Synthesizing Parameterized Self-stabilizing Rings with Constant-Space Processes

Alex P. Klinkhamer[1] and Ali Ebnenasir[2](✉)

[1] Google, Mountain View, CA 94043, USA
apklinkh@mtu.edu

[2] Department of Computer Science, Michigan Technological University, Houghton, MI 49931, USA
aebnenas@mtu.edu

Abstract. This paper investigates the problem of synthesizing parameterized rings that are "self-stabilizing by construction". While it is known that the verification of self-stabilization for parameterized unidirectional rings is undecidable, we present a counterintuitive result that synthesizing such systems is decidable! This is surprising because it is known that, in general, the synthesis of distributed systems is harder than their verification. We also show that synthesizing self-stabilizing bidirectional rings is an undecidable problem. To prove the decidability of synthesis for unidirectional rings, we propose a sound and complete algorithm that performs the synthesis in the local state space of processes. We also generate strongly stabilizing rings where no fairness assumption is made. This is particularly noteworthy because most existing verification and synthesis methods for parameterized systems assume a fair scheduler.

1 Introduction

Developing parameterized Self-Stabilizing (SS) distributed systems is an important and challenging problem since a parameterized SS system must be self-stabilizing regardless of the number of processes. An SS system (i) recovers from *any* configuration/state to a set of legitimate states – that captures the normal behaviors of a system, and (ii) guarantees global recovery to legitimate states solely based on the local actions of its processes (without any central point of coordination). Designing self-stabilization becomes even more challenging for *parameterized* systems that include families of *symmetric* processes, where the code of each process is obtained from a *template* process in a symmetric network. Since the general case synthesis problem is undecidable, several researchers have recently proposed methods where they generate specific parameterized systems from their temporal logic specifications, mainly by exploiting verification techniques (e.g., cutoff theorems [13]) and boundedness assumptions [16]. As the verification of SS parameterized unidirectional rings is known to be undecidable [22], the common understanding has been that synthesizing such systems should also be undecidable. In this paper, we prove otherwise! We show that

© IFIP International Federation for Information Processing 2017
Published by Springer International Publishing AG 2017. All Rights Reserved
M. Dastani and M. Sirjani (Eds.): FSEN 2017, LNCS 10522, pp. 100–115, 2017.
DOI: 10.1007/978-3-319-68972-2_7

synthesizing self-stabilization is actually decidable for parameterized unidirectional rings where all processes follow the same synthesized rules.

Numerous approaches exist for the synthesis of parameterized systems, most of which focus on synthesis from temporal logic specifications while assuming some sort of fairness. For example, Finkbeiner and Schewe [16] present a method where they solve the synthesis problem in a scope-based fashion similar to the scope-based verification methods [19]. They formulate the synthesis problem as a set of constraints that are fed to a Satisfiability Modulo Theory (SMT) solver [9]. Jacobs and Bloem [20] reduce the problem of synthesizing parameterized systems to synthesizing a small network of symmetric processes under the assumption of fair token passing. They exploit bounded synthesis and cutoff theorems to enable a semi-decision procedure that will eventually find a solution if one exists. Additionally, some researchers have investigated the synthesis of parameterized self-stabilizing systems in a problem-specific context. For instance, Bloem *et al.* [6] provide a method for the synthesis of synchronous systems that are SS and tolerate Byzantine failures and their underlying communication topology is a clique. Dolev *et al.* [11] present a verification-based method to generate synchronous and constant-space counting algorithms that are self-stabilizing under Byzantine faults. Lenzen and Rybicki [25] provide an SS and Byzantine-tolerant solution for the counting problem with near-optimal stabilization time and message sizes. What the aforementioned methods have in common is that they synthesize from temporal logic specifications and/or make assumptions about synchrony, fairness and complete knowledge of the network for each process. Moreover, some of them investigate specific problems.

In addition to proving some undecidability results for bidirectional rings, this paper presents an algorithmic method for the synthesis of symmetric unidirectional rings that are self-stabilizing by construction. The proposed algorithm works in a graph-theoretic context rather than synthesis from temporal logic specifications. In our work, we consider processes that are deterministic, self-disabling and constant-space, where a self-disabling process stops executing once it executes an action until it is enabled again by an action of its predecessor.[1] Moreover, we investigate this problem for sets of legitimate states that can be specified as the conjunction of local legitimate states. While our assumptions may seem restrictive, there are important applications for such systems [18,28]. The decidability result of this paper is counterintuitive as in our previous work [22] we have shown that verifying self-stabilization for unidirectional rings is undecidable. This is surprising because it is known [26] that, in general, the synthesis of distributed systems is harder than their verification. We first present a *necessary and sufficient* condition for the existence of a symmetric SS unidirectional ring. Our necessary and sufficient condition states that an SS symmetric unidirectional ring exists if and only if (iff) there is a value in the state space of the template process that can make the locality of each process true. We then use this result and design a sound and complete algorithm. The input to our algorithm includes a set of legitimate states and the size of the state space of the

[1] We have shown [23] that these assumptions uphold the completeness of synthesis.

template process. The output of the proposed algorithm is the symmetric code of the template process so that the entire ring becomes self-stabilizing for any arbitrary (but finite) number of processes. Our approach is easier than synthesis from temporal logic specifications in that we perform the synthesis in a bottom-up fashion by intelligently searching the state space of the template process. Subsequently, we investigate the synthesis of bidirectional symmetric rings that are self-stabilizing, and show that this problem is undecidable. Our proof of undecidability is a reduction from the problem of verifying self-stabilization for unidirectional rings [22]. While we have used our algorithm to synthesize a few example systems in this paper, we are currently investigating the generalization of our algorithm for other topologies and more interesting case studies.

Contributions. This paper makes the following contributions. We

– present a surprising result that synthesizing parameterized SS unidirectional rings under the interleaving semantics and no fairness assumption is decidable;
– provide an algorithm that takes a set of legitimate states and the size of the state space of the template process, and automatically generates the code of the template process, and
– prove that synthesizing SS bidirectional rings is undecidable.

