# Specification-Based Synthesis of Distributed Self-Stabilizing Protocols

Fathiyeh Faghih, Borzoo Bonakdarpour, Sébastien Tixeuil, Sandeep Kulkarni



# Specification-based Synthesis of Distributed Self-Stabilizing Protocols

Fathiyeh Faghih[1], Borzoo Bonakdarpour[1], Sébastien Tixeuil[2], and Sandeep Kulkarni[3]

[1] McMaster University, Canada, Email: {faghihef,borzoo}@mcmaster.ca
[2] UPMC Sorbonne Universités, France, Email: Sebastien.Tixeuil@lip6.fr
[3] Michigan State University, USA, Email: 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 the 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 also extend our approach to synthesize *ideal-stabilizing* protocols, where every state is legitimate. Our proposed methods are implemented and we report successful synthesis of Dijkstra's token ring and a self-stabilizing version of Raymond's mutual exclusion algorithm, as well as ideal-stabilizing leader election and local mutual exclusion.

## 1 Introduction

*Self-stabilization* [4] 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, and (2) *closure* indicates that the system remains in $LS$ thereafter.

As Dijkstra mentions in his belated proof of self-stabilization [5], designing self-stabilizing systems is a complex task, but proving their correctness is even more tedious. Thus, having access to automated methods (as opposed to manual techniques such as [3]) for *synthesizing* correct self-stabilizing systems is highly desirable. However, synthesizing self-stabilizing protocols incurs high time and space complexity [12]. The techniques proposed in [1, 2, 6, 13] 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.

Recently, Faghih and Bonakdarpour [7] 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 [2,6,13] 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 [4] 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 huge 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.

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). We also investigate automated synthesis of *ideal-stabilizing* protocols [14]. These 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 implicit 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. Our synthesis approach is SMT-based; i.e., 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 protocol that satisfies the input specification exists.

We also conduct several case studies using the model finder Alloy [10]. In the case of self-stabilizing systems, we successfully synthesize Dijkstra's [4] token ring and Raymond's [16] 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.

*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. In Section 5, we describe our SMT-based technique, while Section 6 is dedicated to our case studies. We discuss the related work in Section 7, and finally, we make concluding remarks and discuss future work in Section 8.

## 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 program 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 1.** *A* process $\pi$ *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* $\pi$*; i.e., variables that* $\pi$ *can read,*
- $W_\pi \subseteq R_\pi$ *is the* write-set *of* $\pi$*; i.e., variables that* $\pi$ *can write, and*
- $T_\pi$ *is the set of transitions of* $\pi$*, 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:*

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

- $T_\mathcal{D}$ *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 imposes the constraint that for each process $\pi$, each transition in $T_\pi$ depends only on reading the variables that $\pi$ can read. Thus, each transition is 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 $T_\mathcal{D}$ 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* We use the problem of distributed self-stabilizing *mutual exclusion* as a running example to describe the concepts throughout the paper. Let $V = \{c_0, c_1, c_2\}$ be the set of variables, where $D_{c_0} = D_{c_1} = D_{c_2} = \{0, 1, 2\}$. Let $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ be a distributed program, where $\Pi_\mathcal{D} = \{\pi_0, \pi_1, \pi_2\}$. Each process $\pi_i$ $(0 \leq i \leq 2)$ can write variable $c_i$. Also, $R_{\pi_0} = \{c_0, c_1\}$, $R_{\pi_1} = \{c_0, c_1, c_2\}$, and $R_{\pi_2} = \{c_1, c_2\}$. Notice that following Definition 2 and read/write restrictions of $\pi_0$, (arbitrary) transitions

$$t_1 = ([c_0 = 1, c_1 = 1, c_2 = 0], [c_0 = 2, c_1 = 1, c_2 = 0])$$
$$t_2 = ([c_0 = 1, c_1 = 1, c_2 = 2], [c_0 = 2, c_1 = 1, c_2 = 2])$$

are in the same group, since $\pi_0$ cannot read $c_2$. 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 $\pi_0$ depends on the value of $c_2$, which, of course, $\pi_0$ cannot read.

**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. □

### 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 $\pi \in \Pi_{\mathcal{D}}$ is the set of all possible local states of $\pi$: $\Sigma_\pi = \prod_{v \in R_\pi} D_v$.

