SMT-based Synthesis of Distributed Self-Stabilizing Systems

Fathiyeh Faghih[1] and Borzoo Bonakdarpour[2]

[1] School of Computer Science, University of Waterloo, Canada
Email: ffaghihe@uwaterloo.ca
[2] Department of Computing and Software, McMaster University, Canada
Email: borzoo@mcmaster.ca

Abstract. A *self-stabilizing* system is one that guarantees reaching a set of *legitimate states* from any arbitrary initial state. Designing distributed self-stabilizing protocols is often a complex task and developing their proof of correctness is known to be significantly more tedious. In this paper, we propose an SMT-based method that automatically synthesizes a self-stabilizing protocol, given the network topology of distributed processes and description of the set of legitimate states. Our method can synthesize synchronous, asynchronous, symmetric, and asymmetric protocols for two types of stabilization, namely *weak* and *strong*. We also report successful automated synthesis of a set of well-known distributed stabilizing protocols such as Dijkstra's token ring, distributed maximal matching, graph coloring, and mutual exclusion in anonymous networks.

1 Introduction

Self-stabilization is a versatile technique for forward fault recovery. A self-stabilizing system has two key features:

– *Strong convergence.* When a fault occurs in the system and, consequently, reaches some arbitrary state, the system is guaranteed to recover proper behavior within a finite number of execution steps.
– *Closure.* Once the system reaches such good behavior, typically specified in terms of a set of *legitimate states*, it remains in this set thereafter in the absence of new faults.

Self-stabilization has a wide range of application domains, including networking [9] and robotics [23]. The concept of self-stabilization was first introduced by Dijkstra in the seminal paper [6], where he proposed three solutions for designing self-stabilizing token circulation in ring topologies. Twelve years later, in a follow up article [7], he published the correctness proof, where he states that demonstrating the proof of correctness of self-stabilization was more complex than he originally anticipated. Indeed, designing correct self-stabilizing algorithms is a tedious and challenging task, prone to errors. Also, complications in designing self-stabilizing algorithms arise, when there is no commonly accessible data store for all processes, and the system state is based on the valuations of variables distributed among all processes [6]. Thus, it is highly desirable to have access to techniques that can automatically generate self-stabilizing protocols that are correct by construction.

With this motivation, in this paper, we focus on the problem of automated *synthesis* of self-stabilizing protocols. Program synthesis (often called the holy grail of computer science) is an algorithmic technique that takes as input a logical specification and automatically generates as output a program that satisfies the specification. Automated synthesis is generally a highly complex and challenging problem due to the high time and space complexity of its decision procedures. For this reason, synthesis is often used for developing intricate but small-sized components of systems. Synthesizing self-stabilizing distributed protocols involves an additional level of complexity, due to constraints caused by distribution. Examples of such constraints include read-write restriction of processes in the shared-memory model, timing models, and symmetry. These constraints result in combinatorial blowups in the search space of corresponding synthesis problems.

Based on the input specification and the type of output program, there are various synthesis techniques. Our technique in this paper to synthesize self-stabilizing protocols takes as input the following specification:

1. A *topology* that specifies (1) a finite set V of variables allowed to be used in the protocol and their respective finite domains, (2) the number of processes, and (3) read-set and write-set of each process; i.e., subsets of V that each process is allowed to read and write.
2. A set of *legitimate states* in terms of a Boolean expression over V.
3. The *timing model*; i.e., whether the synthesized protocol is synchronous or asynchronous.
4. *Symmetry*; i.e., whether or not all processes should behave identically.
5. *Type of stabilization*; i.e., *strong* convergence guarantees finite-time recovery, while *weak* convergence guarantees only the possibility of recovery from any arbitrary state.

Synthesis of a self-stabilizing protocol is a highly complex problem, since synthesizing strong convergence is shown to be NP-complete in the size of the state space, which itself is exponential in the size of variables of the protocol [19]. Our synthesis approach in this paper, is SMT[3]-based. That is, given the five above input constraints, we encode them as a set of SMT constrains. If the SMT instance is satisfiable, then a witness solution to its satisfiability is a distributed protocol that meets the input specification. If the instance is not satisfiable, then we are guaranteed that there is no protocol that satisfies the input specification. To the best of our knowledge, unlike the work in [3, 10], our approach, is the first sound and complete technique that synthesizes self-stabilizing algorithms. That is, our

[3] *Satisfiability Modulo Theories* (SMT) are decision problems for formulas in first-order logic with equality combined with additional background theories such as arrays, bit-vectors, etc.

approach guarantees synthesizing a protocol that is correct by construction, if theoretically, there exists one. It is also the first that considers synthesizing protocols with different combinations of timing models along with symmetry and types of stabilization.

Our technique for transforming the input specification into an SMT instance consists in developing the following two sets of constraints:

- *State and transition constraints* capture requirements from the input specification that are concerned with each state and transition of the output protocol. For instance, read-write restrictions constrain transitions of each process; i.e., in all transitions, a process should only read and write variables that it is allowed to. Timing models, symmetry, and designation of legitimate states are also state/transition constraints. Encoding these constraints in an SMT instance is relatively straightforward.
- *Temporal constraints* in our work are only concerned with ensuring closure and weak/strong convergence. Our approach to encode weak/strong convergence in an SMT instance is inspired by *bounded synthesis* [12]. In bounded synthesis, temporal logic properties are first transformed into a universal co-Büchi automaton. This automaton is subsequently used to synthesize the next-state function or relation, which in turn identifies the set of transitions of each process.

Solving the satisfiability problem for the conjunction of all above state/transition and temporal properties results in synthesizing a stabilizing protocol. In order to demonstrate the effectiveness of our approach, we conduct a diverse set of case studies for automatically synthesizing well-known protocols from the literature of self-stabilization. These case studies include Dijkstra's token ring [6] (for both three and four state machines), maximal matching [22], weak stabilizing token circulation in anonymous networks [5], and the three coloring problem [14]. Given different input settings (i.e., in terms of the network topology, type of stabilization, symmetry, and timing model), we report and analyze the total time needed for synthesizing these protocols using the constraint solver Alloy [17].

Organization The rest of the paper is organized as follows. In Section 2, we present the preliminary concepts on the shared-memory model and self-stabilization. Formalization of timing models and symmetry in distributed programs are described in Section 3. Then, Section 4 formally states the synthesis problem in the context of self-stabilizing systems. In Section 5, we describe our SMT-based technique, while Section 6 is dedicated to our case studies. Related work is discussed in Section 7. Finally, we make concluding remarks and discuss future work in Section 8. The appendix provides a summary of notations and a lookup table of synthesis SMT constraints with respect to different types of self-stabilizing distributed programs and additional case studies.

2 Preliminaries

2.1 Distributed Programs

Throughout the paper, let V be a finite set of discrete *variables*, where each variable $v \in V$ has a finite domain D_v. A *state* is a valuation of all variables; i.e., a mapping from each variable $v \in V$ to a value in its domain D_v. We call the set of all possible states the *state space*. A *transition* in the state space is an ordered pair (s_0, s_1), where s_0 and s_1 are two states. A *state predicate* is a set of states and a *transition predicate* is a set of transitions. We denote the value of a variable v in state s by $v(s)$.

Definition 1. *A* process π *over a set* V *of variables is a tuple* $\langle R_\pi, W_\pi, T_\pi \rangle$, *where*

- $R_\pi \subseteq V$ *is the* read-set *of* π*; i.e., variables that* π *can read,*
- $W_\pi \subseteq R_\pi$ *is the* write-set *of* π*; i.e., variables that* π *can write, and*
- T_π *is the transition predicate of process* π*, such that* $(s_0, s_1) \in T_\pi$ *implies that for each variable* $v \in V$*, if* $v(s_0) \neq v(s_1)$*, then* $v \in W_\pi$. □

Notice that Definition 1 requires that a process can only change the value of a variable in its write-set (third condition), but not blindly (second condition). We say that a process $\pi = \langle R_\pi, W_\pi, T_\pi \rangle$ is *enabled* in state s_0 if there exists a state s_1, such that $(s_0, s_1) \in T_\pi$.

