On the Verification of Livelock-Freedom and Self-Stabilization on Parameterized Rings

Ali Ebnenasir and Alex Klinkhamer

Michigan Technological University
Computer Science Technical Report
CS-TR-19-01
January 2019

On the Verification of Livelock-Freedom and Self-Stabilization on Parameterized Rings

Ali Ebnenasir and Alex Klinkhamer

January 2019

Abstract

This paper investigates the verification of livelock-freedom and self-stabilization on parameterized rings consisting of symmetric, constant space, deterministic and self-disabling processes. The results of this paper have a significant impact on several fields including scalable distributed systems, resilient and self-* systems, and verification of parameterized systems. First, we identify necessary and sufficient local conditions for the existence of global livelocks in parameterized unidirectional rings with unbounded (but finite) number of processes under the interleaving semantics. Using a reduction from the periodic domino problem, we show that, in general, verifying livelock-freedom of parameterized unidirectional rings is undecidable (specifically, Π_1^0-complete) even for constant space, deterministic and self-disabling processes. This result implies that verifying self-stabilization for parameterized rings of self-disabling processes is also undecidable. We also show that verifying livelock-freedom and self-stabilization remain undecidable under (1) synchronous execution semantics; (2) the FIFO consistency model, and (3) any scheduling policy. We then present a new scope-based method for detecting and constructing livelocks in parameterized rings. The proposed semi-algorithm behind our scope-based verification is based on a novel paradigm for the detection of livelocks that totally circumvents state space exploration. Our experimental results on an implementation of the proposed semi-algorithm are very promising as we have found livelocks in parameterized rings in a few micro seconds on a regular laptop. The results of this paper have significant implications for scalable distributed systems with cyclic topologies.

Contents

1 Introduction

Distributed self-stabilizing systems have the property of recovering from any state to a set of legitimate states (that capture the normal/desired behaviors) without a need for a central point of coordination. Self-stabilization has two requirements, namely closure and convergence [13]. *Convergence* stipulates that, from any state, every execution will eventually reach a legitimate state. In other words, a system converges to its legitimate states if and only if it is deadlock-free and livelock-free in illegitimate states. *Closure* states that, starting from any legitimate state, system executions remain in legitimate states as long as no faults occur. Verifying convergence in non-parametric systems is known to be a difficult task [30]. For this reason, a relaxed version of convergence, called *weak convergence*, requires that from any state, a system has at least one execution that reaches legitimate states. Designing and verifying convergence for parameterized systems are important problems as they have applications in several fields such as network protocols [13, 32], multi-agent systems [28], cloud computing [50], and equilibrium in socioeconomic systems [31]. *Parameterized systems* comprise some families of symmetric processes, where two processes are *symmetric* if the code of one can be obtained from the other by a simple variable renaming/re-indexing. Each family may contain an unbounded (but finite) number of symmetric processes that communicate based on a specific network topology. This paper investigates the verification of self-stabilization for parameterized rings.

There is a rich body of work on the verification of parameterized systems [68, 7, 37, 57, 41, 19, 33, 29, 12, 62, 54, 25] most of which cannot be directly used for the verification of self-stabilization due to the requirements that (1) convergence must be met from *any* state and not just a finite set of initial states, and (2) convergence is a global liveness property rather than a local liveness property (e.g., progress of each thread). For example, numerous methods [27, 20, 9, 24] focus on the verification of safety properties of parameterized systems, where safety requires that nothing bad happens in system executions (e.g., no deadlock state is reached). Other approaches extend the results of Apt and Kozen [3] where they illustrate that, in general, verifying Linear Temporal Logic (LTL) [16] properties for parameterized systems is Π^0_1-complete. Suzuki [64] shows that the verification problem remains Π^0_1-complete even for unidirectional rings. The main contributions of this paper are as follows:

Contributions. In this paper, we extend existing results for the special case where the property of interest is self-stabilization and every system state is under consideration.

