SMT-Based Synthesis of Distributed Self-stabilizing Systems

Fathiyeh Faghih[1] and Borzoo Bonakdarpour[2]

[1] School of Computer Science, University of Waterloo, Canada
ffaghihe@uwaterloo.ca
[2] Department of Computing and Software, McMaster University, Canada
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. We also report successful automated synthesis of Dijkstra's token ring and distributed maximal matching.

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 [8] and robotics [17]. The concept of self-stabilization was first introduced by Dijkstra in the seminal paper [5], where he proposed three solutions for designing self-stabilizing token circulation in ring topologies. Twelve years later, in a follow up article [6], 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 [5]. 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 read-write restriction of processes in the shared-memory model.

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.

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 [14]. Our synthesis approach in this paper, is SMT[1]-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, 9], our approach, is the first sound and complete technique that synthesizes self-stabilizing algorithms. That is, our approach guarantees synthesizing a protocol that is correct by construction, if theoretically, there exists one.

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. Encoding these constraints in an SMT instance is relatively straightforward.

[1] *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.

– *Temporal constraints* in our work are only concerned with ensuring closure and strong convergence. Our approach to encode weak/strong convergence in an SMT instance is inspired by *bounded synthesis* [11]. 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 [5] (for the three-state machine) and maximal matching [16]. Given different input settings (i.e., in terms of the network topology), we report and analyze the total time needed for synthesizing these protocols using the constraint solver Alloy [13].

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. Then, Section 3 formally states the synthesis problem in the context of self-stabilizing systems. In Section 4, we describe our SMT-based technique, while Section 5 is dedicated to our case studies. Related work is discussed in Section 6. Finally, we make concluding remarks and discuss future work in Section 7.

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 that is the union of transition predicates of all processes. I.e.,*

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

$\square$

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 such a way $\mathcal{D}$ resembles an asynchronous distributed program, where process transitions execute in an *interleaving* fashion.

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 $D_{match_0} = \{1, \bot\}$, $D_{match_1} = \{0, 2, \bot\}$, and $D_{match_2} = \{1, \bot\}$. In other words, each process can be matched to one of its adjacent processes, or to no process (i.e., the value $\bot$). 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

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

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.

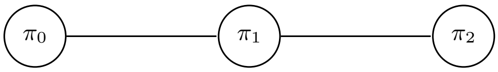


Fig. 1. Example of a maximal matching problem

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* $\{0 \ldots |\Pi_\mathcal{T}| - 1\} \mapsto 2^V$ *from a process index to its read-set,*
- $W_\mathcal{T}$ *is a mapping* $\{0 \ldots |\Pi_\mathcal{T}| - 1\} \mapsto 2^V$ *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 $D_{match_0} = \{1, \bot\}$, $D_{match_1} = \{0, 2, \bot\}$, and $D_{match_2} = \{1, \bot\}$,
- $|\Pi_\mathcal{T}| = 3$,
- $R_\mathcal{T}(0) = \{match_0, match_1\}$, $R_\mathcal{T}(1) = \{match_0, match_1, match_2\}$, $R_\mathcal{T}(2) = \{match_1, match_2\}$, 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* $g : \{0 \ldots |\Pi_\mathcal{T}| - 1\} \mapsto \Pi_\mathcal{D}$ *such that*

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

2.2 Self-Stabilization

Pioneered by Dijkstra [5], 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* linear temporal logic *(LTL) [10] property:*

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

- 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 LTL property:*

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

□

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, LTL formula 3 is evaluated over all possible states. This is why the formula is not of form $\Box(LS \Rightarrow \bigcirc LS)$.

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

$$\begin{aligned} LS = \{\; & [match_0 = 1, match_1 = 0, match_2 = \perp], \\ & [match_0 = \perp, match_1 = 2, match_2 = 1]\} \end{aligned}$$

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 convergence.

3 Problem Statement

Our goal is to synthesize 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 the following as input:

- a topology $\mathcal{T} = \langle V, |\Pi_\mathcal{T}|, R_\mathcal{T}, W_\mathcal{T} \rangle$,
- a set LS of legitimate states,
- the LTL specification of self-stabilization,

and generates a distributed program as output that respects the above input specification.

4 SMT-Based Synthesis Solution

In this section, we propose a technique that transforms the synthesis problem stated in Section 3 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 4.1. SMT constraints appear in Subsection 4.2.

4.1 SMT Entities

Recall that the inputs to our problem are a topology $\mathcal{T} = \langle V, |\Pi_{\mathcal{T}}|, R_{\mathcal{T}}, W_{\mathcal{T}}\rangle$, and a set LS of legitimate states. 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 4.2.