Definition 2. *A* distributed program *is a tuple* $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$*, where*

- $\Pi_\mathcal{D}$ *is a set of processes over a common set* V *of variables, such that:*
  - *for any two distinct processes* $\pi_1, \pi_2 \in \Pi_\mathcal{D}$*, we have* $W_{\pi_1} \cap W_{\pi_2} = \emptyset$
  - *for each process* $\pi \in \Pi_\mathcal{D}$ *and each transition* $(s_0, s_1) \in T_\pi$*, the following* read restriction *holds:*

$$\begin{aligned} \forall s_0', s_1' :\ & (\forall v \in R_\pi : (v(s_0) = v(s_0') \ \wedge\ v(s_1) = v(s_1'))) \wedge \\ & (\forall v \notin R_\pi : v(s_0') = v(s_1'))) \implies (s_0', s_1') \in T_\pi \end{aligned} \qquad (1)$$

- $T_\mathcal{D}$ *is a transition predicate.* □

Intuitively, the read restriction in Definition 2 imposes the constraint that for each process π, each transition in T_π depends only on reading the variables that π can read (i.e. R_π). Thus, each transition in $T_\mathcal{D}$ is in fact an equivalence class in $T_\mathcal{D}$, which we call a *group* of transitions. The key consequence of read restrictions is that during synthesis, if a transition is included (respectively, excluded) in $T_\mathcal{D}$, then its corresponding group must also be included (respectively, excluded) in $T_\mathcal{D}$. Also, notice that $T_\mathcal{D}$ is defined in an abstract fashion. In Section 3, we will discuss what transitions are included in $T_\mathcal{D}$ based on the timing model and symmetry of process in $\Pi_\mathcal{D}$.

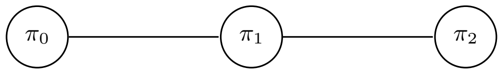


Fig. 1. Example of a maximal matching problem

Example We use the problem of distributed self-stabilizing *maximal matching* as a running example to describe the concepts throughout the paper. In an undirected graph a maximal matching is a maximal set of edges, in which no two edges share a common vertex. Consider the graph in Fig. 1 and suppose each vertex is a process in a distributed program. In particular, let $V = \{match_0, match_1, match_2\}$ be the set of variables and $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ be a distributed program, where $\Pi_{\mathcal{D}} = \{\pi_0, \pi_1, \pi_2\}$. We also have $D_{match_0} = \{1, \perp\}$, $D_{match_1} = \{0, 2, \perp\}$, and $D_{match_2} = \{1, \perp\}$. In other words, each process can be matched to one of its adjacent processes, or to no process (i.e., the value $\perp$). Each process π_i can read and write variable $match_i$ and read the variables of its adjacent processes. For instance, $\pi_0 = \langle R_{\pi_0}, W_{\pi_0}, T_{\pi_0} \rangle$, with $R_{\pi_0} = \{match_0, match_1\}$ and $W_{\pi_0} = \{match_0\}$. Notice that following Definition 2 and read/write restrictions of π_0, (arbitrary) transitions

$$t_1 = ([match_0 = match_2 =\perp, match_1 = 0], [match_0 = 1, match_1 = 0, match_2 =\perp])$$
$$t_2 = ([match_0 =\perp, match_1 = 0, match_2 = 1], [match_0 = match_2 = 1, match_1 = 0])$$

have the same effect as far as π_0 is concerned (since π_0 cannot read $match_2$). This implies that if t_1 is included in the set of transitions of a distributed program, then so should t_2. Otherwise, execution of t_1 by π_0 will depend on the value of $match_2$, which, of course, π_0 cannot read. Notice that the target state in t_2, where $match_0 = 1$, $match_1 = 0$, and $match_2 = 1$, is not a good matching state. However, such states in a distributed program may be reachable due to occurrence of faults or wrong initialization.

Definition 3. *A* computation *of $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ is an infinite sequence of states $\overline{s} = s_0 s_1 \cdots$, such that: (1) for all $i \geq 0$, we have $(s_i, s_{i+1}) \in T_{\mathcal{D}}$, and (2) if a computation reaches a state s_i, from where there is no state $\mathfrak{s} \neq s_i$, such that $(s_i, \mathfrak{s}) \in T_{\mathcal{D}}$, then the computation stutters at s_i indefinitely. Such a computation is called a* terminating computation*.* □

As an example, in maximal matching, computations may terminate when a matching between processes is established.

We now define the notion of *topology*. Intuitively, a topology specifies only the architectural structure of a distributed program (without its set of transitions). The reason for defining topology is that one of the inputs to our synthesis solution is a topology based on which a distributed program is synthesized as output.

Definition 4. *A* topology *is a tuple $\mathcal{T} = \langle V_{\mathcal{T}}, |\Pi_{\mathcal{T}}|, R_{\mathcal{T}}, W_{\mathcal{T}} \rangle$, where*

– *$V_{\mathcal{T}}$ is a finite set of finite-domain discrete variables,*

– $|\Pi_{\mathcal{T}}| \in \mathbb{N}_{\geq 1}$ *is the number of processes,*
– $R_{\mathcal{T}}$ *is a mapping* $\{0 \dots |\Pi_{\mathcal{T}}| - 1\} \mapsto 2^V$ *from a process index to its read-set,*
– $W_{\mathcal{T}}$ *is a mapping* $\{0 \dots |\Pi_{\mathcal{T}}| - 1\} \mapsto 2^V$ *that maps a process index to its write-set, such that* $W_{\mathcal{T}}(i) \subseteq R_{\mathcal{T}}(i)$*, for all* i $(0 \leq i \leq |\Pi_{\mathcal{T}}| - 1)$. □

Example The topology of our matching problem is a tuple $\langle V, |\Pi_{\mathcal{T}}|, R_{\mathcal{T}}, W_{\mathcal{T}} \rangle$, where

– $V = \{match_0, match_1, match_2\}$, with domains $D_{match_0} = \{1, \perp\}$, $D_{match_1} = \{0, 2, \perp\}$, and $D_{match_2} = \{1, \perp\}$,
– $|\Pi_{\mathcal{T}}| = 3$,
– $R_{\mathcal{T}}(0) = \{match_0, match_1\}$, $R_{\mathcal{T}}(1) = \{match_0, match_1, match_2\}$, $R_{\mathcal{T}}(2) = \{match_1, match_2\}$, and
– $W_{\mathcal{T}}(0) = \{match_0\}$, $W_{\mathcal{T}}(1) = \{match_1\}$, and $W_{\mathcal{T}}(2) = \{match_2\}$.

Definition 5. *A distributed program* $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ has *topology* $\mathcal{T} = \langle V_{\mathcal{T}}, |\Pi_{\mathcal{T}}|, R_{\mathcal{T}}, W_{\mathcal{T}} \rangle$, *iff*

– *each process* $\pi \in \Pi_{\mathcal{D}}$ *is defined over* $V_{\mathcal{T}}$
– $|\Pi_{\mathcal{D}}| = |\Pi_{\mathcal{T}}|$
– *there is a mapping* $g : \{0 \dots |\Pi_{\mathcal{T}}| - 1\} \mapsto \Pi_{\mathcal{D}}$ *such that*

$$\forall i \in \{0 \dots |\Pi_{\mathcal{T}}| - 1\} : (R_{\mathcal{T}}(i) = R_{g(i)}) \wedge (W_{\mathcal{T}}(i) = W_{g(i)})$$

□

2.2 Self-Stabilization

Pioneered by Dijkstra [6], a *self-stabilizing system* is one that always recovers a good behavior (typically, expressed in terms of a set of *legitimate states*), even if it starts execution from any arbitrary initial state. Such an arbitrary state may be reached due to wrong initialization or occurrence of transient faults.

