Synthesizing Self-Stabilizing Parameterized Protocols with Unbounded Variables

Ali Ebnenasir

Michigan Technological University
Computer Science Technical Report
CS-TR-22-01
May 2022

Synthesizing Self-Stabilizing Parameterized Protocols with Unbounded Variables

Ali Ebnenasir
Department of Computer Science
Michigan Technological University
Houghton, MI 49931, U.S.A.
Email: aebnenas@mtu.edu

***Abstract*—The focus of this paper is on the synthesis of unidirectional symmetric ring protocols that are self-stabilizing. Such protocols have an unbounded number of processes and unbounded variable domains, yet they ensure recovery to a set of legitimate states from any state. This is a significant problem as many distributed systems should preserve their fault tolerance properties when they scale. While previous work addresses this problem for constant-space protocols where domain size of variables are fixed regardless of the ring size, this work tackles the synthesis problem assuming that both variable domains and the number of processes in the ring are unbounded (but finite). We present a sufficient condition for synthesis and develop a sound algorithm that takes a conjunctive state predicate representing legitimate states, and generates the parameterized actions of a protocol that is self-stabilizing to legitimate states. We characterize the unbounded nature of protocols as semilinear sets, and show that such characterization simplifies synthesis. The proposed method addresses a longstanding problem because recovery is required from any state in an unbounded state space. For the first time, we synthesize some self-stabilizing unbounded protocols, including a near agreement and a parity protocol.**

I. Introduction

This paper investigates the problem of synthesizing Self-Stabilizing unidirectional Symmetric ring protocols with Unbounded number of processes and unbounded variable domains, called SS-SymU protocols (a.k.a. unbounded uni-rings). A *process* contains a set of atomic actions. When an action of a process is executed, it is disabled until enabled again by the neighborning processes; i.e., *self-disabling* actions. In a *symmetric* ring, the actions of each process are generated from a *template* process by a simple variable re-indexing. A self-stabilizing protocol automatically recovers (in a finite number of steps) to a set of legitimate states $\mathcal{I}$ from *any* arbitrary state [1]; i.e., all states are initial states. Such recovery should be achieved without the intervention of a central authority. The significance of this synthesis problem is multi-fold. First, while uni-ring is a simple topology, it is of practical importance in distributed systems where the underlying communication topology may include cyclic structures. Second, the unboundedness of the ring size and variable domains is a requirement where networks scale up and buffer sizes grow. The elegance of many distributed protocols/algorithms (e.g., logical clocks [2], Dijkstra's token passing [1], unbounded registers [3]) is due to the assumption of unbounded variable domains and processes, which makes it significant to develop tools that can synthesize such protocols under the unboundedness assumption. Third, self-stabilization is an important fault tolerance property that enables decentralized recovery in the presence of transient faults, which perturb the system state without causing permanent damages. While previous work [4], [5], [6], [7], [8] addresses the verification and synthesis of parameterized symmetric uni-rings, the domain size of variables remains constant regardless of the ring size. To the best of our knowledge, this paper presents the first method for the synthesis of SS-SymU protocols that are unbounded in terms of both the number of processes and variable domains.

Most existing methods for the synthesis of self-stabilizing protocols either focus on fixed-size protocols or consider an unbounded number of processes only; variable domains are considered bounded. For example, specification-based methods [9] compose a pair of template processes to reason about the global safety and local liveness properties of parameterized synchronization skeletons. Methods for fixed-size synthesis [10], [11], [12], [13] consider a fixed upper bound k on the number of processes, and generate a solution that is correct up to k processes. To enable the synthesis of parameterized self-stabilizing systems where solutions work for an arbitrary number of n processes, some approaches rely on parameterized synthesis [14] where an implementation is generated for a parameterized specification and a parameterized architecture. Such methods employ bounded [15] and SMT-based [11] synthesis to show the correctness of a solution with cutoff number of processes, where a solution exists for a protocol with *cutoff* number of processes *iff* (if and only if) a solution exists for the parameterized protocol with unbounded number of processes. Other methods [7] present cutoffs for the synthesis of self-stabilizing protocols in symmetric networks, however, such cutoffs can be quadratic/exponential in the bounded variable domains depending on the structure of $\mathcal{I}$. Synthesis of parameterized systems with threshold guards [4] starts with a sketch automaton (whose transitions have incomplete guard conditions capturing the number of received messages), and complete the guards towards satisfying program specifications. Our previous work [5] addresses the synthesis of self-stabilizing parameterized protocols where the local state space of the template process remains constant.

Contributions. In contrast to most existing methods, we propose a novel approach based on the synthesis of semilinear sets in the unbounded local state space of the template process of SS-SymU for conjunctive predicates. Specifically, we start with a global state predicate $\mathcal{I} = \forall i \in \mathbb{N} :: L(x_{i-1}, x_i)$ where $L(x_{i-1}, x_i)$ denotes a local state predicate of the template process P_i and x_i is an abstraction of the local state of P_i. We then generate a protocol that self-stabilizes to $\mathcal{I}$ regardless of network size and the domain size of variables. Domain size is of particular importance as some protocols may not exist for specific domain sizes (e.g., Dijkstra's token ring [1] requires a domain size of at least $N-1$ in a ring of N processes). We utilize necessary and sufficient conditions identified in [5], [6] for the livelock-freedom of a solution with constant-space processes in order to impose a structure on the unbounded transition system of the template process. Such conditions require the existence of a value γ in the domain of x_i for which $L(\gamma, \gamma)$ holds. Moreover, necessary and sufficient conditions for livelock-freedom (under an unfair scheduler) require a tree-like structure rooted at γ for the local state transition system of the template process. While these results are for constant-space processes, we generalize them for unbounded domain sizes. Specifically, we show that if the state transition system of the template process is a semilinear set represented as an infinite tree rooted at γ, then a solution exists. A semilinear set is the finite union of a set of linear sets, where a linear set contains periodic integer vectors. Based on this sufficient condition, we develop a sound algorithm that takes $L(x_{i-1}, x_i)$ and generates the periodic linear sets of a semilinear set in a way that their vectors are organized in a potentially infinite tree rooted at γ. Each synthesized linear set represents the unbounded structure of a protocol action. We then use such linear sets to synthesize the parameterized actions of a protocol that self-stabilizes to $\mathcal{I}$ for unbounded number of processes and unbounded domain sizes. We demonstrate the proposed method using a near-agreement and a parity protocol.

Organization. Section II provides some basic concepts. Section III presents the proposed synthesis method. Section IV demonstrates the application of the synthesis method for some example protocols. Section V discusses related work. Section VI makes concluding remarks and discusses future research.

II. Preliminaries

This section represents the definition of state predicates, parameterized protocols and their representation as locality graphs (adopted from [16], [17], [5], [6]), and semilinear sets. Wlog, we use the term *parameterized protocol* to refer to uni-ring symmetric protocols that have both unbounded number of processes and unbounded variable domains. A protocol p includes $N > 1$ *symmetric* processes on a uni-ring, where the code of each process is derived from the code of a template process P_i by variable re-indexing. The template process P_i has a variable x_i whose domain abstracts the set of valuations to all writable variables of P_i. The domain of x_i, denoted $M = Dom(x_i)$, can be unbounded (but finite). Any *local state* of a process (a.k.a. *locality/neighborhood*) is determined by a unique valuation of its readable variables. We assume that any writable variable is also readable. Network topology defines the set of readable variables of a process. For example, in a uni-ring consisting of N processes, each process P_i (where $i \in \mathbb{Z}_N$, i.e., $0 \le i \le N-1$) has a predecessor P_{i-1}, where subtraction is in modulo N. That is, P_i can read the values of x_i and x_{i-1}, but can update only x_i. The *global state* of a protocol is defined by a snapshot of the local states of all processes. The *state space* of a protocol, denoted by Σ_p, is the universal set of all global states. A *state predicate* is a subset of Σ_p. A process *acts* (i.e., *transitions*) when it atomically updates its state based on its locality.

We assume that processes act one at a time (i.e., interleaving semantics). Thus, each *global transition* corresponds to the action of a single process from some global state. An *execution/computation* of a protocol is a sequence of states $s_0, s_1, \ldots, s_k$ where there is a transition from s_i to s_{i+1} for every $i \in \mathbb{Z}_k$. The transition function $\delta : \Sigma_p \times \Sigma_p \to \Sigma_p$ of the template process captures its set of actions $x_{i-1} = a \wedge x_i = b \longrightarrow x_i := c$, which can also be captured as triples of the form (a, b, c). That is, $\delta(a, b) = c$ *iff* (if and only if) P_i has an action $x_{i-1} = a \wedge x_i = b \longrightarrow x_i := c$. An action has two components; a *guard*, which is a Boolean expression in terms of readable variables and a *statement* that atomically updates the state (i.e., writable variables) of the process once the guard holds; i.e., the action is *enabled*. Previous work [18] shows that assuming self-disabling and deterministic processes simplifies synthesis without undermining soundness and completeness. An action (a, b, c) cannot co-exist with action (a, c, d) in a *self-disabling* process for any d. A *deterministic* process cannot have two actions enabled at the same time; i.e., an action (a, b, c) cannot co-exist with an action (a, b, d) where $d \neq c$.

Definition II.1 (Action Graph)**.** For a fixed domain size M, we can depict the set of actions of the template process of a symmetric uni-ring by a labeled directed multigraph $G = (V, A)$, called the *action graph*, where each vertex $v \in V$ represents a value in $\mathbb{Z}_M$, and each arc $(a, c) \in A$ with a label b captures an action $x_{i-1} = a \wedge x_i = b \longrightarrow x_i := c$.