**Definition 4.** *An* interpreted global predicate *of a distributed program $\mathcal{D}$ is a subset of $\Sigma_{\mathcal{D}}$ and an* interpreted local predicate *is a subset of $\Sigma_\pi$, for some $\pi \in \Pi_{\mathcal{D}}$.* □

**Definition 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 $\Sigma_\pi$, for some $\pi \in \Pi_{\mathcal{D}}$.* □

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

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

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

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

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 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* $\{0 \ldots |\Pi_{\mathcal{T}}| - 1\} \mapsto 2^V$ *from a process index to its read-set,*
– $W_{\mathcal{T}}$ *is a mapping* $\{0 \ldots |\Pi_{\mathcal{T}}| - 1\} \mapsto 2^V$ *from a process index to its write-set, such that* $W_{\mathcal{T}}(i) \subseteq R_{\mathcal{T}}(i)$*, for all* $i$ $(0 \leq i \leq |\Pi_{\mathcal{T}}| - 1)$. □

**Definition 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 mapping* $g : \{0 \ldots |\Pi_{\mathcal{T}}| - 1\} \mapsto \Pi_{\mathcal{D}}$ *such that*

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

□

## 3 Formal Characterization of Self- and Ideal-Stabilization

We specify the behavior of a distributed self-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 [4].

### 3.1 The Functional Behavior

We use LTL [15] to specify the functional behavior of a stabilizing program. Since LTL is a commonly-known language, we refrain from presenting its syntax and semantics and continue with our running example (where $\mathbf{F}$, $\mathbf{G}$, $\mathbf{X}$, 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 $\varphi$ 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 $\varphi$, such that all computations of $\mathcal{D}$, starting from a state in $I$ satisfy $\varphi$. 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 $\pi_i$ has a variable $c_i$ with the domain $D_{c_i} = \{0, 1, 2\}$. This problem has two functional requirements:

**Safety** The *safety* requirement for this problem is that in each state, only one process can execute. To formulate this requirement, we assume each process $\pi_i$ is associated with a local uninterpreted predicate $tk_i$, which shows whether $\pi_i$ is enabled. Let $LP = \{tk_i \mid 0 \leq i < n\}$. A process $\pi_i$ 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 $n$:

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

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

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

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 implies that for every process $\pi_i$ and starting from each state, the computation should reach a state, where $\pi_i$ is enabled:

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

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

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

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 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 $\pi_i$ has a Boolean variable $c_i$. The requirements of this problem are as follows:

**Safety** In each state, *(i)* at least one process is enabled (*i.e.*, deadlock-freedom), and *(ii)* no two neighbors are enabled (*i.e.*, mutual exclusion). To formulate this requirement, we associate with each process $\pi_i$ a local uninterpreted predicate $tk_i$, which is true when $\pi_i$ is enabled:

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

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

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

**Fairness** Each process is eventually enabled:

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

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* [4] 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 8.** *A distributed program* $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ *with the state space* $\Sigma_{\mathcal{D}}$ *is* self-stabilizing *for an* LTL *specification* $\psi$ *iff there exists a global predicate* $LS$ *(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: $\mathcal{D}, \Sigma_{\mathcal{D}} \models (LS \Rightarrow \mathbf{X}\, LS)$ □

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.

### 3.3 Ideal-Stabilization

Self-stabilization does not predict program behavior during recovery, which may be undesirable for some applications. A trivial way to integrate program behavior during recovery is to include it in the specification itself, then 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 *ideal stabilizing* [14].

**Definition 9.** *Let* $\psi$ *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* $\psi$ *iff* $\mathcal{D}, \Sigma_{\mathcal{D}} \models \psi$. □

The existence of ideal stabilizing protocols for "classical" specifications (that only mandate legitimate states) is an intriguing question, as one has to find a "clever" transition predicate and an interpretation function for every uninterpreted predicate (if included in the specification), such that the system satisfies the specification. Note that there is a specification for every system to which it ideally stabilizes [14], and that is the specification that includes all of the system computations. In this paper, we do the reverse; meaning that getting a specification $\psi$, we synthesize a distributed system that ideally stabilizes to $\psi$.

## 4 Problem Statement

Our goal is to develop synthesis algorithms that take as input the (1) system topology, and (2) two LTL formulas $\varphi$ and $\psi$ that involve a set $LP$ of uninterpreted predicates, and generate as output a self- 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-stabilization).** Given is

1. a topology $\mathcal{T} = \langle V, |\Pi_{\mathcal{T}}|, R_{\mathcal{T}}, W_{\mathcal{T}} \rangle$;
2. two LTL formulas $\varphi$ and $\psi$ 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-stabilizing for $\psi$.

