Parameterized Synthesis of Self-Stabilizing Protocols in Symmetric Rings

Nahal Mirzaie
University of Tehran, North Kargar st., Tehran, Iran
mirzaienahal@ut.ac.ir

Fathiyeh Faghih
University of Tehran, North Kargar st., Tehran, Iran
f.faghih@ut.ac.ir

Swen Jacobs[1]
CISPA Helmholtz Center i.G., Saarbrücken, Germany
jacobs@cispa.saarland
https://orcid.org/0000-0002-9051-4050

Borzoo Bonakdarpour[2]
Iowa State University, 207 Atanasoff Hall, Ames, IA 50011, USA
borzoo@iastate.edu
https://orcid.org/0000-0003-1800-5419

Abstract

Self-stabilization in distributed systems is a technique to guarantee *convergence* to a set of *legitimate states* without external intervention when a transient fault or bad initialization occurs. Recently, there has been a surge of efforts in designing techniques for automated synthesis of self-stabilizing algorithms that are correct by construction. Most of these techniques, however, are not parameterized, meaning that they can only synthesize a solution for a fixed and predetermined number of processes. In this paper, we report a breakthrough in parameterized synthesis of self-stabilizing algorithms in symmetric rings. First, we develop tight cutoffs that guarantee (1) closure in legitimate states, and (2) deadlock-freedom outside the legitimates states. We also develop a sufficient condition for convergence in *silent* self-stabilizing systems. Since some of our cutoffs grow with the size of local state space of processes, we also present an automated technique that significantly increases the scalability of synthesis in symmetric networks. Our technique is based on SMT-solving and incorporates a loop of synthesis and verification guided by counterexamples. We have fully implemented our technique and successfully synthesized solutions to maximal matching, three coloring, and maximal independent set problems.

2012 ACM Subject Classification Theory of computation → Logic and verification, Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Parameterized synthesis, Self-stabilization, Formal methods.

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.0

[1] This research was supported by the German Research Foundation (DFG) under the project ASDPS (JA 2357/2-1).
[2] This research has been partially supported by the NSF SaTC-1813388 and a grant from Iowa State University.

1 Introduction

Program *synthesis* (often called the "*holy grail*" of computer science) is the problem of automated generation of a computer program from a formally specified set of properties. The program generated in this fashion is guaranteed to be correct by construction. Program synthesis is known to be computationally intractable and, thus, is usually used to deal with small but intricate components of a system, e.g., concurrent/distributed algorithms that may exhibit obscure corner cases, where reasoning about their correctness is not straightforward.

Dijkstra [3] introduced the notion of *self-stabilization* in distributed systems, where the system always converges to a good behavior even if it is arbitrarily initialized or is subject to transient faults. Proof of self-stabilization is, however, often much more complex than what it initially seems like. Dijkstra himself published the proof of correctness of his seminal 3-state machine solution 12 years later [4]. This means that program synthesis can play a prime role in designing and reasoning about the correctness of self-stabilizing algorithms. In [8–12], we introduced a set of algorithms and tools for synthesizing self-stabilizing protocols. Our techniques take as input the network topology, timing model (asynchronous or synchronous), the good behavior of the protocol (either explicitly as a set of *legitimate states* or implicitly as a set of temporal logic formulas), type of symmetry, and type of stabilization (e.g., strong, weak, monotonic, ideal) and generate a set of first-order modulo theory (SMT) constraints. Then, an SMT-solver solves these constraints and, if satisfiable, produces a model that respects the input specification. Our tool Assess [10] has successfully synthesized complex algorithms such as Raymond's distributed mutual exclusion [24], Dijkstra's token ring [3] (for both three and four state machines), maximal matching [23], weak stabilizing token circulation in anonymous networks [2], and the three coloring problem [14]. Our algorithms are *complete* for a predetermined fixed number of processes; i.e., if they fail to find a solution to the synthesis problem, then there does not exist one. This completeness, however, comes at a big cost which is scalability. That is, for most instances, we could only synthesize solutions up to 5 processes at best.

In this paper, our goal is to address scalability as well as the shortcoming that the previous work can synthesize only a fixed and predetermined number of processes. To this end, we focus on automated synthesis of self-stabilizing protocols in *symmetric* and *parameterized* rings, where an unbounded number of processes exhibit identical behavior. We make two main contributions. First, we show how to solve the *parameterized synthesis problem* based on the notion of *cutoffs* [6] that can guarantee properties of distributed systems of arbitrary size by considering only systems of up to a certain fixed size $c \in \mathbb{N}$, and augmented by a sound but incomplete abstraction-based synthesis approach for properties that are known to be undecidable. In particular, we provide:

- cutoffs for the closure and deadlock-freedom properties, under the assumption that the

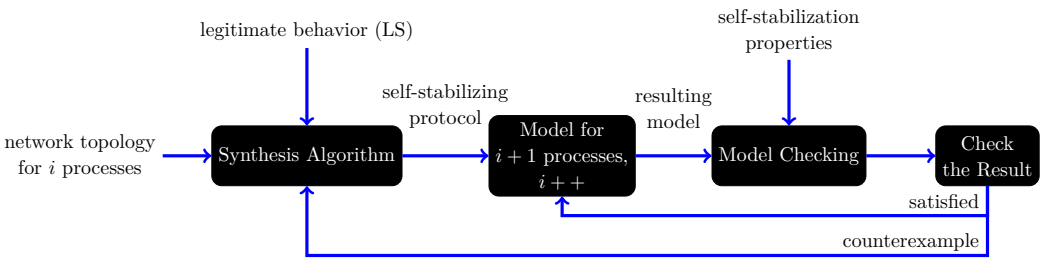

Figure 1 SMT-based Counterexample-Guided Synthesis Technique

set of legitimate states is defined by a conjunction of predicates on the local state of processes; we show that smaller cutoffs are possible under additional assumptions, and that all our cutoffs are tight under the assumptions we consider.
- an abstraction-based method for the convergence property, which is known to be undecidable in general [19, 21]; we show how, for the class of silent algorithms, a sufficient condition for convergence of the parameterized system can be efficiently checked on a finite system that over-approximates the behavior of systems of arbitrary size.

Note that these results can be used for both synthesis and verification. A drawback of our cutoffs is that they are quadratic in the state space of a single process, so even with a cutoff, we need synthesis methods that scale to a large number of processes. Thus, as our second contribution, we propose a counterexample-guided synthesis technique that exploits our symmetry assumption. More specifically, it consists of fours steps (see Fig. 1):

1. First, we *synthesize* a solution for a small network of i processes using existing techniques;
2. Next, we trivially generalize this solution to a larger network with $i + 1$ processes;
3. Then, we *verify* this solution using a model checker, and
4. If verification succeeds, we return to step 2 to attempt a larger network. Otherwise, we obtain a counterexample that is added as a negative constraint for the synthesis algorithm, and we return to step 1 for another round of synthesis with limited search space.

Using this approach and our cutoff results, we successfully synthesized parameterized self-stabilizing protocols for well-known problems including three coloring, maximal matching, and maximal independent set in less than 10 minutes. To our knowledge, this is the first instance of such parameterized synthesis.