For example, consider the Parity protocol introduced in [6]. Each process P_i has a variable $x_i \in \mathbb{Z}_3$ (i.e., $M = 3$) and actions $x_{i-1} = 0 \wedge x_i = 1 \longrightarrow x_i := 0$, $x_{i-1} = 1 \wedge x_i = 2 \longrightarrow x_i := 0$, and $x_{i-1} = 2 \wedge x_i = 1 \longrightarrow x_i := 0$. This protocol ensures that, from any global state of a symmetric uni-ring, a state is reached where processes agree on a common odd/even parity. We formally specify these states as the state predicate $\mathcal{I}_{Par} = \forall i \in \mathbb{Z}_N : ((|x_{i-1} - x_i| \bmod 2 = 0)$. Throughout this paper, the subscript operations are modulo number of processes, and the arithmetic operations in the state predicates, and in the guard and assignment of actions are performed modulo M. Figure 7b illustrates this protocol as an action graph containing arcs $(0, 1, 1), (1, 0, 1), (1, 2, 1)$, and $(2, 1, 0)$. This is a multigraph because of the arcs $(1, 0, 1), (1, 2, 1)$.

Definition II.2 (Self-Stabilization and Convergence)**.** A protocol p is *self-stabilizing* [1] to a state predicate $\mathcal{I}$ *iff* from any state in $\neg\mathcal{I}$, *every* computation of p reaches a state in I (i.e., *convergence*) and remains in $\mathcal{I}$ (i.e., *closure*). A state predicate $\mathcal{I}$ is *closed* in p *iff* there is no transition (s, s'), where $s \in \mathcal{I}$ and $s' \notin \mathcal{I}$. Notice that, convergence of p to $\mathcal{I}$ requires that p does not reach a deadlock, nor does it reach a livelock in $\neg\mathcal{I}$. A *deadlock* state is a global state where no process has any enabled action. A *livelock* of a protocol p is an infinite cyclic computation $l = \langle s_0, s_1, \cdots, s_0\rangle$, where s_i is a global state, for $i \geq 0$.

Definition II.3 (Locality Graph)**.** Consider a global state predicate $\mathcal{I} = \forall i \in \mathbb{Z}_N : L(x_{i-1}, x_i)$ for a protocol, and a domain size M. The local predicate $L(x_{i-1}, x_i)$ captures a set of local states, representing an acceptable relation between the states of each process P_i and the states of its predecessor P_{i-1}. We represent $L(x_{i-1}, x_i)$ as a digraph $G = (V, A)$, called the *locality graph*, such that each vertex $v \in V$ represents a value in $\mathbb{Z}_M$, and an arc (a, b) is in A *iff* $L(a, b)$ holds.

Figure 7a illustrates the locality graph of the Parity protocol introduced in this section for the state predicate $\mathcal{I}_{SN2}$. We have extensively studied [5], [6] the use of locality and action graphs in reasoning about global properties (e.g., livelocks). Our previous work [17], [5] investigates the following synthesis problem, whereas in Section III we solve this problem when its assumption is lifted.

Problem II.4 (Synthesis of Symmetric Uni-Rings)**.**

- **Input**: $L(x_{i-1}, x_i)$, and the domain size M of x_i.
- **Output**: The transition function δ (represented as an action graph or parameterized actions) of a protocol p such that the entire ring is self-stabilizing to $\mathcal{I} = \forall i : i \in \mathbb{Z}_N : L(x_{i-1}, x_i)$ for any ring size $N \geq 3$.
- **Assumption**: M is fixed regardless of the ring size N; i.e., p has *constant-space* processes.

The following theorem (proved in [17], [5]) provides the foundation of a synthesis method for parameterized uni-rings with constant-space processes. In the rest of this section, we present an overview of the synthesis method of [5] since its knowledge is required for our exposition.

Theorem II.5. *There is a symmetric uni-ring protocol p (with deterministic, self-disabling and constant-space processes) that self-stabilizes to $\mathcal{I} = \forall i \in \mathbb{Z}_N$: $L(x_{i-1}, x_i)$ for an unbounded (but finite) number of N processes iff there is a vertex γ in the locality graph G of $L(x_{i-1}, x_i)$, where $L(\gamma, \gamma)$ holds, and the action graph of p is a directed spanning tree of G, sinking at γ as its root [17], [5].*

Algorithm 1 (introduced in [5]) takes as input the local predicate $L(x_{i-1}, x_i)$ and generates the set of parameterized actions of a self-stabilizing uni-ring protocol. For example, Step 1 takes the local predicate $(|x_{i-1} - x_i| \bmod 2 = 0)$ of $\mathcal{I}_{Par}$ in Parity with domain size 3, and initially generates its locality graph illustrated in Figure 7a. This occurs because there is some γ for which $L(\gamma, \gamma)$ holds. Selecting γ as 1, Algorithm 2 generates the spanning tree of Figure 7b in Step 3 (excluding the labels). Notice that, the output of Algorithm 2 is a spanning tree of the locality graph of $L(x_{i-1}, x_i)$ rooted at γ, including a self-loop on γ. Step 4 of Algorithm 1 then includes the arc labels, where a value b becomes a label for an arc (a, c) *iff* $\neg L(a, b) \wedge (b \neq c)$. For example, when labeling the arc $(1, 1)$ in Figure 7b , $a = 1$, and the algorithm looks for any value b in $\mathbb{Z}_M$ such that $(|1 - b| \bmod 2) \neq 0$ modulo 3. For $M = 3$, the values 0 and 2 are the only acceptable labels.

Algorithm 1. *SynUniRing($L(x_{i-1}, x_i)$: state predicate, M: domain size)*

```
1: Check if a value $\gamma \in \mathbb{Z}_M$ exists such that $L(\gamma, \gamma) =$ **true**.
2: If no such $\gamma$ exists, then **return** $\emptyset$ and declare that no solution exists.
3: $\tau :=$ *ConstructSpanningTree*$(L(x_{i-1}, x_i), M, \gamma)$.
4: Transform $\tau$ into an action graph of a protocol by the following step:
       For each arc $(a, c)$ in $\tau$, where $a, c \in \mathbb{Z}_M$, label $(a, c)$ with every value $b$ for which $L(a, b) =$ **false** and $b \neq c$.
5: Return the actions represented by the arcs of $\tau$.
end
```

Algorithm 2. *ConstructSpanningTree($L(x_{i-1}, x_i)$: state predicate, M: positive integer, $\gamma \in \mathbb{Z}_M$)*

```
1: Construct the locality graph $G = (V, A)$ of $L(x_{i-1}, x_i)$ for domain size $M$.
2: Induce a subgraph $G' = (V', A')$ that contains all arcs of $G$ that participate in cycles involving $\gamma$.
3: Construct a spanning tree $\tau$ rooted at $\gamma$ for $G'$. Use backward reachability to construct the spanning tree.
4: For each node $v \in G$ that is absent from $G'$, include an arc from $v$ to the root of $\tau$. The resulting graph would still be a tree, denoted $\tau'$.
5: Include a self-loop $(\gamma, \gamma)$ at the root of $\tau'$.
6: Return $\tau'$.
end
```

Theorem II.5 explains why Algorithm 2 includes a self-loop at the root γ (in Step 7). Moreover, the reason why Algorithm 1 constructs a spanning tree is to ensure deadlock and livelock-freedom. We have shown [5] that the existence of such a spanning tree is necessary and sufficient for convergence to $\mathcal{I}$ in symmetric uni-rings with constant-space processes.

Definition II.6 (Vector)**.** A vector of dimension $d \geq 1$ of non-negative integers is a tuple $(a_1, a_2, \cdots, a_d) \in \mathbb{N}^d$, where $a_i \in \mathbb{N}$ for $1 \leq i \leq d$.

Definition II.7 (Linear Set)**.** Any non-empty subset of $\mathbb{N}^d$ is *linear* [19] if it can be represented as a periodic set of vectors $\mathcal{L} = \{v_b + \Sigma_{i=1}^{n} \lambda_i \cdot p_i : \lambda_i \in \mathbb{N}\}$, $v_b \in \mathbb{N}^d$ is the *base vector* and $\{p_1, \cdots, p_n\} \subseteq \mathbb{N}^d$ is a finite set of *period vectors*.

For example, a singleton set $\mathcal{L}_1 = \{(5, 7)\}$ is linear (with dimension $d = 2$) because the base vector is $(5, 7)$, and there is a unique period vector $(0, 0)$. Moreover, the linear set $\mathcal{L}_2 = \{(3, 2), (4, 3), (5, 4), \cdots\}$ has a base vector $(3, 2)$ and a period vector $p_1 = (1, 1)$. That is, $\mathcal{L}_2 = \{v_b + \lambda p_1 : \lambda \in \mathbb{N}\}$, where $v_b = (3, 2), n = 1, d = 2, p_1 = (1, 1)$, and $\lambda \in \mathbb{N}$.

Definition II.8 (Semilinear Set)**.** A *semilinear* set [19] is a finite union of some linear sets. Semilinear sets provide a finite representation for finite and infinite subsets of $\mathbb{N}^d$.

Ginsburg and Spanier [20] show that semilinear sets capture the sets of integers that are definable in the first-order theory of integers with addition and order; i.e., Presburger arithmetic. Semilinear sets are closed under Boolean operations [20].

III. Synthesis Method

This section first presents a sufficient condition for the existence of a SS-SymU protocol, and then provides a sound algorithm for generating such protocols. We use the Near Agreement (NA) protocol as a running example to ease the presentation of this section.

Problem Statement. We solve Problem II.4 without its assumption of constant-space processes. That is, processes have unbounded (but finite) state spaces due to the unboundedness of variable domains.