- Since convergence entails livelock-freedom, we first study the verification of livelock-freedom under the assumption of processes that are deterministic, have constant state spaces, and are *self-disabling*, i.e., no action of a process is enabled immediately after it acts. (Previous work [24] demonstrates the decidability of deadlock-freedom in parameterized rings.) Specifically, we first illustrate that, even under our restrictive assumptions livelock detection is undecidable (Σ^0_1-complete) on unidirectional ring topologies. The proof of undecidability in our work is based on a reduction from the periodic domino problem [34] because it is unclear how Suzuki's proof [64] can directly be used for fully symmetric self-stabilizing rings. (Please see Section 8 for detailed reasoning). This result implies that verifying self-stabilization for parameterized rings of self-disabling processes is also undecidable. These undecidability results hold for any system that includes parameterized rings as subgraphs of its topology graph (i.e., cyclic topologies). Our results strengthen Apt and Kozen's [3] results in that we consider constant space, deterministic and self-disabling processes, and assume that any state could be an initial state.

- We also show that our undecidability results hold under (1) synchronous and asynchronous execution semantics; (2) the FIFO consistency model, and (3) any scheduling policy. This is a counterintuitive result because the common understanding [30] is that assuming strong fairness ensures livelock-freedom. By contrast, we show that detecting livelocks in symmetric unidirectional rings (a.k.a. uni-rings) remains undecidable even under strong fairness.

- We then present a novel scope-based method for detecting and constructing livelocks in parameterized uni-rings. The proposed semi-algorithm behind our scope-based verification is based on a transformative method for the detection of livelocks that totally circumvents state space exploration. An implementation of our semi-algorithm, called *Prop*, is able to find and construct livelocks in parameterized rings of several hundred processes in a fraction of a second on a regular laptop machine. It is infeasible to achieve such efficiency of verification using existing model checkers for parameterized [12] or non-parameterized [36] systems. We have also developed an enumerator (i.e., Turing recognizer) that searches through all possible protocols to find the ones that have livelocks and have compared its time efficiency with *Prop* in Section 7. The source codes of *Prop* and the enumerator are available at `https://github.com/grencez/protocon/tree/master/src/uni`.

Organization. Section 2 presents some basic concepts. Section 3 provides a local characterization of global livelocks in unidirectional rings. Then, Section 4 presents a well-known undecidable problem that we will use to show the undecidability of detecting livelocks in unidirectional ring protocols. Section 5 gives far-reaching undecidability results for livelock detection and verifying stabilization. Section 6 presents a novel method for scope-based livelock detection in unidirectional rings. Section 7 presents our experimental results. Section 8 discusses related work, and Section 9 summarizes our contributions and outlines some future work.

2 Basic Concepts

This section presents the definition of protocols, action graphs and propagations. Some concepts in this section are adapted from [24, 23]. A *protocol* p is a network of $N > 1$ processes (finite-state machines), where each process P_i owns a finite set of variables whose valuation determines its *state*. The state of a protocol is defined by the current states of all processes. A process *acts* when it atomically changes its state based on its current state and the states of its neighboring processes, where neighbors are defined by the network topology. For example, in a unidirectional ring topology consisting of N processes, each process P_i (where $i \in \mathbb{Z}_N$, i.e., $0 \le i \le N-1$) has a neighbor P_{i-1}, where subtraction is in modulo N. A *global transition* is a change in the system state. We assume that one process acts at a time (i.e., interleaving/asynchronous semantics), therefore each global transition corresponds to the action of a single process from some global state. In a *synchronous* protocol, a global transition is the result of simultaneous execution of all processes. 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. Furthermore, we assume that the state space Σ and rules for each process are independent of the topology (e.g., number of processes).

Definition 2.1 (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 $\xi : \Sigma \times \Sigma \to \Sigma$ as a partial function such that $\xi(a, b) = c$ if and only if (iff) 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.

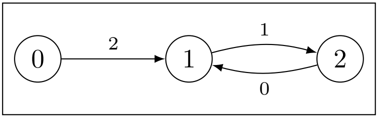


Figure 1: Action graph representing Sum-Not-2 protocol.

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 any process P_i in a unidirectional ring protocol. To visually represent the structure of a process, we depict a protocol by a labeled directed multigraph, called the *action graph* of the protocol, 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 [23]. 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 $(x_{i-1} = 2 \wedge x_i = 0 \longrightarrow x_i := 1)$. We represent this protocol with a graph containing arcs $(0, 2, 1)$, $(1, 1, 2)$, and $(2, 0, 1)$ as shown in Figure 1.