Organization The rest of the paper is organized as follows. Section 2 introduces the preliminary concepts. In Section 3, we present the formal statement of our synthesis problem. The parameterized correctness results are presented in Section 4, while our counterexample-guided synthesis approach is presented in Section 5. Experimental results and case studies are reported in Section 6. Related work is discussed in Section 7, and finally, we make concluding remarks and discuss future work in Section 8.

2 Preliminaries

2.1 Distributed Programs

Most self-stabilizing algorithms are defined in the shared-memory model. Assume V to be the set of all variables in the system, where each variable $v \in V$ has a finite domain D_v. We define a *state* s as a valuation of each variable in V by a value in its domain. The set of all possible states is called the *state space*, and represented by S. A *transition* is defined as an ordered pair (s_0, s_1), where $s_0, s_1 \in S$. We denote the value of a variable v in state s by $v(s)$.

▶ **Definition 1.** A *process* π is a tuple $\langle R_\pi, W_\pi, T_\pi \rangle$, where

- $R_\pi \subseteq V$ is the set of variables that π can read their values and is called the *read-set* of π;
- $W_\pi \subseteq R_\pi$ is the set of variables that π can write to, and is called the *write-set* of π, and
- T_π is the set of transitions of π, where for each transition $(s_0, s_1) \in T_\pi$ and each variable v, such that $v(s_0) \neq v(s_1)$, we have $v \in W_\pi$. ◀

The third condition imposes the constraint that a process can only change the value of a variable in its write-set, and the second condition states that this change cannot be blind. A process $\pi = \langle R_\pi, W_\pi, T_\pi \rangle$ is called *enabled* in state s_0, if there exists a state s_1, such that

$(s_0, s_1) \in T_\pi$. The *local state space* of π is the set of all possible valuations of the variables that π can read: $S_\pi = \prod_{v \in R_\pi} D_v$

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

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

$$\forall s'_0, s'_1 : ((\forall v \in R_\pi : (v(s_0) = v(s'_0) \ \wedge \ v(s_1) = v(s'_1))) \wedge (\forall v \notin R_\pi : v(s'_0) = v(s'_1))) \implies (s'_0, s'_1) \in T_\pi \quad (1)$$

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

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

◀

Intuitively, the read restriction in Definition 2 imposes the constraint that for each process π, each transition in T_π depends only on the variables in the read-set of π. 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 $T_\mathcal{D}$.

Example. We use the problem of distributed self-stabilizing *one-bit maximal matching* as a running example to describe the concepts throughout the paper. Consider a ring of 4 processes (see Fig. 2), and let $V = \{x_0, x_1, x_2, x_3\}$ be the set of variables, where each x_i is a Boolean variable with domain $\{\mathtt{F}, \mathtt{T}\}$. Let $\mathcal{D} = \langle P_\mathcal{D}, T_\mathcal{D} \rangle$ be a distributed program, where $P_\mathcal{D} = \{\pi_0, \pi_1, \pi_2, \pi_3\}$. Each process π_i $(0 \leq i \leq 3)$ can write to the variable x_i (i.e., $W_{\pi_i} = \{x_i\}$), and read the variables of its own and its neighbors ($R_{\pi_i} = \{x_i, x_{(i+1) \mod 4}, x_{(i-1) \mod 4}\}$). Notice that following Definition 2 and read/write restrictions of π_0, (arbitrary) transitions

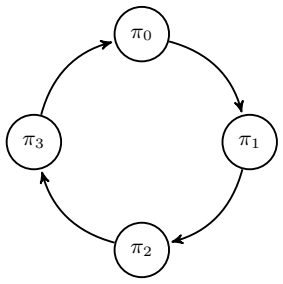

Figure 2 One-bit maximal matching example.

$$t_1 = ([x_0 = \mathtt{F}, x_1 = \mathtt{F}, x_2 = \mathtt{F}, x_3 = \mathtt{F}], [x_0 = \mathtt{T}, x_1 = \mathtt{F}, x_2 = \mathtt{F}, x_3 = \mathtt{F}])$$

$$t_2 = ([x_0 = \mathtt{F}, x_1 = \mathtt{F}, x_2 = \mathtt{T}, x_3 = \mathtt{F}], [x_0 = \mathtt{T}, x_1 = \mathtt{F}, x_2 = \mathtt{T}, x_3 = \mathtt{F}])$$

are in the same group. The reason is that π_0 cannot read x_2, and if, for example, t_1 is included in the set of transitions, while t_2 is not, it implies that the execution in process π_0 depends on the value of x_2, which is not possible.

▶ **Definition 3.** An *uninterpreted local function* for a process maps the *local state space* of a process to a domain D_{lf}. The interpretation of an uninterpreted local function for a process π is a function:

$$S_\pi \to D_{lf}$$

where S_π is the local state space of π. ◀

In the sequel, we use "uninterpreted functions" to refer to uninterpreted local functions.

Example. To formulate the requirements in the one-bit maximal matching example, we assume each process π_i is associated with an uninterpreted local function, called *match*, with the domain $D_{match_i} = \{l, r, n\}$, where l, r, and n correspond to the cases where the process is matched to its left, right, and no neighbor (self-matched), respectively. The interpretation of $match_i$ is a function:

$$(match_i)_{\mathrm{I}} : \{\mathtt{F}, \mathtt{T}\} \times \{\mathtt{F}, \mathtt{T}\} \times \{\mathtt{F}, \mathtt{T}\} \rightarrow \{l, r, n\}$$

In other words, the value of $match_i$ depends on the value of the process and its neighbors' Boolean variables.

▶ **Definition 4.** A *local predicate* of a process maps the *local state space* of a process to a Boolean:

$$S_\pi \rightarrow \{\mathtt{F}, \mathtt{T}\}$$

◀

We use this definition to define legitimate states in Section 2.3.

2.1.1 Network Topology

A topology specifies the communication model of a distributed program.

▶ **Definition 5.** A *topology* is a tuple $\mathcal{T} = \langle V, |P_\mathcal{T}|, R_\mathcal{T}, W_\mathcal{T} \rangle$, where

- V is a finite set of finite-domain discrete variables,
- $|P_\mathcal{T}| \in \mathbb{N}_{\geq 1}$ is the number of processes,
- $R_\mathcal{T}$ is a mapping $\{0 \ldots |P_\mathcal{T}| - 1\} \rightarrow 2^V$ from a process index to its read-set,
- $W_\mathcal{T}$ is a mapping $\{0 \ldots |P_\mathcal{T}| - 1\} \rightarrow 2^V$ from a process index to its write-set, such that $W_\mathcal{T}(i) \subseteq R_\mathcal{T}(i)$, for all i $(0 \leq i \leq |P_\mathcal{T}| - 1)$. ◀

Example. The topology of our maximal matching problem is a tuple $\langle V, |P_\mathcal{T}|, R_\mathcal{T}, W_\mathcal{T} \rangle$:

- $V = \{x_0, x_1, x_2, x_3\}$, with domains $D_{x_0} = D_{x_1} = D_{x_2} = D_{x_3} = \{\mathtt{T}, \mathtt{F}\}$,
- $|P_\mathcal{T}| = 4$,
- $R_\mathcal{T}(0) = \{x_0, x_1, x_3\}$, $R_\mathcal{T}(1) = \{x_1, x_2, x_0\}$, $R_\mathcal{T}(2) = \{x_2, x_3, x_1\}$, $R_\mathcal{T}(3) = \{x_3, x_0, x_2\}$, and
- $W_\mathcal{T}(0) = \{x_0\}$, $W_\mathcal{T}(1) = \{x_1\}$, $W_\mathcal{T}(2) = \{x_2\}$, and $W_\mathcal{T}(3) = \{x_3\}$.