Example: Near Agreement (NA) Protocol. A node P_i in a ring of N symmetric nodes *nearly agrees* with P_{i-1} *iff* $(x_{i-1} = x_i) \vee (x_{i-1} = x_i + 1)$, where subtraction is in modulo N and addition is done modulo M. Thus, the entire ring should self-stabilize to $\mathcal{I}_{NA} = \forall i \in \mathbb{N} :: L(x_{i-1}, x_i)$, where $L(x_{i-1}, x_i) \equiv (x_{i-1} = x_i) \vee (x_{i-1} = x_i + 1)$. Figure 3a illustrates the locality graph of $L(x_{i-1}, x_i)$ for $M = 3$. Our objective is to synthesize a self-stabilizing NA protocol that is correct regardless of the number of processes in the ring and the domain size M.

A. Sufficient Condition for Solvability

Since Algorithm 1 is a sound and complete algorithm for any fixed domain size M, one can enumeratively increase the domain size and utilize Algorithm 1 to generate a self-stabilizing protocol for each particular M. However, such an approach would not bear fruit for unbounded domain sizes unless we can ensure that the structure of the spanning tree (and in turn the action graph) that Algorithm 1 generates for M, will be inductively preserved for $M+1$ and beyond. This is a challenge because when the domain size increases to $M+1$, the locality graph of $L(x_{i-1}, x_i)$ may be totally different; i.e., there will be new arcs and some arcs may be removed. For example, observe how the locality graphs in Figures 2a and 3a change when M is increased from 2 to 3 for the NA protocol. To ensure that the spanning tree's structure would be preserved when domain size increases, one approach is to keep the arcs of the spanning tree τ_M for domain size M, and systematically include one more arc (a, a') in τ_M to derive another spanning tree τ_{M+1} for the domain size $M+1$. In turn, expanding the domain of x_i from $M+1$ to $M+2$ should ensure that τ_{M+2} preserves all arcs of τ_{M+1} and includes an additional arc (b, b') through some function f such that $f[(a, a')] = (b, b')$ and $b = M+1$ modulo $M+2$. Moreover, if $f[(b, b')] = (c, c')$ when the domain size increases to $M+3$, then $c - b = b - a$ and $c' - b' = b' - a'$ must hold. That is, the growth of the spanning tree must be periodic. Moreover, the root remains to be γ. If such conditions are met, then for any domain size M, the conditions of Theorem II.5 hold. Since the vertices of the spanning tree are non-negative integers, each arc (a, b) in a tree is an integer vector. As such, the vector (a, a') would be the base vector of a linear set and $(b - a, b' - a')$ gives the period vector of that linear set. Each one of the arcs in the first tree τ_M for the initial domain size M would also form a finite linear set. Therefore, the arcs of the unbounded spanning tree would form a semilinear set.

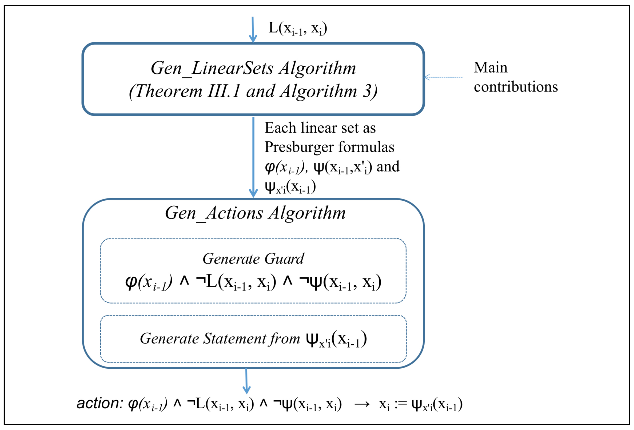
Theorem III.1. *Let $\mathcal{I} = \forall i \in \mathbb{N} :: L(x_{i-1}, x_i)$, and let there be a value γ for which $L(\gamma, \gamma)$ holds starting from some domain size M onward. If the arcs of the γ-rooted spanning trees built for each domain size $k \geq M$ represent the periodic growth of a semilinear set, then there is a symmetric uni-ring protocol that self-stabilizes to $\mathcal{I}$ regardless of the ring size and domain size. (Proof is due to Algorithm 3 and its soundness.)*

B. Overview of the Synthesis Method

An implication of Theorem III.1 is that we no longer have a finite spanning tree. Instead, we have an unbounded set of spanning trees $\tau_0, \tau_1, \cdots$ as the domain size M grows. Put it another way, for an unbounded domain size, we have an unbounded spanning tree that has an unbounded branching factor, or an unbounded depth (or both). *How do we formally represent such unbounded structures to facilitate the synthesis of actions*? Theorem III.1 points us to semilinear sets. For example, Algorithm 1 generates the tree in Figure 2b for the NA protocol and domain size 2, whose arcs represent a set of integer vectors $\{(1,1), (0,1)\}$. Likewise, the trees in Figures 3b to 5b respectively capture these three sets of integer vectors: $\{(1,1), (0,1), (2,1)\}, \{(1,1), (0,1), (2,1), (3,2)\}$ and $\{(1,1), (0,1), (2,1), (3,2), (4,3)\}$ for domain sizes 3 to 5. Notice that, the vectors $(1,1)$ and $(0,1)$ exist in the *intersection* of all four sets and will be there for larger domain sizes too. We call this set of vectors the *common core* of an SS-SymU protocol, denoted $\mathcal{C}$. The remaining vectors can be generalized as the linear set $\mathcal{UC} = \{(2,1), (3,2), \cdots\}$ with the base vector $(2,1)$ and the period vector $(1,1)$. We call the linear set $\mathcal{UC}$ the *unbounded core* of the protocol. Since the common core is finite, each vector in it can be represented as a linear set. Thus, we first generate the linear sets of a semilinear set that represents the unbounded spanning tree of a protocol (Figure 1). Then, we synthesize a parameterized action from each linear set.

C. Generating Linear Sets

This section presents an algorithm for the generation of a semilinear set representing the unbounded spanning tree of a protocol. This problem is divided into the formal specifications of the common and unbounded cores of a protocol as linear sets. A tree is acceptable as long as it has a vertex corresponding to each value in a domain size M and its root is a value $\gamma \in \mathbb{Z}_M$ for which $L(\gamma, \gamma)$ holds. Algorithm 3 generates the linear sets of an unbounded tree as Presburger formulas. Naturally, we start with the domain size of 2. Steps 2 and 3 of Algorithm 3 search for a value γ for which $L(\gamma, \gamma)$ holds for two consecutive odd and even domain sizes. This search continues up to a preset upper bound $\mathcal{B}$. Without such an upper bound, the algorithm may never terminate. Step 4 invokes Algorithm 2 for the construction of a spanning tree for M and γ found in Step 3. The common core $\mathcal{C}$ (see Step 5) then includes the integer vectors corresponding to the arcs of the spanning tree τ built in Step 4. After forming the common core, Algorithm



$L(x_{i-1}, x_i)$
Gen_LinearSets Algorithm (Theorem III.1 and Algorithm 3)
Main contributions
Each linear set as Presburger formulas $\varphi(x_{i-1})$, $\psi(x_{i-1},x'_i)$ and $\psi_{x'_i}(x_{i-1})$
Gen_Actions Algorithm
Generate Guard
$\varphi(x_{i-1}) \land \neg L(x_{i-1}, x_i) \land \neg\psi(x_{i-1}, x_i)$
Generate Statement from $\psi_{x'_i}(x_{i-1})$
action: $\varphi(x_{i-1}) \land \neg L(x_{i-1}, x_i) \land \neg\psi(x_{i-1}, x_i) \rightarrow x_i := \psi_{x'_i}(x_{i-1})$

Fig. 1: Overview of the proposed synthesis method.

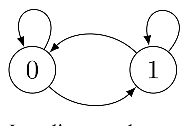
3 increases the domain size in Step 6. Such an increase introduces a new value in the domain of x_i, denoted v_M, which corresponds to a new vertex added to τ. To determine how v_M should be included in the tree, Algorithm 3 identifies the set U of all vertices u for which $L(v_M, u)$ holds. We ignore the arcs $L(u, v_M)$ because connecting any non-leaf node to v_M creates a cycle in the tree. Moreover, connecting a leaf node l to v_M would result in two parents for l. Thus, the only option for connecting v_M to the tree is to include an outgoing arc from v_M to some other tree node. If the set U is empty (Step 7), then v_M is directly connected to the root γ; i.e., an arc (v_M, γ) is included in τ. In this case, we consider (v_M, γ) as the base vector of a linear set and $(1, 0)$ as the period vector. Such a linear set captures the unbounded growth of the domain size as new arcs connected to the root. That is, the root γ would have an unbounded number of children. If U is non-empty (Step 8), then a value $w \in U$ is randomly selected to be the parent of v_M in the tree; i.e., the arc (v_M, w) is included in the tree. Every time the domain size increases, the value of v_M is incremented. For this reason, the first element of the period vector must be 1. For simplicity, we consider the growth of w in an incremental fashion too. That is, the period vector is $(1, 1)$ and the base vector is (v_M, w). Overall, Steps 7 and 8 determine the values of the base vector (b, b') and the period vector (p, p') of the unbounded core.