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 $d \neq c$. That is, in action graphs, the labels of outgoing arcs from any vertex have distinct labels.
Livelock, deadlock, and closure. A *legitimate* state is a state that we want the system to be in. 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 protocol p when no transition exists that brings the system from a state in I to a state in $\neg I$. These concepts allow us to define self-stabilization and weak stabilization (adapted from [30]).

Definition 2.2 (Self-Stabilization)**.** A protocol p is *self-stabilizing* with respect to its legitimate state predicate I *iff* from every illegitimate state, all executions reach (i.e., *convergence*) and remain (i.e., *closure*) in the set of legitimate states. That is, p is livelock-free and deadlock-free, and I is closed under p.

Definition 2.3 (Silent Stabilization)**.** A protocol p is *silent-stabilizing* with respect to its legitimate state predicate I *iff* p is self-stabilizing to I and no process is enabled to act within I.

As an example, the Sum-Not-2 protocol converges to states where the sum of each two consecutive x values does not equal 2 (i.e., the state predicate $\forall i : (x_{i-1} + x_i \neq 2)$).

Definition 2.4 (Strong Fairness)**.** We adopt Gouda's definition of *strong fairness* [30], where an execution is *strongly* fair *iff* for any state C_0 and a transition (C_0, C_1), if C_0 is reached infinitely often, then (C_0, C_1) is executed infinitely often.

Definition 2.5 (Weak Stabilization)**.** A protocol p is *weakly stabilizing* with respect to its legitimate state predicate I *iff* from each illegitimate state, an execution exists to a legitimate state, and I is closed under p.

Definition 2.6 (Weak Fairness)**.** An execution is *weakly* fair *iff* any action that is continuously enabled is executed infinitely often.

Now, we re-state one of our previous results [24] about the relation between livelocks in uni-rings and weak fairness.

Theorem 2.7. *A unidirectional ring protocol p has a livelock L for some ring size under no fairness iff p has the livelock L for the same ring size on a weakly fair scheduler.*

The proof in [24] is intuitively based on the fact that, in any livelocked execution, no process/action of a uni-ring is continuously enabled, thereby vacuously meeting the requirement of weak fairness.
Propagations and Collisions. When a process acts and enables its successor, it propagates its ability to act. The successor may enable its own successor by acting, and the pattern may continue indefinitely. In other words, such behaviors can be represented as sequences of actions that are propagated in a ring, called *propagations.* Consider a propagation $\langle (a, b, c), (d, e, f) \rangle$ of length 2 that says a state exists that allows some P_i to perform an action (a, b, c) that then enables P_{i+1} to perform (d, e, f). Since P_i assigns c to its variable x_i and P_{i+1} is then enabled to perform (d, e, f), which relies on $x_i = d$ and $x_{i+1} = e$, we know $c = d$. We therefore write the jth action of a propagation as (a_{j-1}, b_j, a_j). It follows that a propagation is a walk through the protocol's 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 if its jth action and $(j+n)$th action are the same for every index j. A periodic propagation with period n corresponds to a closed walk of length n in the action graph. The Sum-Not-2 protocol has such a propagation of period 2: $\langle (1, 1, 2), (2, 0, 1) \rangle$.

By contrast, 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 (due to the assumption of self-disabling processes), and P_{i+1} becomes disabled too because x_i is no longer equal to b. The net result is that two enabled processes become disabled in one action; i.e., a collision.

3 Local Characterization of Global Livelocks

This section presents a local characterization of global livelocks in parameterized rings, which is an extension of [24, 23]. This characterization is based on the notion of propagations and a *leads* relation between the propagations. We shall use propagations and the *leads* relation to identify necessary and sufficient conditions for the existence of livelocks in symmetric unidirectional ring protocols of self-disabling processes.