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

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

4.2 SMT Constraints

In this section, we present the SMT constraints formulated based on our synthesis problem.

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:

$$\forall s_0, s_1 \in S \ : (s_0 \neq s_1) \implies (\exists v \in V \ : \ v\_val(s_0) \neq v\_val(s_1)) \tag{4}$$

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

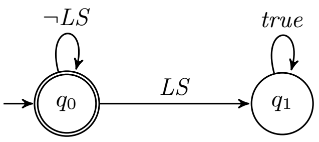
$$\begin{aligned}\forall s_0, s_1 \in S \ : \ (s_0 \neq s_1) \implies & (match_0\_val(s_0) \neq match_0\_val(s_1)) \vee \\ & (match_1\_val(s_0) \neq match_1\_val(s_1)) \vee \\ & (match_2\_val(s_0) \neq match_2\_val(s_1))\end{aligned}$$

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

$$\forall s, s' \in S \ : \ (LS(s) \wedge (s, s') \in T_{\mathcal{D}}) \implies LS(s') \tag{5}$$

Strong Convergence (*SC*). Our formulation of the SMT constraints for *SC* is an adaptation of the concept of *bounded synthesis* [11]. 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). 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 [11]:

– **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 $SC = \Diamond LS$.



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

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

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 [15]. 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 [11] for developing an *annotation function* $\lambda : Q \times S \mapsto \mathbb{N} \cup \{\bot\}$, such that the following three conditions hold:

$$\forall q_0 \in Q_0 : \forall s_0 \in S_0 \ : \ \lambda(q_0, s_0) \in \mathbb{N} \tag{6}$$

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

$$\forall s' \in S : (s, s') \in T_\mathcal{D} \implies (\lambda(q', s') \neq\bot \ \wedge \lambda(q', s') \geq \lambda(q, s)) \tag{7}$$

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

$$\forall s' \in S \ : \ (s, s') \in T_\mathcal{D} \implies (\lambda(q', s') \neq\bot \ \wedge \lambda(q', s') > \lambda(q, s)) \tag{8}$$

It is shown in [11] 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:

$$\forall s \in S \;:\; \lambda(q_0, s) \neq \perp \tag{9}$$

$$\begin{aligned}\forall s, s' \in S :\; & (\lambda(q_0, s) \neq \perp \wedge LS(s) \wedge (s, s') \in T_{\mathcal{D}}) \implies \\ & (\lambda(q_1, s') \neq \perp \wedge \lambda(q_1, s') \geq \lambda(q_0, s))\end{aligned} \tag{10}$$

$$\begin{aligned}\forall s, s' \in S \;:\; & (\lambda(q_1, s) \neq \perp \wedge \mathit{true} \wedge (s, s') \in T_{\mathcal{D}}) \implies \\ & (\lambda(q_1, s') \neq \perp \wedge \lambda(q_1, s') \geq \lambda(q_1, s))\end{aligned} \tag{11}$$

$$\begin{aligned}\forall s, s' \in S \;:\; & (\lambda(q_0, s) \neq \perp \wedge \neg LS(s) \wedge (s, s') \in T_{\mathcal{D}}) \implies \\ & (\lambda(q_0, s') \neq \perp \wedge \lambda(q_0, s') > \lambda(q_0, s))\end{aligned} \tag{12}$$

Notice that Constraint 9 is obtained from Constraint 6 (since in a self-stabilizing system, every state can be an initial state). Similarly, Constraints 10 and 11 are instances of Constraint 7 for transitions (q_0, q_1) and (q_1, q_1), respectively. Also, Constraint 12 is an instance of Constraint 8 for transition (q_0, q_0) (see Fig 2). We now claim that Constraints 10 and 11 can be eliminated.

Lemma 1. *There always exists a non-trivial annotation function λ, which evaluates Constraints 10 and 11 as true.*

Proof. We show that we can always find an annotation function that satisfies Constraints 10 and 11 without violating the other constraints. To this end, assume that there is an annotation that satisfies all properties except for the Constraint 10. 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 9 and 12. This assignment can be done in a fixpoint iteration, until no more violation exists. We can develop a similar proof for Constraint 11. 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 10 and 11 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 9, the value $\perp$ is irrelevant in the range of the annotation functions. Hence, we define our annotation function as:

$$\psi \;:\; S \mapsto \mathbb{N} \tag{13}$$

As a result, one can simplify Constraints 9-12 as follows:

$$\forall s, s' \in S \;:\; \neg LS(s) \wedge (s, s') \in T_{\mathcal{D}} \implies \psi(s') > \psi(s) \tag{14}$$