Algorithm 3. *Gen_LinearSets($L(x_{i-1}, x_i)$: state predicate, $\mathcal{B}$: positive integer)*

```
 1: $M := 2$.
 2: If $M \geq \mathcal{B}$ then declare that $\gamma$ could not be found
    and exit; // Upper bound reached.
 3: If there is a solution for some value $\gamma$ where
    $L(\gamma, \gamma)$ holds modulo $M$ and $M + 1$, then go
    to Step 4; otherwise, $M := M + 1$ and go to Step
    2.
 4: $\tau :=$ ConstructSpanningTree$(L(x_{i-1}, x_i), M, \gamma)$.
 5: $\mathcal{C} := S_\tau$ where $S_\tau$ represents the set of arcs of $\tau$
    as a set of integer vectors. // The common core
    detected
 6: $M' := M + 1$ and let $v_M$ denote the new vertex
    (i.e., value $M$ modulo $M'$) due to domain size
    increase. Calculate the set $U = \{u \mid L(v_M, u)$
    holds $\}$;
 7: If $U = \emptyset$ then include arc $(v_M, \gamma)$ every time
    the domain is increased. Set the base vector to
    $(v_M, \gamma)$, and the period vector to $(1, 0)$. Thus,
    $(b, b') := (v_M, \gamma)$, and $(p, p') := (1, 0)$. // Unbounded core $\mathcal{UC}$.
 8: Else select an arc $(v_M, w)$ for some value $w \in U$
    as the base vector. Set the base vector to $(v_M, w)$,
    and the period vector to $(1, 1)$. Thus, $(b, b') :=$
    $(v_M, w)$, and $(p, p') := (1, 1)$. // Unbounded core
    $\mathcal{UC}$.
 9: For each integer vector $(c, d) \in \mathcal{C}$, return formulas
    $\phi(x_{i-1}) \overset{\text{def}}{=} (x_{i-1} = c)$, $\psi(x'_i) \overset{\text{def}}{=} (x'_i = d)$, and
    $\psi_{x'_i}(x_{i-1}) \overset{\text{def}}{=} d$.
10: Corresponding to the unbounded core $\mathcal{UC}$ con-
    structed in Steps 7 and 8, return formulas
    $\phi(x_{i-1}) \overset{\text{def}}{=} (x_{i-1} = b + \lambda p)$, $\psi(x_{i-1}, x'_i) \overset{\text{def}}{=} (x'_i =$
    $x_{i-1} + (b' - b) + \lambda(p' - p))$, and $\psi_{x'_i}(x_{i-1}) \overset{\text{def}}{=}$
    $(x_{i-1} + (b' - b) + \lambda(p' - p))$, where $\lambda \in \mathbb{N}$.
end
```

Steps 9 and 10 specify the linear sets corresponding to the common core and the unbounded core as Presburger formulas [20]. Each integer vector (a, b) in a linear set actually represents an atomic action of the protocol specified as $x_{i-1} = a \land C(x_{i-1}, x_i) \rightarrow x_i := b$, where $C(x_{i-1}, x_i)$ is a Boolean expression specified in terms of x_i and x_{i-1}. Since the second element of each vector (a, b) represents the updated value of x_i, we use the notation x'_i instead of x_i when formally specifying the linear sets of a semilinear set. For example, we specify the linear set $\{(0, 1)\}$ as $x_{i-1} = 0 \land x'_i = 1$. Each such formula provides an incomplete sketch of an action, which should be completed in subsequent steps of synthesis. In general, we specify a linear set $\mathcal{L}$ with the base vector (b, b') and the period vector (p, p') as $\{(x_{i-1}, x'_i) \mid \forall\lambda \in \mathbb{N} :: (x_{i-1} = b + \lambda p) \land (x'_i = b' + \lambda p')\}$. Since x_{i-1} and x'_i are free variables and λ is known to be a natural value, we eliminate the quantifications in Steps 9 and 10 of Algorithm 3. Let $\mathcal{F}_1 = (x_{i-1} = b + \lambda p)$ and $\mathcal{F}_2 = (x'_i = b' + \lambda p')$. Subtracting $\mathcal{F}_1$ from $\mathcal{F}_2$ relates x'_i with x_{i-1} as $\psi(x_i, x'_i) \stackrel{\text{def}}{=} x'_i = x_{i-1} + (b' - b) + \lambda(p' - p)$ (Step 10). Factoring out x'_i, we get $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} + (b' - b) + \lambda(p' - p))$. In fact, $\psi_{x'_i}(x_{i-1})$ represents the expression that should be assigned to x_i in the action corresponding to the linear set $\mathcal{L}$.

The NA protocol. Figures 2a and 2b respectively represent the locality graph and the spanning tree of NA for $M = 2$. Figures 3 to 5 illustrate the locality graphs and the spanning trees for domain sizes 3 to 5. The semilinear set of the NA protocol can be specified as the union of the following linear sets:


(a) Locality graph representing predicate $L(x_{i-1}, x_i)$ in NA.

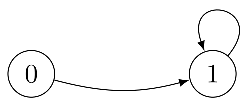
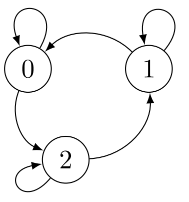

(b) A spanning tree rooted at 1.

Fig. 2: Locality graph and a spanning tree of NA for $M = 2$.


(a) Locality graph representing predicate $L(x_{i-1}, x_i)$ in NA.

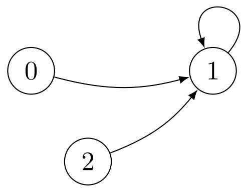
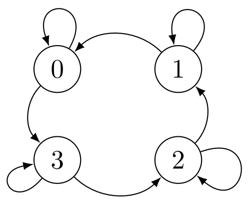

(b) A spanning tree rooted at 1.

Fig. 3: Locality graph and a spanning tree of NA for $M = 3$.

- *Linear set 1*: The base vector is $(1, 1)$, and the period vector is $(0, 0)$. That is, for the unbounded domain M, this set would be equal to $\{(x_{i-1}, x'_i) \mid x_{i-1} = (1 + \lambda 0) = 1$ and $x'_i = (1 + \lambda 0) = 1$ where $\lambda \in \mathbb{N}\}$. Since the period vector is $(0, 0)$, this set includes just a single vector; i.e., $\{(1, 1)\}$. Thus, we have $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} = 1)$, $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = 1)$ and $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} 1$ for this linear set.
- *Linear set 2*: The base vector is $(0, 1)$, and the period vector is $(0, 0)$. Thus, we have $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} = 0)$, $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = 1)$ and $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} 1$.


(a) Locality graph representing predicate $L(x_{i-1}, x_i)$ in NA.

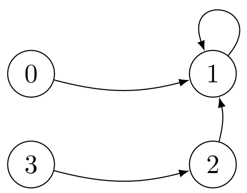
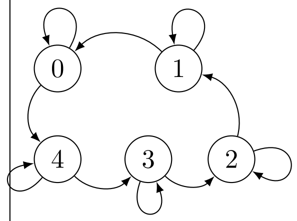

(b) A spanning tree rooted at 1.

Fig. 4: Locality graph and a spanning tree of NA for $M = 4$.


(a) Locality graph representing predicate $L(x_{i-1}, x_i)$ in NA.

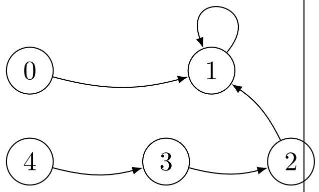

(b) A spanning tree rooted at 1.

Fig. 5: Locality graph and a spanning tree of NA for $M = 5$.

- *Linear set 3*: Using the base vector $(2, 1)$, and the period vector $(1, 1)$, this linear set is specified as $\{(x_{i-1}, x'_i) \mid x_{i-1} = 2 + \lambda$ and $x'_i = 1 + \lambda$ where $\lambda \in \mathbb{N}\}$. Step 10 gives $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} = 2 + \lambda)$, which means $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} \geq 2)$. Moreover, we have $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = x_{i-1} - 1)$, and $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} - 1)$.

The union of the above linear sets forms a semilinear set that captures the unbounded spanning tree of the NA protocol.

Theorem III.2. *Algorithm 3 terminates.*

Proof. Since the upper bound $\mathcal{B}$ is a finite value, Algorithm 3 exits in Step 2 if there is no $\gamma \in \mathbb{Z}_M$ for any M up to $\mathcal{B}$. Otherwise, Algorithm 3 jumps to Step 4, and moves through the remaining unconditional statements and terminates. $\square$

Theorem III.3. *Algorithm 3 is sound. That is, it correctly generates a semilinear set representing an unbounded spanning tree rooted at γ.*

Proof. We prove two cases. First, we show that the common core $\mathcal{C}$ constructed in Step 5 is a finite union of some linear sets. Due to the correctness of Algorithm 2 (shown in [5]), the structure τ generated in Step 4 is actually a finite spanning tree for the value of M (in Step 4). Thus, the set of arcs of τ, denoted S_τ, is a finite set of integer vectors. Each vector $(a, b) \in S_\tau$ can be considered as the base vector of a linear set with the period vector $(0, 0)$. That is, S_τ is a finite union of some linear sets.

Second, we show that the union of τ and the unbounded core generated in Steps 7 and 8 is an unbounded spanning tree rooted at γ. We show this by induction on M.

- *Base Case*: The initial value of M, denoted M_{init}, is the value of M in Step 4. The correctness of Algorithm 2 ensures that τ generated in Step 4 is a spanning tree of $L(x_{i-1}, x_i)$ rooted at γ for M_{init}. Moreover, as mentioned before, the common core is a finite union of linear sets; i.e., a semilinear set.
- *Inductive hypothesis*: For a domain size $k > M_{init}$, the structure built by the linear set generated in Steps 7 and 8 is a spanning tree rooted at γ for domain size k. Let τ_k denote this spanning tree. Since k is a finite value, a reasoning similar to the base case shows that the set of vectors of τ_k form a semilinear set.
- *Inductive step*: We start with τ_k. By the hypothesis, $\tau_k = (V_k, A_k)$ is a spanning tree rooted at γ for domain size k, where V_k denotes the set of vertices of τ_k and A_k represents its set of arcs. Incrementing the domain size to $k + 1$, we add a new vertex u_k to V_k, where u_k is actually equal to the value k modulo $k + 1$. The vertex u_k is connected to τ_k by one of the two arcs identified in Steps 7 and 8. That is, we include either an arc (u_k, γ) or an arc (u_k, w) for some value w modulo k, but not both. The arc (u_k, γ) preserves the tree structure and connects u_k to τ_k. The resulting structure is a spanning tree modulo $k + 1$. If we include (u_k, w), then the tree structure is preserved because $0 \leq w \leq k - 1$; i.e., the parent of u_k is a value modulo k. This means that an existing node in τ_k would have the new child u_k. The process of including a new vertex and arc does not change the root; i.e., the root remains γ. Thus, no cycles are formed and the resulting tree, denoted τ_{k+1}, is a connected graph over $V_k \cup \{u_k\}$ rooted at γ.

