Logical Methods in Computer Science
Vol. 14(1:12)2018, pp. 1–25
https://lmcs.episciences.org/

Submitted Dec. 01, 2016
Published Jan. 30, 2018

AUTOMATED SYNTHESIS OF DISTRIBUTED SELF-STABILIZING PROTOCOLS

FATHIYEH FAGHIH [a], BORZOO BONAKDARPOUR [b], SÉBASTIEN TIXEUIL [c], AND SANDEEP KULKARNI [d]

[a] College of Engineering, University of Tehran, Iran
e-mail address: f.faghih@ut.ac.ir

[b] Iowa State University, USA
e-mail address: borzoo@iastate.edu

[c] LIP6, UPMC Sorbonne Universités, France
e-mail address: sebastien.tixeuil@lip6.fr

[d] Michigan State University, USA
e-mail address: sandeep@cse.msu.edu

Abstract. In this paper, we introduce an SMT-based method that automatically synthesizes a distributed *self-stabilizing* protocol from a given high-level specification and network topology. Unlike existing approaches, where synthesis algorithms require the *explicit* description of the set of legitimate states, our technique only needs the temporal behavior of the protocol. We extend our approach to synthesize *ideal-stabilizing* protocols, where every state is legitimate. We also extend our technique to synthesize *monotonic-stabilizing* protocols, where during recovery, each process can execute an most once one action. Our proposed methods are fully implemented and we report successful synthesis of well-known protocols such as Dijkstra's token ring, a self-stabilizing version of Raymond's mutual exclusion algorithm, ideal-stabilizing leader election and local mutual exclusion, as well as monotonic-stabilizing maximal independent set and distributed Grundy coloring.

1. Introduction

Self-stabilization [7] has emerged as one of the prime techniques for forward fault recovery. A self-stabilizing protocol satisfies two requirements: (1) *Convergence* ensures that starting from any arbitrary state, the system reaches a set of *legitimate states* (denoted in the sequel by LS) with no external intervention within a finite number of execution steps, provided no new faults occur; (2) *Closure* indicates that the system remains in LS thereafter.

As Dijkstra mentions in his belated proof of self-stabilization [8], designing self-stabilizing systems is a complex task. Proving the correctness of these algorithms is even more tedious. Thus, having access to automated methods (as opposed to manual techniques such as [6]) for *synthesizing* correct self-stabilizing systems is highly desirable. However, synthesizing self-stabilizing protocols incurs high time and space complexity [21]. The techniques proposed

A preliminary version of the paper has appeared in [13].

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.23638/LMCS-14(1:12)2018
© F. Faghih, B. Bonakdarpour, S. Tixeuil, and S. Kulkarni
Creative Commons

in [1, 4, 10, 22] attempt to cope with this complexity using heuristic algorithms, but none of these algorithms are complete; i.e., they may fail to find a solution although there exists one.

1.1. **Motivation.** Recently, Faghih and Bonakdarpour [14] proposed a sound and complete method to synthesize finite-state self-stabilizing systems based on SMT-solving. However, the shortcoming of this work as well as the techniques in [4, 10, 22] is that an *explicit* description of LS is needed as an input to the synthesis algorithm. The problem is that developing a formal predicate for legitimate states is not at all a straightforward task. For instance, the predicate for the set of legitimate states for Dijkstra's token ring algorithm with three-state machines [7] for three processes is the following:

$$\begin{aligned} LS = {} & ((x_0 + 1 \equiv_3 x_1) \ \wedge \ (x_1 + 1 \not\equiv_3 x_2)) \vee \\ & ((x_1 = x_0) \ \wedge \ (x_1 + 1 \not\equiv_3 x_2)) \vee \\ & ((x_1 + 1 \equiv_3 x_0) \ \wedge \ (x_1 + 1 \not\equiv_3 x_2)) \vee \\ & ((x_0 + 1 \not\equiv_3 x_1) \ \wedge \ (x_1 + 1 \not\equiv_3 x_0) \wedge (x_1 + 1 \equiv_3 x_2)) \end{aligned}$$

where $\equiv_3$ denotes modulo 3 equality and variable x_i belongs to process i. Obviously, developing such a predicate requires substantial expertise and insight and is, in fact, the key to the solution. Ideally, the designer should only express the basic requirements of the protocols (i.e., the existence of a unique token and its fair circulation), instead of an obscure predicate such as the one above.

1.2. **Contributions.** In this paper, we propose an automated approach to synthesize self-stabilizing systems given (1) the network topology, and (2) the high-level specification of legitimate states in the linear temporal logic (LTL) [25]. We also investigate automated synthesis of two important refinements of self-stabilization, namely *ideal stabilization* [24] and *monotonic stabilization* [28]. Ideally stabilizing protocols address two drawbacks of self-stabilizing protocols, namely exhibiting unpredictable behavior during recovery and poor compositional properties. In order to keep the specification as abstract as possible, the input LTL formula may include a set of uninterpreted predicates. In designing ideal-stabilizing systems, the transition relation of the system and interpretation function of uninterpreted predicates must be found such that the specification is satisfied in every state.

Monotonic stabilization [28] relates to the behavior of a self-stabilizing system during stabilization, as it mandates a participating processor to change its output at most once after a transient fault occurs. So, a legitimate state is reached after at most one output change at every process. Intuitively, monotonic stabilization prevents unnecessary oscillations during stabilization, and guarantees recovery in a monotonic way (the system always moves closer to a legitimate state). Generic approaches to monotonic stabilization [28] require huge memory and time resources as the monotonic stabilization layer is added to an existing protocol. Finding specific monotonically stabilizing protocols that are memory and time efficient is notoriously difficult, yet highly appealing. These difficulties further motivate the need for developing methods that can automatically synthesize distributed self-, ideal-, and monotonic-stabilizing protocols.

Our synthesis approach is inspired by bounded-synthesis [15], where we transform the input specification into a set of SMT constraints. If the SMT instance is satisfiable, then a witness solution to its satisfiability encodes a distributed protocol that meets the input specification and topology. If the instance is not satisfiable, then we are guaranteed that no

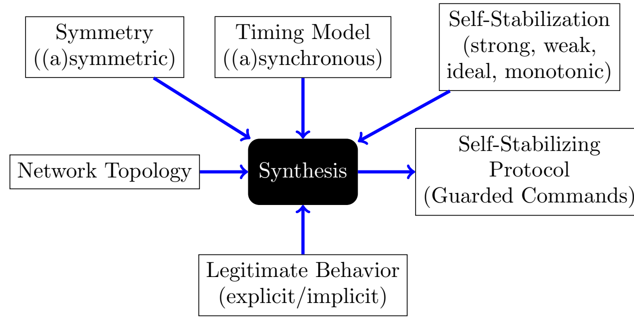

Figure 1: Input and output of our synthesis method.

protocol that satisfies the input specification exists. The inputs and output of our synthesis method are depicted in Fig. 1.

We also conduct several case studies using the model finder Alloy [19]. In the case of self-stabilizing systems, we successfully synthesize Dijkstra's [7] (three-state machine) token ring and Raymond's [26] mutual exclusion algorithms without explicit legitimate states as input. We also synthesize ideal-stabilizing leader election and local mutual exclusion (in a line topology) protocols, as well as monotonic-stabilizing distributed maximal independent set protocols and Grundy coloring.

Comparison to the conference version. A preliminary version of this article appeared in the 36th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE'16). This article extends the conference version as follows:

- We extend our synthesis approach to synthesize monotonic-stabilizing protocols.
- We conduct three case studies on synthesizing monotonic-stabilizing distributed maximal independent set, and Grundy coloring.

More precisely, Subsections 3.4, 6.3.4, and 7.3 are our added contributions.

Organization. In Sections 2 and 3, we present the preliminary concepts on the shared-memory model and self-stabilization. Section 4 formally states the synthesis problems. Formalization of timing models and symmetry in distributed programs are described in Section 5. In Section 6, we describe our SMT-based technique, while Section 7 is dedicated to our case studies. In Section 8, we respond to some of the questions often raised about this line of work. We discuss the related work in Section 9 and make concluding remarks Section 10.

2. Model of Computation

2.1. **Distributed Programs.** Throughout the paper, let V be a finite set of discrete *variables*. Each variable $v \in V$ has a finite domain D_v. A *state* is 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. We denote the value of a variable v in state s by $v(s)$.