**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 $\varphi$ and $\psi$ 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 SMT-based Synthesis Solution

Our technique is inspired by our SMT-based work in [7]. 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 the witness generated by the SMT solver is the answer to our synthesis problem. We describe the SMT entities obtained in our transformation in Subsection 5.1. SMT constraints appear in Subsections 5.2- 5.3. Note that using our approach in [7], 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. In a weak-stabilizing protocol there is only the *possibility* of recovery [9].

### 5.1 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, $lp\_val : S \mapsto Bool$.
- A relation $T_i$ that represents the transition relation for process $\pi_i$ in the synthesized program.
- An uninterpreted function $\gamma$, 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:

- $D_{c_0} = D_{c_1} = D_{c_2} = \{0, 1, 2\}$, $Bool = \{true, false\}$, set $S$, where $|S| = 27$
- $c_0\_val : S \mapsto D_{c_0}$, $c_1\_val : S \mapsto D_{c_1}$, $c_2\_val : S \mapsto D_{c_2}$
- $T_0 \subseteq S \times S$, $T_1 \subseteq S \times S$, $T_2 \subseteq S \times S$ , $\gamma : \ S \mapsto \mathbb{N}$ , $LS \ : \ S \mapsto Bool$

### 5.2 General SMT Constraints

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

$$\forall s_0, s_1 \in S \ : (s_0 \neq s_1) \ \Rightarrow \ (\exists v \in V \ : \ v\_val(s_0) \neq v\_val(s_1)) \tag{2}$$

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

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

*Example* For Example 3.1, we add the following constraint for process $\pi_1$:

$$\forall s, s' \in S : (x_0(s) = x_0(s')) \wedge (x_1(s) = x_1(s')) \wedge (x_2(s) = x_2(s')) \ \Rightarrow \ (tk_1(s) = tk_1(s')) \tag{3}$$

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

$$\forall (s_0, s_1) \in T_i \ : \forall v \notin W_{\mathcal{T}}(i) \ : \ v\_val(s_0) = v\_val(s_1) \tag{4}$$

Constraint 4 ensures that in each relation $T_i$, only process $\pi_i$ can execute. By introducing $|\Pi_{\mathcal{T}}|$ transition relations, we consider all possible interleaving of processes executions.

**5.2.4 Read Restrictions** To ensure that $\mathcal{D}$ meets the read restrictions given by $\mathcal{T}$ and Definition 2, we add the following constraint for each process index:

$$\forall (s_0, s_1) \in T_i : \forall s'_0, s'_1 : ((\forall v \in R_\pi : (v(s_0) = v(s'_0) \ \wedge \ v(s_1) = v(s'_1))) \wedge (\forall v \notin R_\pi : v(s'_0) = v(s'_1))) \Rightarrow (s'_0, s'_1) \in T_i \tag{5}$$

### 5.3 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 $\psi$ and $\varphi$ formulas (given as input) as $\psi_{SMT}$ and $\varphi_{SMT}$, and add them to the SMT instance.

**5.3.1 SMT Formulation of an LTL Formula** SMT formulation of an LTL formula is presented in [8]. 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 [8] needs to be applied.

*SMT formulation of* $\mathbf{X}$*:* A formula of the form $\mathbf{X}\,P$ is translated to an SMT constraint as below [4]:

$$\forall s, s' \in S \ : \ \forall i \in \{0, \ldots, |\Pi_\mathcal{T}| - 1\} \ : \ (s, s') \in T_i \ \Rightarrow \ P(s') \tag{6}$$

*SMT formulation of* $\mathbf{U}$*:* Inspired by *bounded synthesis* [8], 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:

$$\forall s, s' \in S \ : \forall i \in \{0, \ldots, |\Pi_\mathcal{T}| - 1\} \ : \ \neg Q(s) \ \wedge \ (s, s') \in T_i \ \Rightarrow (P(s) \ \wedge \ \gamma_i(s') > \gamma_i(s)) \tag{7}$$

$$\forall s \in S \ : \ \neg Q(s) \ \Rightarrow \ \exists i \in \{0, \ldots, |\Pi_\mathcal{T}| - 1\} \ : \ \exists s' \in S \ : \ (s, s') \in T_i \tag{8}$$

The intuition behind Constraints 7 and 8 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 8 ensures that there is no deadlock state in $\neg Q$ states.

**5.3.2 Synthesis of Self-Stabilizing Systems** In this section, we present the constraints specific to Problem Statement 1.