Definition 6. *A distributed program* $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ *is* self-stabilizing *for a set* LS *of* legitimate states *iff the following two conditions hold:*

– Strong convergence: *In any computation* $\overline{s} = s_0 s_1 \cdots$ *of* $\mathcal{D}$*, where* s_0 *is an arbitrary state of* $\mathcal{D}$*, there exists* $i \geq 0$*, such that* $s_i \in LS$*. That is, the* computation-tree logic *(CTL) [11] property:*

$$SC \;=\; \mathbf{A} \Diamond LS \tag{2}$$

– Closure: *For all transitions* $(s_0, s_1) \in T_{\mathcal{D}}$*, if* $s_0 \in LS$*, then* $s_1 \in LS$ *as well. That is, the CTL property:*

$$CL \;=\; LS \Rightarrow \mathbf{A} \bigcirc LS \tag{3}$$

□

Notice that the strong convergence property ensures that starting from any state, any computation will converge to a legitimate state of $\mathcal{D}$ within a finite number of steps. The closure property ensures that starting from any legitimate state, execution of the program remains within the set of legitimate states. Also, since all states in a self-stabilizing distributed program are considered as initial states, CTL formula 3 is evaluated over all possible states. This is why the formula is not of form $\mathbf{A}\Box(LS \Rightarrow \mathbf{A} \bigcirc LS)$.

Example In our maximal matching problem, the set of legitimate states is:

$$LS = \{\ [match_0(s) = 1, match_1(s) = 0, match_2(s) = \perp],$$
$$[match_0(s) = \perp, match_1(s) = 2, match_2(s) = 1]\}$$

There exist several results on impossibility of distributed self-stabilization (e.g., in token circulation and leader election in anonymous networks [15]). Thus, less strong forms of stabilization have been introduced in the literature of distributed computing. One example is *weak-stabilizing* distributed programs [13], where there only exists the *possibility* of convergence.

Definition 7. *A distributed program $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ is* weak-stabilizing *for a set LS of* legitimate states *iff the following two conditions hold:*

– Weak convergence: *For each state s_0 in the state space of $\mathcal{D}$, there exists a computation $\overline{s} = s_0 s_1 \cdots$ of $\mathcal{D}$, where there exists $i \geq 0$, such that $s_i \in LS$. That is, the CTL property:*

$$WC \ = \ \mathbf{E}\Diamond LS \tag{4}$$

– Closure: *As defined in Definition 6.* □

Notice that unlike self-stabilizing programs, in a weak-stabilizing program, there may exist execution cycles outside the set of legitimate states. In the rest of the paper, we use 'strong self-stabilization' (respectively, 'strong convergence') and 'self-stabilization' (respectively, 'convergence') interchangeably.

Notation We denote the fact that a distributed program $\mathcal{D}$ satisfies a temporal logic property φ by $\mathcal{D} \models \varphi$. For example, $\mathcal{D} \models SC$ means that distributed program $\mathcal{D}$ satisfies strong convergence.

3 Timing Models and Symmetry in Distributed Programs

Our synthesis solution takes as input the type of timing model as well as symmetry requirements among processes. These constraints are defined in Subsections 3.1 and 3.2.

3.1 Timing Models

Two commonly-considered timing models in the literature of distributed computing are *synchronous* and *asynchronous* programs [21]. In an asynchronous distributed program, every transition of the program is a transition of one and only one of its processes.

Definition 8. *A distributed program $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ is* asynchronous *iff the following condition holds:*

$$\begin{aligned} ASYN \ =& \ \forall (s_0, s_1) \in T_{\mathcal{D}} : ((\exists \pi \in \Pi_{\mathcal{D}} : (s_0, s_1) \in T_\pi) \vee \\ & ((s_0 = s_1) \ \wedge \ \forall \pi \in \Pi_{\mathcal{D}} : \forall \mathfrak{s} : (s_0, \mathfrak{s}) \notin T_\pi)) \end{aligned} \tag{5}$$

Thus, the transition predicate of an asynchronous program is simply the union of transition predicates of all processes. That is,

$$T_{\mathcal{D}} = \bigcup_{\pi \in \Pi_{\mathcal{D}}} T_{\pi}$$

An asynchronous distributed program resembles a system, where process transitions execute in an *interleaving* fashion.

In a synchronous distributed program, on the other hand, in every step, all enabled processes have to take a step simultaneously.

Definition 9. *A distributed program $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ is* synchronous *iff the following condition holds:*

$$\begin{aligned} SYN = & \forall (s_0, s_1) \in T_{\mathcal{D}} : \forall \pi \in \Pi_{\mathcal{D}} : \\ & (\exists \mathfrak{s} : ((s_0, \mathfrak{s}) \in T_{\pi}) \wedge \forall v \in W_{\pi} : v(s_1) = v(\mathfrak{s})) \vee \\ & (\forall \mathfrak{s} : ((s_0, \mathfrak{s}) \notin T_{\pi}) \wedge \forall v \in W_{\pi} : v(s_0) = v(s_1)) \end{aligned} \tag{6}$$

□

In other words, a distributed program is synchronous, if and only if each transition $(s_0, s_1) \in T_{\mathcal{D}}$ is obtained by execution of all enabled processes (the ones that have a transition starting from s_0). Hence, the value of the variables in their write-sets change in s_1 accordingly. Also, for all non-enabled processes, the value of the variables in their write-sets do not change from s_0 to s_1.

3.2 Symmetry

Symmetry in distributed programs refers to similarity of behavior of different processes.

Definition 10. *A distributed program $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ is called* symmetric *iff for any two distinct processes $\pi, \pi' \in \Pi_{\mathcal{D}}$, there exists a bijection $f : R_{\pi} \to R_{\pi'}$, such that the following condition holds:*

$$\begin{aligned} SYM = & \forall (s_0, s_1) \in T_{\pi} : \exists (s'_0, s'_1) \in T_{\pi'} : \\ & (\forall v \in R_{\pi} : (v(s_0) = f(v)(s'_0))) \wedge (\forall v \in W_{\pi} : (v(s_1) = f(v)(s'_1))) \end{aligned} \tag{7}$$

□

In other words, in a symmetric distributed program, the transitions of a process can be determined by a simple variable mapping from another process. A distributed program is called *asymmetric* if it is not symmetric.

4 Problem Statement

Our goal is to synthesize strong/weak self-stabilizing distributed programs by starting from the description of its set of legitimate states and the architectural structure of processes. Formally, the goal is to devise a synthesis algorithm that takes as input the following:

– a topology $\mathcal{T} = \langle V, |\Pi_{\mathcal{T}}|, R_{\mathcal{T}}, W_{\mathcal{T}} \rangle$
– a set LS of legitimate states
– the specification of the timing model, type of self-stabilization, and symmetry of the resulting system.

The synthesis algorithm is required to generate as output a distributed program $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$, such that, based on the given input specification: (1) $\mathcal{D}$ has topology $\mathcal{T}$, (2) $\mathcal{D} \models SC \wedge CL$ or $\mathcal{D} \models WC \wedge CL$, and (3) $T_{\mathcal{D}}$ respects $ASYN$ or SYN, and if symmetry is required, it also respects SYM.

5 SMT-based Synthesis Solution

In this section, we propose a technique that transforms the synthesis problem stated in Section 4 into an SMT solving problem. An SMT instance consists of two parts: (1) a set of *entity* declarations (in terms of sets, relations, and functions), and (2) first-order modulo-theory *constraints* on the entities. An SMT-solver takes as input an SMT instance and determines whether or not the instance is satisfiable; i.e., whether there exists concrete SMT entities (also called an *SMT model*) that satisfy the constraints. We transform the input to our synthesis problem into an SMT instance. If the SMT instance is satisfiable, then the witness generated by the SMT solver is the answer to our synthesis problem. We describe the SMT entities obtained in our transformation in Subsection 5.1. Constraints that appear in all SMT instances regardless of the timing model, type of symmetry and stabilization are presented in Subsection 5.2, while constraints depending on these factors are discussed in Subsection 5.3.

5.1 SMT Entities

Recall that the inputs to our problem are a topology $\mathcal{T} = \langle V, |\Pi_{\mathcal{T}}|, R_{\mathcal{T}}, W_{\mathcal{T}} \rangle$, a set LS of legitimate states, and the program type. Let $D = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ denote the distributed program to be synthesized that has topology $\mathcal{T}$ and legitimate states LS. In our SMT instance, we include:

– A set D_v for each $v \in V$, which contains the elements in the domain of v.
– A set called S, whose cardinality is

$$\left| \prod_{v \in V} D_v \right|$$

(i.e., the Cartesian product of all variable domains). This set represents the state space of the synthesized distributed program. Notice that in a self-stabilizing program, any arbitrary state can be an initial state and, hence, we need to include the entire state space in the SMT instance.