"Leads" Relation. Consider two actions A_1 and A_2 in a process P_i. We say the action A_1 *leads* A_2 if and only if 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) if and only if $e = c$. For example, in Figure 1, action $A_1 = (0, 2, 1)$ leads action $A_2 = (1, 1, 2)$ because A_1 sets x_i to 1, which is the value required for A_2 to execute. Symmetrically, action A_2 also leads action A_1 because it sets x_i to 2, which is a requirement for A_1 to be enabled and executed. Similarly, a propagation leads another if and only if, for every index j, its jth action leads the jth action of the other propagation. Thus, if we have a propagation whose jth action is (a_{j-1}, b_j, a_j) that leads another propagation whose jth action is (d_{j-1}, e_j, d_j), then we know $e_j = a_j$ and write the led action as (d_{j-1}, a_j, d_j). In the context of the protocol action graph, this corresponds to two walks (representing propagations) where the jth destination node label of the first walk matches the jth arc label of the second walk for each index j. After some first propagation executes through a ring segment $P_q, \ldots, P_{q+n-1}$, a second propagation can execute through the same segment only if the first propagation leads the second. This is true since each process P_{q+j} performs the jth action of the first propagation, assigning its variable x_{q+j} to some value a_j. If the second propagation executes through the segment, each P_{q+j} must perform the jth action of the second propagation from a state where $x_{q+j} = a_j$. As such, each jth action of the first propagation must lead the jth action of the second propagation. Thus, the first propagation itself must lead the second.

Example 3.1. *Sum-Not-Odd protocol and its periodic propagations.*

Consider a unidirectional ring where each process P_i has a variable $x_i \in \mathbb{Z}_5$. Figure 2 illustrates the action graph of a protocol for the set of legitimate states $\forall i : ((x_{i-1} + x_i) \texttt{ mod } 2) = 0$; hence the term *Sum-Not-Odd*. We find values that show Lemma 3.2 holds for this protocol. Consider the propagation $\mathcal{P}_1 = (4, 1, 2), (2, 3, 0), (0, 3, 2), (2, 1, 4)$. This propagation can occur in four consecutive process P_i, P_{i+1}, P_{i+2} and P_{i+3}, where x_i, x_{i+1}, x_{i+2} and x_{i+3} are respectively equal to 1, 3, 3, 1. Starting at the node labeled 4, this propagation sets the values of four consecutive nodes to $2, 0, 2$, and 4. Notice that, when a process takes the value 4, its successor will be enabled only if it has the values 3 and 1 (corresponding to the two outgoing arcs from Node 4). Now, the propagation $\mathcal{P}_1$ can lead another propagation, say $\mathcal{P}_2$, if $\mathcal{P}_2$ respectively carries arc labels 2, 0, 2, and 4 in the action graph of the protocol. The top graph in Figure 2 shows that there is such a propagation $\mathcal{P}_2$ containing the actions $(3, 2, 3), (3, 0, 1), (1, 2, 1), (1, 4, 3)$. The propagation $\mathcal{P}_2$ will result in values 3, 1, 1, and 3 in the same segment of the ring. By the same reasoning, we can find values for the third propagation $\mathcal{P}_3 = (0, 3, 2), (2, 1, 4), (4, 1, 2), (2, 3, 0)$ (led by $\mathcal{P}_2$), which will leave the values 2, 4, 2 and 0 in the same segment of the ring. Finally, the propagation $\mathcal{P}_3$ leads propagation $\mathcal{P}_4 = (1, 2, 1), (1, 4, 3), (3, 2, 3), (3, 0, 1)$, which will get the segment back to values 1, 3, 3, 1. That is, $\mathcal{P}_4$ leads $\mathcal{P}_1$.

We focus on scenarios where for some positive integers m and n, there are m periodic propagations with period n where the ith propagation leads the $(i+1)$th propagation for each $i \in \mathbb{Z}_m$ (and the last propagation leads the first). This case can be represented succinctly.

Lemma 3.2 (Periodic Propagations that Successively Lead)**.** *Consider a unidirectional ring protocol of symmetric, self-disabling processes that may exhibit m propagations of period n where each ith propagation leads the $(i + 1)$th propagation (and $i \in \mathbb{Z}_m$). Such propagations exists iff there exist a^i_j values along with superscript and subscript indices i and j modulo m and n (respectively) such that $(a^i_{j-1}, a^{i-1}_j, a^i_j)$ would denote the jth action of ith propagation.*