  We now show that the growth of the unbounded core is periodic. Observe that, either in Step 7 or in Step 8, Algorithm 3 identifies the period vector to be either $(1, 0)$ or $(1, 1)$. This means that every pair of vectors (a, a') and (b, b') that are *consecutively* included in the unbounded core satisfy either $(b-a = 1) \wedge (b' = a')$ or $(b-a = 1) \wedge (b'-a' = 1)$, where a, a', b, b' are non-negative integers. Such periodicity implies the existence of a linear set corresponding to the unbounded core with the base vector (b, b') and the period vector (p, p') identified in Step 7 or 8.

$\square$

D. Synthesizing Parameterized Actions from Linear Sets

This section presents a method for the synthesis of parameterized actions of self-stabilizing protocols from linear sets. Each linear set in the semilinear set represents the structure of an individual action in a protocol with deterministic and self-disabling process. However, such a structure lacks details of the guard and statement of each action. Thus, the question is: *how do we synthesize the guard of each action*? and *how do we synthesize the statement of each action*? The guard of each action includes three components: (1) its structure (taken from a linear set); (2) $\neg L(x_{i\ominus 1}, x_i)$, and (3) the self-disabling condition, which is the negation of the statement of the action. Since a linear set contains integer vectors (a, b) where a represents the value that x_{i-1} should have before the value of x_i is updated to b, the first component of a guard includes all values of x_{i-1} that make the formula $\phi(x_{i-1})$ true, and the statement of the guard should make $\psi(x_{i-1}, x'_i)$ true. Moreover, an action is enabled for all values of x_i (in the current state of a process) that make $L(x_{i-1}, x_i)$ false, which is why $\neg L(x_{i-1}, x_i)$ is a part of the guard condition. The statement of the action should make $L(x_{i-1}, x_i)$ true. Moreover, once an action is executed, it should disable itself; i.e., self-disabling assumption. This means that the guard of an action should contain the negation of the expression that holds after the execution of the action. Thus, the third component of a guard is $\neg\psi(x_{i-1}, x_i)$. In the computation of $\psi(x_{i-1}, x_i)$, Algorithm 4 uses the values of x_{i-1} and x_i in the current state of process P_i, before x_i is updated. In summary, the guard of each action would be equal to $\phi(x_{i-1}) \wedge \neg L(x_{i-1}, x_i) \wedge \neg(x_i = \psi_{x'_i}(x_{i-1}))$ (see Algorithm 4). Since x'_i represents the updated value of x_i in $\psi(x_{i-1}, x'_i)$, one can refactor $\psi(x_{i-1}, x'_i)$ in order to generate $\psi_{x'_i}(x_{i-1})$, which denotes $\psi(x_{i-1}, x'_i)$ modulo x'_i. That is, $\psi_{x'_i}(x_{i-1})$ treats x'_i as a function of x_{i-1}. This way, we create the assignment $x_i := \psi_{x'_i}(x_{i-1})$ in Line 2 of Algorithm 4.

Algorithm 4. *Gen_Actions($\phi(x_{i-1}), \psi(x_{i-1}, x'_i)$): Presburger formula corresponding to a linear set,*

```
L(x_{i-1}, x_i): State predicate)
    1: $\mathcal{G} \overset{\text{def}}{=} \phi(x_{i-1}) \wedge \neg L(x_{i-1}, x_i) \wedge \neg(x_i = \psi_{x'_i}(x_{i-1}))$
    2: $\mathcal{A} \overset{\text{def}}{=} (x_i := \psi_{x'_i}(x_{i-1}))$
    3: Return $\mathcal{G} \to \mathcal{A}$
end
```

1) Example: Synthesis of the Actions of the NA Protocol: We first demonstrate how we generate the action corresponding to the linear set $(1, 1)$. We take the output of Algorithm 3 for this linear set (i.e., $\phi(x_{i-1}) \overset{\text{def}}{=} (x_{i-1} = 1)$, $\psi(x_{i-1}, x'_i) \overset{\text{def}}{=} (x'_i = 1)$ and $\psi_{x'_i}(x_{i-1}) \overset{\text{def}}{=} 1)$ and generate its action. The components of the guard of the first action capturing the self-loop on 1 include the following:

- $\neg L(x_{i-1}, x_i)$: Since $L(x_{i-1}, x_i) = (x_{i-1} = x_i) \vee (x_{i-1} = x_i + 1)$, we include the constraint $(x_{i-1} \neq x_i) \wedge (x_{i-1} \neq x_i + 1)$ in the guard of this action.
- *Linear set constraint*: This linear set imposes the constraint $\phi(x_{i-1}) \equiv (x_{i-1} = 1)$ on the guard of the action.
- *Self-disabling constraint*: We use $\psi(x_{i-1}, x'_i) \overset{\text{def}}{=} (x'_i = 1)$ to specify this constraint. To this end, we first determine the assignment of the action using $\psi_{x'_i}(x_{i-1}) \overset{\text{def}}{=} 1$. Thus, the assignment is just $x_i := 1$. As a result, the self-disabling constraint is the negation of $x_i = 1$; i.e., $x_i \neq 1$.

Thus, the synthesized action is $(x_{i-1} = 1) \wedge (x_{i-1} \neq x_i) \wedge (x_{i-1} \neq x_i + 1) \wedge (x_i \neq 1) \to x_i := 1$. Likewise, the action generated from the linear set $\{(0, 1)\}$ is $(x_{i-1} = 0) \wedge (x_{i-1} \neq x_i) \wedge (x_{i-1} \neq x_i + 1) \wedge (x_i \neq 1) \to x_i := 1$.

We now generate the action corresponding to the linear set $\{(x_{i-1}, x'_i) \mid x_{i-1} = 2 + \lambda$ and $x'_i = 1 + \lambda$ where $\lambda \in \mathbb{N}\}$. Corresponding to this unbounded linear set, Algorithm 3 generates $\phi(x_{i-1}) \overset{\text{def}}{=} (x_{i-1} \geq 2)$, $\psi(x_{i-1}, x'_i) \overset{\text{def}}{=} (x'_i = x_{i-1} - 1)$ and $\psi_{x'_i}(x_{i-1}) \overset{\text{def}}{=} (x_{i-1} - 1)$. Similar to the previous action, we first synthesize the three components of the guard of this action, and then generate its assignment.

- $\neg L(x_{i-1}, x_i)$: This part is again $(x_{i-1} \neq x_i) \wedge (x_{i-1} \neq x_i + 1)$ for the same reason discussed for the first action.
- *Linear set constraint*: The constraint $\phi(x_{i-1})$ requires that we include $(x_{i-1} \geq 2)$ as part of the guard condition.
- *Self-disabling constraint*: Using $\psi_{x'_i}(x_{i-1}) \overset{\text{def}}{=} (x_{i-1} - 1)$, we realize that the assignment of this action establishes the condition $(x_i = x_{i-1} - 1)$. Thus, we include the constraint $(x_i \neq x_{i-1} - 1)$ in the guard, and $x_i := x_{i-1} - 1$ as the assignment of this action.

Putting everything together, we get the following action for this unbounded linear set: $(x_{i-1} \geq 2) \wedge (x_{i-1} \neq x_i) \wedge (x_i \neq x_{i-1} - 1) \to x_i := x_{i-1} - 1$.

Sample executions. Consider a computation of a ring of four processes for a domain size $M = 4$ (i.e., $x_i \in \mathbb{Z}_4$) starting at the state $s_0 = \langle \underline{0}, \underline{2}, 1, \underline{3} \rangle$, where the underlined values indicate the enabled processes based on the synthesized actions. That is, processes P_0, P_1 and P_3 are enabled. For example, P_0 is enabled because $x_0 = 0 \wedge x_3 = 3$ and the third action is enabled. Using a similar reasoning, one can figure out why P_1 and P_3 are enabled at s_0. For brevity, we demonstrate a synchronous execution of this ring, but one can extract an asynchronous interleaving of processes that converges to the same final state. Starting at s_0, all three enabled processes can execute, where the entire ring transitions to the state $s_1 = \langle \underline{2}, 1, 1, 1 \rangle$, and then reaches the state $s_2 = \langle 1, 1, 1, 1 \rangle$, where everyone agrees with its predecessor. For a domain size $M = 5$ and an arbitrary start state $\langle \underline{0}, \underline{2}, \underline{0}, \underline{3} \rangle$, the NA protocol generates the following computation: $\langle \underline{2}, 1, 1, 1 \rangle$, $\langle 1, 1, 1, 1 \rangle$. As another example, consider a larger ring of five processes and $M = 5$. Starting at $\langle 0, 4, \underline{2}, \underline{3}, 1 \rangle$, the NA protocol will converge through the following states: $\langle \underline{0}, 4, 3, \underline{1}, \underline{2} \rangle$, $\langle 1, \underline{4}, 3, 2, 1 \rangle$, $\langle 1, 1, \underline{3}, 2, 1 \rangle$, $\langle 1, 1, 1, \underline{2}, 1 \rangle$, $\langle 1, 1, 1, 1, 1 \rangle$. Yet another example includes a case of $M = 7$ and six processes in the ring. Starting at $\langle \underline{6}, \underline{2}, \underline{0}, \underline{3}, \underline{6}, \underline{4} \rangle$, the NA protocol has the following converging computation: $\langle \underline{3}, \underline{5}, \underline{1}, 1, \underline{2}, \underline{5} \rangle$, $\langle \underline{4}, \underline{2}, \underline{4}, \underline{1}, 1, 1 \rangle$, $\langle 1, \underline{3}, \underline{1}, \underline{3}, 1, 1 \rangle$, $\langle 1, 1, \underline{2}, 1, 1, 1 \rangle$, $\langle 1, 1, 1, 1, 1, 1 \rangle$. Observe that, the synthesized NA protocol is self-stabilizing for different ring sizes and domain sizes.