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

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

$$\forall s, s' \in S \ : \ \forall i \in \{0 \cdots |\Pi_{\mathcal{T}}| - 1\} \ : \ (LS(s) \ \wedge \ (s, s') \in T_i) \ \Rightarrow \ LS(s') \tag{9}$$

*Strong Convergence (SC):* Similar to the constraints presented in Section 5.3.1, our SMT formulation for *SC* is an adaptation of the concept of *bounded synthesis* [8]. The two following constraints ensure strong self-stabilization in the resulting model:

$$\forall s, s' \in S : \forall i \in \{0 \cdots |\Pi_{\mathcal{T}}| - 1\} : \neg LS(s) \ \wedge \ (s, s') \in T_i \ \Rightarrow \ \gamma(s') > \gamma(s) \tag{10}$$

$$\forall s \in S \ : \ \neg LS(s) \ \Rightarrow \ \exists i \in \{0 \cdots |\Pi_{\mathcal{T}}| - 1\} \ : \ \exists s' \in S \ : \ (s, s') \in T_i \tag{11}$$

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

$$\forall s \in S \ : \ \varphi_{SMT} \tag{12}$$

*Constraints on LS:* Another input to our problem is the LTL formula, $\psi$ 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 5.3.1. Considering $\psi_{SMT}$ to be the SMT formulation of the $\psi$ formula, we add the following SMT constraint to the SMT instance:

$$\forall s \in S \ : \ LS(s) \ \Rightarrow \ \psi_{SMT} \tag{13}$$

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

$$\forall s \in S \ : \ \forall i \in \{0 \cdots n-1\} \ : \ tk_i(s) \iff (\forall j \in \{0 \cdots n-1\} \ : \ j \neq i \ \Rightarrow \ \nexists s' \in S \ : \ (s, s') \in T_j)$$

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:

$$\forall s \in S : LS(s) \ \Rightarrow \ (\exists i \in \{0 \cdots n-1\} : (tk_i(s) \ \wedge \ \forall j \neq i : \neg tk_j(s)))$$

$$\forall s \in S \ : \ LS(s) \ \Rightarrow \ \forall i \in \{0 \cdots n-1\} : (tk_i(s) \wedge (s, s') \in T_i) \Rightarrow tk_{(i+1 \bmod n)}(s')$$

**5.3.3 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 $\varphi$ and $\psi$. To this end, we add the following to our SMT instance:

$$\forall s \in S \ : \ \varphi_{SMT} \wedge \psi_{SMT} \tag{14}$$

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

$$\forall s \in S \ : (\exists i \in \{0 \cdots n-1\} : (tk_i(s) \ \wedge \ \forall j \neq i : \neg tk_j(s)))$$

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

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

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

## 6 Case Studies and Experimental Results

We used the Alloy [10] model finder tool for our experiments. Alloy performs the relational reasoning over quantifiers, which means that we did not have to unroll quantifiers over their domains. 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 normally due to the impossibility of self- or ideal-stabilization for certain problems.

### 6.1 Case Studies for Synthesis of Self-Stabilizing Systems

**6.1.1 Self-stabilizing Token Ring** Synthesizing a self-stabilizing system for Example 3.1 leads to automatically obtaining Dijkstra [4] *three-state* algorithm in a bi-directional ring. Each process $\pi_i$ 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 $\pi_1$, $R_{\mathcal{T}}(1) = \{x_0, x_1, x_2\}$ 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. In the symmetric cases, we synthesized protocols with symmetric middle (not top nor bottom) processes. We present one of the solutions we found for the token ring problem in ring of three processes [5]. First, we present the interpretation functions for the uninterpreted local predicates.

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

Next, we present the synthesized transition relations for each process:

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

Note that our synthesized solution is similar to Dijkstra's $k$-state solution.

[5] 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 | 71.19 |
| 4 | weak | asynchronous | asymmetric | 73.55 |
| 4 | strong | asynchronous | symmetric | 178.6 |

**Table 1.** Results for synthesizing Dijkstra's three-state 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).

**6.1.2 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 $\pi_j$ has a variable $h_j$ with domain $\{i \mid \pi_i \text{ is a neighbor of } \pi_j\} \cup \{j\}$. If $h_j = j$, then $\pi_j$ has the token. Otherwise, $h_j$ contains the process id of one of the process's neighbors. The holder variable forms a directed path from any process in the tree to the process currently holding the token. The problem specification is the following:

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

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

**Fairness** Each process $\pi_i$ is eventually enabled:

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

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

Using the above specification, we synthesized a synchronous self-stabilizing systems, which resembles Raymond's mutual exclusion algorithm on a tree [16]. 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 $\pi_0$ to be the root process, and $\pi_1$ and $\pi_2$ 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. 1 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.