Organization. Section 2 presents some basic concepts. Section 3 shows that synthesizing SS unidirectional rings is decidable. Section 4 investigates the synthesis of SS bidirectional rings and proves that this problem is undecidable. Section 5 examines related work. Finally, Sect. 6 makes concluding remarks and discusses future extensions of this work.

2 Basic Concepts

This section presents the definition of parameterized rings, their representation as action graphs, and self-stabilization. Wlog, we use the term *protocol* to refer to parameterized rings as we conduct our investigation in the context of network protocols. A protocol p for a computer network includes $N > 1$ processes (finite-state machines), where each process P_i has a finite set of readable and writeable variables. 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 writeable variable is also readable. The *global state* of the protocol is defined by a snapshot of the local states of all processes. The *state space* of a protocol, denoted by Σ, is the universal set of all global states. A *state predicate* is a subset of Σ. A process *acts* (i.e., *transitions*) when it atomically updates its state based on its locality. The locality of a process is defined by the network topology. In this paper, our focus is on the ring topology. For example, in a unidirectional ring consisting of N processes, each process P_i (where $i \in \mathbb{Z}_N$, i.e., $0 \leq i \leq N-1$) has a neighbor P_{i-1}, where subtraction and addition are in modulo N. 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 $C_0, C_1, \ldots, C_k$ where there is a transition from C_i to C_{i+1} for every $i \in \mathbb{Z}_k$. We consider *symmetric* protocols, where processes have identical rules for changing their state and the rules are *parameterized.* That is, the code of each process can be instantiated from a *template* process. We use triples of the form (a, b, c) to denote actions $(x_{i-1} = a \wedge x_i = b \longrightarrow x_i := c;)$ of the template process P_i in a unidirectional ring protocol. 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 of the process once the guard evaluates to *true*; i.e., the action is *enabled.*

Definition 1 (Transition Function). *Let P_i be any process with a state variable x_i in a unidirectional ring protocol p. We define its transition function $\xi : \Sigma \times \Sigma \rightarrow \Sigma$ as a partial function such that $\xi(a, b) = c$ if and only if P_i has an action $(x_{i-1} = a \wedge x_i = b \longrightarrow x_i := c;)$. In other words, ξ can be used to define all actions of P_i in the form of a single parametric action:*

$$((x_{i-1}, x_i) \in \mathsf{Pre}(\xi)) \longrightarrow x_i := \xi(x_{i-1}, x_i);$$

where $(x_{i-1}, x_i) \in \mathsf{Pre}(\xi)$ checks to see if the current x_{i-1} and x_i values are in the preimage of ξ.

Visually, we depict the actions of a process (hence a protocol) by a labeled directed multigraph, called the *action graph*, where each action (a, b, c) in the protocol appears as an arc from node a to node c labeled b in the graph. For example, consider the self-stabilizing Sum-Not-2 protocol given in [15]. Each process P_i has a variable $x_i \in \mathbb{Z}_3$ and actions $(x_{i-1} = 0 \wedge x_i = 2 \longrightarrow x_i := 1)$, $(x_{i-1} = 1 \wedge x_i = 1 \longrightarrow x_i := 2)$, and $(x_{i-1} = 2 \wedge x_i = 0 \longrightarrow x_i := 1)$. This protocol converges to a state where the sum of each two consecutive x values does not equal 2. The set of such states is formally specified as the state predicate $\forall i : (x_{i-1} + x_i \neq 2)$. We represent this protocol with a graph containing arcs $(0, 2, 1)$, $(1, 1, 2)$, and $(2, 0, 1)$ as shown in Fig. 1.

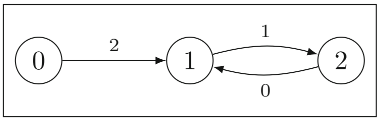

$$\begin{tikzcd}
0 \arrow[r, "2"] & 1 \arrow[r, bend left, "1"] & 2 \arrow[l, bend left, "0"]
\end{tikzcd}$$

Fig. 1. Graph representing Sum-Not-2 protocol.

Since protocols consist of *self-disabling* processes, an action (a, b, c) cannot coexist with action (a, c, d) for any d. Moreover, when the protocol is deterministic, a process cannot have two actions enabled at the same time; i.e., an action (a, b, c) cannot coexist with an action (a, b, d) where $d \neq c$.

Livelock, deadlock, and closure. A *legitimate* state captures a state to which a protocol is required to recover. Let I be a predicate representing the legitimate states for some protocol p. A *livelock* of p is an infinite execution that never reaches I. When legitimate states are not specified, we assume a livelock is any infinite execution. A *deadlock* of p is a state in $\neg I$ that has no outgoing transition; i.e., no process is enabled to act. The state predicate I is *closed* under p when no transition exists that brings the protocol from a state in I to a state in $\neg I$.

Definition 2 (Self-stabilization). *A protocol p is self-stabilizing [10] with respect to its legitimate state predicate I* iff *from each illegitimate state in $\neg I$, all executions reach a state in I (i.e., convergence) and remain in I (i.e., closure). That is, p is livelock-free and deadlock-free in $\neg I$, and I is closed under p.*

Definition 3 (Weak Stabilization). *A weakly stabilizing protocol ensures that from each illegitimate state in $\neg I$, there is some execution that reaches a state in I (i.e., reachability) and remains in I.*

Next, we represent some of our previous result that we shall use in this paper.