Definition 2.1. *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 set of transitions of 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 2.1 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 2.2. *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:*

$$\forall s_0', s_1' : \Bigg(\Big(\forall v \in R_\pi : \big(v(s_0) = v(s_0') \ \wedge \ v(s_1) = v(s_1') \big) \Big) \wedge \Big(\forall v \notin R_\pi : v(s_0') = v(s_1') \Big) \Bigg) \implies (s_0', s_1') \in T_\pi \tag{2.1}$$

- *$T_\mathcal{D}$ is the set of transitions and is the union of transitions of all processes:*

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

Intuitively, the read restriction in Definition 2.2 imposes the constraint that for each process π, each transition in T_π depends only on reading the variables that π can read. Thus, each transition forms 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 entire corresponding group must be included (respectively, excluded) in $T_\mathcal{D}$ as well. Also, notice that $T_\mathcal{D}$ is defined in such a way that $\mathcal{D}$ resembles an *asynchronous* distributed program, where process transitions execute in an *interleaving* fashion.

Example. Let $V = \{x_0, x_1, x_2\}$ be the set of variables, where $D_{x_0} = D_{x_1} = D_{x_2} = \{0, 1, 2\}$. Let $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ be the distributed program, where $\Pi_\mathcal{D} = \{\pi_0, \pi_1, \pi_2\}$. Each process π_i $(0 \leq i \leq 2)$ can write variable x_i. Also, $R_{\pi_0} = \{x_0, x_1\}$, $R_{\pi_1} = \{x_0, x_1, x_2\}$, and $R_{\pi_2} = \{x_1, x_2\}$. Notice that following Definition 2.2 and read/write restrictions of π_0, the following (arbitrary) transitions

$$t_1 = \big([x_0 = 1, x_1 = 1, x_2 = 0], [x_0 = 2, x_1 = 1, x_2 = 0] \big)$$
$$t_2 = \big([x_0 = 1, x_1 = 1, x_2 = 2], [x_0 = 2, x_1 = 1, x_2 = 2] \big)$$

are in the same group, since π_0 cannot read x_2. This implies that if t_1 is included in the set of transitions of a distributed program, then so should be t_2. Otherwise, execution of t_1 by π_0 will depend on the value of x_2, which, of course, π_0 cannot read.

Definition 2.3. *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. □

2.2. **Predicates.** Let $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ be a distributed program over a set V of variables. The *global state space* of $\mathcal{D}$ is the set of all possible global states of $\mathcal{D}$:

$$\Sigma_\mathcal{D} = \prod_{v \in V} D_v$$

The *local state space* of a process $\pi \in \Pi_\mathcal{D}$ is the set of all possible local states of π, that is, the states that π can fully read:

$$\Sigma_\pi = \prod_{v \in R_\pi} D_v$$

Definition 2.4. *An* interpreted global predicate *of a distributed program* $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ *is a subset of* $\Sigma_\mathcal{D}$ *and an* interpreted local predicate *of a process* $\pi \in \Pi_\mathcal{D}$ *is a subset of* Σ_π. □

Definition 2.5. *Let* $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ *be a distributed program. An* uninterpreted global predicate *up is an uninterpreted Boolean function from* $\Sigma_\mathcal{D}$. *An* uninterpreted local predicate *lp is an uninterpreted Boolean function from* Σ_π*, for some* $\pi \in \Pi_\mathcal{D}$. □

The *interpretation* of an uninterpreted global predicate is a Boolean function from the set of all states:

$$up_\mathrm{I} : \Sigma_D \mapsto \{\mathsf{true}, \mathsf{false}\}$$

Similarly, the interpretation of an uninterpreted local predicate for a process π is a Boolean function:

$$lp_\mathrm{I} : \Sigma_\pi \mapsto \{\mathsf{true}, \mathsf{false}\}$$

Throughout the paper, we use 'uninterpreted predicate' to refer to either uninterpreted global or local predicate, and use 'global (local) predicate' to refer to interpreted global (local) predicate.

2.3. **Topology.** A topology specifies the communication model of a distributed program.

Definition 2.6. *A* topology *is a tuple* $\mathcal{T} = \langle V, |\Pi_\mathcal{T}|, R_\mathcal{T}, W_\mathcal{T} \rangle$*, where*
- V *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* $[0, |\Pi_\mathcal{T}| - 1] \mapsto 2^V$ *from a process index to its read-set, and*
- $W_\mathcal{T}$ *is a mapping* $[0, |\Pi_\mathcal{T}| - 1] \mapsto 2^V$ *from a process index to its write-set, such that* $W_\mathcal{T}(i) \subseteq R_\mathcal{T}(i)$*, for all* $i \in [0, |\Pi_\mathcal{T}| - 1]$. □

Definition 2.7. *A distributed program* $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ has *topology* $\mathcal{T} = \langle V, |\Pi_\mathcal{T}|, R_\mathcal{T}, W_\mathcal{T} \rangle$ *iff*
- *each process* $\pi \in \Pi_\mathcal{D}$ *is defined over* V
- $|\Pi_\mathcal{D}| = |\Pi_\mathcal{T}|$
- *there is a bijective function* $g : [0, |\Pi_\mathcal{T}| - 1] \mapsto \Pi_\mathcal{D}$ *such that*

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

3. Formal Characterization of Self-, Monotonic-, and Ideal-Stabilization

We specify the behavior of a distributed stabilizing program based on (1) the *functional* specification, and (2) the *recovery* specification. The functional specification is intended to describe what the program is required to do in a fault-free scenario (e.g., mutual exclusion or leader election). The recovery behavior stipulates Dijkstra's idea of self-stabilization in spite of distributed control [7].

3.1. **The Functional Behavior.**

We use LTL [25] to specify the functional behavior of a stabilizing program. Since LTL is a commonly-known language, we refrain from presenting its syntax and semantics ($\mathbf{F}$, $\mathbf{G}$, $\mathbf{X}$, and $\mathbf{U}$ denote the 'finally', 'globally', 'next', and 'until' operators, respectively). In our framework, an LTL formula may include uninterpreted predicates. Thus, we say that a program $\mathcal{D}$ satisfies an LTL formula φ from an initial state in the set I, and write $\mathcal{D}, I \models \varphi$ iff there exists an interpretation function for each uninterpreted predicate in φ, such that all computations of $\mathcal{D}$, starting from a state in I satisfy φ. Also, the semantics of the satisfaction relation is the standard semantics of LTL over Kripke structures (i,e., computations of $\mathcal{D}$ that start from a state in I).

Example 3.1. Consider the problem of *token passing* in a ring topology (i.e., token ring), where each process π_i has a variable x_i with the domain $D_{x_i} = \{0, 1, 2\}$. This problem has two functional requirements:

Safety: The *safety* requirement for this problem is that in each state, only one process has the token and, hence, can execute. To formulate this requirement, we assume that each process π_i is associated with a local uninterpreted predicate tk_i, which shows whether π_i is enabled. Let $LP = \{tk_i \mid 0 \leq i < n\}$. Thus, a process π_i can execute a transition, if and only if tk_i is true. The LTL formula, $\varphi_{\mathbf{TR}}$, expresses the above requirement for a ring of size n:

$$\varphi_{\mathbf{TR}} = \forall i \in [0, n-1] : tk_i \iff \big(\forall val \in \{0, 1, 2\} : (x_i = val) \Rightarrow \mathbf{X}\,(x_i \neq val)\big)$$

Using the set of uninterpreted predicates, the safety requirement can be expressed by the following LTL formula:

$$\psi_{\mathbf{safety}} = \exists i \in [0, n-1] : (tk_i \ \wedge\ \forall j \neq i : \neg tk_j)$$

Note that although safety requirements generally need the $\mathbf{G}$ operator, we do not need it, as every state in a stabilizing system can be an initial state.

Fairness: This requirement express that for every process π_i and starting from each state, the computation should reach a state, where π_i is enabled:

$$\psi_{\mathbf{fairness}} = \forall i \in [0, n-1] : (\mathbf{F}\, tk_i)$$

Another way to guarantee this requirement is that processes get enabled in a clockwise order in the ring, which can be formulated as follows:

$$\psi_{\mathbf{fairness}} = \forall i \in [0, n-1] : (tk_i \Rightarrow \mathbf{X}\, tk_{(i+1 \bmod n)})$$

Note that the latter approach is a stronger constraint, and would prevent us to synthesize bidirectional protocols, such as Dijkstra's three-state solution.

Thus, the functional requirements of the token ring protocol is

$$\psi_{\mathbf{TR}} = \psi_{\mathbf{safety}} \ \wedge \ \psi_{\mathbf{fairness}}$$

Observe that following Definition 2.3, $\psi_{\mathbf{TR}}$ ensures deadlock-freedom as well.