Proof. The proof is by combining the notations used in this section when defining periodic propagations and the leads relation. Using $\texttt{X}$ as a wildcard value (i.e., any value, do not assume $\texttt{X} = \texttt{X}$), recall that an action is defined to lead another action $(\mathtt{X}, a, \mathtt{X})$ *iff* it has the form $(\mathtt{X}, \mathtt{X}, a)$. Also recall that a propagation of period n has the form $\langle(a_{n-1}, \mathtt{X}, a_0), (a_0, \mathtt{X}, a_1), \ldots, (a_{n-2}, \mathtt{X}, a_{n-1})\rangle$. Thus, we can write each ith propagation as $\langle(a^i_{n-1}, \mathtt{X}, a^i_0), (a^i_0, \mathtt{X}, a^i_1), \ldots, (a^i_{n-2}, \mathtt{X}, a^i_{n-1})\rangle$ and can determine the $\mathtt{X}$ values as $\langle(a^i_{n-1}, a^{i-1}_0, a^i_0), (a^i_0, a^{i-1}_1, a^i_1), \ldots, (a^i_{n-2}, a^{i-1}_{n-1}, a^i_{n-1})\rangle$. This makes $(a^i_{j-1}, a^{i-1}_j, a^i_j)$ the jth action of the ith propagation. □

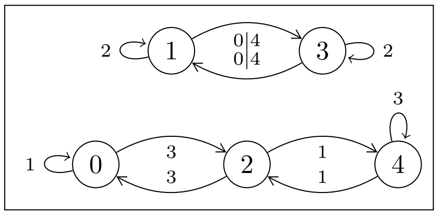


Figure 2: Action graph representing the *Sum-Not-Odd* protocol.

Lemma 3.3. *Let p be a uni-ring protocol where processes are symmetric and self-disabling. An execution of p from some state C to itself exists (i.e., a livelock exists) for some ring size* iff *an execution of p exists from C to some C' obtained by rotating the values of C by some $k \geq 0$ positions.*

Proof. $\Rightarrow$: An execution from C to C can simply repeat to make an infinite execution; i.e., a livelock. For $k = 0$ and $C' = C$, such an infinite execution must visit $C = C'$ infinitely often.

$\Leftarrow$: To finish the proof, assume an execution from C to C' exists for some k with the goal of showing that an execution from C to C exists. Since processes are symmetric, another execution exists that rotates values by another k positions, leaving the C values rotated by $2k$ positions. With N being the ring size, this can continue for N/k total rotations to return the system to state C. Emerson and Namjoshi [18] similarly use this notion of rotational symmetry to reason about rings of symmetric processes. □

Lemma 3.4. *Let p be a unidirectional ring protocol of symmetric, self-disabling processes. The protocol p has a livelock for some ring size iff for some $m, n > 1$, protocol p has m propagations with some period n, where the $(i-1)$-th propagation leads the ith propagation for each index i modulo m; i.e., the propagations successively lead each other modulo m.*

Proof. The proof is broken into two parts that show existence of a livelock is sufficient and necessary for any m propagations of period n to exist such that each propagation leads the next.

$\Rightarrow$ Assume a livelock exists on a ring of size N. By Lemma 3.3, an execution exists that revisits some state C. Let m denote the number of enabled processes at C, and let $i_0, \ldots, i_{m-1}$ denote their positions in the ring. The number of enabled processes is m in all states between two visitations of C in the livelock since, when processes in a unidirectional ring are self-disabling, the number of enabled processes will not increase or decrease in a livelock [24] (a process enables its successor when it disables itself). Thus, between two visitations of C, for each i_j, actions propagate forward through indices $i_j, i_{j+1}, \ldots, i_{j+k-1}$ for some k, leaving process $P_{i_{j+k}}$ enabled. After revisiting C a total of m times, for each i_j, the propagation that started at index i_j leaves $P_{i_{j+mk}} = P_{i_j}$ enabled. Since each of the m propagations has reached the same position in the same state after km total transitions, they can repeat with period k.