- An uninterpreted function v_val for each variable v, $v\_val : S \mapsto D_v$ that maps each state in the state-space to a valuation of that variable.
- A relation $T_{\mathcal{D}}$ that represents the transition relation of the synthesized distributed program (i.e., $T_{\mathcal{D}} \subseteq S \times S$). Obviously, the main challenge in synthesizing $\mathcal{D}$ is identifying $T_{\mathcal{D}}$, since variables (and, hence, states) and read/write-sets of $\Pi_{\mathcal{D}}$ are given by topology $\mathcal{T}$.
- A Boolean function $LS : S \mapsto \{0,1\}$. $LS(s)$ is true iff s is a legitimate state.
- An uninterpreted function ψ, from each state to a natural number ($\psi : S \mapsto \mathbb{N}$). We will discuss this function in detail in Subsection 5.3.

Example In our maximal matching problem, the SMT entities are as follows:

- $D_{match_0} = \{\bot, 1\}$, $D_{match_1} = \{\bot, 0, 2\}$, $D_{match_2} = \{\bot, 1\}$
- set S, where $|S| = 12$
- $match_0\_val : S \mapsto D_{match_0}$, $match_1\_val : S \mapsto D_{match_1}$, $match_2\_val : S \mapsto D_{match_2}$
- $T_{\mathcal{D}} \subseteq S \times S$
- $\psi : S \mapsto \mathbb{N}$

5.2 General Constraints

In this section, we present the constraints that appear in all SMT instances regardless of the timing model and type of symmetry and stabilization.

State Distinction As mentioned, we specify the size of the state space in the model. The first constraint in our SMT instance stipulates that any two distinct states differ in the value of some variable:

$$\forall s_0, s_1 \in S : (s_0 \neq s_1) \implies (\exists v \in V : v\_val(s_0) \neq v\_val(s_1)) \tag{8}$$

Example In our maximal matching problem, the state distinction constraint is:

$$\begin{aligned}\forall s_0, s_1 \in S : (s_0 \neq s_1) \implies & (match_0\_val(s_0) \neq match_0\_val(s_1)) \vee \\ & (match_1\_val(s_0) \neq match_1\_val(s_1)) \vee \\ & (match_2\_val(s_0) \neq match_2\_val(s_1))\end{aligned}$$

Read Restrictions To ensure that $\mathcal{D}$ meets the read restrictions given by $\mathcal{T}$, we add the following constraint for each process index $i \in \{0, \ldots, |\Pi_{\mathcal{T}}| - 1\}$:

$$\begin{aligned}&\forall s_0, s_1 \in S : \big((s_0, s_1) \in T_{\mathcal{D}} \wedge \exists v \in W_{\mathcal{T}}(i) : v\_val(s_0) \neq v\_val(s_1)\big) \implies \\ &\forall s_0', s_1' \in S : \big((\forall v' \in R_{\mathcal{T}}(i) : v'\_val(s_0) = v'\_val(s_0') \wedge \\ &\qquad \forall v' \in W_{\mathcal{T}}(i) : v'\_val(s_1) = v'\_val(s_1'))\big) \implies (s_0', s_1') \in T_{\mathcal{D}}\end{aligned} \tag{9}$$

Note that Constraint 9 is formulated differently from the definition of read restriction in Condition 1. The reason is that Definition 2 corresponds to an asynchronous system. To cover both synchronous and asynchronous systems, we formalize read restrictions as Constraint 9, which can be used in addition to Constraint 22 to synthesize asynchronous systems. This will be discussed in Subsection 5.3.

Example In our maximal matching problem, the read restriction for process 0 is the following constraint:

$$\begin{aligned}
&\forall s_0,\, s_1 \in S : \big((s_0, s_1) \in T_{\mathcal{D}} \;\wedge\; match_0\_val(s_0) \neq match_0\_val(s_1)\big) \implies \\
&\forall s'_0, s'_1 \in S :\; \big(match_0\_val(s_0) = match_0\_val(s'_0) \wedge \\
&\qquad\qquad match_1\_val(s_0) = match_1\_val(s'_0) \wedge \\
&\qquad\qquad match_0\_val(s_1) = match_0\_val(s'_1)\big) \implies (s'_0, s'_1) \in T_{\mathcal{D}}
\end{aligned}$$

Closure (*CL*) The formulation of the *CL* constraint in our SMT instance is as follows:

$$\forall s, s' \in S \;:\; (LS(s) \,\wedge\, (s, s') \in T_{\mathcal{D}}) \implies LS(s') \tag{10}$$

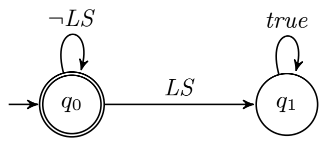
5.3 Program-specific Constraints

We now present the model constraints that depend on the specific timing model, type of symmetry, and stabilization (i.e., strong/weak-stabilization, asynchronous and symmetric programs).

Strong Convergence (*SC*) Our formulation of the SMT constraints for *SC* is an adaptation of the concept of *bounded synthesis* [12]. Inspired by bounded model checking techniques [4], the goal of bounded synthesis is to synthesize an implementation that realizes a set of linear-time temporal logic (LTL) properties, where the size of the implementation is bounded (in terms of the number of states). We emphasize that although strong convergence (Constraint 2) is stated in CTL, it can also be stated by an equivalent LTL property:

$$\mathcal{D} \models \mathbf{A}\Diamond LS \;\;\Leftrightarrow\;\; \mathcal{D} \models \Diamond LS$$

for any distributed program $\mathcal{D}$. One difficulty with bounded model checking and synthesis is to make an estimate on the size of reachable states of the program under inspection. We argue that this difficulty is not an issue in the context of synthesizing self-stabilizing systems, since it is assumed that any arbitrary state is either reachable or can be an initial state. Hence, the bound will be equal to the size of the state space; i.e., the size is a priori known by the input topology. The bounded synthesis technique for synthesizing a state-transition system from a set of LTL properties consists in two steps [12]:



$Q = \{q_0, q_1\}$, $Q_0 = \{q_0\}$, $\Delta = \{(q_0, q_0), (q_0, q_1), (q_1, q_1)\}$, $G(q_0, q_0) = \{\neg LS\}$, $G(q_0, q_1) = \{LS\}$, $G(q_1, q_1) = \{true\}$

Fig. 2. Universal co-Büchi automaton for strong convergence $\varphi = \Diamond LS$.

- **Step 1: Translation to universal co-Büchi automaton.** First, we transform each LTL property φ into a universal co-Büchi automaton B_φ. Roughly speaking, a universal co-Büchi automaton is a tuple $B_\varphi = \langle Q, Q_0, \Delta, G\rangle$, where Q is a set of states, $Q_0 \subseteq Q$ is the set of initial states, $\Delta \subseteq Q \times Q$ is a set of transitions, and G maps each transition in Δ to propositional conditions. Each state could be accepting (depicted by a circle), or rejecting (depicted by a double-circle). For instance, Fig. 2 shows the universal co-Büchi automaton for the strong convergence property $SC = \Diamond LS$.
  Let $ST = \langle S, S_0, T_{\mathcal{D}}\rangle$ be a state-transition system, where S is a set of states, $S_0 \subseteq S$ is the set of initial states, and $T_{\mathcal{D}} \subseteq S \times S$ is a set of transitions. We say that B_φ accepts ST iff on every infinite path of ST running on B_φ, there are only finitely many visits to the set of rejecting states in B_φ [20]. For instance, if a state-transition system is self-stabilizing for the set LS of legitimate states, all its infinite paths visit a state in $\neg LS$ only finitely many times. Hence, the automaton in Fig. 2 accepts such a system.
- **Step 2: SMT encoding.** In this step, the conditions for the co-Büchi automaton to satisfy a state-transition system are formulated as a set of SMT constraints. To this end, we utilize the technique proposed in [12] for developing an *annotation function* $\lambda : Q \times S \mapsto \mathbb{N} \cup \{\bot\}$, such that the following three conditions hold:

$$\forall q_0 \in Q_0 : \forall s_0 \in S_0 \; : \; \lambda(q_0, s_0) \in \mathbb{N} \tag{11}$$