Example 3.2. Consider the problem of *local mutual exclusion* on a line topology, where each process π_i has a Boolean variable x_i. The requirements of this problem are as follows:

Safety: In each state, (1) at least one process is enabled, that is, (i.e., deadlock-freedom), and (2) no two neighboring processes are enabled (i.e., local mutual exclusion). To formulate these requirements, we associate with each process π_i a local uninterpreted predicate tk_i, which is true when π_i is enabled:

$$\varphi_{\mathbf{LME}} = \forall i \in [0, n-1] : tk_i \iff \Big((x_i \Rightarrow \mathbf{X} \neg x_i) \ \wedge \ (\neg x_i \Rightarrow \mathbf{X}\, x_i) \Big)$$

Thus, $LP = \{tk_i \ \mid \ 0 \leq i < n\}$ and the safety requirement can be formulated by the following LTL formula:

$$\psi_{\mathbf{safety}} = (\exists i \in [0, n-1] : tk_i) \ \wedge \ (\forall i \in [0, n-2] : \neg(tk_i \wedge tk_{(i+1)}))$$

Fairness: Each process is eventually enabled:

$$\psi_{\mathbf{fairness}} = \forall i \in [0, n-1] : (\mathbf{F}\, tk_i)$$

Thus, the functional requirement of the local mutual exclusion protocol is

$$\psi_{\mathbf{LME}} = \psi_{\mathbf{safety}} \ \wedge \ \psi_{\mathbf{fairness}}$$

3.2. **Self-Stabilization.** A *self-stabilizing system* [7] is one that always recovers a good behavior (typically, expressed in terms of a set of *legitimate states*), starting from any arbitrary initial state.

Definition 3.1. *A distributed program* $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ *is* self-stabilizing *for* LTL *functional specification* ψ *iff there exists a global predicate* LS [1] *(called the set of* legitimate states*), such that:*

- Functional behavior: $\mathcal{D}, LS \models \psi$
- Strong convergence: $\mathcal{D}, \Sigma_{\mathcal{D}} \models \mathbf{F}\, LS$
- Closure: $\mathcal{D}, \Sigma_{\mathcal{D}} \models (LS \Rightarrow \mathbf{X}\, LS)$ □

[1] LS is a set of states, which is implicitly specified in terms of an LTL formula. When it appears on the left-hand side of $\models$, it means that the formula on the right-hand side needs to hold in LS, rather than in the whole state-space Σ_D. When LS appears in right-hand side of $\models$, it is part of the formula, which we need to hold.

Notice that the strong convergence property ensures that starting from any state, any computation converges to a legitimate state of $\mathcal{D}$ within a finite number of steps. The closure property ensures that execution of the program is closed in the set of legitimate states. In the sequel, we will omit the state space $\Sigma_\mathcal{D}$ and LTL specification ψ, when they are clear from the context or they are irrelevant.

There exist several results on impossibility of distributed self-stabilization (e.g., in token circulation and leader election in anonymous networks [18]). Thus, weaker forms of stabilization have been introduced in the literature of distributed computing. One example is *weak-stabilizing* distributed programs [16], which is defined as follows.

Definition 3.2. *A distributed program* $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ *is* weak-stabilizing *for a set LS of* legitimate states *if and only if the following 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* $s_i \in LS$*.*[2]
- Functional behavior: *As defined in Definition 3.1*
- Closure: *As defined in Definition 3.1.* □

Notice that unlike strong 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.

3.3. **Ideal-Stabilization.** Self-stabilization does not predict program behavior during recovery, which may be undesirable for some applications. A simple way to integrate program behavior during recovery is to include it in the specification itself. This way, the protocol must ensure that every configuration in the specification is legitimate (so, the only recovery behaviors are those included in the specification). Such a protocol is called *ideal stabilizing* [24].

Definition 3.3. *Let* ψ *be an* LTL *specification and* $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ *be a distributed program. We say that* $\mathcal{D}$ *is* ideal stabilizing *for* ψ *iff* $\mathcal{D}, \Sigma_\mathcal{D} \models \psi$*.* □

The existence of ideal stabilizing protocols for LTL specifications (that only mandate legitimate states) is an intriguing question, as one has to find a "clever" set of transitions and an interpretation function for every uninterpreted predicate (if included in the specification), such that the system satisfies the specification at all times. Note that there is a specification for every system to which it ideally stabilizes [24], and that is the specification that includes all of the system computations ($\psi = \mathsf{true}$). In this paper, we do the reverse; meaning that getting an LTL specification ψ, we synthesize a distributed system that ideally stabilizes to ψ.

3.4. **Monotonic-Stabilization.** Monotonic stabilization [28] also relates to prescribing program behavior during recovery, as it requires every process to change its variable at most once after a transient fault occurs. This simple requirement induces desirable properties for fault recovery. For example, processes cannot go back and forth between states: once a variable has been changed, it remains so until a legitimate state is reached, improving stability while recovering.

[2] Observe that weak convergence cannot be expressed in LTL.

A generic approach to monotonic stabilization [28] is for each process to collect variable information at some distance that depends on the considered problem, and change its variable only if it is absolutely sure that it should do so. The space and time required to implement such a scheme is huge, even for relatively simple problems. In order to design a viable monotonically stabilizing protocol, a problem specific approach is necessary. This indeed makes manual design of monotonic-stabilizing protocols a tedious task.

Definition 3.4. *A distributed program* $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ *is* monotonic-stabilizing *iff*

- $\mathcal{D}$ *is self-stabilizing with some set* LS *of legitimate states, and*
- *for every (recovery) computation* $\overline{s} = s_0 s_1 \cdots s_n$ *of* $\mathcal{D}$*, where for all* $i \in [0, n-1]$*, we have* $s_i \in \neg LS$ *and* $s_n \in LS$*, the following holds:*

$$\exists j \in [0, n-1] : \exists \pi \in \Pi_\mathcal{D} : \exists v \in W_\pi : v(s_j) \neq v(s_{j+1}) \;\Rightarrow\; \forall k \in [0, n-1] - \{j\} : \forall v \in W_\pi : v(s_k) = v(s_{k+1})$$

□

4. Problem Statement

Our goal is to develop a synthesis algorithm that takes as input (1) system topology, and (2) two LTL formulas φ and ψ that involve a set LP of uninterpreted predicates, and generate as output a self-, monotonic-, or ideal-stabilizing protocol. For instance, in token passing on a ring, $\psi_{\mathbf{TR}}$ includes safety and fairness, which should hold in the set of legitimate states, while $\varphi_{\mathbf{TR}}$ is a general requirement that we specify on every uninterpreted predicate tk_i. Since in the case of self-stabilizing systems, we do not get LS as a set of states (global predicate), we refer to our problem as "synthesis of self-stabilizing systems with *implicit* LS".

Problem statement 1 (self/monotonic-stabilization). Given is
(1) a topology $\mathcal{T} = \langle V, |\Pi_\mathcal{T}|, R_\mathcal{T}, W_\mathcal{T} \rangle$;
(2) two LTL formulas φ and ψ that involve a set LP of uninterpreted predicates.

The synthesis algorithm is required to identify as output (1) a distributed program $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$, (2) an interpretation function for every local predicate $lp \in LP$, and (3) the global state predicate LS, such that $\mathcal{D}$ has topology $\mathcal{T}$, $\mathcal{D}, \Sigma_\mathcal{D} \models \varphi$, and $\mathcal{D}$ is self/monotonic-stabilizing for ψ.

Problem statement 2 (ideal-stabilization). Given is
(1) a topology $\mathcal{T} = \langle V, |\Pi_\mathcal{T}|, R_\mathcal{T}, W_\mathcal{T} \rangle$
(2) two LTL formulas φ and ψ that involve a set LP of uninterpreted predicates.

The synthesis algorithm is required to generate as output (1) a distributed program $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$, and (2) an interpretation function for every local predicate $lp \in LP$, such that $\mathcal{D}$ has topology $\mathcal{T}$ and $\mathcal{D}, \Sigma_\mathcal{D} \models (\varphi \wedge \psi)$.

5. Timing Models and Symmetry in Distributed Programs

We would like our synthesis solution to also take as input the timing model as well as symmetry requirements among processes. These constraints are defined in Subsections 5.1 and 5.2.

5.1. **Timing Models.** Two commonly-considered timing models in the literature of distributed computing are *synchronous* and *asynchronous* programs [23]. In an asynchronous distributed program, every transition of the program is a transition of one and only one of its processes (central daemon model).

Definition 5.1. *A distributed program $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ is* asynchronous *if and only if the following condition holds:*

$$ASYN = \forall (s_0, s_1) \in T_\mathcal{D} : \Big((\exists \pi \in \Pi_\mathcal{D} : (s_0, s_1) \in T_\pi) \vee \\ ((s_0 = s_1) \wedge \forall \pi \in \Pi_\mathcal{D} : \forall \mathfrak{s} : (s_0, \mathfrak{s}) \notin T_\pi) \Big) \quad (5.1)$$

Thus, the set of transitions of an asynchronous program is simply the union of transitions of all its 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 5.2. *A distributed program $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ is* synchronous *if and only if the following condition holds:*

$$SYN = \forall (s_0, s_1) \in T_\mathcal{D} : \forall \pi \in \Pi_\mathcal{D} : \\ \big(\exists \mathfrak{s} : ((s_0, \mathfrak{s}) \in T_\pi) \wedge \forall v \in W_\pi : v(s_1) = v(\mathfrak{s})\big) \vee \quad (5.2) \\ \big(\forall \mathfrak{s} : ((s_0, \mathfrak{s}) \notin T_\pi) \wedge \forall v \in W_\pi : v(s_0) = v(s_1)\big)$$

□

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.