▶ **Definition 6.** A distributed program $\mathcal{D} = \langle P_\mathcal{D}, T_\mathcal{D} \rangle$ *has* topology $\mathcal{T} = \langle V, |P_\mathcal{T}|, R_\mathcal{T}, W_\mathcal{T} \rangle$ iff

- each process $\pi \in P_\mathcal{D}$ is defined over V
- $|P_\mathcal{D}| = |P_\mathcal{T}|$
- there is a mapping $g : \{0 \ldots |P_\mathcal{T}| - 1\} \rightarrow P_\mathcal{D}$ such that

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

◀

2.2 Symmetric Networks

Roughly speaking, a topology is symmetric, if the read-set and write-set of any two distinct processes can be swapped (i.e., there is a bijection that maps read/write variables of a process to another).

▶ **Definition 7.** A topology $\mathcal{T} = \langle V, |P_\mathcal{T}|, R_\mathcal{T}, W_\mathcal{T} \rangle$ is *symmetric*, iff for any distinct $i, j \in \{0 \dots |P_\mathcal{T}| - 1\}$, there exists

- a bijection $f : R_\mathcal{T}(i) \to R_\mathcal{T}(j)$, such that $\forall v \in R_\mathcal{T}(i) : \; D_v = D_{f(v)}$, and
- a bijection $g : W_\mathcal{T}(i) \to W_\mathcal{T}(j)$, such that $\forall v \in W_\mathcal{T}(i) : \; D_v = D_{g(v)}$. ◀

We call a symmetric topology a (bi-directional) *ring* (of size $k = |P_\mathcal{T}|$) if for every $i \in \{0 \dots |P_\mathcal{T}| - 1\}$, we have $R_\mathcal{T}(i) = W_\mathcal{T}(i-1 \mod k) \cup W_\mathcal{T}(i) \cup W_\mathcal{T}(i+1 \mod k)$. Since in this paper we only deal with rings, to simplify notation throughout the paper, arithmetic on process indices is implicitly modulo the size of the ring.

Example. The topology of our one-bit maximal matching example is symmetric, and a ring of size 4 (Fig. 2). For any two numbers i and j, function g is the mapping from x_i to a x_j, and function f maps $x_i \mapsto x_j$, $x_{(i+1)} \mapsto x_{(j+1)}$, and $x_{(i-1)} \mapsto x_{(j-1)}$.

▶ **Definition 8.** A distributed program $\mathcal{D} = \langle P_\mathcal{D}, T_\mathcal{D} \rangle$ is called *symmetric* iff

- it has a symmetric topology, and
- for any two distinct processes $\pi, \pi' \in P_\mathcal{D}$, the following condition holds:

$$\begin{array}{l} \forall (s_0, s_1) \in T_\pi : \; \exists (s'_0, s'_1) \in T_{\pi'} : \\ \Big(\forall v \in R_\pi : \big(v(s_0) = f(v)(s'_0)\big) \big) \; \wedge \; \big(\forall v \in W_\pi : (v(s_1) = g(v)(s'_1)) \big) \Big) \end{array} \tag{2}$$

where f and g are the functions defined in Definition 7. ◀

In other words, in a symmetric distributed program the read- and write-sets of all processes are identical up to renaming, and so are their transitions. Therefore, we also write $\mathcal{T}^\pi$ for a symmetric distributed program that has topology $\mathcal{T}$ and where all processes are identical up to renaming to π.

2.3 Self-Stabilization

Given a subset of the state space, called the set of *legitimate states* (denoted by LS), a *self-stabilizing* [3] program always recovers to a state in LS from any arbitrary state (e.g., due to bad initialization or occurrence of transient faults) in a finite number of steps, and stays in LS thereafter.

▶ **Definition 9.** A *computation* of $\mathcal{D} = \langle P_\mathcal{D}, T_\mathcal{D} \rangle$ 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.* ◀

▶ **Definition 10.** A distributed program $\mathcal{D} = \langle P_\mathcal{D}, T_\mathcal{D} \rangle$ is *self-stabilizing* for a set LS of legitimate states iff

1. *(Convergence)* For any computation $\overline{s} = s_0 s_1 \cdots$, there exists a state $s_j \in \overline{s}$ $(j \geq 0)$, such that $s_j \in LS$.

2. *(Closure)* For any transition $(s_0, s_1) \in T_{\mathcal{D}}$, if $s_0 \in LS$, then $s_1 \in LS$. ◀

▶ **Definition 11.** A distributed program $\mathcal{D} = \langle P_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ is *silent* with respect to a given LS if for any transition $(s_0, s_1) \in T_{\mathcal{D}}$, if $s_0 \in LS$, then $s_1 = s_0$. ◀

▶ **Definition 12.** A set of legitimate states is *locally defined* if it can be defined by

$$s \in LS \quad \text{if and only if} \quad \forall i \in \{0 \dots |P_{\mathcal{T}}| - 1\} : LS_i(s),$$

where LS_i is a predicate on the read-set of process π_i. ◀

Example. In a directed graph, a maximal matching is a maximal set of edges, in which no two edges share a common vertex. In a ring topology, each process can be matched to one of its two adjacent processes. To formulate this requirement, we assume each process π_i is associated with a local uninterpreted function, called $match_i$, with the domain $D_{match_i} = \{l, r, n\}$. LS can be locally defined with

$$\begin{aligned} LS_i = \big\{ s \mid\ & (match_{i-1}(\Pi_{R_{\pi_{i-1}}}(s)) = r \wedge match_i(\Pi_{R_{\pi_i}}(s)) = l \wedge match_{i+1}(\Pi_{R_{\pi_{i+1}}}(s)) = n) \vee \\ & (match_{i-1}(\Pi_{R_{\pi_{i-1}}}(s)) = n \wedge match_i(\Pi_{R_{\pi_i}}(s)) = r \wedge match_{i+1}(\Pi_{R_{\pi_{i+1}}}(s)) = l) \vee \\ & (match_{i-1}(\Pi_{R_{\pi_{i-1}}}(s)) = l \wedge match_i(\Pi_{R_{\pi_i}}(s)) = n \wedge match_{i+1}(\Pi_{R_{\pi_{i+1}}}(s)) = r) \vee \\ & (match_{i-1}(\Pi_{R_{\pi_{i-1}}}(s)) = l \wedge match_i(\Pi_{R_{\pi_i}}(s)) = r \wedge match_{i+1}(\Pi_{R_{\pi_{i+1}}}(s)) = l) \vee \\ & (match_{i-1}(\Pi_{R_{\pi_{i-1}}}(s)) = r \wedge match_i(\Pi_{R_{\pi_i}}(s)) = l \wedge match_{i+1}(\Pi_{R_{\pi_{i+1}}}(s)) = r) \big\} \end{aligned}$$

The system is in its legitimate state, if and only if all processes are in their local legitimate states. For example, in a ring of size three with the set of processes $P = \{\pi_0, \pi_1, \pi_2\}$, the set of legitimate states can be formulated as the following:

$$\{s \mid LS_0(s) \wedge LS_1(s) \wedge LS_2(s)\}$$

Note how uninterpreted functions can be used to easily express LS. Without $match_i$, the user has to explicitly specify the cases where a process is matched to its left, right or itself, using the Boolean variables of its own and its adjacent processes (its read set).