  If (1) $\lambda(q, s) \neq \bot$ for some $q \in Q$ and $s \in S$, (2) there exists $q' \in Q$ such that q' is an accepting state and $(q, q') \in \Delta$ with the condition $g \in G$, and (3) g is satisfied in the state s, then

$$\forall s' \in S : (s, s') \in T_{\mathcal{D}} \implies (\lambda(q', s') \neq \bot \; \wedge \lambda(q', s') \geq \lambda(q, s)) \tag{12}$$

  and if q' is a rejecting state in the co-Büchi automaton, then

$$\forall s' \in S \; : \; (s, s') \in T_{\mathcal{D}} \implies (\lambda(q', s') \neq \bot \; \wedge \lambda(q', s') > \lambda(q, s)) \tag{13}$$

It is shown in [12] that the acceptance of a finite-state state-transition system by a universal co-Büchi automaton is equivalent to the existence of an annotation function λ. The natural number assigned to (q, s) by λ can represent the maximum number of rejecting states that occur on some path to (q, s) when running the state-transition system on the universal co-Büchi automaton.

To ensure that the synthesized distributed program $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ satisfies strong convergence, we use the bounded synthesis technique explained above. In the first step, we construct the universal co-Büchi automaton for the LTL property $\Diamond LS$ (see Fig. 2). The annotation constraints for the transitions in $T_\mathcal{D}$ with the set of states S for the automaton in Fig. 2 are as follows:

$$\forall s \in S \ : \ \lambda(q_0, s) \neq \perp \tag{14}$$

$$\forall s, s' \in S : \ (\lambda(q_0, s) \neq \perp \ \wedge LS(s) \ \wedge \ (s, s') \in T_\mathcal{D}) \implies (\lambda(q_1, s') \neq \perp \ \wedge \lambda(q_1, s') \geq \lambda(q_0, s)) \tag{15}$$

$$\forall s, s' \in S \ : \ (\lambda(q_1, s) \neq \perp \ \wedge \mathit{true} \ \wedge \ (s, s') \in T_\mathcal{D}) \implies (\lambda(q_1, s') \neq \perp \ \wedge \lambda(q_1, s') \geq \lambda(q_1, s)) \tag{16}$$

$$\forall s, s' \in S \ : \ (\lambda(q_0, s) \neq \perp \ \wedge \neg LS(s) \ \wedge \ (s, s') \in T_\mathcal{D}) \implies (\lambda(q_0, s') \neq \perp \ \wedge \lambda(q_0, s') > \lambda(q_0, s)) \tag{17}$$

Notice that Constraint 14 is obtained from Constraint 11 (since in a self-stabilizing system, every state can be an initial state). Similarly, Constraints 15 and 16 are instances of Constraint 12 for transitions (q_0, q_1) and (q_1, q_1), respectively. Also, Constraint 17 is an instance of Constraint 13 for transition (q_0, q_0) (see Fig 2). We now claim that Constraints 15 and 16 can be eliminated.

Lemma 1. *There always exists a non-trivial annotation function λ, which evaluates Constraints 15 and 16 as true.*

Proof. We show that we can always find an annotation function that satisfies Constraints 15 and 16 without violating the other constraints. To this end, assume that there is an annotation that satisfies all properties except for the Constraint 15. Hence, we have:

$$\exists s, s' \in S \ : \ LS(s) \ \wedge \ (s, s') \in T_\mathcal{D} \ \wedge (\lambda(q_1, s') = \perp \ \vee \lambda(q_1, s') < \lambda(q_0, s))$$

We can simply assign $\lambda(q_0, s)$ to $\lambda(q_1, s')$, without violating Constraints 14 and 17. This assignment can be done in a fixpoint iteration, until no more violation exists. We can develop a similar proof for Constraint 16. Intuitively, for each state s, we assign to $\lambda(q_1, s)$, the maximum number assigned to $\lambda(q_1, s')$, for every state s' in any path reaching s. $\square$

Following Lemma 1, since Constraints 15 and 16 can be removed from the SMT instance, all constraints involving λ will have q_0 as their first argument. This observation results in replacing λ by a simpler annotation function ψ as follows:

– Function ψ takes only one argument, since the state of the co-Buchi automaton is always q_0.

– Due to Constraint 14, the value $\perp$ is irrelevant in the range of the annotation functions. Hence, we define our annotation function as:

$$\psi \ : \ S \mapsto \mathbb{N} \tag{18}$$

As a result, one can simplify Constraints 14-17 as follows:

$$\forall s, s' \in S \ : \ \neg LS(s) \ \wedge \ (s, s') \in T_{\mathcal{D}} \implies \psi(s') > \psi(s) \tag{19}$$

The intuition behind Constraints 18 and 19 can be understood easily. If we can assign a natural number to each state, such that along each outgoing transition from a state in $\neg LS$, the number is strictly increasing, then the path from each state in $\neg LS$ should finally reach LS or get stuck in a state, since the size of state space is finite. Also, there can not be any loops whose states are all in $\neg LS$, as imposed by the annotation function.

Finally, the following constraint ensures that there is no deadlock state in $\neg LS$:

$$\forall s \in S \ : \ \neg LS(s) \implies \exists s' \in S \ : \ (s, s') \in T_{\mathcal{D}} \tag{20}$$

Weak Convergence (*WC*) To synthesize a weak self-stabilizing system, the SMT instance should encode property $WC = \exists \Diamond LS$ rather than SC. Notice that WC is not an LTL property and, hence, cannot be transformed into an SMT constraint using the 2-step approach introduced in Subsection 5.3. To this end, we refine the constraints developed for strong convergence as follows. Since in weak convergence, for each state in $\neg LS$, a path should exist to a state in LS, we utilize the following constraint:

$$\forall s \in S \ : \ \neg LS(s) \implies \exists s' \in S \ : \ (s, s') \in T_{\mathcal{D}} \ \wedge \ \psi(s') > \psi(s) \tag{21}$$

where ψ is the annotation Function 18. It is straightforward to prove using induction that if a transition system satisfies Constraint 21, then for each state in $\neg LS$, there exists a path to a state in LS.

Constraints for an Asynchronous System The transition relation obtained using the constraints introduced in the previous Subsection does not impose any requirements on which process can execute in each state. In fact, since $T_{\mathcal{D}}$ encodes a next-state function, all processes that can execute a local transition while respecting the read-write restrictions would take a step. Such a program stipulates a synchronous program, where all processes execute a local transition at the same time (if there exists one). To synthesize an asynchronous distributed program, instead of a transition function $T_{\mathcal{D}}$, we introduce a transition relation T_i for each process index $i \in \{0, \ldots, |\Pi_{\mathcal{T}}| - 1\}$ $T_{\mathcal{D}} = T_0 \cup \cdots \cup T_{|\Pi_{\mathcal{T}}|-1})$, and add the following constraint for each transition relation:

$$\forall (s_0, s_1) \in T_i \ : \forall v \notin W_{\mathcal{T}}(i) \ : \ v\_val(s_0) = v\_val(s_1) \tag{22}$$

Constraint 22 ensures that in each relation T_i, only process π_i can execute. By introducing $|\Pi_{\mathcal{T}}|$ transition relations, we consider all possible interleaving of processes execution.