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

Definition 5.3. *A distributed program $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ is called* symmetric *if and only if 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:*

$$SYM = \forall (s_0, s_1) \in T_\pi : \exists (s'_0, s'_1) \in T_{\pi'} : \\ \big(\forall v \in R_\pi : (v(s_0) = f(v)(s'_0))\big) \wedge \big(\forall v \in W_\pi : (v(s_1) = f(v)(s'_1))\big) \quad (5.3)$$

□

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.

6. SMT-based Synthesis Solution

Our technique is inspired by our SMT-based method in [14]. In particular, we transform the problem input into an *SMT instance*. 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. If so, then a witness generated by the SMT solver is the answer to our synthesis problem. We describe the SMT entities obtained in our transformation in Subsection 6.1. SMT constraints appear in Subsections 6.2- 6.3. Note that using our approach in [14], we can synthesize different systems considering types of timing models (i.e., synchronous and asynchronous), symmetric and asymmetric, as well as strong- and weak-stabilizing protocols.

6.1. **SMT Entities.** Recall that the inputs to our problems include a topology $\mathcal{T} = \langle V, |\Pi_\mathcal{T}|, R_\mathcal{T}, W_\mathcal{T}\rangle$, and two LTL formulas on a set LP of uninterpreted predicates. Let $D = \langle \Pi_\mathcal{D}, T_\mathcal{D}\rangle$ denote a distributed program that is a solution to our problem. 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 $Bool$ that contains the elements true and false.
- A set called S, whose cardinality is $\left|\prod_{v\in V} D_v\right|$. This set represents the state space of the synthesized distributed program.
- An uninterpreted function v_val for each variable v; i.e., $v_val : S \mapsto D_v$.
- An uninterpreted function lp_val for each uninterpreted predicate $lp \in LP$; i.e, $lp_val : S \mapsto Bool$.
- An uninterpreted relation $T_i \subseteq S \times S$ that represents the transition relation for process π_i in the synthesized program.
- An uninterpreted function γ, from each state to a natural number ($\gamma : S \mapsto \mathbb{N}$). This function is used to capture convergence to the set of legitimate states.
- An uninterpreted function $LS : S \mapsto Bool$.

The last two entities are only included in the case of Problem Statement 1.

Example. For Example 3.1, we include the following SMT entities:

- $D_{x_0} = D_{x_1} = D_{x_2} = \{0, 1, 2\}$, $Bool = \{\mathsf{true}, \mathsf{false}\}$, set S, where $|S| = 27$
- $x_0_val : S \mapsto D_{x_0}$, $x_1_val : S \mapsto D_{x_1}$, $x_2_val : S \mapsto D_{x_2}$
- $T_0 \subseteq S \times S$, $T_1 \subseteq S \times S$, $T_2 \subseteq S \times S$, $\gamma :\ S \mapsto \mathbb{N}$, $LS : S \mapsto Bool$

6.2. **General SMT Constraints.**

6.2.1. *State Distinction.* Any two states differ in the value of some variable:

$$\forall s, s' \in S \;:(s \neq s') \;\Rightarrow\; (\exists v \in V \;:\; v_val(s) \neq v_val(s')) \tag{6.1}$$

6.2.2. *Local Predicates Constraints.* Let LP be the set of uninterpreted predicates used in formulas φ and ψ. For each uninterpreted local predicate lp_π, we need to ensure that its interpretation function is a function of the variables in the read-set of π. To guarantee this requirement, for each $lp_\pi \in LP$, we add the following constraint to the SMT instance:

$$\forall s, s' \in S : \big(\forall v \in R_\pi : (v(s) = v(s')) \;\Rightarrow\; (lp_\pi(s) = lp_\pi(s'))\big)$$

Example. For Example 3.1, we add the following constraint for process π_1:

$$\begin{aligned}\forall s, s' \in S : \big((x_0(s) = x_0(s')) \wedge (x_1(s) = x_1(s')) \wedge (x_2(s) = x_2(s'))\big) \;\Rightarrow \\ (tk_1(s) = tk_1(s'))\end{aligned} \tag{6.2}$$

6.2.3. *Constraints for an Asynchronous System.* To synthesize an asynchronous distributed program, we add the following constraint for each transition relation T_i:

$$\forall (s, s') \in T_i \;: \forall v \notin W_{\mathcal{T}}(i) \;:\; v_val(s) = v_val(s') \tag{6.3}$$

Constraint 6.3 ensures that in each relation T_i, only process π_i can take a transition. By introducing $|\Pi_{\mathcal{T}}|$ transition relations, we consider all possible interleaving of processes taking transitions. Note that this constraint is a formulation of the third item in Definition 2.1.

6.2.4. *Read Restrictions.* To ensure that $\mathcal{D}$ meets the read restrictions given by $\mathcal{T}$ and Definition 2.2, we add the following constraint for each process index:

$$\begin{aligned}\forall (s_0, s_1) \in T_i :\; \forall s'_0, s'_1 :\; & \Big(\big(\forall v \in R_\pi : (v(s_0) = v(s'_0) \;\wedge\; v(s_1) = v(s'_1))\big) \wedge \\ & \big(\forall v \notin R_\pi : v(s'_0) = v(s'_1)\big)\Big) \Rightarrow (s'_0, s'_1) \in T_i\end{aligned} \tag{6.4}$$

6.3. **Specific SMT Constraints for Self- and Ideal-Stabilizing Problems.** Before presenting the constraints specific to each of our problem statements, we present the formulation of an LTL formula as an SMT constraint. We use this formulation to encode the ψ and φ formulas (given as input) as ψ_{SMT} and φ_{SMT}, and add them to the SMT instance.

6.3.1. *SMT Formulation of an* LTL *Formula.* SMT formulation of an LTL formula is presented in [15]. Below, we briefly discuss the formulation of LTL formulas without nested temporal operators. For formulas with nested operators, the formulation based on universal co-Büchi automata [15] needs to be applied.

SMT formulation of $\mathbf{X}$: A formula of the form $\boldsymbol{X}P$ is translated to an SMT constraint as below [3]:

$$\forall s, s' \in S \;:\; \forall i \in [0, |\Pi_{\mathcal{T}}| - 1] \;:\; (s, s') \in T_i \;\Rightarrow\; P(s') \tag{6.5}$$

[3]Note that for a formula P, $P(s)$ is acquired by replacing each variable v with $v(s)$.

SMT formulation of $\mathbf{U}$: Inspired by *bounded synthesis* [15], for each formula of the form $P\,\mathbf{U}\,Q$, we define an uninterpreted function $\gamma_i : S \mapsto \mathbb{N}$ and add the following constraints to the SMT instance:

$$\forall s, s' \in S \;: \forall i \in [0, |\Pi_{\mathcal{T}}| - 1] : \neg Q(s) \;\wedge\; (s, s') \in T_i \;\Rightarrow (P(s) \;\wedge\; \gamma_i(s') > \gamma_i(s)) \tag{6.6}$$

$$\forall s \in S \;:\; \neg Q(s) \;\Rightarrow\; \exists i \in [0, |\Pi_{\mathcal{T}}| - 1] \;:\; \exists s' \in S \;:\; (s, s') \in T_i \tag{6.7}$$

The intuition behind Constraints 6.6 and 6.7 can be understood easily. If we can assign a natural number to each state, such that along each outgoing transition from a state in $\neg Q$, the number is strictly increasing, then the path from each state in $\neg Q$ should finally reach Q or get stuck in a state, since the size of state space is finite. Also, there cannot be any loops whose states are all in $\neg Q$, as imposed by the annotation function. Finally, Constraint 6.7 ensures that there is no deadlock state in $\neg Q$ states.

6.3.2. *Synthesis of Self-Stabilizing Systems.* In this section, we present the constraints specific to synthesizing self-stabilizing systems.