3 Problem Statement

Our goal is to propose an automated method for parameterized synthesis of silent self-stabilizing protocols in symmetric ring networks. That is, we consider a problem where the size of the topology is a parameter, and we want to automatically synthesize the transition predicate and the interpretation of the uninterpreted function of each process, such that the resulting distributed program is silent self-stabilizing for any value of the parameter.

Formally, a *parameterized topology* is a sequence of symmetric topologies $\mathcal{T}_1, \mathcal{T}_2, \dots$, where for all n we have $|P_{\mathcal{T}_n}| = n$ and bijections read-sets and write-sets, as required in Definition 7, also exist between process indices from different elements of the sequence. A *parameterized program* is a sequence of symmetric distributed programs $\mathcal{D}_1, \mathcal{D}_2, \dots$ such that $\mathcal{D}_i = \mathcal{T}_i^{\pi}$ for a parameterized topology $\mathcal{T}_1, \mathcal{T}_2, \dots$, and some process π.

The *parameterized synthesis problem* takes as input:

- a parameterized topology, and
- a set of locally defined legitimate states LS,

and generates as output:

- a process π such that for every element $\mathcal{T}_n$ of the topology, the program $\mathcal{D}_n = \mathcal{T}_n^\pi$ is self-stabilizing to LS.

▶ **Definition 13.** For a given parameterized topology and a property under consideration, a *cutoff* is a natural number c such that for any given process π and a locally defined LS the following holds: $\mathcal{D}_n = \mathcal{T}_n^\pi$ satisfies the property wrt. LS for all $n \in \mathbb{N}$ iff $\mathcal{D}_i = \mathcal{T}_i^\pi$ satisfies the property wrt. LS for all $i \in \{1 \dots c\}$. ◀

Note that cutoffs can be used for both parameterized verification and synthesis. In Section 4, we will present cutoffs for two properties: i) closure, and ii) the absence of deadlocks outside of LS. Moreover, we will introduce an abstraction-based method that can be combined with the cutoffs to solve the parameterized synthesis problem.

4 Parameterized Synthesis of Self-Stabilization in Symmetric Rings

In this section, we show how to reduce reasoning about parameterized programs to reasoning about a finite number of finite programs. To prove self-stabilization, we need to prove that the algorithm has the two properties of closure and convergence. We split the latter into two properties: (1) the absence of deadlocks outside of LS, and (2) the absence of cycles outside of LS. In the following, we provide tight cutoffs for closure and deadlocks outside of LS, as well as a sound abstraction to prove the absence of cycles outside of LS. Finally, we provide our main theorem that combines these results into a method for parameterized synthesis of self-stabilizing algorithms in rings.

4.1 Cutoffs for Closure

Assume that the write-set of each process has l valuations. In other words, if process π_i has $W_\mathcal{T}(i) = \{v_1, \dots, v_n\}$, then $l = |D_{v_1}| \times \cdots \times |D_{v_n}|$.

▶ **Lemma 1.** For self-stabilizing algorithms on a ring topology, the following are cutoffs for the closure property:

- $c = l^2 + 1$, if LS is locally defined;
- $c = l + 1$, if LS is locally defined and LS_i only depends on $W_\mathcal{T}(i)$ and $W_\mathcal{T}(i+1)$, and
- $c = 3$, if LS is locally defined and LS_i only depends on $W_\mathcal{T}(i)$.

All of the cutoffs are tight under their respective assumptions.

Proof. We provide the proof idea for the first case. Consider a ring of size $M > l^2 + 1$. Assume there exists a transition from $s \in LS$ to $s' \notin LS$. WLOG, assume this action is taken by π_0. Now, consider the $M-1$ pairs of consecutive processes (π_i, π_{i+1}), where $i \in [0, M-2]$. Note that π_0 appears in just one tuple. Based on the pigeonhole principle, at least two of these pairs of processes have the same valuation of their write-sets in s, since we have $M - 1 \geq l^2 + 1$ tuples and only l^2 possible valuations of the write-sets of a tuple. Assume that (π_i, π_{i+1}) and (π_j, π_{j+1}) have the same valuation of their write-sets. Then we can consider a smaller ring composed of $\pi_0, \dots, \pi_i, \pi_{j+1}, \dots, \pi_M$ with local valuations as in state s (Fig. 3). Note that this construction does not change the valuations of the read-sets of the remaining processes, and hence, the smaller ring will also be in a state in LS, and π_0 can still take a transition that leads to a state outside of LS. If necessary, we can repeat the removal of processes until we arrive at a ring of size at most $c = l^2 + 1$.

To observe tightness, define a total order on pairs of valuations of write- sets, and assume that LS is defined such that it requires the valuations along the ring to be compatible with

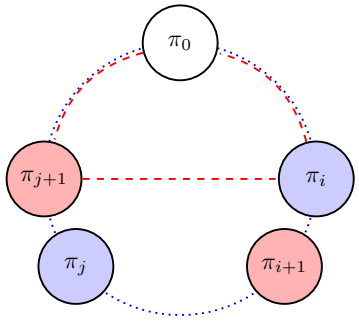

Figure 3 Reducing a ring (blue dotted) to a smaller ring (red dashed).

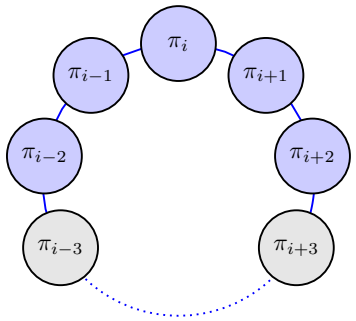

Figure 4 Blue processes act based on the synthesized algorithm and grey (with slanted lines) processes act randomly.

successor relation of that order. Then a state can only be in LS in a ring of size at least $c = l^2 + 1$, and therefore violation of closure is not possible in any smaller ring.

For the second case, the same construction works even if we only consider single valuations of write-sets instead of pairs, except for one special case: if the processes with identical valuation are π_0 and π_1, then π_1 cannot be removed, as this might make the transition of π_0 impossible. Therefore, the cutoff is not l, but $l + 1$. Tightness follows by a similar argument as above.

Finally, in the third case we mainly need to ensure that π_0 can still take its transition, as for every process LS_i only depends on its own valuation. Thus, $c = 3$ is sufficient (and in general necessary, as otherwise the transition may not be possible). ◀

4.2 Cutoffs for Deadlock Detection

▶ **Lemma 2.** For self-stabilizing algorithms on a ring topology, the following are cutoffs for the detection of deadlocks outside of LS:

- $c = l^2 + 1$, if LS is locally defined;
- $c = l + 1$, if LS is locally defined and transitions of processes only depend on $W_{\mathcal{T}}(i)$ and $W_{\mathcal{T}}(i + 1)$ (i.e., the ring is uni-directional), and
- $c = 3$, if LS is locally defined and transitions of processes only depend on $W_{\mathcal{T}}(i)$ (i.e., processes are completely independent).

All of the cutoffs are tight under their respective assumptions.