Example To synthesize an asynchronous version of our maximal matching example, we define three relations T_0, T_1, and T_2 and add a constraint for each to the SMT instance. For example, the constraint for T_0 is:

$$\begin{aligned}\forall(s_0, s_1) \in T_0 \ &: (match_1\_val(s_0) = match_1\_val(s_1)) \wedge \\ &\quad (match_2\_val(s_0) = match_2\_val(s_1))\end{aligned}$$

Constraints for Symmetric Systems To synthesize a symmetric distributed program, processes should have a symmetric topology as well, meaning that the number of read variables and write variables, as well as their domains, should be similar in all processes (see Constraint 7). Let us assume that the size of read-set and write-set of all processes are $|R_p|$ and $|W_p|$, respectively. Also, assume R_p and W_p to be a set of variables with the same domains as the read-set and write-set of each process. We define an uninterpreted relation T_p that represents how processes execute in a symmetric distributed program:

$$T_p \subseteq (\prod_{(v \in R_p)} D_v) \times (\prod_{(v \in W_p)} D_v) \tag{23}$$

Let

$$V\_val : S \mapsto \prod_{v \in V} D_v$$

be the set of all state valuations for the variables in V for a given state. We define a function

$$f : \mathbb{N} \mapsto V\_val$$

that gets a process index and maps it to the valuation function of all variables in the read-set of the process. Likewise, we define a function

$$g : \mathbb{N} \mapsto V\_val$$

which does a similar task for the variables in the write-set of each process. We add the Constraint 24 for each process index $i \in \{0, \dots, |\Pi_{\mathcal{T}}|-1\}$ to ensure that all processes act symmetrically:

$$\begin{aligned}&\exists i \in \{0, \dots, |\Pi_{\mathcal{T}}| - 1\}, \forall s_0, s_1 \in S \ : \ (s_0, s_1) \in T_i \iff \\ &\quad \forall j \in \{0, \dots, |\Pi_{\mathcal{T}}| - 1\} : (f(j)(s_0) \upharpoonright R_{\mathcal{T}}(i), g(j)(s_1) \upharpoonright W_{\mathcal{T}}(i)) \in T_p\end{aligned} \tag{24}$$

Note that synthesis of symmetric systems does not need the read restriction constraints. The reason is that the next value of write variables of a process π is specified by a relation (T_p) based on the values of read variables of the process π. We should also mention that Constraint 24 corresponds to an asynchronous system. The constraint could be easily rewritten for a synchronous system, where there is only one transition relation.

Example In order to synthesize a symmetric program for our matching problem, we assume there are four processes π_0, π_1, π_2, and π_3 on a ring, and that all domains are similar and equal to $\{l, s, r\}$, meaning that a process can be matched to its left or right neighbor, or to itself (no matching). Thus, we define the uninterpreted relation $T_p \subseteq \{l, s, r\} \times \{l, s, r\} \times \{l, s, r\} \times \{l, s, r\}$. Function f maps each process to the values of matching of its right neighbor, itself, and left neighbor, and function g maps each process to value of the only variable in the write-set. For each process, we add an instance of Constraint 24 to the SMT instance. For example, for process π_0, the following constraint is added to the SMT instance:

$$\forall s_0, s_1 \in S \; : \; (s_0, s_1) \in T_0 \iff \\ ((match_4\_val(s_0), match_0\_val(s_0), match_1\_val(s_0)), match_0\_val(s_1)) \in T_p$$

6 Case Studies and Experimental Results

We evaluate our synthesis method using several case studies from well-known distributed self-stabilizing problems. We consider cases where synthesis succeeds and cases where synthesis fails to find a solution for the given topology. Failure of synthesis is normally due to impossibility of self-stabilization for certain problems. We emphasize that although our case studies deal with synthesizing a small number of processes (due to high complexity of synthesis), having access to a solution for a small number of processes can give key insights to designers of self-stabilizing protocols to generalize the protocol for any number of processes. For example, our method can be applied in cases where there exists a *cut-off point* [18]. We should also mention that the maximum number of processes in the system we could synthesize differs from problem to problem. This number solely depends on the complexity of the input specification and, hence, the SMT instance. That means there is no fixed maximum number of processes that this method can handle.

We used the Alloy [17] model finder tool for our experiments. Alloy solver performs the relational reasoning over quantifiers, which means that we did not have to unroll quantifiers over their domains. All experiments in this section are run on a machine with Intel Core i5 2.6 GHz processor with 8GB of RAM. We note that since our synthesis method is deterministic, we do not replicate experiments for statistical confidence. We also conducted experiments using Z3 [2] and Yices [1] SMT solvers as well. In the majority of cases studies Alloy was the fastest solver.

6.1 Maximal Matching

Our first case study is our running example, distributed self-stabilizing *maximal matching* [16, 22, 25]. Recall that each process maintains a *match* variable with domain of all its neighbors and an additional value $\bot$ that indicates the process is not matched to any of its neighbors. The set of legitimate states is the disjunction

Topology	# of Processes	Self-Stabilization	Timing Model	Time (sec)
line	3	strong	synchronous	0.44
line	4	strong	synchronous	5.18
line	4	weak	synchronous	3.29
line	5	weak	synchronous	340.62
star	4	strong	asynchronous	2.95
star	4	weak	asynchronous	2.93
star	5	strong	asynchronous	53.75
star	5	weak	asynchronous	41.80

Table 1. Results for synthesizing maximal matching for line and star topologies.

of all possible maximal matchings on the given topology. As an example, for the graph shown in Fig. 1, we have:

$$(match_0(s) = 1 \wedge match_1(s) = 0 \wedge match_2(s) = \perp) \vee$$
$$(match_0(s) = \perp \wedge match_1(s) = 2 \wedge match_2(s) = 1)$$

Table 1 presents our results for different sizes of line and star topologies. Obviously, such topologies are inherently asymmetric. As expected, by increasing the number of processes, synthesis time also increases. Another observation is that synthesizing a solution for the star topology is in general faster than the line topology. This is because a protocol that intends to solve maximal matching for the star topology deals with a significantly smaller problem space. Also, synthesizing a weak-stabilizing protocol is faster than a self-stabilizing protocol, as the former has more relaxed constraints.

6.2 Dijkstra's Token Ring with Three-State Machines

In the *token ring* problem, a set of processes are placed on a ring network. Each process has a so-called privilege (token), which is a Boolean function of its neighbors' and its own states. When this function is true, the process has the privilege.

Dijkstra [6] proposed three solutions for the token ring problem. In the *three-state token ring*, each process π_i maintains a variable x_i with domain $\{0, 1, 2\}$. The read-set of a process is its own and its neighbors' variables, and its write-set contains its own variable. As an example, for process π_1, $R_{\mathcal{T}}(1) = \{x_0, x_1, x_2\}$ and $W_{\mathcal{T}}(1) = \{x_1\}$. Token possession is formulated using the conditions on a machine and its neighbors [6]. Briefly, in a state s, process π_0 (called the *bottom* process) has the token, when $x_0(s)+1 \mod 3 = x_1(s)$, process $\pi_{(|\Pi_{\mathcal{T}}|-1)}$ (called the *top* process) has the token, when $(x_0(s) = x_{(|\Pi_{\mathcal{T}}|-2)}(s)) \wedge (x_{(|\Pi_{\mathcal{T}}|-2)}(s) + 1 \mod 3 \neq x_{(|\Pi_{\mathcal{T}}|-1)}(s))$, and any other process π_i owns the token, when either $x_i(s) + 1 \mod 3$ equals to the variable of its left or right process. The set of legitimate states are those in which exactly one process has the token. For

# of Processes	Self-Stabilization	Timing Model	Symmetry	Time (sec)
3	strong	asynchronous	asymmetric	1.26
3	weak	asynchronous	asymmetric	1.06
4	strong	asynchronous	asymmetric	63.02
4	weak	asynchronous	asymmetric	62.13

Table 2. Results for synthesizing three-state token ring.

example, for a ring of size three, the set of legitimate states is formulated by the following expression:

$$\begin{aligned}
&((x_0(s)+1 \mod 3 = x_1(s)) \ \wedge \ (x_1(s)+1 \mod 3 \neq x_2(s))) \vee \\
&((x_1(s) = x_0(s)) \ \wedge \ (x_1(s)+1 \mod 3 \neq x_2(s))) \vee \\
&((x_0(s)+1 \mod 3 \neq x_1(s)) \ \wedge \ (x_1(s)+1 \mod 3 = x_0(s)) \vee \\
&(x_1(s)+1 \mod 3 = x_2(s)))
\end{aligned}$$

Table 2 presents the result for synthesizing solutions for the three-state version. Obviously, symmetry is not studied, because the top and bottom processes do not behave similar to other processes. We note that the synthesized strong stabilizing programs using our technique are identical to Dijkstra's solution in [6]. Also, notice that the time needed to synthesize weak and strong stabilizing solutions for the same number of process is almost identical. This is due to the fact that the search space for solving the corresponding SMT instances are of the same complexity.