The intuition behind Constraints 13 and 14 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 $\neg LS$:

$$\forall s \in S \;:\; \neg LS(s) \implies \exists s' \in S \;:\; (s, s') \in T_{\mathcal{D}} \tag{15}$$

Constraints for an Asynchronous System. To synthesize an asynchronous distributed program, instead of a transition relation $T_{\mathcal{D}}$, we introduce a transition relation T_i for each process index $i \in \{0, \dots, |\Pi_{\mathcal{T}}|-1\}$ ($T_{\mathcal{D}} = T_0 \cup \cdots \cup T_{|\Pi_{\mathcal{T}}|-1}$), and add the following constraint for each transition relation:

$$\forall (s_0, s_1) \in T_i \;: \forall v \notin W_{\mathcal{T}}(i) \;:\; v\_val(s_0) = v\_val(s_1) \tag{16}$$

Constraint 16 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:

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

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

$$\begin{aligned}\forall (s_0, s_1) \in T_i :\; \forall s_0', s_1' \in S :\; & (\forall v \in R_\pi : (v(s_0) = v(s_0') \;\wedge\; v(s_1) = v(s_1'))) \wedge \\ & (\forall v \notin R_\pi : v(s_0') = v(s_1'))) \implies (s_0', s_1') \in T_i\end{aligned} \tag{17}$$

which is similar to Condition 1 in Definition 2.

5 Case Studies and Experimental Results

We used the Alloy [13] 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.

5.1 Maximal Matching

Our first case study is our running example, distributed self-stabilizing *maximal matching* [12,16,18]. Table 1 presents our results for different sizes of line and star topologies. 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.

Table 1. Results for synthesizing maximal matching

Topology	**# of Processes**	**Time (sec)**
line	3	0.19
star	4	2.95
line	4	3.5
star	5	53.75
line	5	65.88

5.2 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 [5] 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 [5]. 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 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. Results for synthesizing three-state token ring

# of Processes	Time (sec)
3	1.26
4	63.02

Table 2 presents the result for synthesizing solutions for the three-state version. We note that the synthesized stabilizing programs using our technique are identical to Dijkstra's solution in [5].

6 Related Work

In [14], 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 [9] also proposed an automated method to synthesize self-stabilizing algorithms. Our work is different in that the method in [9] 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.

In bounded synthesis [11], 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. For other constraints, we used a different approach from bounded synthesis. The other difference of our work with bounded synthesis 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 increase 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. In [7], 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.

7 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 assumption 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.

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 of 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.

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. Bonakdarpour, B., Kulkarni, S.S., Abujarad, F.: Symbolic synthesis of masking fault-tolerant programs. Springer Journal on Distributed Computing 25(1), 83–108 (2012)
4. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design 19(1), 7–34 (2001)
5. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974)
6. Dijkstra, E.W.: A belated proof of self-stabilization. Distributed Computing 1(1), 5–6 (1986)
7. Dimitrova, R., Finkbeiner, B.: Synthesis of fault-tolerant distributed systems. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 321–336. Springer, Heidelberg (2009)
8. Dolev, S., Schiller, E.: Self-stabilizing group communication in directed networks. Acta Informatica 40(9), 609–636 (2004)
9. Ebnenasir, A., Farahat, A.: A lightweight method for automated design of convergence. In: Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 219–230 (2011)

10. Emerson, E.A.: Handbook of Theoretical Computer Science. Temporal and Modal Logics, vol. B, ch. 16. Elsevier Science Publishers B. V., Amsterdam (1990)
11. Finkbeiner, B., Schewe, S.: Bounded synthesis. International Journal on Software Tools for Technology Transfer (STTT) 15(5-6), 519–539 (2013)
12. Hsu, S.-C., Huang, S.-T.: A self-stabilizing algorithm for maximal matching. Information Processing Letters 43(2), 77–81 (1992)
13. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press Cambridge (2012)
14. Klinkhamer, A., Ebnenasir, A.: On the complexity of adding convergence. In: Arbab, F., Sirjani, M. (eds.) FSEN 2013. LNCS, vol. 8161, pp. 17–33. Springer, Heidelberg (2013)
15. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Proceedings of 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS), pp. 531–542 (2005)
16. Manne, F., Mjelde, M., Pilard, L., Tixeuil, S.: A new self-stabilizing maximal matching algorithm. Theoretical Computer Science 410(14), 1336–1345 (2009)
17. Ooshita, F., Tixeuil, S.: On the self-stabilization of mobile oblivious robots in uniform rings. In: Richa, A.W., Scheideler, C. (eds.) SSS 2012. LNCS, vol. 7596, pp. 49–63. Springer, Heidelberg (2012)
18. Tel, G.: Maximal matching stabilizes in quadratic time. Information Processing Letters 49(6), 271–272 (1994)