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

$$\forall s, s' \in S \;:\; \forall i \in [0, |\Pi_{\mathcal{T}}| - 1] \;:\; (LS(s) \;\wedge\; (s, s') \in T_i) \;\Rightarrow\; LS(s') \tag{6.8}$$

Strong Convergence (SC): Similar to the constraints presented in Section 6.3.1, our SMT formulation for SC is a simplification of Constraints 6.6 and 6.7 (recall that $\mathbf{F}\,LS = \mathsf{true}\,\mathbf{U}\,LS$):

$$\forall s, s' \in S : \forall i \in \{0 \cdots |\Pi_{\mathcal{T}}| - 1\} : \neg LS(s) \;\wedge\; (s, s') \in T_i \;\Rightarrow\; \gamma(s') > \gamma(s) \tag{6.9}$$

$$\forall s \in S \;:\; \neg LS(s) \;\Rightarrow\; \exists i \in \{0 \cdots |\Pi_{\mathcal{T}}| - 1\} \;:\; \exists s' \in S \;:\; (s, s') \in T_i \tag{6.10}$$

General Constraints on Uninterpreted Predicates: As mentioned in Section 4, one of the inputs to our problem is an LTL formula, φ describing the role of uninterpreted predicates. Considering φ_{SMT} to be the SMT formulation of φ, we add the following SMT constraint to the SMT instance:

$$\forall s \in S \;:\; \varphi_{SMT} \tag{6.11}$$

Constraints on LS: Another input to our problem is the LTL formula ψ that includes requirements, which should hold in the set of legitimate states. We formulate this formula as SMT constraints using the method discussed in Section 6.3.1. Considering ψ_{SMT} to be the SMT formulation of the ψ formula, we add the following SMT constraint to the SMT instance:

$$\forall s \in S \;:\; LS(s) \;\Rightarrow\; \psi_{SMT} \tag{6.12}$$

Example. Continuing with Example 3.1, we add the following constraints to encode $\varphi_{\mathbf{TR}}$:

$$\forall s \in S \;:\; \forall i \in [0, n-1] \;:\; tk_i(s) \iff (\forall j \in [0, n-1] \;:\; j \neq i \;\Rightarrow\; \nexists s' \in S \;:\; (s, s') \in T_j)$$

Note that the asynchronous constraint does not allow change of x_i for T_j, where $j \neq i$. The other requirements of the token ring problem are $\psi_{\mathbf{safety}}$ and $\psi_{\mathbf{fairness}}$, which should hold in the set of legitimate states. To guarantee them, the following SMT constraints are added to the SMT instance:

$$\forall s \in S : LS(s) \;\Rightarrow\; (\exists i \in [0, n-1] : (tk_i(s) \;\wedge\; \forall j \neq i : \neg tk_j(s)))$$
$$\forall s \in S \;:\; LS(s) \;\Rightarrow\; \forall i \in [0, n-1] : (tk_i(s) \wedge (s, s') \in T_i) \Rightarrow tk_{(i+1 \bmod n)}(s')$$

6.3.3. *Synthesis of Ideal-Stabilizing Systems.* We now present the constraints specific to Problem Statement 2. The only such constraints is related to the two LTL formulas φ and ψ. To this end, we add the following to our SMT instance:

$$\forall s \in S \;:\; \varphi_{SMT} \wedge \psi_{SMT} \tag{6.13}$$

Example. We just present $\psi_{\mathbf{LME}}$ for Example 3.2, as $\varphi_{\mathbf{LME}}$ is similar to Example 3.1:

$$\forall s \in S \;:\Big((\exists i \in [0, |\Pi_{\mathcal{T}}|-1] : tk_i(s)) \;\wedge\; (\forall i \in [0, |\Pi_{\mathcal{T}}|-2] : \neg(tk_i(s) \wedge tk_{(i+1)}(s)))\Big)$$

$$\forall s, s' \in S \;:\; \forall i, j \in [0, |\Pi_{\mathcal{T}}|-1] \;:\; \neg tk_i(s) \;\wedge\; (s, s') \in T_j \implies \gamma_i(s') > \gamma_i(s)$$

$$\forall s \in S \;:\; \forall i \in \{0, |\Pi_{\mathcal{T}}|-1] \;:\; \neg tk_i(s) \implies \exists j \in [0, |\Pi_{\mathcal{T}}|-1] \;:\; \exists s' \in S \;:\; (s, s') \in T_j$$

Note that adding a set of constraints to an SMT instance is equivalent to adding their conjunction.

6.3.4. *Synthesis of Monotonic-Stabilizing Systems.* In order to synthesize a monotonic-stabilizing protocol, we need to add a constraint to guarantee that in each recovery path, each process executes at most once transition. In order to enforce this property, for each process π_i, we define a Boolean function

$$\mathit{flag}_i : S \mapsto \{\mathsf{true}, \mathsf{false}\}$$

and include the following constraint to the SMT instance:

$$\forall s, s' \in S \;:\; \forall i \in [0, |\Pi_{\mathcal{T}}|-1] \;: (\neg LS(s) \wedge (s, s') \in T_i) \implies (\mathit{flag}_i(s) \;\wedge\; \neg \mathit{flag}_i(s')) \tag{6.14}$$

$$\forall s \in S \;:\; \forall i, j \in \{0, \ldots, |\Pi_{\mathcal{T}}|-1\} \;: (\neg LS(s) \;\wedge\; i \neq j \;\wedge\; (s, s') \in T_j \wedge \neg \mathit{flag}_i(s)) \implies \neg \mathit{flag}_i(s') \tag{6.15}$$

The above two constraints guarantee that in every path starting from a state in $\neg LS$, each process executes at most once. This can easily be proved by contradiction. Assume that in the set of executions of the resulting protocol, there exists a recovery path from a state in $\neg LS$ to a state in LS, in which a process (assume W.L.O.G π_i) executes more than once. Based on Constraint 6.14, each time π_i executes a transition, the value of flag_i should change from true to false. First time, π_i executes, this change in the value of flag_i happens.

Also, based on Constraint 6.15, it is guaranteed that in the execution of any other process, $flag_i$ does not change. Now, based on Constraint 6.14, in the second execution of π_i, $flag_i$ should change from true to false. But we concluded that $flag_i$ is already set to false, and cannot be changed by the execution of any other process, which is a contradiction.

7. Case Studies and Experimental Results

We used the Alloy [19] model finder tool for our experiments. Alloy performs relational reasoning over quantifiers, which means that we did not have to unroll quantifiers over their domains. Note that Alloy may convert the model into a SAT instance. The results presented in this section are based on experiments on a machine with Intel Core i5 2.6 GHz processor with 8GB of RAM. We report our results in both cases of success and failure for finding a solution. Failure is due to the impossibility of self-, monotonic-, or ideal-stabilization for certain problems. All of our case studies are available within our tool Assess [11]. We present our results for self-, ideal-, and monotonic-stabilization in Sections 7.1, 7.2, and 7.3, respectively.

7.1. **Case Studies for Self-Stabilization.**

7.1.1. *Self-Stabilizing Token Ring.* In Example 3.1, 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. For example, in case of three processes for π_1, $R_{\mathcal{T}}(1) = \{x_0, x_1, x_2\}$ and $W_{\mathcal{T}}(1) = \{x_1\}$. Token possession and mutual exclusion constraints follow Example 3.1. Table 1 presents our results for different input settings. We present one of the solutions we found for the asynchronous strong stabilizing token ring problem in a ring of three processes [4]. First, we present the interpretation functions for the uninterpreted local predicates.

$$tk_0 \Leftrightarrow x_0 = x_2, \quad tk_1 \Leftrightarrow x_1 \neq x_0, \quad tk_2 \Leftrightarrow x_2 \neq x_1$$

The synthesized solution for transition relations for each process is the following:

$$\begin{array}{llcl}
\pi_0 : & (x_0 = x_2) & \rightarrow & x_0 := (x_0 + 1) \bmod 3 \\
\pi_1 : & (x_1 \neq x_0) & \rightarrow & x_1 := x_0 \\
\pi_2 : & (x_2 \neq x_1) & \rightarrow & x_2 := x_1
\end{array}$$

Note that our synthesized solution is identical to that of Dijkstra's k-state solution. We could not synthesize the three-state solution, as in this protocol, the token does not always circulate in one direction (it changes its circulation direction), but we have this constraint in $\psi_{\mathbf{fairness}}$, as presented in Example 3.1.

[4] We manually simplified the output of Alloy for presentation, although this task can be also automated.

# of Processes	Self-Stabilization	Timing Model	Symmetry	Time (sec)
3	strong	asynchronous	asymmetric	4.21
3	weak	asynchronous	asymmetric	1.91
4	strong	asynchronous	asymmetric	748.81
4	weak	asynchronous	asymmetric	100.03

Table 1: Results for synthesizing token ring.

# of Processes	Self-Stabilization	Timing Model	Time (sec)
3	strong	synchronous	0.84
4	strong	synchronous	16.07
4	weak	synchronous	26.8

Table 2: Results for synthesizing mutual exclusion on a tree (Raymond's algorithm).