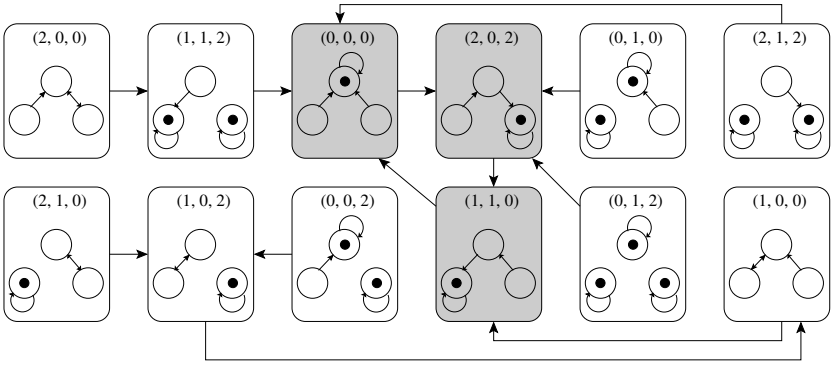


**Fig. 1.** Self-stabilizing mutual exclusion in a tree of size 3 (Raymond's algorithm).

| # 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.

### 6.2 Case Studies for Synthesis of Ideal-Stabilizing Systems

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

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

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 solution for the case of three processes on a line, where each process $\pi_i$ has a Boolean variable $c_i$. Since the only specification for this problem

| # 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.

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 $l_i$.

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

**6.2.2 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 $\pi_i$ 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 four processes on a ring. Note that for each process $\pi_i$, when $tk_i$ is true, the transition $T_i$ changes the value of $c_i$. Hence, having the interpretation 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 $tk_i$.

$$\begin{aligned}
tk_0 &= (c_0 \wedge c_1) \vee (\neg c_0 \wedge \neg c_1) \\
tk_1 &= (\neg c_0 \wedge c_1 \wedge c_2) \vee (c_0 \wedge \neg c_1 \wedge \neg c_2) \\
tk_2 &= (\neg c_1 \wedge c_2 \wedge \neg c_3) \vee (c_1 \wedge \neg c_2 \wedge c_3) \\
tk_3 &= (c_2 \wedge c_3) \vee (\neg c_2 \wedge \neg c_3)
\end{aligned}$$

## 7 Related Work

*Bounded Synthesis* In bounded synthesis [8], 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.

*Synthesis of Self-Stabilizing Systems* In [12], 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 [6] also proposed an automated method to synthesize self-stabilizing algorithms. Our work is different in that the method in [6] 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 [13]. The limitations of this work compared to ours is: (1) Unlike the approach in [13], we do not need the explicit description of the set of legitimate states, and (2) The method in [13] 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 [11].

*Automated Addition of Fault-Tolerance* The proposed algorithm in [2] 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 [2] is not complete.

## 8 Conclusion

In this paper, we proposed an automated SMT-based technique for synthesizing self- and ideal-stabilizing algorithms. In both cases, we assume that only a high-level specification of the algorithm is given in the linear temporal logic (LTL). 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.

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 [11]. 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 counter-example guided inductive synthesis to improve the scalability of the synthesis process.

## 9 Acknowledgments

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

## References

1. F. Abujarad and S. S. Kulkarni. Multicore constraint-based automated stabilization. In *Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 47–61, 2009.
2. 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.
3. Murat Demirbas and Anish Arora. Specification-based design of self-stabilization. *IEEE Transactions on Parallel Distributed Systems*, 27(1):263–270, 2016.
4. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
5. E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, 1986.
6. A. Ebnenasir and A. Farahat. A lightweight method for automated design of convergence. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 219–230, 2011.
7. 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.
8. B. Finkbeiner and S. Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer (STTT)*, 15(5-6):519–539, 2013.
9. M. G. Gouda. The theory of weak stabilization. In *International Workshop on Self-Stabilizing Systems*, pages 114–123, 2001.
10. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press Cambridge, 2012.
11. S. Jacobs and R. Bloem. Parameterized synthesis. *Logical Methods in Computer Science*, 10(1), 2014.
12. A. Klinkhamer and A. Ebnenasir. On the complexity of adding convergence. In *Fundamentals of Software Engineering (FSEN)*, pages 17–33, 2013.
13. A. Klinkhamer and A. Ebnenasir. Synthesizing self-stabilization through superposition and backtracking. In *Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 252–267, 2014.
14. Mikhail Nesterenko and Sébastien Tixeuil. Ideal stabilisation. *IJGUC*, 4(4):219–230, 2013.
15. A. Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.
16. Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.