Propagations and Collisions. When a process acts and enables its successor in a unidirectional ring, it propagates its ability to act. The successor may enable its own successor by acting, and the pattern may continue indefinitely. Such behaviors can be represented as sequences of actions that are propagated in a ring, called *propagations*. A propagation is a walk through the action graph. For example, the Sum-Not-2 protocol has a propagation $\langle(0,2,1),(1,1,2),(2,0,1),(1,1,2)\rangle$ whose actions can be executed in order by processes P_i, P_{i+1}, P_{i+2}, and P_{i+3} from a state $(x_{i-1},x_i,x_{i+1},x_{i+2},x_{i+3}) = (0,2,1,0,1)$. A propagation is *periodic* with period n *iff* its j-th action and $(j+n)$-th action are the same for every index j. A periodic propagation corresponds to a *closed walk* of length n in the graph. The Sum-Not-2 protocol has such a propagation of period 2: $\langle(1,1,2),(2,0,1)\rangle$. A *collision* occurs when two consecutive processes, say P_i and P_{i+1}, have enabled actions; e.g., (a,b,c) and (b,e,f), where $b \neq c$. In such a scenario, $x_{i-1} = a, x_i = b, x_{i+1} = e$. A collision occurs when P_i executes and assigns c to x_i. If that occurs, P_i will be disabled (because processes are self-disabling), and P_{i+1} becomes disabled too because x_i is no longer equal to b. As a result, two enabled processes become disabled by one action.

"Leads" Relation. Consider two actions A_1 and A_2 in a process P_i. We say the action A_1 *leads* A_2 *iff* the value of the variable x_i after executing A_1 is the same as the value required for P_i to execute A_2. Formally, this means an action (a,b,c) leads (d,e,f) *iff* $e = c$. Similarly, a propagation leads another *iff* for every index j, its j-th action leads the j-th action of the other propagation. In the action graph, this corresponds to two walks where the label of the destination node of the j-th arc in the first walk matches the arc label of the j-th arc in the second walk (for each index j). In [22], we prove the following theorem:

Theorem 1. *A unidirectional ring protocol of symmetric, self-disabling processes has a livelock for some ring size* iff *there exist some m propagations with some period n, where the $(i-1)$-th propagation leads the i-th propagation for each index i modulo m; i.e., the propagations successively lead each other modulo m.*

Undecidability of Verification. We have shown [15] that verifying deadlock-freedom in unidirectional rings is decidable. However, checking livelock-freedom is an undecidable problem (specifically Π_1^0-complete) for unidirectional ring protocols (with self-disabling and deterministic processes) [22]. The following results hold for cases where the set of legitimate states I is a conjunctive predicate; i.e., $I = \forall i : L(x_{i-1}, x_i)$, where $L(x_{i-1}, x_i)$ captures the locality of process P_i, which depends on its own state and that of its predecessor. Varghese [28,29] presents methods for specifying some global state predicates as conjunctive predicates.

Theorem 2. *Verifying livelock-freedom in a parameterized unidirectional ring protocol (with self-disabling and deterministic processes) is undecidable [22].*

We have also shown that verifying livelock-freedom remains undecidable even for a special type of livelocks, where exactly one process is enabled to execute in every state of the livelocked computation; i.e., *deterministic livelocks* [22].

Theorem 3. *Verifying livelock-freedom in a parameterized unidirectional ring protocol (with self-disabling and deterministic processes) remains undecidable even for deterministic livelocks* [22].

The above results imply the undecidability of verifying self-stabilization for parameterized unidirectional rings.

Theorem 4. *Verifying self-stabilization for a parameterized unidirectional ring protocol (with self-disabling and deterministic processes) is undecidable* [22].

3 Decidability of Synthesizing Unidirectional Rings

In this section, we show that synthesizing SS unidirectional rings is decidable.

Theorem 5. *Given a predicate $I \overset{\text{def}}{=} (\forall i : L(x_{i-1}, x_i))$ and variable domain M for a unidirectional ring, $L(\gamma, \gamma)$ is true for some γ if and only if there exists a protocol that stabilizes to I.*

Proof. $\Rightarrow$: Assume that no γ exists such that $L(\gamma, \gamma)$ is true. This implies that $\forall i : x_{i-1} \neq x_i$ in I. In this case, a stabilizing protocol would be a coloring protocol, which Bernard *et al.* [5] have shown is impossible for a unidirectional ring of size greater than M. (If the ring has at most M processes, then assigning unique values modulo M will provide a coloring.) This means if we check the entire domain $\mathbb{Z}_M$ and find no value that makes L true, then using Bernard *et al.*'s result we can *decide* that no solution exists. That is, the problem is decidable when $L(\gamma, \gamma)$ is false for all $\gamma \in \mathbb{Z}_M$. We are left to show how to construct a stabilizing protocol p when some γ can make $L(\gamma, \gamma)$ true. One could argue that a stabilizing protocol could contain just an action $\neg L(x_{i-1}, x_i) \rightarrow x_i := \gamma$, but this protocol is just *weakly* stabilizing.

Find a γ such that $L(\gamma, \gamma)$ is true. Assuming such a γ exists, it is trivial to find it by trying each value in $\mathbb{Z}_M$. Intuitively, we will make the stabilizing protocol p converge to $(\forall i : x_i = \gamma)$ unless it reaches some other state that satisfies I.

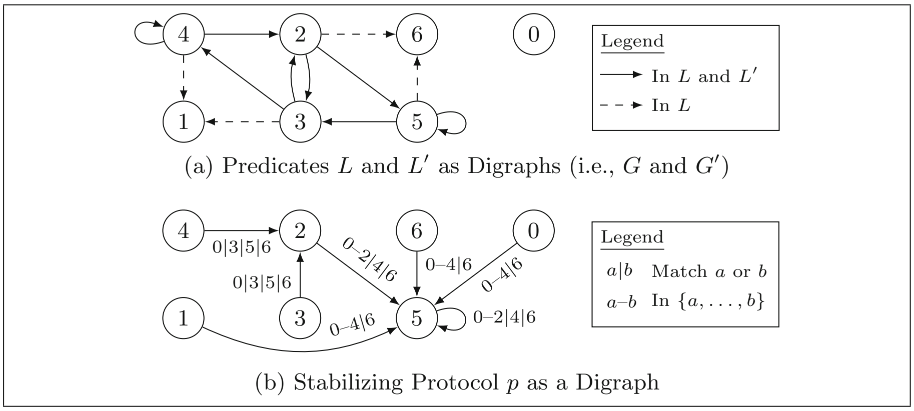

Legend
In L and L'
In L
(a) Predicates L and L' as Digraphs (i.e., G and G')
0|3|5|6
0|3|5|6
0–2|4|6
0–4|6
0–4|6
0–4|6
0–2|4|6
Legend
$a|b$ Match a or b
a–b In $\{a, \ldots, b\}$
(b) Stabilizing Protocol p as a Digraph