7.1.2. *Mutual Exclusion in a Tree.* In the second case study, the processes form a directed rooted tree, and the goal is to design a self-stabilizing protocol, where at each state of LS, one and only one process is enabled. In this topology, each process π_j has a variable h_j with domain $\{i \mid \pi_i \text{ is a neighbor of } \pi_j\} \cup \{j\}$. The problem specification is the following:

Safety: We assume each process π_i is associated with an uninterpreted local predicate tk_i, which shows whether π_i is enabled. Thus, mutual exclusion is the following formula:

$$\psi_{\textbf{safety}} = \exists i \in [0, n-1] : (tk_i \ \wedge \ \forall j \neq i : \neg tk_j)$$

Fairness: Each process π_i is eventually enabled:

$$\psi_{\textbf{fairness}} = \forall i \in [0, n-1] : (\mathsf{F}\, tk_i)$$

The formula, $\psi_{\mathbf{R}}$ given as input is $\psi_{\mathbf{R}} = \psi_{\textbf{safety}} \ \wedge \ \psi_{\textbf{fairness}}$.

Using the above specification, we synthesized a synchronous self-stabilizing systems, which resembles Raymond's mutual exclusion algorithm on a tree [26]. Table 2 shows the experimental results. We present one of our solutions for token circulation on a tree, where there is a root with two leaves. The interpretation functions for the uninterpreted local predicates are as follows:

$$\forall i \ : \ tk_i \Leftrightarrow h_i = i$$

Another part of the solution is the transition relation. Assume π_0 to be the root process, and π_1 and π_2 to be the two leaves of the tree. Hence, the variable domains are $D_{h_0} = \{0, 1, 2\}$, $D_{h_1} = \{0, 1\}$, and $D_{h_2} = \{0, 2\}$. Fig. 2 shows the transition relation over states of the form (h_0, h_1, h_2) as well as pictorial representation of the tree and token, where the states in LS are shaded.

7.2. **Case Studies for Ideal-Stabilization.**

7.2.1. *Leader Election.* In leader election, a set of processes choose a leader among themselves. Normally, each process has a subset of states in which it is distinguished as the leader. In a legitimate state, exactly one process is in its leader state subset, whereas the states of all other processes are outside the corresponding subset.

We consider line and tree topologies. Each process has a variable c_i and we consider domains of size two and three to study the existence of an ideal-stabilizing leader election

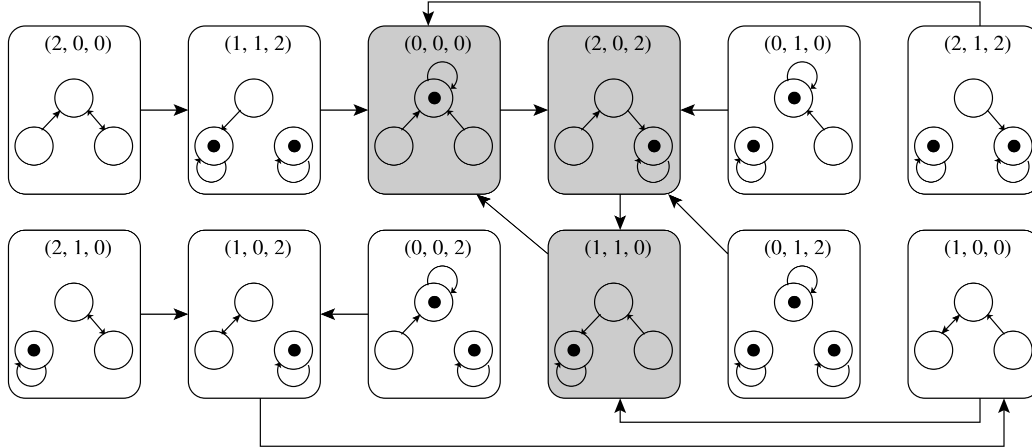

Figure 2: Self-stabilizing mutual exclusion in a tree of size 3 (Raymond's algorithm).

# of Proc.	Timing Model	Topology	Time (sec)
3	asynchronous	line/2-state	0.034
4	asynchronous	line/2-state	0.73
4	asynchronous	line/3-state	115.21
4	asynchronous	tree/2-state	0.63
4	asynchronous	tree/3-state	12.39

Table 3: Results for ideal-stabilizing leader election.

protocol. To synthesize such a protocol, we associate an uninterpreted local predicate l_i for each process π_i, whose value shows whether or not the process is the leader. Based on the required specification, in each state of the system, there is one and only one process π_i, for which $l_i = \mathsf{true}$:

$$\psi_{\mathbf{safety}} = \exists i \in [0, n-1] : (l_i \ \wedge \ \forall j \neq i : \neg l_j)$$

The results for this case study are presented in Table 3. In the topology column, the structure of the processes along with the domain of variables is reported. In the case of 4 processes on a line topology and tree/2-state, no solution is found. The time we report in the table for these cases are the time needed to report unsatisfiability by Alloy.

We present the asynchronous solution for the case of three processes on a line, where each process π_i has a Boolean variable c_i. Since the only specification for this problem is state-based (safety), there is no constraint on the transition relations, and hence, we only present the interpretation function for each uninterpreted local predicate l_i.

$$l_0 = (c_0 \wedge \neg c_1) \quad l_1 = (\neg c_0 \wedge \neg c_1) \vee (c_1 \wedge \neg c_2) \quad l_2 = (c_1 \wedge c_2)$$

7.2.2. *Local Mutual Exclusion.* Our next case study is local mutual exclusion, as discussed in Example 3.2. We consider a line topology in which each process π_i has a Boolean variable c_i. The results for this case study are presented in Table 4.

The solution we present for the local mutual exclusion problem corresponds to the case of asynchronous system with four processes on a ring. Note that for each process π_i, when tk_i is true, the transition T_i changes the value of c_i. Hence, having the interpretation

# of Proc.	Timing Model	Symmetry	Time (sec)
3	asynchronous	asymmetric	0.75
4	asynchronous	asymmetric	24.44

Table 4: Results for synthesizing ideal stabilizing local mutual exclusion.

functions of tk_i, the definition of transitions T_i are determined as well. Below, we present the interpretation functions of the uninterpreted local predicates tk_i.

$$tk_0 = (c_0 \wedge c_1) \vee (\neg c_0 \wedge \neg c_1)$$
$$tk_1 = (\neg c_0 \wedge c_1 \wedge c_2) \vee (c_0 \wedge \neg c_1 \wedge \neg c_2)$$
$$tk_2 = (\neg c_1 \wedge c_2 \wedge \neg c_3) \vee (c_1 \wedge \neg c_2 \wedge c_3)$$
$$tk_3 = (c_2 \wedge c_3) \vee (\neg c_2 \wedge \neg c_3)$$

7.3. **Case Studies for Monotonic-Stabilization.**

7.3.1. *Maximal Independent Set.* Given an undirected graph $G = (V, E)$, we say that $S \subseteq V$ is an independent set of G, if no two vertices in S share an edge in E. The set S is a maximal independent set (MIS), if it is not a proper subset of any other independent set. We use a similar topology as used in the literature [27]. Assuming processes to be the vertices of the graph, we consider a Boolean variable Ind_i for each process π_i. The value of Ind_i determines whether or not π_i is part of the independent set or not. A legitimate state is the one where processes with true values of their Ind variables form an independent set. For example, considering a ring of four processes, the set of legitimate states can be specified by the following predicate:

$$(Ind_0(s) \wedge \neg Ind_1(s) \wedge Ind_2(s) \wedge \neg Ind_3(s)) \vee$$
$$(\neg Ind_0(s) \wedge Ind_1(s) \wedge \neg Ind_2(s) \wedge Ind_3(s))$$

In this case study, our goal is to synthesize monotonic-stabilizing MIS protocols for ring topologies, where each process can read its own variable, as well as the variables of its neighbors, and can only write to its own variable. The results of this case study are presented in Table 5. The last column indicates whether or not Alloy is able to find a solution. Note that since our method is complete, unsatisfiability means that there exists no protocol satisfying the specified requirements. The following is the synthesized symmetric asynchronous protocol for the case of three processes in a ring topology. Note that in the case of symmetric protocol, all processes execute similarly.

$$\pi_i : \quad \neg Ind_i \wedge \neg Ind_l \wedge \neg Ind_r \rightarrow Ind_i := \mathsf{true}$$
$$Ind_i \wedge Ind_l \rightarrow Ind_i := \mathsf{false}$$

In the above synthesized protocol, r is the index of the right process, or $r = (i + 1) \mod 3$, and l is the index of the left process, or $l = (i - 1) \mod 3$. With a simple observation of the above synthesized protocol, we can see that in any path starting from a non-legitimate state, each process takes at most one action. We also present one of the solutions for the case of 4 processes in a ring topology.