IV. Case Studies

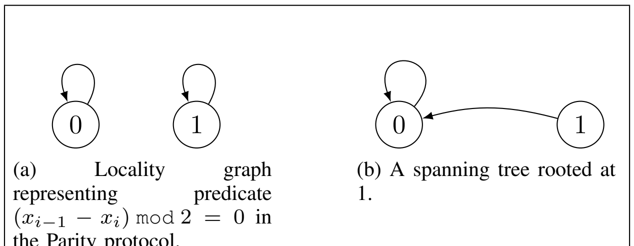
This section presents three more case studies for the synthesis of self-stabilizing parameterized protocols with unbounded variables. Section IV-A presents the synthesis of a parity protocol. Then, Section IV-B discusses the SumNot2 protocols, and Section IV-C introduces a self-stabilizing SumNotOdd protocol. For all examples, the subscript operations are modulo number of processes, and the arithmetic operations in the guard and assignment of an action are performed modulo M.

A. The Parity Protocol

This section demonstrates the synthesis of a Parity protocol, where processes in the uni-ring should converge to an agreed-upon parity starting from any arbitrary state. Formally, the entire ring should self-stabilize to states where $\forall i : i \in \mathbb{N} : (|x_{i-1} - x_i| \bmod 2) = 0$ holds. (Notice that, $|x_{i-1} - x_i|$ =max$(x_{i-1} - x_i, x_i -$

$x_{i-1})$.) Figures 6 to 9 illustrate how the spanning tree of Parity grows as the domain size increases. The common core is $\{(0,0),(1,0)\}$. We synthesize an action corresponding to each linear set.

- *Linear set 1*: The self-loop on 0 can be represented as a linear set with the base vector $(0,0)$ and the period vector $(0,0)$. Algorithm 3 outputs $\phi(x_{i-1}) \overset{\text{def}}{=} (x_{i-1} = 0)$, $\psi(x_{i-1}, x'_i) \overset{\text{def}}{=} (x'_i = 0)$, and $\psi_{x'_i}(x_{i-1}) \overset{\text{def}}{=} 0$. Thus, the assignment of the action is $x_i := 0$, and the requirement of having self-disabling actions would be $x_i \neq 0$. The constraint $\neg L(x_{i-1}, x_i)$ provides $(|x_{i-1} - x_i| \bmod 2) \neq 0$. Thus, the synthesized action is $(x_{i-1} = 0) \wedge ((|x_{i-1} - x_i| \bmod 2) \neq 0) \wedge (x_i \neq 0) \rightarrow x_i := 0$.
- *Linear set 2*: The base vector of this linear set is $(1,0)$ and its period vector is $(0,0)$. As a result, we have $\phi(x_{i-1}) \overset{\text{def}}{=} (x_{i-1} = 1)$, $\psi(x_{i-1}, x'_i) \overset{\text{def}}{=} (x'_i = 0)$, and $\psi_{x'_i}(x_{i-1}) \overset{\text{def}}{=} 0$. The assignment of the action is $x_i := 0$, which leads to the self-disabling constraint $x_i \neq 0$. The constraint $\neg L(x_{i-1}, x_i)$ provides $((|x_{i-1} - x_i| \bmod 2) \neq 0)$. Thus, the synthesized action is $(x_{i-1} = 1) \wedge ((|x_{i-1} - x_i| \bmod 2) \neq 0) \wedge (x_i \neq 0) \rightarrow x_i := 0$.
- *Linear set 3*: Using the base vector $(2,0)$ and the period vector $(1,1)$, this linear set contains integer vectors $S_3 = \{(x_{i-1}, x'_i) \mid (x_{i-1} = 2 + \lambda) \wedge (x'_i = \lambda)$ where $\lambda \in \mathbb{N}\}$. Algorithm 3 gives us $\phi(x_{i-1}) \overset{\text{def}}{=} (x_{i-1} = 2 + \lambda)$, which can be written as $\phi(x_{i-1}) \overset{\text{def}}{=} (x_{i-1} \geq 2)$. Algorithm 3 also outputs $\psi(x_{i-1}, x'_i) \overset{\text{def}}{=} (x'_i = x_{i-1} - 2)$, and $\psi_{x'_i}(x_{i-1}) \overset{\text{def}}{=} (x_{i-1} - 2)$. The assignment of the action is obtained from $\psi_{x'_i}(x_{i-1}) \overset{\text{def}}{=} (x_{i-1} - 2)$, leading to $x_i := x_{i-1} - 2$. Thus, the synthesized action for this linear set is $(x_{i-1} \geq 2) \wedge ((|x_{i-1} - x_i| \bmod 2) \neq 0) \wedge (x_i \neq x_{i-1} - 2) \rightarrow x_i := x_{i-1} - 2$.

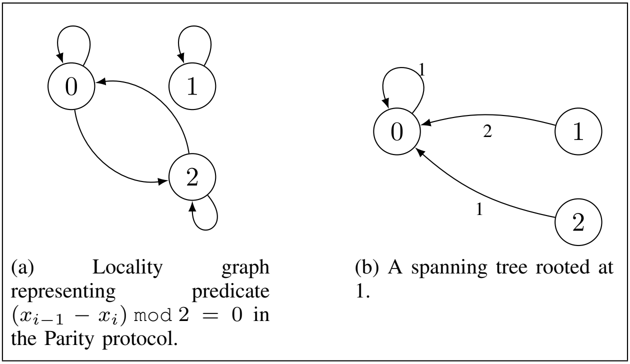


(a) Locality graph representing predicate $(x_{i-1} - x_i) \bmod 2 = 0$ in the Parity protocol.
(b) A spanning tree rooted at 1.

Fig. 6: Locality graph and a spanning tree of the Parity protocol for domain size 2.

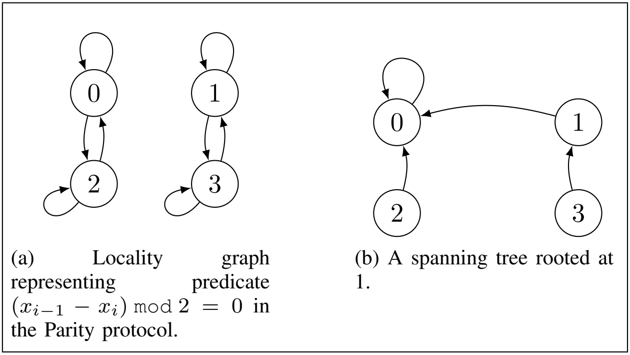
B. SumNot2 Protocol

The SumNot2 protocol is a simple but non-trivial protocol to synthesize, initially introduced in [21]. To illustrate the intricacies of synthesizing unbounded protocols, we use the SumNot2 protocol. Starting from any arbitrary state, the entire ring should self-stabilize to states where $\mathcal{I}_{SN2} = \forall i \in \mathbb{N} :: (x_{i-1} + x_i) \neq 2$ holds. We follow Algorithm 3 to synthesize the semilinear set of SumNot2 and then generate its parameterized actions. Figures 10 to 13 illustrate the locality graphs and spanning trees for domain sizes 2 to 5. These figures show how the unbounded spanning tree grows when M is increased. The common core of SumNot2 contains the arcs of the tree in Figure 10b; i.e., $\{(0,0),(1,0)\}$. We now specify the linear sets corresponding to the growth of the spanning tree for unbounded domains.



(a) Locality graph representing predicate $(x_{i-1} - x_i) \bmod 2 = 0$ in the Parity protocol.
(b) A spanning tree rooted at 1.

Fig. 7: Locality graph and a spanning tree of the Parity protocol for domain size 3.



(a) Locality graph representing predicate $(x_{i-1} - x_i) \bmod 2 = 0$ in the Parity protocol.
(b) A spanning tree rooted at 1.

Fig. 8: Locality graph and a spanning tree of the Parity protocol for domain size 4.

- *Linear set 1*: The self-loop on 0 can be represented as a linear set with the base vector $(0,0)$ and the period vector $(0,0)$. This set can be specified as formulas $\phi(x_{i-1}) \overset{\text{def}}{=} (x_{i-1} = 0)$ and $\psi(x_{i-1}, x'_i) \overset{\text{def}}{=} (x'_i = 0)$, which results in $\psi_{x'_i}(x_{i-1}) \overset{\text{def}}{=} 0$. Notice that, $\psi(x_{i-1}, x'_i) \overset{\text{def}}{=} (x'_i = 0)$ means that x_i should be updated to 0 by this action. Thus, the assignment of this action would be $x_i := 0$. The

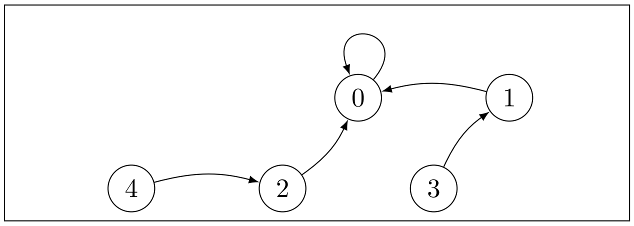
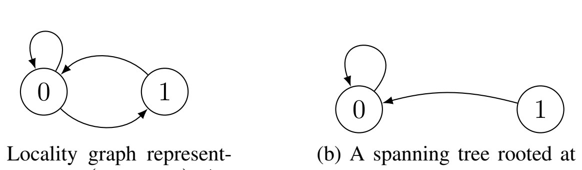


Fig. 9: A spanning tree of the Parity protocol for domain size 5.

guard of the action corresponding to this linear set includes three constraints, namely the constraint imposed by this linear set (i.e., $\phi(x_{i-1})$), the constraint $\neg L(x_{i-1}, x_i)$ and the self-disabling constraint $(x_i \neq 0)$. We also know that $\neg L(x_{i-1}, x_i) \equiv (x_{i-1} + x_i = 2)$. As a result, the synthesized action is $(x_{i-1} = 0) \wedge (x_{i-1} + x_i = 2) \wedge (x_i \neq 0) \rightarrow x_i := 0$.

