally checkable systems, where the set of legitimate states can be decomposed into a conjunction of a set of local state predicates. By contrast, our approach is more general in that we add stabilization to non-locally checkable and/or non-locally correctable programs. More importantly, our heuristic for the addition of strong stabilization enables the design of programs that may oscillate for a finite
number of times before converging to the set of legitimate states, whereas most existing design methods rely on strictly decreasing ranking/variant functions.

We are currently investigating several extensions of our work. First, as our experimental results demonstrate, the bottleneck in automated design of strong stabilization is cycle detection and resolution. As such, we plan to devise more intelligent heuristics for cycle resolution. Second, we will focus on identifying sufficient conditions (e.g., locally correctable programs) for efficient addition of strong stabilization. Third, we will investigate the parallelization of our algorithms towards exploiting the computational resources of computer clusters for automated design of self-stabilization. We will also integrate STSyn with modeling languages such as SAL [45] to facilitate the model-driven development of self-stabilizing network protocols.

References