# of Proc.	Timing Model	Symmetry	Time (sec)	Result
3	asynchronous	asymmetric	0.35	sat
3	asynchronous	symmetric	0.09	sat
3	synchronous	asymmetric	0.08	sat
4	asynchronous	asymmetric	1.35	sat
4	asynchronous	symmetric	0.9	unsat
5	asynchronous	asymmetric	6.33	sat
6	asynchronous	asymmetric	63.24	sat

Table 5: Results for monotonic stabilizing maximal independent set in ring.

$$
\begin{array}{lrcl}
\pi_0 : & Ind_0 \wedge Ind_3 & \rightarrow & Ind_0 := \mathsf{false} \\
\pi_1 : & \neg Ind_1 \wedge \neg Ind_2 & \rightarrow & Ind_1 := \mathsf{true} \\
\pi_2 : & Ind_2 \wedge Ind_3 & \rightarrow & Ind_2 := \mathsf{false} \\
 & Ind_2 \wedge \neg Ind_3 \wedge Ind_1 & \rightarrow & Ind_2 := \mathsf{false} \\
\pi_3 : & \neg Ind_3 \wedge \neg Ind_0 & \rightarrow & Ind_3 := \mathsf{true} \\
 & \neg Ind_3 \wedge Ind_0 \wedge \neg Ind_2 & \rightarrow & Ind_3 := \mathsf{true}
\end{array}
$$

7.3.2. *Maximal Independent Set in Unidirectional Rings.* Yamauchi and Tixeuil [28] state that monotonic stabilization requires additional information exchange between processes. In our second case study, we attempt to limit information exchange in maximal independent set and see whether we can still synthesize monotonic-stabilizing protocols for this problem. We considered unidirectional rings for this case study. In other words, each process can only read its own variable and the variable of its left process, and can write to its own variable. For example, for a ring of three processes, $R_{\pi_0} = \{Ind_0, Ind_2\}$ and $W_{\pi_0} = \{Ind_0\}$. The results for this case study are presented in Table 6. As can be seen, for the case of asymmetric asynchronous topologies, a protocol is found for rings of even size (4 and 6), but not for rings of odd size (3 and 5). Although, our solution is not general, but it can give an intuition to protocol designers for a general monotonic-stabilizing protocol to solve maximal independent in unidirectional rings. For the case of synchronous protocol with three processes, we synthesized the following solution:

$$
\begin{array}{lrcl}
\pi_0 : & \mathsf{true} & \rightarrow & Ind_0 := \mathsf{false} \\
\pi_1 : & \mathsf{true} & \rightarrow & Ind_1 := \mathsf{false} \\
\pi_2 : & \mathsf{true} & \rightarrow & Ind_2 := \mathsf{true}
\end{array}
$$

As can be simply observed, the synthesized topology takes every state (legitimate or non-legitimate) to one state $\langle \mathsf{false}, \mathsf{false}, \mathsf{true} \rangle$. A question that may raise for the reader is that why a similar protocol does not work in the case of asynchronous systems. The answer is that in the case of asynchronous systems, each step of the system is the execution of exactly one process, and hence, one execution of such a protocol may take the system from LS to a non-legitimate state (closure violation). For example, asynchronous execution of

# of Proc.	Timing Model	Symmetry	Time (sec)	Result
3	asynchronous	asymmetric	0.06	unsat
3	asynchronous	symmetric	0.05	unsat
3	synchronous	asymmetric	0.07	sat
4	asynchronous	asymmetric	0.74	sat
4	asynchronous	symmetric	0.44	unsat
5	asynchronous	asymmetric	8.73	unsat
6	asynchronous	asymmetric	85.14	sat

Table 6: Results for monotonic stabilizing maximal independent set in unidirectional ring.

the synthesized actions for the synchronous case will take the system from $\langle \mathsf{true}, \mathsf{false}, \mathsf{false} \rangle$, which is legitimate to $\langle \mathsf{false}, \mathsf{false}, \mathsf{false} \rangle$, which is non-legitimate (the first action is taken).

7.3.3. *Grundy Coloring.* Our third case study is the problem of Grundy coloring. Considering a graph $G = (V, E)$, and a coloring function

$$color : V \to [1, k]$$

a vertex $v \in V$ is called a *Grundy* node if

$$color(v) = \min \big\{ l \in [1, k] \mid \forall u : (v, u) \in E \implies (color(u) \neq l) \big\}$$

Simply speaking, v is colored with the smallest color not taken by any neighbor. A Grundy coloring for a graph is one in which every node is a Grundy node.

To synthesize a monotonic-stabilizing protocol for this problem, we consider a set of processes as the nodes of the graph, such that each process has a *color* variable. The designer can specify the domain of the *color* variables. Each process can read its own variable, and the variables of its neighbors, and can write to its own variable. The set of legitimate states are those, in which each process is a Grundy node. For example, considering a ring of 4 processes, where the domain of *color* variables is $\{1, 2, 3\}$, the set of legitimate states can be specified by the following predicate:

$$\begin{aligned}
\forall i \in \{0,1,2,3\} : \big(&color_i(s) \neq color_{(i+1 \mod 4)}(s)\big) \wedge \\
&\big(color_i(s) = 2 \implies (color_{(i+1 \mod 4)}(s) = 1 \vee color_{(i-1 \mod 4)}(s) = 1)\big) \wedge \\
&\big(color_i(s) = 3 \implies ((color_{(i+1 \mod 4)}(s) = 2 \vee color_{(i-1 \mod 4)}(s) = 2) \wedge \\
&\qquad (color_{(i+1 \mod 4)}(s) = 1 \vee color_{(i-1 \mod 4)}(s) = 1))\big)
\end{aligned}$$

Note that the last three lines of the above predicate are ensuring that the assigned color to each node is the minimum available one. Our results for this case study are presented in Table 7. Our synthesized protocol for the case of a symmetric protocol with three processes

# of Proc.	Timing Model	Topology	Symmetry	Time (sec)	Result
3	asynchronous	ring	asymmetric	3.02	sat
3	asynchronous	ring	symmetric	2.89	sat
3	synchronous	ring	asymmetric	4.29	sat
3	asynchronous	line	asymmetric	3.93	sat
4	asynchronous	ring	asymmetric	102.46	sat
4	asynchronous	ring	symmetric	152.69	unsat
4	asynchronous	line	asymmetric	144.16	sat

Table 7: Results for monotonic stabilizing Grundy coloring.

in a ring is the following:

$$\begin{array}{lrcl}
\pi_i \;: & (color_i = 1) \wedge (color_l \neq 2) \wedge (color_r = 1) & \rightarrow & color_i := 2 \\
& (color_i = 1) \wedge (color_l = 2) \wedge (color_r = 1) & \rightarrow & color_i := 3 \\
& (color_i = 3) \wedge (color_l = 3) \wedge (color_r \neq 2) & \rightarrow & color_i := 2 \\
& (color_i = 3) \wedge (color_l = 2) \wedge (color_r = 3) & \rightarrow & color_i := 1 \\
& (color_i = 2) \wedge (color_l = 2) \wedge (color_r = 3) & \rightarrow & color_i := 1 \\
& (color_i = 2) \wedge (color_l \neq 3) \wedge (color_r = 2) & \rightarrow & color_i := 3
\end{array}$$

In the above synthesized protocol, r is the index of the right process, or $r = (i+1) \mod 3$, and l is the index of the left process, or $l = (i-1) \mod 3$. Note that in this case, Grundy coloring is the same as the three-coloring problem [17].

We also present our synthesized model for the case of asynchronous protocol with 4 processes in a line topology:

$$\begin{array}{lrcl}
\pi_0 \;: & (color_0 \neq 1) \wedge (color_1 = 3) & \rightarrow & color_0 := 1 \\
& (color_0 = 3) \wedge (color_1 = 1) & \rightarrow & color_0 := 2 \\
\pi_1 \;: & (color_0 = 3) \wedge (color_1 = 2) & \rightarrow & color_1 := 3 \\
& (color_0 = 2) \wedge (color_1 = 2) & \rightarrow & color_1 := 1 \\
& (color_0 = 1) \wedge (color_1 \neq 3) \wedge (color_2 = 2) & \rightarrow & color_1 := 3 \\
& (color_0 = 1) \wedge (color_1 = 1) \wedge (color_2 \neq 2) & \rightarrow & color_1 := 3 \\
\pi_2 \;: & (color_1 = 3) \wedge (color_2 \neq 2) \wedge (color_3 = 1) & \rightarrow & color_2 := 2 \\
& (color_1 \neq 2) \wedge (color_2 = 1) \wedge (color_3 = 2) & \rightarrow & color_2 := 2 \\
& (color_1 \neq 3) \wedge (color_2 = 1) \wedge (color_3 = 1) & \rightarrow & color_2 := 2 \\
& (color_1 = 1) \wedge (color_2 = 3) \wedge (color_3 \neq 2) & \rightarrow & color_2 := 2 \\
\pi_3 \;: & (color_2 \neq 1) \wedge (color_3 \neq 1) & \rightarrow & color_3 := 1 \\
& (color_2 = 1) \wedge (color_3 = 3) & \rightarrow & color_3 := 1
\end{array}$$

8. Discussion

In this section, we address a few points often raised about this line of work.