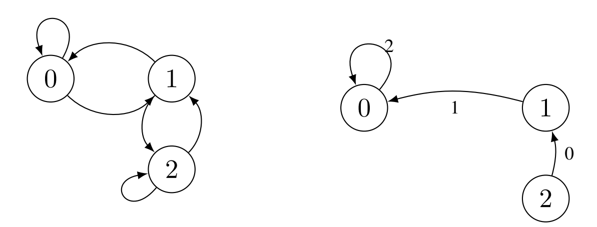
- *Linear set 2*: The base vector is $(1, 0)$ and the period vector is $(0, 0)$. The formulas $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} = 1)$ and $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = 0)$ represent this linear set. Thus, we have $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} 0$. A similar reasoning to that of the previous case would give us the action $(x_{i-1} = 1) \wedge (x_{i-1} + x_i = 2) \wedge (x_i \neq 0) \rightarrow x_i := 0$.
- *Linear set 3*: The base vector is $(2, 1)$ and the period vector is $(1, 1)$. This linear set is a periodic set of integer vectors $\{(x_{i-1}, x'_i) \mid (x_{i-1} = 2 + \lambda) \wedge (x'_i = 1 + \lambda)$ where $\lambda \in \mathbb{N}\}$. Thus, Algorithm 3 returns $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} = 2 + \lambda)$, which can be represented as $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} \geq 2)$. Moreover, we have $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = 1 + \lambda)$, which means $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = x_{i-1} - 1)$, and $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} - 1)$. Thus, the assignment of the action is $x_i := x_{i-1} - 1$, and the self-disabling constraint would be $x_i \neq x_{i-1} - 1$. Thus, we get the action: $(x_{i-1} \geq 2) \wedge (x_{i-1} + x_i = 2) \wedge (x_i \neq x_{i-1} - 1) \rightarrow x_i := x_{i-1} - 1$.



(a) Locality graph representing predicate $(x_{i-1}+x_i) \neq 2$ in the SumNotTwo protocol.

(b) A spanning tree rooted at 0.

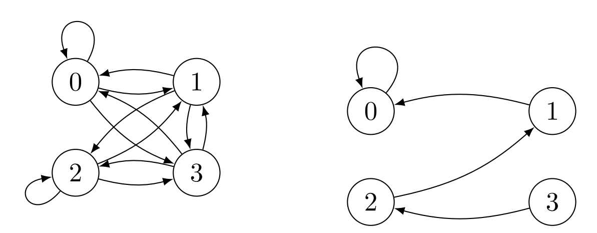
Fig. 10: Locality graph and a spanning tree of the SumNotTwo protocol for domain size 3.



(a) Locality graph representing predicate $(x_{i-1}+x_i) \neq 2$ in the SumNotTwo protocol.

(b) A spanning tree rooted at 1.

Fig. 11: Locality graph and a spanning tree of the SumNotTwo protocol for domain size 3.



(a) Locality graph representing predicate $(x_{i-1}+x_i) \neq 2$ in the SumNotTwo protocol.

(b) A spanning tree rooted at 1.

Fig. 12: Locality graph and a spanning tree of the SumNotTwo protocol for domain size 4.

C. SumNotOdd Protocol

A more general form of SumNot2 includes the two protocols SumNotOdd and SumNotEven. In this section, we synthesize a self-stabilizing SumNotOdd protocol on unbounded uni-rings. The self-stabilization of SumNotOdd requires that reovery is achieved to $\mathcal{I}_{\mathcal{SNO}} = \forall i \in \mathbb{N} :: (x_{i-1} + x_i) \bmod 2 = 0$ from any state in the unbounded state space of SumNotOdd. Steps 2 and 3 result in $\gamma = 1$. Figure 14 illustrates the locality graph and the spanning tree τ generated in Step 4. Increasing the domain size to $M = 3$, we get the locality graph and the spanning tree in Figure 15. Thus, the common core is $\{(0, 1), (1, 1)\}$. Moreover, since $(2, 1)$ is included as the base vector in Step 8 of Algorithm 3 (see Figure 15), the period vector is set to $(1, 1)$.

We now synthesize the action corresponding to each linear set.

- *Linear set 1*: Similar to previous case studies, the self-loop on 1 can be represented as a linear set with the base vector $(1, 1)$ and the period vector $(0, 0)$. Considering that $\neg L(x_{i-1}, x_i)$ is $(x_{i-1} + x_i) \bmod 2 \neq 0$, we get the following action for this linear set: $(x_{i-1} = 1) \wedge ((x_{i-1} + x_i) \bmod 2 \neq$

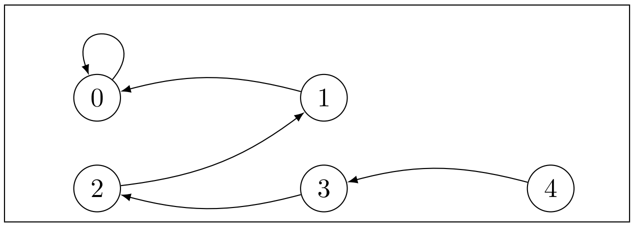
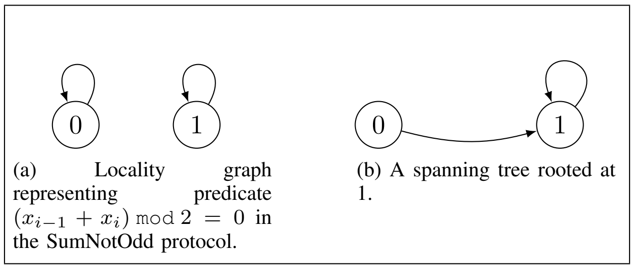


Fig. 13: A spanning tree of the SumNotTwo protocol for domain size 5.



(a) Locality graph representing predicate $(x_{i-1} + x_i) \bmod 2 = 0$ in the SumNotOdd protocol.

(b) A spanning tree rooted at 1.

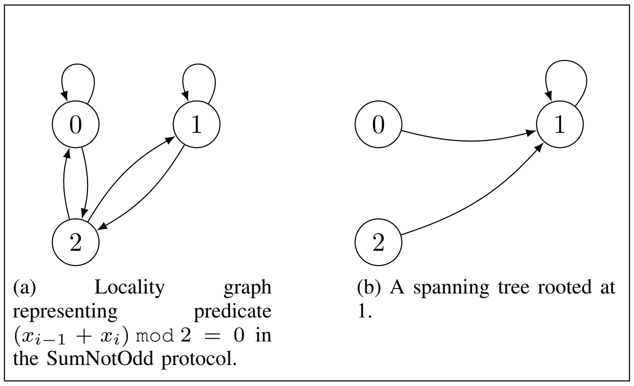
Fig. 14: Locality graph and a spanning tree of the SumNotOdd protocol for domain size 2.

$0) \wedge (x_i \neq 1) \rightarrow x_i := 1$.

- *Linear set 2*: The base vector is $(0, 1)$ and the period vector is $(0, 0)$ would give us the action $(x_{i-1} = 0) \wedge ((x_{i-1} + x_i) \bmod 2 \neq 0) \wedge (x_i \neq 1) \rightarrow x_i := 1$.
- *Linear set 3*: This linear set captures the unbounded core of the protocol with the base vector $(2, 1)$ and the period vector $(1, 1)$. Thus, we specify this linear set as $\{(x_{i-1}, x'_i) \mid (x_{i-1} = 2 + \lambda) \wedge (x'_i = 1 + \lambda)$ where $\lambda \in \mathbb{N}\}$. Step 10 of Algorithm 3 gives us $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} = 2+\lambda)$; i.e., $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} \geq 2)$. Further, we get $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = x_{i-1} - 1)$ and $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} - 1)$. Thus, the synthesized action for this linear set is $(x_{i-1} \geq 2) \wedge ((x_{i-1} + x_i) \bmod 2 \neq 0) \wedge (x_i \neq x_{i-1} - 1) \rightarrow x_i := x_{i-1} - 1$.

V. Related Work

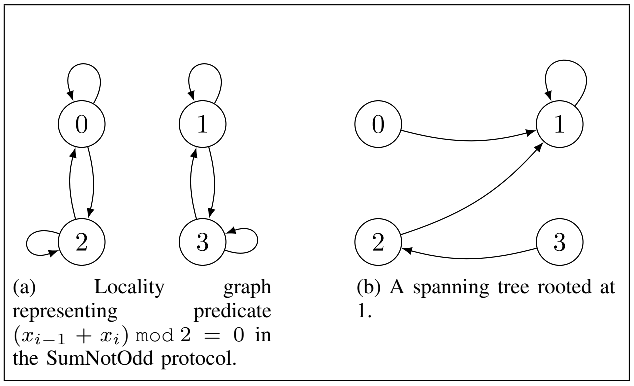
This section discusses the state-of-the-art in the verification and synthesis of parameterized systems, especially unbounded and infinite-state systems. For example, predicate abstraction [22], [23] enables a method for creating a finite-state representation of infinite-state systems where safety properties can be verified. Constraint language programming [24] enables the verification of safety properties of concurrent systems with unbounded data. Approaches for reachability analysis of generalized Petri nets [25], [26] apply over-approximation towards generating a finite model, and then develop an efficient semi-decision procedure for forward reachability analysis. Counter abstraction [27] utilizes integer counters to count the number of processes in a specific state, but such abstractions are too coarse for the design of self-stabilizing protocols where recovery must be ensured from every concrete state. Environment abstraction [28] extends counter abstraction in order to model the abstract state and the environment of each process. Invisible invariants [29], [30] infer an invariant of a parameterized system by examining a few small instantiations of protocols. Indexed predicates [31] provide a method for the generation and verification of invariant predicates specified in terms of the process indices in infinite-state systems. The aforementioned methods mostly aim at the verification of safety and local liveness properties, and it is unclear how they can synthesize self-stabilizing unbounded protocols.