Proof. We only provide the proof idea for the first case, the other cases are variants similar to those seen above. Consider a ring of size $M > l^2 + 1$ and assume that with the given program we have a deadlock state $s \notin LS$; that is $s \notin LS_i$ for at least one of the processes, and there is no active transition for any processes. We assume that $i = 0$, and by the same construction as in the proof of Lemma 1 we can obtain a smaller ring where every remaining process has the same valuation of its read-set. Therefore, the state in the smaller ring is not in LS_0, and no transitions are enabled. Tightness can be observed by a similar construction as above, except that now we define the transitions of processes such that no transition is enabled iff the order is respected. ◀

4.3 Process Abstraction for Convergence

As mentioned before, to prove self-stabilization of a parameterized program, we need to prove closure and convergence. Closure can be proved based on Lemma 1, and Lemma 2 shows how to deal with deadlocks outside of LS. Thus, the missing part is a method to prove that there are no cycles outside of LS that prevents a computation to eventually reach LS. In contrast to the two previous problems, we now consider infinite behaviors of the system. Since parameterized verification and synthesis of symmetric self-stabilization in rings is known to be undecidable [19, 21], we cannot obtain cutoffs for this property. Therefore, we resort to proving the absence of cycles based on a sound abstraction of the system behavior.

The basic idea is the following: we check whether there is a loop that starts and ends in the same *local* state for an arbitrary process. If we can show that this is not possible, then certainly no global loop is possible. Note that this is a stronger property than what we want to prove; it proves that there could not be any loops in the protocol, neither inside nor outside LS. It is obvious that this property can only be satisfied for silent protocols. To this end, we fix five processes (see Fig. 4), and define the following property:

$$S \Rightarrow \ (\Diamond\Box S \ \vee \ \neg\Box\Diamond S),$$

where S is the local state of π_i (i.e., the valuation of its read-set), $\Diamond$ is the 'eventually' operator and $\Box$ is the 'always' operator in temporal logic. That is, given a local state in S, any future extension either reaches a state where S is continually true, or, S does not become true infinitely often. Next, we attempt to prove the property in a ring of size 7, where 5 processes behave according to the synthesized protocol, and the other two processes have the same write-set, but can execute arbitrary transitions. The idea is that these two processes over-approximate the possible behavior of all other processes. If we can prove the property above in this abstraction of the system, then this implies that no loops are possible in a concrete system in a ring of size ≥ 5. Note that the precision of the abstraction can be refined by increasing the number of processes that behave according to the protocol, or by including the local state of additional processes into S. For the problems we considered in our experiments (see Sect. 6), the fixed abstraction with $5 + 2$ processes was sufficient.

4.4 Parameterized Self-Stabilization

Based on Lemmas 1 and 2, and the approach in Section 4.3, we obtain our main result.

▶ **Theorem 14.** *Let $\mathcal{T}_1, \mathcal{T}_2, \ldots$ be a parameterized ring topology, π a process, and let LS be locally defined by LS_i. Let c_1 and c_2 be cutoffs for closure and deadlock detection wrt. LS, respectively. If (1) closure holds in rings of size up to c_1, (2) deadlocks outside of LS are impossible in rings of size up to c_2, and (3) the absence of cycles can be proven in rings of up to size 4 and in an abstract system as above, then every instance of the parameterized program $\mathcal{D}_1 = \mathcal{T}_1^\pi, \mathcal{D}_2 = \mathcal{T}_2^\pi, \ldots$ is self-stabilizing to LS.* ◀

5 SMT-based Counterexample-Guided Synthesis

5.1 General Idea

In [8, 9, 11], we introduced SMT-based methods to solve the synthesis problem for self-stabilizing systems. In a nutshell, our techniques generate a set of SMT constraints from the input synthesis instance and produce a model that represents a self-stabilizing protocol. In order to scale up these technique to synthesize solutions up to the cutoff point efficiently, in

this section, we propose a method, where we find a solution for a larger topology using a solution for a smaller topology. Let us first entertain a naïve idea, where we first synthesize a protocol for a small topology and then simply use this solution for larger topologies with the hope that since the protocol is symmetric, a small solution works in a larger network as well. We now show that this approach is not conceivable even for very simple protocols.

Example. When applying our latest algorithm [11] to the one-bit maximal matching example, the first synthesized solution for 4 processes is the following transition relation encoded by guarded commands for each process π_i:

$$\begin{aligned} \pi_i : \quad & (x_i = \mathtt{F}) \wedge (x_{(i+1)} = \mathtt{F}) \wedge (x_{(i-1)} = \mathtt{F}) \quad \rightarrow \quad x_i := \mathtt{T} \\ & (x_i = \mathtt{T}) \wedge (x_{(i+1)} = \mathtt{T}) \quad \rightarrow \quad x_i := \mathtt{F} \end{aligned}$$

and the following interpretation for uninterpreted function $match_i$:

$$\begin{aligned} match_i : \quad & (x_i = \mathtt{T}) \wedge \big((x_{(i+1)} = \mathtt{T}) \vee (x_{(i-1)} = \mathtt{F})\big) \quad \mapsto \quad l \\ & (x_{(i+1)} = \mathtt{T}) \wedge (x_{(i-1)} = \mathtt{F}) \quad \mapsto \quad l \\ & (x_i = \mathtt{T}) \wedge (x_{(i+1)} = \mathtt{F}) \wedge (x_{(i-1)} = \mathtt{T}) \quad \mapsto \quad r \\ & (x_i = \mathtt{F}) \wedge (x_{(i+1)} = \mathtt{T}) \wedge (x_{(i-1)} = \mathtt{T}) \quad \mapsto \quad r \\ & (x_i = \mathtt{F}) \wedge (x_{(i+1)} = \mathtt{F}) \quad \mapsto \quad n \end{aligned}$$

Now, if we trivially use the synthesized protocol on a topology with 5 processes, the resulting protocol is incorrect. In particular, the following is a counterexample (i.e., a finite computation that violates the specification) in terms of predicate *match*:

$$\begin{aligned} & \Big([match_0 = n, match_1 = n, match_2 = n, match_3 = n, match_4 = n], \\ & \ [match_0 = l, match_1 = n, match_2 = n, match_3 = n, match_4 = l], \\ & \ [match_0 = l, match_1 = r, match_2 = l, match_3 = n, match_4 = l]\Big) \end{aligned}$$

This computation violates convergence, as it reaches a deadlock state in $\neg LS$. This example shows that a synthesized symmetric solution cannot be trivially extended to larger topologies.

5.2 The Counterexample-Guided Synthesis Algorithm

In order to limit the search space of SMT-solvers for a solution, we incorporate a synthesis-verification loop guided by counterexamples. Our approach consists of the following steps:

1. Given a topology with i processes and a set of legitimate states, we use our existing approach [8, 9, 11] to formulate the synthesis problem as an SMT instance.
2. We use an SMT solver to find a solution for the SMT instance, as a transition relation and an interpretation for each uninterpreted function. Note that due to symmetry, the transition relations and the interpretation functions are identical for all processes.
3. Next, we generalize the solution for a topology with $i+1$ processes and verify this solution using a model checker.
4. If the result of verification is positive, then we go back to step 3 to check the properties for a topology with $i+2$ processes. Otherwise, we transform the generated counterexample into an SMT constraint and add it to the initial SMT instance (step 1) and return to step 2.