8.1. **Applicability.** First, notice that a user of our technique has to give the network topology in terms of read/write restrictions as input. This in turn means that the user has to choose a set of variables and their domains. Although choosing variables and their domains may seem challenging, in most cases, the user can have educated guesses about the variables and their domains from the specification (similar to programming practices), as in the maximal matching example. This process may involve some trial and error though. For example, one can start by assigning each process a Boolean variable, and if no solution is found, increase the number of variables or their domains (increase the local state space of each process).

8.2. **Scalability.** It is obvious that scalability is an issue in this method. However, note that although our case studies deal with synthesizing a small number of processes (due to the high complexity of synthesis), having access to a solution for a small number of processes may 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 [20], and we can theoretically prove that the solution works for any number of processes. Also, in cases, where we find that there is no solution for the problem, this may be a hint for a general impossibility result.

One way to improve scalability is by using a counterexample-guided inductive synthesis loop, where an over-approximation is quickly synthesized and then later refined by identifying counterexamples.

8.3. **The Choice of SMT-solver.** It is noteworthy to mention that we have conducted experiments using Z3 [5] and Yices [9] SMT solvers as well, and in the majority of our cases studies, Alloy was the fastest model solver. 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. Note that the maximum number reported in this paper is the maximum number of processes we could find a solution for each case study in less than an hour.

8.4. **Synthesis under Worst-Case Recovery Time Constraint.** There are quantitative metrics in stabilizing systems that are as crucial as closure and convergence in practice. One of these metrics is *recovery time*, which is essentially the length of the path starting in a state in $\neg LS$ and ending in a state in LS. Recovery time is crucial in designing stabilizing systems for some applications, such as in network protocols. In such applications, it may not be desirable for the recovery time to exceed a specific number of steps, say w. Thus, we can include a constraint based on the "worst-case recovery time" to our model. The constraint is the following:

$$\forall s \in S : (0 \leq \gamma(s)) \wedge (\gamma(s) \leq w) \tag{8.1}$$

Based on the above constraint, since γ is incremented in every step, and its range is w, the worst-case recovery time in the synthesized system cannot exceed w.

9. Related Work

9.1. **Bounded Synthesis.** In bounded synthesis [15], given is a set of LTL properties, a system architecture, and a set of bounds on the size of process implementations and their composition. The goal is to synthesize an implementation for each process, such that their composition satisfies the given specification. The properties 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 and also the specification of legitimate states. For other constraints, such as the ones for synthesis of weak convergence, asynchronous and symmetric systems, we used a different approach from bounded synthesis. 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 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.

9.2. **Synthesis of Self-Stabilizing Systems.** In [21], the authors show that adding strong convergence is NP-complete in the size of the state space, which itself is exponential in the number 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. A complete synthesis technique for self-stabilizing systems is introduced in [22]. The limitations of this work compared to ours is: (1) unlike the approach in [22], we do not need the explicit description of the set of legitimate states, and (2) the method in [22] needs the set of actions on the underlying variables in the legitimate states. We also emphasize that although our experimental results deal with small numbers of processes, our approach can give key insights to designers of self-stabilizing protocols to generalize the protocol for any number of processes [20].

Another line of research is the work in [2]. The authors in this paper also introduce a technique to synthesize self-stabilizing protocols based on bounded synthesis, but their main focus is on Byzantine failures. To this end, they use a counterexample-guided inductive synthesis loop for networks of fixed size.

9.3. **Automated Addition of Fault-Tolerance.** The proposed algorithm in [4] 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 [4] is not complete. Bonakdarpour and Kulkarni studied the complexity of synthesizing timed multi-phased fault recovery in [3]. Finally, we introduced efficient symbolic heuristics for timed multi-phase recovery in [12].

10. Conclusion

In this paper, we proposed an automated SMT-based technique for synthesizing self-, ideal-, and monotonic-stabilizing algorithms. The required input to our approach is a high-level specification of the algorithm, given in the linear temporal logic (LTL) and the network topology. In the particular case of self-stabilization, this means that the detailed description of the set of legitimate states is not required. This relaxation is significantly beneficial, as developing a detailed predicate for legitimate states can be a tedious task. Our approach is sound and complete for finite-state systems; i.e., it ensures correctness by construction and if it cannot find a solution, we are guaranteed that there does not exist one. We demonstrated the effectiveness of our approach by automatically synthesizing Dijkstra's token ring, Raymond's mutual exclusion, and ideal-stabilizing leader election and local mutual exclusion algorithms as well as monotonic-stabilizing maximal independent set and Grundy coloring.

We note that our approach can be easily extended to incorporate additional properties of self-stabilizing systems. For instance, one can impose a worst-case recovery time constraint by putting an upperbound on the number of recovery steps. This can be simply achieved by including a constraint on the γ function (i.e., Constraint 8.1).

For future, 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 note that parameterized synthesis of distributed systems, when there is a cut-off point is studied in [20]. Our goal is to study parameterized synthesis for self-stabilizing systems, and we plan to propose a general method that works not just for cases with cut-off points. We would also like to investigate the application of techniques such as counterexample-guided inductive synthesis to improve the scalability of the synthesis process.

11. Acknowledgments

This research was supported in part by Canada NSERC Discovery Grant 418396-2012 and NSERC Strategic Grant 430575-2012. We would also like to acknowledge the anonymous referees who carefully reviewed the article and provided us with many constructive comments.

References

[1] F. Abujarad and S. S. Kulkarni. Multicore constraint-based automated stabilization. In *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 47–61, 2009.

[2] R. Bloem, N. Braud-Santoni, and Swen Jacobs. Synthesis of self-stabilising and byzantine-resilient distributed systems. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV)*, pages 157–176, 2016.

[3] B. Bonakdarpour and S. S. Kulkarni. Synthesizing bounded-time 2-phase fault recovery. *Formal Aspects of Computing*, 27(1):1–31, 2015.

[4] 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.

[5] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[6] Murat Demirbas and Anish Arora. Specification-based design of self-stabilization. *IEEE Transactions on Parallel Distributed Systems*, 27(1):263–270, 2016.

[7] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
[8] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, 1986.
[9] Bruno Dutertre. Yices 2.2. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, pages 737–744, 2014.
[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] F. Faghih and B. Bonakdarpour. Assess: A tool for automated synthesis of distributed self-stabilizing algorithms. In *Proceedings of the 19th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2017. To appear.
[12] F. Faghih and B. Bonakdarpour. Symbolic synthesis of timed models with strict 2-phase fault recovery. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, to appear.
[13] F. Faghih, B. Bonakdarpour, S. Tixeuil, and S. Kulkarni. Specification-based synthesis of distributed self-stabilizing protocols. In *Proceedings of the International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, pages 124–141, 2016.
[14] Fathiyeh Faghih and Borzoo Bonakdarpour. SMT-based synthesis of distributed self-stabilizing systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(3):21, 2015.
[15] B. Finkbeiner and S. Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer (STTT)*, 15(5-6):519–539, 2013.
[16] M. G. Gouda. The theory of weak stabilization. In *Proceedings of the 5th Workshop on Self-Stabilizing Systems*, pages 114–123, 2001.
[17] 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.
[18] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.
[19] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press Cambridge, 2012.
[20] S. Jacobs and R. Bloem. Parameterized synthesis. *Logical Methods in Computer Science*, 10(1), 2014.
[21] A. Klinkhamer and A. Ebnenasir. On the complexity of adding convergence. In *Proceedings of the 5th IPM International Conference on Fundamentals of Software Engineering (FSEN)*, pages 17–33, 2013.
[22] A. Klinkhamer and A. Ebnenasir. Synthesizing self-stabilization through superposition and backtracking. In *Proceedings of the 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 252–267, 2014.
[23] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.
[24] Mikhail Nesterenko and Sébastien Tixeuil. Ideal stabilisation. *IJGUC*, 4(4):219–230, 2013.
[25] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.
[26] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.
[27] S. K. Shukla, D. J. Rosenkrantz, and S. S. Ravi. Observations on self-stabilizing graph algorithms for anonymous networks. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 1–15, 1995.
[28] Y. Yamauchi and S. Tixeuil. Monotonic stabilization. In *Proceedings of the 14th International Conference on Principles of Distributed Systems (OPODIS)*, pages 475–490, 2010.

This work is licensed under the Creative Commons Attribution-NoDerivs License. To view a copy of this license, visit https://creativecommons.org/licenses/by-nd/4.0/ or send a letter to Creative Commons, 171 Second St, Suite 300, San Francisco, CA 94105, USA, or Eisenacher Strasse 2, 10777 Berlin, Germany