Fig. 2. Synthesis of stabilization to $\forall i : L(x_{i-1}, x_i)$, where $L(x_{i-1}, x_i) \overset{\text{def}}{=} \big((x_{i-1}^2 + x_i^3) \texttt{ mod } 7 = 3\big)$ and $x_i \in \mathbb{Z}_7$.

Figure 2 provides a running example where $L(x_{i-1}, x_i) \overset{\text{def}}{=} ((x_{i-1}^2 + x_i^3) \texttt{ mod } 7 = 3)$ and variables have domain size $M = 7$. We arbitrarily choose $\gamma = 5$ (instead of $\gamma = 4$) to satisfy $L(\gamma, \gamma)$; i.e., the solution is not unique.

Construct relation L' from arcs that form cycles in the digraph of L. The relation L can be represented as a digraph such that each arc (a, b) is in the graph *iff* $L(a, b)$ is true. Let G be this digraph (e.g., formed by both solid and dashed lines in Fig. 2a). Closed walks in G characterize all states in $(\forall i : L(x_{i-1}, x_i))$ [15]. Derive a digraph G' (and corresponding relation L') from G by removing all arcs that are not part of a cycle (e.g., arcs $(4, 1)$, $(3, 1)$, $(2, 6)$, and $(5, 6)$ in Fig. 2a). Since closed walks of G characterize states in I, we know that for every arc (a, b) in G that is not part of a cycle, no legitimate state contains $x_{i-1} = a \wedge x_i = b$ at any index i. All closed walks of G are retained by G', which means $I \overset{\text{def}}{=} (\forall i : L'(x_{i-1}, x_i))$.

Construct a bottom-up spanning tree τ with γ at the root. Let τ be a function that returns the parent of a node in a tree; i.e., $\tau(a) = c$ means that c is the parent of a. First, let $\tau(\gamma) \overset{\text{def}}{=} \gamma$ represent the root of the tree. Next, create a tree by backward reachability from γ in G', and assign $\tau(a) \overset{\text{def}}{=} c$ for each a that has a path $a, c, \ldots, \gamma$ in G'. Finally, let $\tau(a) \overset{\text{def}}{=} \gamma$ for each node a that has no path to γ in G'. These extra arcs of τ create no cycles. Since all arcs of G' are involved in cycles, any walk in G' can find its way back to a previously visited node. Therefore, if a node cannot reach γ in G', then γ cannot reach that node. Since the extra arcs of τ would not introduce cycles in G', we know that $(\forall i : (L'(x_{i-1}, x_i) \vee \tau(x_{i-1}) = x_i))$ is yet another equivalent way to write I.

Construct each action (a, b, c) of p by labeling each arc (a, c) of τ with all b values such that $(\neg L'(a, b) \wedge \tau(a) \neq b)$. In this way, τ defines how a process P_i in p will assign x_i when it detects an illegitimate state. Figure 2b illustrates the solution protocol for our example, as well as τ if we ignore the arc labels. The protocol p is written succinctly by the following action for each process P_i.

$$\neg L'(x_{i-1}, x_i) \wedge \tau(x_{i-1}) \neq x_i \longrightarrow x_i := \tau(x_{i-1});$$

This protocol p stabilizes to I. Deadlock-freedom in $\neg I$ and closure of I hold because each process P_i is enabled to act *iff* $(\neg L'(x_{i-1}, x_i) \wedge \tau(x_{i-1}) \neq x_i)$ holds. Livelock-freedom holds because all periodic propagations of p consist of actions of the form (γ, b, γ) where $L(\gamma, b)$ is false (e.g., the self-loops of Node 5 in Fig. 2b). Obviously none of these (γ, b, γ) actions lead each other since $b \neq \gamma$; i.e., no periodic propagations exist. Thus, based on Theorem 1, no livelocks exist in $\neg I$ for any ring size greater than M. Therefore, the protocol p stabilizes to I for any number of processes.

Proof $\Leftarrow$: Let p be a protocol p that stabilizes to I for all ring sizes. Thus, closure of I in p, deadlock-freedom and livelock-freedom of p in $\neg I$ must hold. Since processes are deterministic and self-disabling, each process P_i contains some actions that are enabled in $\neg L(x_{i-1}, x_i)$. After the execution of such actions $L(x_{i-1}, x_i)$ holds by setting x_i to some value $\lambda \in M$, and P_i becomes disabled. Due to livelock-freedom of p and Theorem 1, no periodic propagations should exists in p. That is, there are no closed walks in the action graph of p other than self-loops over λ. The existence of such self-loops means $L(\lambda, \lambda)$ holds. □

Using the proof of Theorem 5, we present Algorithm 1. Since this algorithm is self-explanatory, we just prove its soundness and completeness.

Theorem 6 (Soundness). *Algorithm 1 is sound.*

Proof. The proof of soundness includes two parts, namely proof of closure of I and convergence to I, where $I = \forall i : L(x_{i-1}, x_i)$. Step 7 of the algorithm guarantees closure. Steps 4 to 7 ensure that starting from any state where $L(x_{i-1}, x_i)$ does not hold, process P_i will eventually set the value of x_i to γ, hence evaluating $L(x_{i-1}, x_i)$ to true. Likewise, every process would perform local recovery, thereby eventually ensuring that $\forall i : L(x_{i-1}, x_i)$ holds. □

Theorem 7 (Completeness). *Algorithm 1 is complete.*

Proof. This algorithm declares failure only in Step 2, where no value γ exists that can satisfy $L(x_{i-1}, x_i)$. The non-existence of some value that can make $L(x_{i-1}, x_i)$ true implies that no process can recover to its local invariant; hence self-stabilization to I is impossible. □

We now present some case studies for the synthesis of parameterized unidirectional symmetric rings using Algorithm 1.