6.3 Dijkstra's Token Ring with Four-State Machines

In this Subsection, we consider Dijkstra's *four-state machine* solution for token ring [6]. Each process π_i has two Boolean variables; x_i and up_i, where $up_0 = true$ and $up_{(|\Pi_\mathcal{T}|-1)} = false$. Process π_0 is called the *bottom* and process $\pi_{(|\Pi_\mathcal{T}|-1)}$ is called the *top* process. The read-set and write-set of a process is similar to the three-state case in Section 6.2. Token possession is defined based on the variables of a process and its neighbors. Briefly, in a state, say s, the bottom process has the token, if $(x_0(s) = x_1(s)) \wedge (\neg up_1(s))$, the top process has the token, if $x_{(|\Pi_\mathcal{T}|-1)}(s) \neq x_{(|\Pi_\mathcal{T}|-2)}(s)$, and the condition for any other process π_i is $(x_i(s) \neq x_{(i-1)}(s)) \vee (x_i(s) = x_{(i+1)}(s) \wedge up_i(s))$. The legitimate states are those where exactly one process has the token. For example, for a ring of three processes, *LS* is the defined by the following expression:

$$\begin{aligned}
&(x_0(s) = x_1(s)) \wedge \neg up_1(s) \ \wedge (x_2(s) = x_1(s)) \ \vee \\
&(x_0(s) = x_1(s)) \wedge up_1(s) \ \wedge (x_2(s) \neq x_1(s)) \ \vee \\
&(up_1(s) \vee (x_0(s) \neq x_1(s))) \ \wedge (x_2(s) = x_1(s))
\end{aligned}$$

Our results on token ring with four-state machines are presented in Table 3.

# of Processes	Self-Stabilization	Timing Model	Symmetry	Time (sec)
3	strong	asynchronous	asymmetric	0.86
3	weak	asynchronous	asymmetric	0.33
4	strong	asynchronous	asymmetric	30.32
4	weak	asynchronous	asymmetric	29.16

Table 3. Results for synthesizing four-state token ring

6.4 Token Circulation in Anonymous Networks

Token circulation in a *unidirectional* ring is one of the most studied self-stabilizing problems. Herman [15] showed that there is no non-probabilistic self-stabilizing algorithm for this problem in an anonymous network. In [5], Devismes et. al. proposed a weak self-stabilizing solution for this problem. We assume a similar topology to the one used in [5]. In a ring of size $|\Pi_\mathcal{T}|$, each process π_i has a variable dt_i with the domain $\{0, \ldots, m_{(|\Pi_\mathcal{T}|)} - 1\}$, where $m_{(|\Pi_\mathcal{T}|)}$ is the smallest integer not dividing $|\Pi_\mathcal{T}|$. The read-set of a process is its own variable and the variable of its left neighbor, and its write-set contains its own variable. A process holds a token, if and only if $dt_i \neq dt_{pred_i} + 1 \mod m_{(|\Pi_\mathcal{T}|)}$, where $pred_i$ represents the index of the left neighbor. A legitimate state is one where exactly one process has the token. For example, for a ring of size 3, LS can be formulated by the following expression:

$$\begin{aligned}
&(\, \neg(dt_0(s) = dt_2(s) + 1 \mod 2) \wedge (dt_1(s) = dt_0(s) + 1 \mod 2) \\
&\quad \wedge (dt_2(s) = dt_1(s) + 1 \mod 2) \,) \vee \\
&(\, \neg(dt_1(s) = dt_0(s) + 1 \mod 2) \wedge (dt_0(s) = dt_2(s) + 1 \mod 2) \\
&\quad \wedge (dt_2(s) = dt_1(s) + 1 \mod 2) \,) \vee \\
&(\, \neg(dt_2(s) = dt_1(s) + 1 \mod 2) \wedge (dt_0(s) = dt_2(s) + 1 \mod 2) \\
&\quad \wedge (dt_1(s) = dt_0(s) + 1 \mod 2) \,)
\end{aligned}$$

As can be seen in Table 4, for 3 processes, synthesizing a symmetric algorithm for strong self-stabilization is possible. However, For 4 and 5 processes, Alloy returns “unsatisfiable”, which shows the impossibility of strong self-stabilizing for these topologies. For 4 and 5 processes, our method synthesized the same weak-stabilizing algorithm as the one proposed in [5]. Using the “next instance” feature of Alloy, we could also synthesize another protocol for 4 processes in less than 3 seconds. We note that the size of the state space for 5 processes is less than the size for 4 process, as $m_{(|\Pi_\mathcal{T}|)}$ is 3 for 4, while it equals 2 for 5 processes. This is the reason why the synthesis time has decreased from 4 to 5 processes. Also, unlike the previous case study, synthesizing an asymmetric program in an anonymous network is not reasonable, because otherwise the network would lose its anonymity.

# of Processes	Self-Stabilization	Timing Model	Symmetry	Time (sec)
3	strong	asynchronous	symmetric	1.59
4	weak	asynchronous	symmetric	114.56
5	weak	asynchronous	symmetric	10.83

Table 4. Results for synthesizing token circulation in anonymous networks.

# of Processes	Self-Stabilization	Timing Model	Symmetry	Time (sec)
3	strong	asynchronous	symmetric	1.53
3	weak	asynchronous	symmetric	1.85
4	strong	synchronous	asymmetric	106.6
4	strong	synchronous	symmetric	139.35
4	strong	asynchronous	symmetric	83.19
4	weak	asynchronous	symmetric	64.13
4	weak	asynchronous	asymmetric	42.29

Table 5. Results for synthesizing the maximal matching problem on a ring.

6.5 Maximal Matching on Rings

Another case study is the maximal matching problem for ring topologies [14]. This is a special case of the case study in Section 6.1, where the topology allows synthesizing symmetric solutions. The *match* variable for each process has domain $\{l, r, s\}$, where values l and r represent matching with left and right processes, respectively, and value s shows that the process is self-matched.

Table 5 shows the results of our experiments. Note that although the topology is symmetric, the synthesized protocol can be symmetric or asymmetric. We observe that synthesizing an asymmetric protocol is faster than a symmetric protocol, since for the latter, the SMT-solver has to search deeper in the state space to rule out asymmetric solutions. In other words, there exist more asymmetric protocols for the given input.

6.6 The Three-Coloring Problem

In the *three coloring problem* [14], we have a set of processes connected in a ring topology. Each process π_i has a variable c_i, with the domain $\{0, 1, 2\}$. Each value of the variable c_i represents a distinct color. A process can read and write its own variable. It can also read, but not write the variables of its left and right processes. For example, in a ring of four processes, the read-set and write-set of π_0 are $R_{\mathcal{T}}(0) = \{c_3, c_0, c_2\}$ and $W_{\mathcal{T}}(0) = \{c_0\}$, respectively. The set of legitimate states is those where each process has a color different from its left and right neighbors. Thus, for a ring of four processes, LS is defined by the following predicate:

$$\neg(c_0(s) = c_1(s) \,\vee\, c_1(s) = c_2(s) \,\vee\, c_2(s) = c_3(s) \,\vee\, c_3(s) = c_0(s))$$

# of Processes	Self-Stabilization	Timing Model	Symmetry	Time (sec)
3	strong	synchronous	asymmetric	0.56
3	strong	asynchronous	asymmetric	0.93
3	weak	synchronous	asymmetric	0.55
3	weak	asynchronous	symmetric	1.33
4	strong	synchronous	asymmetric	78.71
4	weak	synchronous	asymmetric	96.32
4	strong	asynchronous	asymmetric	35.09
4	strong	asynchronous	symmetric	60.35

Table 6. Results for three-coloring

Our synthesis results for the three coloring problem are reported in Table 6. The results in this case study and the previous ones show that synthesizing asynchronous systems are generally faster compared to the synchronous ones, although we cannot claim if this is always the case. Also, we found an impossibility result for this problem as well. Based on [24], there is no symmetric synchronous deterministic algorithm for the three-coloring problem. We found the same impossibility result in our experiment on 4 processes. Note that for synthesis of a deterministic algorithm, we just changed the transition relation in our SMT model to a transition function.