For reasons of space, we do not include the details of our SMT-based synthesis technique [8, 9, 11]. We now analyze the nature of counterexamples. In the context of closure and convergence, a model checker may generate a counterexample of the form $\overline{s} = s_0 s_1 \cdots s_n$. Observe that $\overline{s}$ is one of the following three types of counterexamples:

- If closure is violated, then $\overline{s} = (s_0, s_1)$, where $s_0 \in LS$ and $s_1 \notin LS$.
- If convergence is violated by $\overline{s} = s_0 s_1 \cdots s_n$, where for all $i \in [0, n]$, we have $s_i \notin LS$ and either
  - $s_0 = s_n$; i.e., a loop exists outside the set of legitimate states, or
  - there does not exist a state $\mathfrak{s}$, where $(s_n, \mathfrak{s})$ is a valid transition; i.e., s_n is a *deadlock* state outside the set of legitimate states.

For example, the counterexample presented in Section 5.1 is of the third type.

Dealing with the first type of counterexamples is pretty straightforward: we only add a constraint to the SMT instance that disallows transition (s_0, s_1) in the transition relation. To address deadlocks, we need to add a constraint to the SMT instance to enforce a change in the resulting synthesized model, so that s_n is not a deadlock state. To this end, we propose two sets of heuristics to change either the transition relation or the interpretation of uninterpreted functions in Section 5.3. Dealing with loops is a bit more complicated. For example, one can remove a transition from the loop to break it, but the choice of transition may involve a combinatorial enumeration to find the right transition. This type of counterexamples is not our focus in this paper and we leave it for future work. Interestingly, all of our case studies in Section 6 do not involve loop counterexamples.

5.3 Heuristics Considering Transition Relations

The simplest method to resolve a deadlock is to formulate a constraint imposing the existence of an outgoing transition from s_n. Since in this paper, our focus is on asynchronous systems, a transition is the execution of one of the processes. We propose two strategies for selecting a process to have an outgoing transition from a deadlock state.

Progress Heuristic. In this approach, we add a constraint stating that at least one of the proceses should have an outgoing transition from s_n. More formally, assume that the current topology includes i processes, where the read-set of each process has r variables, with domains $D_0, \ldots, D_{r-1}$, and the write-set of each process includes w variables, with domains $D'_0, \cdots, D'_{w-1}$. Note that since the goal is to synthesize a symmetric program, all processes execute similarly according to the function T_p:

$$T_p : (\prod_{j \in [0, r-1]} D_j) \rightarrow (\prod_{j \in [0, w-1]} D'_j)$$

and function f is of type:

$$f : \mathbb{N} \rightarrow \Big(S \rightarrow (\prod_{j \in [0, r-1]} D_j)\Big)$$

Then, the constraint to be added to the SMT instance can be written as:

$$\forall j \in [0, w-1] : \exists val_j \in D'_j : \bigvee_{k \in [0, i)} \Big(\Big(f(k)(s_n), [val_0, val_1, \cdots, val_{w-1}]\Big) \in T_p\Big)$$

Example. Consider the counterexample mentioned in Section 5.1. Each process can read three Boolean variables and write to one Boolean variable and, hence, T_p is defined as follows:

$$T_p \ : \ \{\mathtt{F},\mathtt{T}\} \times \{\mathtt{F},\mathtt{T}\} \times \{\mathtt{F},\mathtt{T}\} \rightarrow \{\mathtt{F},\mathtt{T}\}$$

Note that for each process π_j, $f(j)$ returns $[x_{(j-1)}, x_j, x_{(j+1)}]$. In the counterexample we presented in the previous example, the last state where the deadlock happens is:

$$[x_0 = \mathtt{T}, x_1 = \mathtt{F}, x_2 = \mathtt{T}, x_3 = \mathtt{F}, x_4 = \mathtt{F}]$$

Thus, we add the following constraint to the SMT instance:

$$\exists val \in \{\mathtt{F},\mathtt{T}\} \ : \ \Big(([\mathtt{F},\mathtt{T},\mathtt{F}], val) \in T_p \ \vee \ ([\mathtt{T},\mathtt{F},\mathtt{T}], val) \in T_p \ \vee \ ([\mathtt{F},\mathtt{T},\mathtt{F}], val) \in T_p \vee$$
$$([\mathtt{T},\mathtt{F},\mathtt{F}], val) \in T_p \ \vee \ ([\mathtt{F},\mathtt{F},\mathtt{T}], val) \in T_p\Big)$$

In the above constraint, the jth clause imposes a constraint on T_p to have an outgoing transition considering the local state of the jth process. (Note that the first and third clauses are the same, and we just put them for clarity.)

Local LS Heuristic. As mentioned in Section 2, we focus on sets LS that can be locally defined, i.e., the set of legitimate states can be described as a conjunction over local legitimate states of processes. In this case, a deadlock can be resolved by checking the local state of each process and imposing a constraint to have an outgoing transition for at least one of those processes that are not in their local legitimate states.

Example. For the counterexample of one-bit maximal matching with 4 processes, the local set of legitimate states is the following:

$$(match_{(i-1)} = l \ \wedge \ match_i = n \ \wedge \ match_{(i+1)} = r) \vee$$
$$(match_{(i-1)} = r \ \wedge \ match_i = l \ \wedge \ match_{(i+1)} = n) \vee$$
$$(match_{(i-1)} = r \ \wedge \ match_i = l \ \wedge \ match_{(i+1)} = r) \vee$$
$$(match_{(i-1)} = l \ \wedge \ match_i = r \ \wedge \ match_{(i+1)} = l) \vee$$
$$(match_{(i-1)} = n \ \wedge \ match_i = r \ \wedge \ match_{(i+1)} = l)$$

and the deadlock state is:

$$[x_0 = \mathtt{T}, x_1 = \mathtt{F}, x_2 = \mathtt{T}, x_3 = \mathtt{F}, x_4 = \mathtt{F}]$$

For checking the local state of each process, we should first note the values of uninterpreted functions $match_i$ in this state:

$$[match_0 = l, match_1 = r, match_2 = l, match_3 = n, match_4 = l]$$

Processes π_0, π_1, π_3, and π_4 are not in a local legitimate state, and hence, the added constraint to the original SMT model will be as follows:

$$\exists val \in \{\mathtt{F},\mathtt{T}\} \ : \ \Big(([\mathtt{F},\mathtt{T},\mathtt{F}], val) \in T_p \ \vee \ ([\mathtt{T},\mathtt{F},\mathtt{T}], val) \in T_p \vee$$
$$([\mathtt{T},\mathtt{F},\mathtt{F}], val) \in T_p \ \vee \ ([\mathtt{F},\mathtt{F},\mathtt{T}], val) \in T_p\Big)$$

Note that although this method seems more efficient than the progress approach in terms of having shorter constraints, it has the drawback of missing some solutions that the previous approach can find. More specifically, for a process being in a legitimate local state in a deadlock state, it may be the case that taking a transition by this process leads to a state, from which its neighbors can take other transitions that finally leads to a legitimate state.