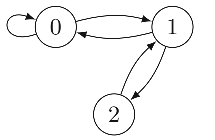
Sum-Not-2 protocol. The Sum-Not-2 protocol is a simple but interesting protocol that illustrates the complexities of designing self-stabilizing systems. This is again a protocol on unidirectional parameterized rings with a domain of $M = 3$ values; i.e., $\{0, 1, 2\}$. The invariant of the protocol specifies the legitimate

Algorithm 1. Synthesizing parameterized self-stabilizing unidirectional rings.

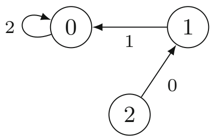
```
SynUniRing($L(x_{i-1}, x_i)$: state predicate, $M$: domain size)
1: Check if a value $\gamma \in \mathbb{Z}_M$ exists such that $L(\gamma, \gamma) = \textbf{true}$.
2: If no such $\gamma$ exists, then return $\emptyset$ since no solution exists for systems with more
   than $M$ processes due to [5].
3: Construct relation $L$ as a graph $G = (V, E)$, where each vertex $v \in V$ represents a
   value in $\mathbb{Z}_M$ and each $e \in E$ captures an arc $(v, v')$ from value $v$ to $v'$ if and only
   if $L(v, v') = \textbf{true}$.
4: Induce a subgraph $G' = (V', E')$ that contains all nodes of $G$ that participate in
   cycles involving $\gamma$.
5: Compute a spanning tree of $G'$ rooted at $\gamma$.
6: For each node $v \in G$ that is absent from $G'$, include an arc from $v$ to the root of
   the spanning tree of $G'$. The resulting graph would still be a tree, denoted $T$.
7: Include a self-loop $(\gamma, \gamma)$ at the root of $T$.
8: Transform $T$ into an action graph of a protocol by the following step:

       For each arc $(a, c)$ in $T$, where $a, c \in \mathbb{Z}_M$, label $(a, c)$ with every value $b$
       for which $L(a, b) = \textbf{false}$ and $b \neq c$.

9: Return the actions represented by the arcs of $T$.
```

states where $\forall i : (x_{i-1} + x_i) \neq 2$ holds, where addition and subtraction are in modulo 3. As such, for each process P_i, we have $L(x_{i-1}, x_i) \overset{\text{def}}{=} (x_{i-1} + x_i) \neq 2$. Figure 3a illustrates the directed graph representing L in the locality of a process. (Notice that processes are symmetric.) In this case, there are two candidate values for γ, where $L(\gamma, \gamma)$ holds; i.e., values of 0 and 2. Wlog, we choose $\gamma = 0$ and form the spanning tree of the graph G with the root of 0. Stripping the graph in Fig. 3b from the labels on its arcs would give us the spanning tree of G, and the graph with the labels is the action graph of the synthesized self-stabilizing protocol (in Fig. 3c).

Even Difference. The Even Difference protocol specifies the local invariant of each process P_i as $L(x_{i-1}, x_i) \overset{\text{def}}{=} ((x_{i-1} - x_i) \texttt{ mod } 2) = 0$, where $M = 4$. Thus, the set of legitimate states is $\forall i : ((x_{i-1} - x_i) \texttt{ mod } 2) = 0$. Notice that if there is an even (respectively, odd) value in the ring, then all values will be even (respectively, odd) in a legitimate state. As such, from any state, Even Difference will converge to either an all-odd or an all-even state. This protocol has applications in choosing a common parity policy in a distributed system, where from an arbitrary state all nodes will agree on a common parity policy. Figure 4a represents the graph corresponding to the predicate L. All four values in the domain M are candidate values for γ. We choose $\gamma = 1$, and generate the action graph of Fig. 4b. Figure 4c illustrates the actions of the self-stabilizing protocol. Please notice that this protocol would recover to global states where all values are odd. Symmetrically, one could generate a protocol that would stabilize to states where all values are even. This could be achieved by strengthening $L(x_{i-1}, x_i)$ by an additional constraint $(x_i \texttt{ mod } 2 = 0)$ (respectively, $(x_i \texttt{ mod } 2 \neq 0)$).



(a) Graph representing predicate $L(x_{i-1}, x_i) = ((x_{i-1} + x_i) \neq 2)$ where each $x_j \in \mathbb{Z}_3$

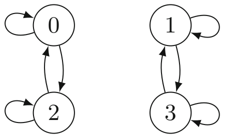


(b) Action graph of the self-stabilizing protocol.

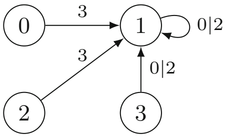
$$x_{i-1}{=}0 \wedge x_i{=}2 \longrightarrow x_i := 0;$$
$$x_{i-1}{=}1 \wedge x_i{=}1 \longrightarrow x_i := 0;$$
$$x_{i-1}{=}2 \wedge x_i{=}0 \longrightarrow x_i := 1;$$

(c) Actions of each process P_i.

Fig. 3. Synthesis of parameterized Sum-Not-2.



(a) Graph representing predicate $L(x_{i-1}, x_i) = ((x_{i-1} - x_i) \texttt{ mod } 2 = 0)$ where each $x_j \in \mathbb{Z}_4$.



(b) Action graph of the self-stabilizing protocol.

$$(x_{i-1}{=}1 \vee x_{i-1}{=}3) \wedge (x_i{=}0 \vee x_i{=}2) \longrightarrow x_i := 1;$$
$$(x_{i-1}{=}0 \vee x_{i-1}{=}2) \wedge x_i{=}3 \longrightarrow x_i := 1;$$

(c) Actions of each process P_i.