7 Related Work

In [19], the authors show that adding strong convergence is NP-complete in the size of the state space, which itself is exponential in the size of variables of the protocol. Ebnenasir and Farahat [10] also proposed an automated method to synthesize self-stabilizing algorithms. Our work is different in that the method in [10] is not complete for strong self-stabilization. This means that if it cannot find a solution, it does not necessarily imply that there does not exist one. However, in our method, if the SMT-solver declares "unsatisfiability", it means that no self-stabilizing algorithm that satisfies the given input constraints exists. Also, using our approach, one can synthesize synchronous and asynchronous programs, while the method in [10] synthesizes asynchronous systems only. Finally, our method is based on the constantly-evolving technique of SMT solving. We expect our technique to become more efficient as more efficient SMT solvers emerge.

The distinction of our work and bounded synthesis is that in [12], given is a set of LTL properties, which are translated to a universal co-Büchi automaton, and then a set of SMT constraints are derived from the automaton. Our work is inspired by this idea for finding the SMT constraints for strong convergence. We also used a similar idea for synthesizing weak convergence (although weak-convergence cannot be expressed in LTL). For distribution and timing models, we used a different approach from bounded synthesis, as they are not temporal properties. The other difference is that the main idea in bounded synthesis is to put a bound on the number of states in the resulting state-transition systems, and then increasing the bound if a solution is not found. In our work, since the purpose is to synthesize a self-stabilizing system, the bound is the number of all possible states, derived from the given topology.

The other line of work related to the synthesis of self-stabilizing algorithms is the area of synthesizing fault-tolerant systems. The proposed algorithm in [3] synthesizes a fault-tolerant distributed algorithm from its fault-intolerant version. The distinction of our work with this study is (1) we emphasize on self-stabilizing systems, where any system state could be reachable due to the occurrence of any possible fault, (2) the input to our problem is just a system topology, and not a fault-intolerant system, and (3), the proposed algorithm in [3] is not complete. The other work in synthesis of fault-tolerant systems is the one presented in [8]. In this work, a synthesis algorithm is proposed to determine whether a fault-tolerant implementation exists for a fully connected topology and a temporal specification, and, in case the answer is positive, automatically derives such an implementation. Our work is different in (1) considering any kind of distributed topology, and (2) focusing on self-stabilizing systems.

8 Conclusion

In this paper, we proposed an automated technique for synthesis of finite-size self-stabilizing algorithms using SMT-solvers. The first benefit of our technique is that it is sound and complete; i.e., it generates distributed programs that are correct by construction and, hence, no proof of correctness is required, and if it fails to find a solution, we are guaranteed that there does not exist one. The latter is due to the fact that all quantifiers range over finite domains and, hence, finite memory is needed for process implementations. This assumptions basically ensures decidability of the problem under investigation. Secondly, our method is fully automated and can save huge effort from designers, specially when there is no solution for the problem. Third, the underlying technique is based on SMT-solving, which is a fast evolving area, and hence, by introducing more efficient SMT-solvers, we expect better results from our proposed method. We also reported highly encouraging results of experiments on a diverse set of case studies on some of the well-known problems in self-stabilization.

For future work, we plan to work on synthesis of probabilistic self-stabilizing systems. Another challenging research direction is to devise synthesis methods where the number of distributed processes is parameterized as well as cases where the size of state space processes is infinite. We would also like to investigate techniques such as counter-example guided inductive synthesis (CEGIS) that may be an interesting solution to the problem of scaling the synthesis process for larger number of processes.

9 Acknowledgements

This research was supported in part by Canada NSERC Discovery Grant 418396-2012 and NSERC Strategic Grant 430575-2012.

References

1. Yices: An SMT Solver. `http://yices.csl.sri.com`.
2. Z3: An efficient theorem prover. `http://research.microsoft.com/en-us/um/redmond/projects/z3/`.
3. B. Bonakdarpour, S. S. Kulkarni, and F. Abujarad. Symbolic synthesis of masking fault-tolerant programs. *Springer Journal on Distributed Computing*, 25(1):83–108, March 2012.
4. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
5. S. Devismes, S. Tixeuil, and M. Yamashita. Weak vs. self vs. probabilistic stabilization. In *Proceedings of the 28th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 681–688, 2008.
6. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
7. E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, 1986.
8. R. Dimitrova and B. Finkbeiner. Synthesis of fault-tolerant distributed systems. In *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 321–336, 2009.
9. S. Dolev and E. Schiller. Self-stabilizing group communication in directed networks. *Acta Informatica*, 40(9):609–636, 2004.
10. A. Ebnenasir and A. Farahat. A lightweight method for automated design of convergence. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 219–230, 2011.
11. E. A Emerson. *Handbook of Theoretical Computer Science*, volume B, chapter 16: Temporal and Modal Logics. Elsevier Science Publishers B. V., Amsterdam, 1990.
12. B. Finkbeiner and S. Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer (STTT)*, 15(5-6):519–539, 2013.
13. M. G. Gouda. The theory of weak stabilization. In *International Workshop on Self-Stabilizing Systems*, pages 114–123, 2001.
14. M. G. Gouda and H. B. Acharya. Nash equilibria in stabilizing systems. In *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 311–324, 2009.
15. T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.
16. S.-C. Hsu and S.-T. Huang. A self-stabilizing algorithm for maximal matching. *Information Processing Letters*, 43(2):77–81, 1992.
17. D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press Cambridge, 2012.
18. S. Jacobs and R. Bloem. Parameterized synthesis. *Logical Methods in Computer Science*, 10(1), 2014.
19. A. Klinkhamer and A. Ebnenasir. On the complexity of adding convergence. In *Proceedings of the International Conference Fundamentals of Software Engineering*, pages 17–33, 2013.

20. O. Kupferman and M. Y. Vardi. Safraless decision procedures. In *Proceedings of 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 531–542, 2005.
21. N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Mateo, CA, 1996.
22. F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A new self-stabilizing maximal matching algorithm. *Theoretical Computer Science*, 410(14):1336–1345, 2009.
23. F. Ooshita and S. Tixeuil. On the self-stabilization of mobile oblivious robots in uniform rings. In *International Symposium on Stabilization, Safety, and Security of Distribted Systems (SSS)*, pages 49–63, 2012.
24. S. Shukla, D. Rosenkrantz, and S. Ravi. Developing self-stabilizing coloring algorithms via systematic randomization. In *Proceedings of the International Workshop on Parallel Processing*, page 668673, 1994.
25. G. Tel. Maximal matching stabilizes in quadratic time. *Information Processing Letters*, 49(6):271–272, 1994.

A Summary of Notations

V	set of variables
D_v	domain of variable v
s	state
$v(s)$	value of variable v in state s
π	process
R	read-set
W	write-set
T	transition predicate
$\mathcal{D}$	distributed program
Π	set of processes
$\mathcal{T}$	topology
$\overline{s}$	computation
LS	set of legitimate states
SC	strong convergence constraint
CL	closure constraint
WC	weak convergence constraint
$ASYN$	asynchronous constraint
SYN	synchronous constraint
SYM	symmetry constraint
v_val	valuation function of the variable v

B Table of Constraints for Program Types

The set of constraints needed for each asymmetric system based on the timing model, and type of self-stabilization are presented in Table 7. For synthesizing a symmetric system of each type, constraint 24 should be added to the set of constraints.

	Strong Self-Stabilization	**Weak Self-Stabilization**
Synchronous	8, 9, 10, 19, 20	8, 9, 10, 21
Asynchronous	8, 9, 10, 19, 20, 22	8, 9, 10, 21, 22

Table 7. SMT Constraints for different asymmetric systems