5.4 Heuristics Considering Uninterpreted Functions

Our second class of heuristics focus on uninterpreted functions. That is, we impose a constraint to change the interpretation function of at least one uninterpreted function in the deadlock state. Similar to the heuristics introduced for transition relations, we introduce two approaches for selecting at least one process to change the interpretation of its uninterpreted function. Because of the similarity to the previous heuristics, we skip the details of this heuristic.

6 Case Studies and Experimental Results

We used the model finder Alloy [15] and model checker NuSMV [7] to implement our counterexample-guided synthesis approach. Our experimental platform is an 2.9 GHz Intel Core i7 processor, with 16 GB of RAM. Our synthesis results are reported in Table 1.

6.1 Three Coloring

We consider the *three coloring problem* [14] on a ring, where each process π_i is associated with a variable c_i with domain $\{0, 1, 2\}$. Each value of the variable c_i represents a distinct color. A process can read and write its own variable. It can also read the variables of its neighbors. *LS* includes all states, where each process has a color different from its both neighbors. Thus, for a ring of 4 processes, *LS* is defined by the following predicate:

$$c_0(s) \neq c_1(s) \ \wedge \ c_1(s) \neq c_2(s) \ \wedge \ c_2(s) \neq c_3(s) \ \wedge \ c_3(s) \neq c_0(s)$$

Observe that the closure/deadlock-freedom cutoff point for this case study is $3^2 + 1 = 10$ and, hence, we need to synthesize a solution for 10 processes. The synthesis time reported in Table 1 is a bit smaller in the case of local *LS* heuristic, which is probably due to the smaller constraints added in this case. The resulting protocols for the two heuristics are different. Following is the one synthesized for the case of local LS:

$$\begin{aligned}
\pi_i : \quad & (c_i = 2) \wedge (c_{(i+1)} = 2) \wedge (c_{(i-1)} \neq 0) \quad \rightarrow \quad c_i := 0 \\
& (c_i \neq 0) \wedge (c_{(i+1)} = 1) \wedge (c_{(i-1)} = 2) \quad \rightarrow \quad c_i := 0 \\
& (c_i = 1) \wedge (c_{(i+1)} = 1) \wedge (c_{(i-1)} \neq 2) \quad \rightarrow \quad c_i := 2 \\
& (c_i = 0) \wedge (c_{(i+1)} \neq 2) \wedge (c_{(i-1)} = 0) \quad \rightarrow \quad c_i := 2 \\
& (c_i \neq 1) \wedge (c_{(i+1)} = 2) \wedge (c_{(i-1)} = 0) \quad \rightarrow \quad c_i := 1
\end{aligned}$$

6.2 One-Bit Maximal Matching

This case study is the running example in this paper with cutoff point of $2^2 + 1 = 5$ processes. Note that using the heuristics considering transition relations, we could not synthesize a protocol for this problem (Alloy reports unsatisfiablity after adding the counterexample constraints). The interesting point about this case study is that the progress heuristic has better efficiency compared to the local *LS*. The reason may be due to the fact that the constraints added in the local *LS* heuristic are too restrictive, and hence, Alloy needs to search more in order to find a solution. The synthesized solutions using both heuristics are the same for this case study, where the transition relation is the following:

$$\begin{aligned}
\pi_i : \quad & (x_i = \mathtt{F}) \wedge (x_{(i+1)} = \mathtt{F}) \wedge (x_{(i-1)} = \mathtt{F}) \quad \rightarrow \quad x_i := \mathtt{T} \\
& (x_i = \mathtt{T}) \wedge (x_{(i+1)} = \mathtt{T}) \quad \rightarrow \quad x_i := \mathtt{F}
\end{aligned}$$

Problem	cutoff #	Heuristic	Synthesis Time	Model Checking Time
Three Coloring	10	Local LS	7m 3sec	16 msec
Three Coloring	10	Progress	9m 5sec	16 msec
One-Bit MM	5	Local LS	1m 48sec	27 msec
One-Bit MM	5	Progress	1m 44sec	33 msec
Maximal Matching	10	Local LS	7m 59sec	36 msec
Maximal Matching	10	Progress	4m 57sec	37 msec
Maximal Independent Set	5	Local LS	10sec	18 msec

Table 1 Results for parameterized synthesis.

and the interpretation function for $match_i$ is the following:

$$
\begin{aligned}
match_i: \quad & (x_i = \mathtt{T}) \wedge (x_{(i+1)} = \mathtt{T}) \wedge (x_{(i-1)} = \mathtt{T}) \mapsto l \\
& (x_{(i+1)} = \mathtt{F}) \wedge (x_{(i-1)} = \mathtt{F}) \mapsto l \\
& (x_i = \mathtt{T}) \wedge (x_{(i+1)} = \mathtt{F}) \wedge (x_{(i-1)} = \mathtt{T}) \mapsto r \\
& (x_i = \mathtt{F}) \wedge (x_{(i+1)} = \mathtt{T}) \mapsto r \\
& (x_i = \mathtt{T}) \wedge (x_{(i+1)} = \mathtt{T}) \wedge (x_{(i-1)} = \mathtt{F}) \mapsto n \\
& (x_i = \mathtt{F}) \wedge (x_{(i+1)} = \mathtt{F}) \wedge (x_{(i-1)} = \mathtt{T}) \mapsto n
\end{aligned}
$$

6.3 Maximal Matching

In this case study, we used the same problem as in Section 6.2, but instead of using one Boolean variable for each process, we use a variable with three values $\{l, r, n\}$ and, hence, we do not need the uninterpreted functions anymore. The resulting protocols for the two heuristics are different. As an example, the synthesized protocol for the case of local LS is the following:

$$
\begin{aligned}
\pi_i: \quad & (x_i = n) \wedge (x_{(i+1)} = n) \wedge (x_{(i-1)} = n) \rightarrow x_i := r \\
& (x_i \neq r) \wedge (x_{(i+1)} \neq r) \wedge (x_{(i-1)} = l) \rightarrow x_i := r \\
& (x_i \neq n) \wedge (x_{(i+1)} = r) \wedge (x_{(i-1)} = l) \rightarrow x_i := n \\
& (x_i = n) \wedge (x_{(i-1)} = r) \rightarrow x_i := l \\
& (x_i = r) \wedge (x_{(i+1)} = r) \wedge (x_{(i-1)} \neq l) \rightarrow x_i := l
\end{aligned}
$$

6.4 Maximal Independent Set

An *independent set* in a graph is a subset of vertices in which no pair of vertices are adjacent. To synthesize a protocol that finds a maximal independent set, we consider a set of processes connected in a ring topology, where each process has a Boolean variable, the value of which shows whether or not it is included in the maximal independent set. The set of legitimate states include those states, where the processes whose variables have the true value form a maximal independent set. As an example, if c_i is the variable of the process π_i, then the set of legitimate states for the case of four processes is formulated by the following predicate:

$$(c_0(s) = \mathtt{T} \wedge c_1(s) = \mathtt{F} \wedge c_2(s) = \mathtt{T} \wedge c_3(s) = \mathtt{F}) \vee (c_0(s) = \mathtt{F} \wedge c_1(s) = \mathtt{T} \wedge c_2(s) = \mathtt{F} \wedge c_3(s) = \mathtt{T})$$