Fig. 4. Synthesis of parameterized Even Difference.

4 Undecidability of Synthesizing Bidirectional Rings

While synthesizing parameterized self-stabilizing protocols is decidable for unidirectional rings, we show that synthesis is undecidable for bidirectional rings.

Theorem 8. *Given a predicate $I \overset{\text{def}}{=} (\forall i : L(x_{i-1}, x_i, x_{i+1}))$ and variable domain M (such that each $x_i \in \mathbb{Z}_M$) for a bidirectional ring, it is undecidable (Π^0_1-complete) whether a protocol can stabilize to I for all ring sizes.*

Proof. To show undecidability, we reduce the problem of verifying livelock freedom of a unidirectional ring protocol p to the problem of synthesizing a bidirectional ring protocol p' that stabilizes to I', where I' has some form determined by p. We construct I' such that exactly one bidirectional ring protocol p' resolves all deadlocks without breaking closure, but it only stabilizes to I' if p is livelock-free. Thus, p' is the only candidate solution for the synthesis procedure, and the synthesis succeeds *iff* p is livelock-free. Our reduction is broken into two parts: (1) showing that exactly one particular p' resolves all deadlocks without breaking closure, and (2) showing that p' is livelock-free *iff* p is livelock-free.

Silent stabilization. Wlog, we present our proof for *silent* stabilizing protocols where the protocol p' does not take any actions in I'.

Assumptions about p. We assume that p (1) has a deterministic livelock that (2) involves all actions and (3) includes all values. These assumptions do not affect the undecidability of verifying livelock freedom in p. First, by Theorem 3, deterministic livelock detection is undecidable in unidirectional rings. Second, deterministic livelock detection remains undecidable when the livelock involves all actions; otherwise, we could detect deterministic livelocks by checking each subset of actions. Third, deterministic livelock detection is undecidable even when the livelock involves all values; otherwise, we could detect deterministic livelocks by checking each subset of values. Thus, verifying livelock-freedom under our assumptions for p remains undecidable.

Forming I' from p. To form I', we augment each process P_i with a new variable $x'_{i-1} \in \mathbb{Z}_M$, which is a local copy of x_{i-1}, along with its $x_i \in \mathbb{Z}_M$, making its effective domain size $M' \stackrel{\text{def}}{=} M^2$. Since p' is a bidirectional ring, P_i can read x_{i-1} and x'_{i-2} from P_{i-1} and can read x_{i+1} and x'_i from P_{i+1}. For each action $(a, b, c) \in \xi$, we use $x_{i-1} = a$ and $x'_i = b$ to encode the precondition of a P_i action (a, b, c), and $x_i = c$ to encode its assignment. Notice that x'_i is from P_{i+1} as depicted in Fig. 5. Thus, we must ensure that x'_i eventually obtains a copy of x_i. The resulting $I' \stackrel{\text{def}}{=} (\forall i : L'(x_{i-1}, x_i))$ is as follows with instances of x_i replaced with x'_i and a condition that x'_{i-1} is a copy of x_{i-1}.

$$\begin{aligned} L'(x_{i-1}, x_i) \stackrel{\text{def}}{=} \big(& (x_{i-1}, x'_i) \in \mathsf{Pre}(\xi) \\ & \implies x'_{i-1} = x_{i-1} \wedge x_i = \xi(x_{i-1}, x'_i) \big) \end{aligned}$$

Forming p' from I'. We want to show that a particular p' stabilizes to I' when p is livelock-free, and it is the only bidirectional ring protocol that resolves deadlocks without breaking closure. This p' has the following action for each P_i.

$$\begin{aligned} & (x_{i-1}, x'_i) \in \mathsf{Pre}(\xi) \wedge \big(x'_{i-1} \neq x_{i-1} \vee x_i \neq \xi(x_{i-1}, x'_i) \big) \\ & \qquad \longrightarrow \quad x'_{i-1} := x_{i-1};\ x_i := \xi(x_{i-1}, x'_i); \end{aligned}$$

Notice that p' is deadlock-free and preserves closure since a process P_i can act *iff* its $L'(x_{i-1}, x_i)$ is unsatisfied. We now show that this p' is the only such protocol. Consider a ring of 5 processes executing p' where a process P_2 and its readable variables from P_1 and P_3 have arbitrary values. By our earlier assumptions about p, it has an action (a, b, c) for any given a or c (not both), and

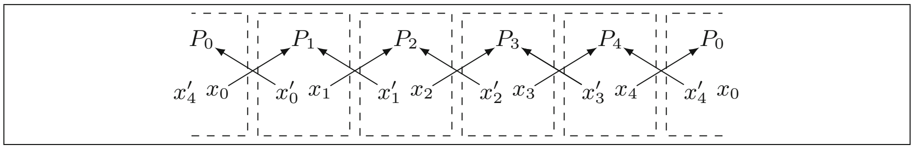


Fig. 5. Topology for bidirectional ring protocol p' in Theorem 8. Each process P_i owns x'_{i-1} and x_i.

$(a, c) \notin \mathsf{Pre}(\xi)$ because processes of p are self-disabling. Thus, we can choose x_0 of P_0 to make $(x_0, x'_1) \notin \mathsf{Pre}(\xi)$ for P_1, and we can choose x'_3 of P_4 to make $(x_2, x'_3) \notin \mathsf{Pre}(\xi)$ for P_3. We have satisfied L'_1 and L'_3, and we can likewise satisfy L'_0 and L'_4 by choosing values of x_4 and x'_4 respectively. Thus, p' is in a legitimate state *iff* L'_2 is satisfied. Therefore, if L'_2 is satisfied, then P_2 cannot act without adding a transition within I' (i.e., breaking closure). As a consequence, no other process but P_2 can act if L'_2 is not satisfied. Since processes are symmetric, each P_i of p' must have the above action to ensure $x'_{i-1} = x_{i-1}$ and $x_i = \xi(x_{i-1}, x'_i)$ when $(x_{i-1}, x'_i) \notin \mathsf{Pre}(\xi)$.