$\Leftarrow$ Assume that m propagations of period n exist where each $(i-1)$th propagation leads the ith for every index i modulo m. From Lemma 3.2, we can write the jth action of the ith propagation as $(a^i_{j-1}, a^{i-1}_j, a^i_j)$ using some appropriate a^i_j values from the process domains. We will use these a^i_j values to find an execution from some state to itself to form a livelock. Construct a ring of mn processes where the initial state is defined with $x_{((m-1-i)n+j)} = a^i_j$ for all $i \in \mathbb{Z}_m$ and $j \in \mathbb{Z}_n$. This initial state and future ones in an execution are shown as rows in Figure 3. Conceptually, we use the first propagation to initialize the last n processes $(x_{((m-1)n)}, \ldots, x_{(mn-1)}) = (a^0_0, \ldots, a^0_{n-1})$, then use the second propagation to initialize the preceding n processes $(x_{((m-2)n)}, \ldots, x_{((m-1)n-1)}) = (a^1_0, \ldots, a^1_{n-1})$, and repeat this pattern until the last propagation is used to initialize the first n processes $(x_0, \ldots, x_{n-1}) = (a^{m-1}_0, \ldots, a^{m-1}_{n-1})$.

x_0	x_1	$\dots$	x_{n-1}	$\dots\dots$	$x_{(m-2)n}$		$\dots$		$x_{(m-1)n}$		$\dots$	
a_0^{m-1}	a_1^{m-1}	$\dots$	a_{n-1}^{m-1}	$\dots\dots$	a_0^1	a_1^1	$\dots$	a_{n-1}^1	a_0^0	a_1^0	$\dots$	a_{n-1}^0
a_0^0	a_1^{m-1}	$\dots$	a_{n-1}^{m-1}	$\dots\dots$	a_0^2	a_1^1	$\dots$	a_{n-1}^1	a_0^1	a_1^0	$\dots$	a_{n-1}^0
a_0^0	a_1^0	$\dots$	a_{n-1}^{m-1}	$\dots\dots$	a_0^2	a_1^2	$\dots$	a_{n-1}^1	a_0^1	a_1^1	$\dots$	a_{n-1}^0
a_0^0	a_1^0	$\dots$	a_{n-1}^0	$\dots\dots$	a_0^2	a_1^2	$\dots$	a_{n-1}^2	a_0^1	a_1^1	$\dots$	a_{n-1}^1

Figure 3: A livelock on $m \cdot n$ processes using m propagations of period n that lead each other circularly.

In the initial state, every process whose index is a multiple of n is enabled; all other processes are disabled because their predecessors have the stable value in their corresponding propagation. Each such process $P_{((m-i)n)}$ has $x_{((m-i)n-1)} = a^i_{n-1}$ and $x_{((m-i)n)} = a^{i-1}_0$, and it is enabled to assign $x_{((m-i)n)} := a^i_0$ since actions have form $(a^i_{j-1}, a^{i-1}_j, a^i_j)$. These positions are highlighted in the first row of Figure 3, and the second row shows the result of all enabled processes acting. Let these m processes act in any order without creating collisions. The system is now in a state where process P_1 is enabled along with every nth process after it. For each such process $P_{((m-i)n+1)}$, we know $x_{((m-i)n+1)} = a^{i-1}_1$ is true due to our choice of initial state. Since the previous actions have assigned $x_{((m-i)n)} := a^i_0$, each $P_{((m-i)n+1)}$ is enabled to assign $x_{((m-i)n+1)} := a^i_1$. After these m processes act, the system is in a state where process P_2 is enabled along with every nth process after it.

As shown in the last two rows of Figure 3, continuing this pattern will eventually enable P_{n-1} and every nth process after it to act, reaching a state that is a rotated version of our initial state. For each such process $P_{((m-i)n+(n-1))}$, we know $x_{((m-i)n+(n-1))} = a^{i-1}_{n-1}$ is true due to our choice of initial state. Since the previous actions have assigned $x_{((m-i)n+(n-2))} := a^i_{n-2}$, each $P_{((m-i)n+(n-1))}$ is enabled to assign $x_{((m-i)n+(n-1))} := a^i_{n-1}$. After mn total steps, this execution has reached a state where each $x_{((m-i)n+j)} = a^i_j$. This state is a copy of initial state (where $x_{((m-1-i)n+j)} = a^i_j$) that is rotated by m positions. Thus, the execution can revisit the initial state by Lemma 3.3, and a livelock exists. □

Example 3.5. *Livelock freedom of the Sum-Not-2 protocol.*

Recall from Figure 1 that the sum-not-2 protocol consists of three parameterized actions $(0, 2, 1)$, $(1, 1, 2)$, and $(2, 0, 1)$. Every periodic propagation in this protocol alternates between actions $(2, 0, 1)$ and $(1, 1, 2)$. These propagations require x_i values to alternate between 0 and 1 for each subsequent i. However, these propagations assign x_i values alternating between 1 and 2 for each subsequent i. Clearly no periodic propagation can execute through a ring segment of alternating 1 and 2 values, therefore no propagation leads another in this protocol. For any ring size, an infinite execution requires that actions propagate around the ring. This is not possible since no propagation leads another, therefore the protocol is livelock-free for all ring sizes.