(a) Locality graph representing predicate $(x_{i-1} + x_i) \bmod 2 = 0$ in the SumNotOdd protocol.

(b) A spanning tree rooted at 1.

Fig. 15: Locality graph and a spanning tree of the SumNotOdd protocol for domain size 3.



(a) Locality graph representing predicate $(x_{i-1} + x_i) \bmod 2 = 0$ in the SumNotOdd protocol.

(b) A spanning tree rooted at 1.

Fig. 16: Locality graph and a spanning tree of the SumNotOdd protocol for domain size 4.

Most methods for the synthesis of parameterized unbounded systems provide little results for the synthesis of unbounded self-stabilizing protocols, where a global liveness property (i.e., convergence) must be met from *any* state in an unbounded state space. For example, synthesis of Petri nets [32], [33], [34] mainly focuses on the transformation of behavioral specifications in the form of labeled transition systems to Petri nets. UCLID5 [35], [36] provides a framework for modular verification and synthesis of the artifacts (e.g., invariants, assume-guarantee conditions) that are used during verification. Syntax-Guided Synthesis (SyGus) [37] generates the implementation of a set of functions (each adhering to a grammar) in the specification of a system for a background logic theory. It is unclear how one can use SyGus to synthesize the actions of SS-SymU protocols which must interact asynchronously to ensure convergence in a specific topology. Moreover, methods that combine SyGus with reactive synthesis are mostly applied to centralized systems [38]. Oracle-Guided Inductive Synthesis (OGIS) [39], [40], [41] is based on iterative query-response interactions between a learner and a teacher towards synthesizing a system that adheres to formal specifications. Utilizing OGIS in the synthesis of self-stabilizing unbounded systems may not converge to a solution that must recover from any state rather than recovery from a proper set of initial states. While the existing synthesis methods inspire our work, the novelty of our approach mainly lies in the characterization of unbounded actions as semilinear sets for the synthesis of SS-SymU.

VI. Conclusions and Future Work

This paper investigated the problem of synthesizing self-stabilizing symmetric protocols (SS-SymU) on uni-rings, where a ring can have an unbounded number of processes and processes have unbounded variables. While previous research [5] has addressed this problem for rings of unbounded size, we are not aware of any work that synthesizes self-stabilizing protocols having unbounded variables too. We first showed that the ability to represent unbounded actions of a protocol as semilinear sets is sufficient for synthesis. This reduces the synthesis of SS-SymU to the synthesis of semilinear sets. Then, we presented a sound algorithm that generates a semilinear set for a protocol from which the parameterized actions of the protocol are derived. We demonstrated how our algorithm can generate SS-SymU protocols (e.g., near agreement and parity on unbounded uni-rings) that were previously infeasible. We are currently implementing the proposed method as a synthesizer and are investigating the feasibility of synthesis for more complicated protocols and topologies. We would also like to know how semilinear sets can be utilized for the verification and synthesis of unbounded protocols that satisfy general temporal properties (instead of just self-stabilization).

References

[1] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, vol. 17, no. 11, pp. 643–644, 1974.

[2] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[3] L. Lamport and N. Lynch, *Handbook of Theoretical Computer Science: Chapter 18, Distributed Computing: Models and Methods*. Elsevier Science Publishers B. V., 1990.

[4] M. Lazic, I. Konnov, J. Widder, and R. Bloem, "Synthesis of distributed algorithms with parameterized threshold guards," in *21st International Conference on Principles of Distributed Systems (OPODIS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[5] A. Ebnenasir and A. P. Klinkhamer, "Topology-specific synthesis of self-stabilizing parameterized systems with constant-space processes," *IEEE Transactions on Software Engineering*, vol. 47, no. 3, pp. 614–629, 2019.

[6] A. Klinkhamer and A. Ebnenasir, "On the verification of livelock-freedom and self-stabilization on parameterized rings," *ACM Transactions on Computational Logic*, vol. 20, no. 3, pp. 1–36, 2019.

[7] N. Mirzaie, F. Faghih, S. Jacobs, and B. Bonakdarpour, "Parameterized synthesis of self-stabilizing protocols in symmetric networks," *Acta Informatica*, vol. 57, no. 1, pp. 271–304, 2020.

[8] H. Moloodi, F. Faghih, and B. Bonakdarpour, "Parameterized distributed synthesis of fault-tolerance using counter abstraction," in *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2021, pp. 67–77.

[9] P. C. Attie and E. A. Emerson, "Synthesis of concurrent systems with many similar processes," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 1, pp. 51–115, 1998.

[10] A. Farahat and A. Ebnenasir, "A lightweight method for automated design of convergence in network protocols," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 7, no. 4, pp. 38:1–38:36, Dec. 2012.

[11] F. Faghih and B. Bonakdarpour, "SMT-based synthesis of distributed self-stabilizing systems," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2015, to appear.

[12] A. Klinkhamer and A. Ebnenasir, "Shadow/puppet synthesis: A stepwise method for the design of self-stabilization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 11, pp. 3338 – 3350, Feb. 2016.

[13] R. Bloem, N. Braud-Santoni, and S. Jacobs, "Synthesis of self-stabilising and byzantine-resilient distributed systems," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 157–176.

[14] S. Jacobs and R. Bloem, "Parameterized synthesis," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2012, pp. 362–376.

[15] B. Finkbeiner and S. Schewe, "Bounded synthesis," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5-6, pp. 519–539, 2013.

[16] A. Klinkhamer and A. Ebnenasir, "Verifying livelock freedom on parameterized rings and chains," in *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, 2013, pp. 163–177.

[17] A. P. Klinkhamer and A. Ebnenasir, "Synthesizing parameterized self-stabilizing rings with constant-space processes," in *International Conference on Fundamentals of Software Engineering*. Springer, 2017, pp. 100–115.

[18] A. Klinkhamer and A. Ebnenasir, "Shadow/puppet synthesis: A stepwise method for the design of self-stabilization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 11, pp. 3338–3350, 2016.

[19] R. J. Parikh, "On context-free languages," *Journal of the ACM (JACM)*, vol. 13, no. 4, pp. 570–581, 1966.

[20] S. Ginsburg and E. H. Spanier, "Bounded algol-like languages," *Transactions of the American Mathematical Society*, vol. 113, no. 2, pp. 333–368, 1964.

[21] A. Farahat, "Automated design of self-stabilization," Ph.D. dissertation, Michigan Technological University, July 2012.

[22] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of c programs," in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 2001, pp. 203–213.

[23] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in c," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 388–402, 2004.

[24] G. Delzanno, "An overview of msr (c): A clp-based framework for the symbolic verification of parameterized concurrent systems," *Electronic Notes in Theoretical Computer Science*, vol. 76, pp. 65–82, 2002.

[25] W. Czerwiński, S. Lasota, R. Lazić, J. Leroux, and F. Mazowiecki, "The reachability problem for petri nets is not elementary," *Journal of the ACM (JACM)*, vol. 68, no. 1, pp. 1–28, 2020.

[26] N. Amat, S. D. Zilio, and T. Hujsa, "Property directed reachability for generalized petri nets," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022, pp. 505–523.

[27] A. Pnueli, J. Xu, and L. D. Zuck, "Liveness with (0, 1, infty)-counter abstraction," in *International Conference on Computer Aided Verification (CAV)*, 2002, pp. 107–122.

[28] E. Clarke, M. Talupur, and H. Veith, "Environment abstraction for parameterized verification," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2006, pp. 126–141.

[29] A. Pnueli, S. Ruah, and L. Zuck, "Automatic deductive verification with invisible invariants," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2001, pp. 82–97.

[30] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck, "Liveness with invisible ranking," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2004, pp. 223–238.

[31] S. K. Lahiri and R. E. Bryant, "Indexed predicate discovery for unbounded system verification," in *International Conference on Computer Aided Verification*. Springer, 2004, pp. 135–147.

[32] P. Darondeau, "Unbounded petri net synthesis," in *Advanced Course on Petri Nets*. Springer, 2003, pp. 413–438.

[33] E. Badouel, L. Bernardinello, and P. Darondeau, *Petri net synthesis*. Springer, 2015.

[34] E. Best and R. Devillers, "Pre-synthesis of petri nets based on prime cycles and distance paths," *Science of Computer Programming*, vol. 157, pp. 41–55, 2018.

[35] S. A. Seshia and P. Subramanyan, "Uclid5: Integrating modeling, verification, synthesis and learning," in *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE, 2018, pp. 1–10.

[36] F. Mora, K. Cheang, E. Polgreen, and S. A. Seshia, "Synthesis in uclid5," *arXiv preprint arXiv:2007.06760*, 2020.

[37] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design (FMCAD), 2013*. IEEE, 2013, pp. 1–17.

[38] W. Choi, "Can reactive synthesis and syntax-guided synthesis be friends?" in *Companion Proceedings of the 2021 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 2021, pp. 3–5.

[39] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1. IEEE, 2010, pp. 215–224.

[40] S. Jha and S. A. Seshia, "A theory of formal synthesis via inductive learning," *Acta Informatica*, vol. 54, no. 7, pp. 693–726, 2017.

[41] E. Polgreen, A. Reynolds, and S. A. Seshia, "Satisfiability and synthesis modulo oracles," in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2022, pp. 263–284.