The point of this case study is that the resulting model for 4 processes worked for the size 5 as well, and hence, no counterexample is found. Therefore, the result for both heuristics is

the same. The resulting protocol is the following:

$$\pi_i: \quad (x_i = \mathtt{F}) \wedge (x_{(i+1)} = \mathtt{F}) \wedge (x_{(i-1)} = \mathtt{F}) \rightarrow x_i := \mathtt{T}$$
$$(x_i = \mathtt{T}) \wedge (x_{(i+1)} = \mathtt{T}) \rightarrow x_i := \mathtt{F}$$

7 Related Work

Regarding the synthesis of self-stabilizing algorithms, one approach is to *add* self-stabilization to a given algorithm. In contrast to our approach, the technique proposed by Ebnenasir and Farahat [5] starts from a given non-stabilizing algorithm, it requires a more explicit specification of the legitimate states, and it is not complete, i.e., it may fail to find a solution even though there exists one. Klinkhammer and Ebnenasir 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 [18], and introduce a new method for adding self-stabilization that is complete, but otherwise has the same limitations as mentioned above [20]. For ring topologies, they have shown that parameterized verification of self-stabilization is undecidable in uni-directional rings [19], while the parameterized synthesis problem is undecidable in bi-directional rings, but surprisingly remains decidable in uni-directional rings [21]. Faghih and Bonakdarpour introduced an SMT-based synthesis technique for automatically synthesizing self-stabilizing systems [8, 9] that is complete and not based on existing non-stabilizing algorithms. An extension of this work [11] allows to symbolically specify the legitimate states as a set of requirements, and supports the synthesis of ideal-stabilizing systems.

While these approaches are promising and can automatically synthesize a number of well-known self-stabilizing systems, they all suffer from the problem of scalability, as the complexity of the problem increases exponentially in the number of processes. For example, all results reported by Faghih and Bonakdarpour [8, 9, 11] correspond to automatically synthesis of self-stabilizing systems with at most 5 processes. One way to address this scalability issue in synthesis is to use a counterexample-guided synthesis method, as it has been proposed for the completion of program sketches [25], for the lazy synthesis of reactive systems [13], and for the synthesis of Byzantine-resilient systems [1]. The latter approach also supports the synthesis of self-stabilizing systems, but counterexamples are only used to guide the encoding of Byzantine-resilience, and the approach is limited to synchronous systems. In all of these examples, a counterexample-guided approach can solve problems that are out of reach for existing approaches. In our work, we for the first time used counterexamples to guide synthesis for an increasing size of the topology, which allows us to scale the SMT-based synthesis of self-stabilizing algorithms to systems with up to 200 processes.

Finally, the problem of scalability in the number of processes can be solved once and for all by using a parameterized synthesis approach, as introduced by Jacobs and Bloem [16] for (non-stabilizing) reactive systems. The approach relies on cutoff results, similar to the ones we introduced in this work for closure and deadlock detection. Different techniques are introduced in [17] to improve scalability of this approach in the complexity of the specification, including the modular application of cutoff results in synthesis. An extension of the approach [1] also supports the parameterized synthesis of self-stabilizing systems, but only for synchronous systems, and not in all cases resulting in a completely symmetric system. Finally, Lazic et al. [22] propose a method for synthesizing parameterized fault-tolerant distributed algorithms. In contrast to our approach, synthesis is based on a sketch of an asynchronous threshold-based fault-tolerant distributed algorithm, and the goal is to find the right values for coefficients that may be missing in the guards.

8 Conclusion

In this paper, we proposed a new method for parameterized synthesis of self-stabilizing algorithms in symmetric rings using cutoff points. Furthermore, in order to scale the existing synthesis solutions [8–12] up to the cutoff point, we introduced an iterative loop of synthesis and verification guided by counterexamples. We demonstrated the effectiveness of our approach by synthesizing parameterized self-stabilizing protocols for well-known problems including self-stabilizing three coloring, maximal matching, and maximal independent set in less then 10 minutes. For future, we plan to work on asymmetric and dynamic networks as well as the case, where the protocol is live in the set of legitimate states.

References

1. R. Bloem, N. Braud-Santoni, and S. Jacobs. Synthesis of self-stabilising and byzantine-resilient distributed systems. In *CAV*, pages 157–176, 2016.
2. S. Devismes, S. Tixeuil, and M. Yamashita. Weak vs. self vs. probabilistic stabilization. In *ICDCS*, pages 681–688, 2008.
3. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
4. E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, 1986.
5. A. Ebnenasir and A. Farahat. A lightweight method for automated design of convergence. In *IPDPS*, pages 219–230, 2011.
6. E. A. Emerson and K. S. Namjoshi. On reasoning about rings. *International Journal on Foundations of Computer Science.*, 14(4):527–550, 2003.
7. A. Cimatti et. al. Nusmv 2: An opensource tool for symbolic model checking. In *CAV*, pages 359–364, 2002.
8. F. Faghih and B. Bonakdarpour. SMT-based synthesis of distributed self-stabilizing systems. In *SSS*, pages 165–179, 2014.
9. F. Faghih and B. Bonakdarpour. SMT-based synthesis of distributed self-stabilizing systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(3):21, 2015.
10. F. Faghih and B. Bonakdarpour. ASSESS: A tool for automated synthesis of distributed self-stabilizing algorithms. In *SSS*, pages 219–233, 2017.
11. F. Faghih, B. Bonakdarpour, S. Tixeuil, and S. Kulkarni. Specification-based synthesis of distributed self-stabilizing protocols. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, pages 124–141, 2016.
12. F. Faghih, B. Bonakdarpour, S. Tixeuil, and S. Kulkarni. Specification-based synthesis of distributed self-stabilizing protocols. *Logical Methods in Computer Science*, To appear.
13. B. Finkbeiner and S. Jacobs. Lazy synthesis. In *VMCAI*, 2012.
14. M. G. Gouda and H. B. Acharya. Nash equilibria in stabilizing systems. In *SSS*, pages 311–324, 2009.
15. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press Cambridge, 2012.
16. S. Jacobs and R. Bloem. Parameterized synthesis. *Logical Methods in Computer Science*, 10(1), 2014.
17. A. Khalimov, S. Jacobs, and R. Bloem. Towards efficient parameterized synthesis. In *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 108–127. Springer, 2013.
18. A. Klinkhamer and A. Ebnenasir. On the complexity of adding convergence. In *Fundamentals of Software Engineering (FSEN)*, pages 17–33, 2013.
19. A. Klinkhamer and A. Ebnenasir. Verifying livelock freedom on parameterized rings and chains. In *SSS*, pages 163–177, 2013.

599 **20** A. Klinkhamer and A. Ebnenasir. Synthesizing self-stabilization through superposition and
600 backtracking. In *SSS*, pages 252–267, 2014.
601 **21** A. Klinkhamer and A. Ebnenasir. Synthesizing parameterized self-stabilizing rings with
602 constant-space processes. In *Fundamentals of Software Engineering (FSEN)*, pages 100–
603 115, 2017.
604 **22** Marijana Lazic, Igor Konnov, Josef Widder, and Roderick Bloem. Synthesis of distributed
605 algorithms with parameterized threshold guards. In *OPODIS*, 2017.
606 **23** F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A new self-stabilizing maximal matching
607 algorithm. *Theoretical Computer Science*, 410(14):1336–1345, 2009.
608 **24** Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transac-*
609 *tions on Computer Systems*, 7(1):61–77, 1989.
610 **25** Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.