If p has a livelock, then p' has a livelock. Assume p has a livelock. We show that p' has a livelock too. We prove this by showing that p' can simulate the livelock of p. By assumption, p has a deterministic livelock from some state $C = (c_0, \ldots, c_{N-1})$ on a ring of size N where only the first process is enabled; i.e., $(c_{i-1}, c_i) \in \mathsf{Pre}(\xi)$ only for $i = 0$. Let $C' = (c'_0, \ldots, c'_{N-1})$ be the state of this system after all processes act once. That is, $c'_0 = \xi(c_{N-1}, c_0)$ and $c'_i = \xi(c'_{i-1}, c_i)$ for all other $i > 0$. We can construct a livelock state of p' from the same $x_i = c_i$ values for all i and $x'_i = c_i$ for all $i < N - 1$. The value of x'_{N-1} can be c_{N-1}, but can be anything else such that $(x_{N-2}, x'_{N-1}) \notin \mathsf{Pre}(\xi)$. In this state of p', only P_0 is enabled since we assumed that $(c_{i-1}, c_i) \in \mathsf{Pre}(\xi)$ only holds for $i = 0$. P_0 then performs $x_0 := c'_0$ and $x'_{N-1} := c_{N-1}$. This does not enable P_{N-1}, but does enable P_1 to perform $x_1 := c'_1$ and $x'_0 := c'_0$. The execution continues for $P_2, \ldots, P_{N-1}$ to assign $x_i := c'_i$ and $x'_{i-1} := c'_{i-1}$ for all $i > 1$. At this point the system is in a state where $x_i = c'_i$ for all i and $x'_i = c'_i$ for all $i < N - 1$. The value of x'_{N-1} is c_{N-1}, which leaves it disabled. This state of p' matches the state C' of p using the same constraints as we used to match the initial state C. Therefore, p' can continue to simulate p, showing that it has a livelock.

If p is livelock-free, then p' is livelock-free. Assume p is livelock-free. We show that p' is livelock-free too. First, notice that if P_{i+1} acts immediately after P_i in p', then P_i will not become enabled because $x_i = x'_i$ and self-disabling processes of p ensure that $(a, c) \notin \mathsf{Pre}(\xi)$ for every action (a, b, c). This means that in a livelock, if an action of P_{i+1} enables P_i, then P_{i-1} must have acted since the last action of P_i. As such, an action of P_{i-1} must occur between every two actions of P_i in a livelock of p'. The number of such propagations clearly cannot increase, and thus must remain constant in a livelock. In order to avoid collisions, an action of P_{i+1} must occur between every two actions of P_i. Since P_{i+1} always acts before P_i in a livelock of p', it ensures that $x'_i = x_i$ when P_i acts. By making this substitution, we see that P_i is only enabled when $(x_{i-1}, x_i) \in \mathsf{Pre}(\xi)$, and assigns $x_i := \xi(x_{i-1}, x_i)$, which is equivalent to the behavior of protocol p. Since p is livelock-free, p' must also be livelock-free, hence self-stabilizing iff p is livelock-free. Therefore, synthesizing stabilization on bidirectional rings is undecidable.

5 Related Work

This section discusses existing work related to verification and synthesis of parameterized systems.

Verification. The literature for the verification of parameterized systems can broadly be classified into undecidability results and verification methods for decidable cases. In their seminal work, Apt and Kozen [2] prove that verifying an Linear Temporal Logic (LTL) formula for a parameterized system is in general undecidable. Suzuki [27] extends their results by showing that the verification problem remains undecidable for unidirectional ring protocols of symmetric processes. While Farahat and Ebnenasir [15] show that verifying deadlock-freedom of parameterized rings is decidable, Fabret and Petit [14] prove that if the underlying communication graph is a planar grid, then deadlock-freedom becomes undecidable. In our previous work [22], we show that verifying livelock-freedom is undecidable even on a symmetric ring of self-disabling and deterministic processes. Our results imply the undecidability of verifying self-stabilization on unidirectional rings. Several researchers present cutoff theorems that reduce the verification of parameterized systems to the verification of a small-scale instantiation (i.e., cutoff) thereof such that the parameterized system meets a specific property iff its cutoff instantiation satisfies the desired property. For example, Emerson and Namjoshi [13] provide a cutoff theorem for the verification of LTL without the next-state operator in token passing rings. Several other researchers [3,12,17,24] extend Emerson and Namjoshi's results for other topologies and for different properties/systems. Methods based on regular model checking [1,7] represent states of parameterized rings as strings of arbitrary length, and a protocol is represented by a finite state transducer. The properties such as deadlock and livelock-freedom are formulated in an automata-theoretic context. The aforementioned approaches are mostly used to verify local properties that are specified in terms of the locality of a process or a proper subset of processes, whereas self-stabilization includes a global liveness property that must be met by local actions of all processes.

Synthesis. Existing synthesis methods can be classified into problem-specific and general approaches. The problem-specific methods focus on generating a parameterized solution for a specific problem (e.g., counting [11,25], consensus [4], sorting [8], etc.). General methods [16,20] for the synthesis of parameterized systems are mainly *specification-based* in that they provide a decision procedure for extracting the skeleton of symmetric processes from their temporal logic specifications. Some existing methods [6] exploit cutoff theorems to generate the template code of parameterized systems. Moreover, several researchers [6,11,21] utilize SMT/SAT solvers for synthesis where they either directly encode the synthesis problem as a set of constraints fed into the solver, or exploit counter-example guided search [11] to find solutions in a bounded scope.

While existing methods are effective in their stated objectives, they often make restrictive assumptions (e.g., synchrony, fairness) to mitigate the complexity of synthesis. We believe that part of this complexity is because of the way synthesis is conceived; that is, generate code skeleton from temporal logic specifications. By contrast, we think that synthesis of parameterized systems must be done on a *property-based* fashion where we devise methods for the synthesis of systems that meet a specific property (e.g., self-stabilization). Such an investigation can be extended to different network topologies (e.g., tree, mesh).

6 Conclusions and Future Work

In this paper, we investigated the problem of synthesizing parameterized rings that have the property of self-stabilization. The ring processes are deterministic and have constant state space. Moreover, we consider self-disabling processes, where a process disables itself after executing an action until it is enabled again by the actions of other processes. While it is known that verifying self-stabilization of unidirectional rings is undecidable [22], in this paper, we present a surprising result that synthesizing self-stabilizing unidirectional rings is actually decidable. We present a sound and complete algorithm for the synthesis of self-stabilizing unidirectional rings, and apply our algorithms to a few case studies. We also show that the synthesis problem becomes undecidable if we assume bidirectional rings. As an extension to this work, we are investigating the application of our approach to other topologies such as trees and meshes. Furthermore, we are integrating our algorithms in Protocon (http://asd.cs.mtu.edu/projects/protocon/), which is a framework for the synthesis of self-stabilizing systems.

References

1. Abdulla, P.A., Jonsson, B., Nilsson, M., Saksena, M.: A survey of regular model checking. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 35–48. Springer, Heidelberg (2004). doi:10.1007/978-3-540-28644-8_3
2. Apt, K.R., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. **22**(6), 307–309 (1986)
3. Außerlechner, S., Jacobs, S., Khalimov, A.: Tight cutoffs for guarded protocols with fairness. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 476–494. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49122-5_23
4. Berman, P., Garay, J.A., Perry, K.J.: Towards optimal distributed consensus. In: 30th Annual Symposium on Foundations of Computer Science, pp. 410–415. IEEE (1989)
5. Bernard, S., Devismes, S., Potop-Butucaru, M.G., Tixeuil, S.: Optimal deterministic self-stabilizing vertex coloring in unidirectional anonymous networks. In: 23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, 23–29 May 2009, pp. 1–8. IEEE (2009)

6. Bloem, R., Braud-Santoni, N., Jacobs, S.: Synthesis of self-stabilising and byzantine-resilient distributed systems. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 157–176. Springer, Cham (2016). doi:10.1007/978-3-319-41528-4_9
7. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000). doi:10.1007/10722167_31
8. Bundala, D., Závodný, J.: Optimal sorting networks. In: Dediu, A.-H., Martín-Vide, C., Sierra-Rodríguez, J.-L., Truthe, B. (eds.) LATA 2014. LNCS, vol. 8370, pp. 236–247. Springer, Cham (2014). doi:10.1007/978-3-319-04921-2_19
9. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM **54**(9), 69–77 (2011)
10. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM **17**(11), 643–644 (1974)
11. Dolev, D., Korhonen, J.H., Lenzen, C., Rybicki, J., Suomela, J.: Synchronous counting and computational algorithm design. In: Higashino, T., Katayama, Y., Masuzawa, T., Potop-Butucaru, M., Yamashita, M. (eds.) SSS 2013. LNCS, vol. 8255, pp. 237–250. Springer, Cham (2013). doi:10.1007/978-3-319-03089-0_17
12. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: McAllester, D. (ed.) CADE 2000. LNCS (LNAI), vol. 1831, pp. 236–254. Springer, Heidelberg (2000). doi:10.1007/10721959_19
13. Emerson, E.A., Namjoshi, K.S.: On reasoning about rings. Int. J. Found. Comput. Sci. **14**(4), 527–550 (2003)
14. Fabret, A.-C., Petit, A.: On the undecidability of deadlock detection in families of nets. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 479–490. Springer, Heidelberg (1995). doi:10.1007/3-540-59042-0_98
15. Farahat, A., Ebnenasir, A.: Local reasoning for global convergence of parameterized rings. In: IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 496–505 (2012)
16. Finkbeiner, B., Schewe, S.: Bounded synthesis. Int. J. Softw. Tools Technol. Transfer **15**(5–6), 519–539 (2013)
17. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM **39**, 675–735 (1992)
18. Gouda, M.G., Haddix, F.F.: The stabilizing token ring in three bits. J. Parallel Distrib. Comput. **35**(1), 43–48 (1996)
19. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2), 256–290 (2002)
20. Jacobs, S., Bloem, R.: Parameterized synthesis. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 362–376. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28756-5_25
21. Khalimov, A., Jacobs, S., Bloem, R.: Towards efficient parameterized synthesis. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 108–127. Springer, Heidelberg (2013). doi:10.1007/978-3-642-35873-9_9
22. Klinkhamer, A., Ebnenasir, A.: Verifying livelock freedom on parameterized rings and chains. In: International Symposium on Stabilization, Safety, and Security of Distributed Systems, pp. 163–177 (2013)
23. Klinkhamer, A., Ebnenasir, A.: Shadow/puppet synthesis: a stepwise method for the design of self-stabilization. IEEE Trans. Parallel Distrib. Syst. **27**(11), 3338–3350 (2016)
24. Kurshan, R.P., McMillan, K.L.: A structural induction theorem for processes. Inf. Comput. **117**(1), 1–11 (1995)

25. Lenzen, C., Rybicki, J.: Near-optimal self-stabilising counting and firing squads. arXiv preprint arXiv:1608.00214 (2016)
26. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesis. In: Proceedings of 31st IEEE Symposium on Foundation of Computer Science, Computer Society, pp. 746–757. IEEE, Washington, DC (1990)
27. Suzuki, I.: Proving properties of a ring of finite-state machines. Inf. Process. Lett. **28**(4), 213–214 (1988)
28. Varghese, G.: Self-stabilization by local checking and correction. PhD thesis. MIT (1993)
29. Varghese, G.: Self-stabilization by counter flushing. In: The 13th Annual ACM Symposium on Principles of Distributed Computing, pp. 244–253 (1994)