We form the same argument in terms of walks in the protocol's graph. Every closed walk in the graph alternates between visiting node 1 and node 2 indefinitely. No closed walk exists that alternates between visiting arcs labeled 1 and 2, therefore no periodic propagation leads another in the this protocol. As such, no livelock exists (based on Lemma 3.4).

Example 3.6. *Sum-Not-Odd has a livelock for some ring size.*

We previously demonstrated that there are $m = 4$ periodic propagations with the period $n = 4$ in the *Sum-Not-Odd* protocol. These propagations act on the following values: $(1, 3, 3, 1), (2, 0, 2, 4), (3, 1, 1, 3), (2, 4, 2, 0)$ in a circular fashion. Using the technique presented in the proof of Lemma 3.4, we construct a ring of size $m \times n = 16$ and demonstrate that this ring has a livelock. Each row in the following matrix represents a global state of the ring, starting from the initial state in the first row. Moreover, the underlined values denote enabled processes, and each enabled process takes an action based on the action graph of Figure 2. Notice that after 16 rounds this computation gets back to the starting state and none of the rows represents a legitimate state.

$\underline{2}$	4	2	0	$\underline{3}$	1	1	3	$\underline{2}$	0	2	4	$\underline{1}$	3	3	1
1	$\underline{4}$	2	0	2	$\underline{1}$	1	3	3	$\underline{0}$	2	4	2	$\underline{3}$	3	1
1	3	$\underline{2}$	0	2	4	$\underline{1}$	3	3	1	$\underline{2}$	4	2	0	$\underline{3}$	1
1	3	3	$\underline{0}$	2	4	2	$\underline{3}$	3	1	1	$\underline{4}$	2	0	2	$\underline{1}$
$\underline{1}$	3	3	1	$\underline{2}$	4	2	0	$\underline{3}$	1	1	3	$\underline{2}$	0	2	4
2	$\underline{3}$	3	1	1	$\underline{4}$	2	0	2	$\underline{1}$	1	3	3	$\underline{0}$	2	4
2	0	$\underline{3}$	1	1	3	$\underline{2}$	0	2	4	$\underline{1}$	3	3	1	$\underline{2}$	4
2	0	2	$\underline{1}$	1	3	3	$\underline{0}$	2	4	2	$\underline{3}$	3	1	1	$\underline{4}$
$\underline{2}$	0	2	4	$\underline{1}$	3	3	1	$\underline{2}$	4	2	0	$\underline{3}$	1	1	3
3	$\underline{0}$	2	4	2	$\underline{3}$	3	1	1	$\underline{4}$	2	0	2	$\underline{1}$	1	3
3	1	$\underline{2}$	4	2	0	$\underline{3}$	1	1	3	$\underline{2}$	0	2	4	$\underline{1}$	3
3	1	1	$\underline{4}$	2	0	2	$\underline{1}$	1	3	3	$\underline{0}$	2	4	2	$\underline{3}$
$\underline{3}$	1	1	3	$\underline{2}$	0	2	4	$\underline{1}$	3	3	1	$\underline{2}$	4	2	0
2	$\underline{1}$	1	3	3	$\underline{0}$	2	4	2	$\underline{3}$	3	1	1	$\underline{4}$	2	0
2	4	$\underline{1}$	3	3	1	$\underline{2}$	4	2	0	$\underline{3}$	1	1	3	$\underline{2}$	0
2	4	2	$\underline{3}$	3	1	1	$\underline{4}$	2	0	2	$\underline{1}$	1	3	3	$\underline{0}$
$\underline{2}$	4	2	0	$\underline{3}$	1	1	3	$\underline{2}$	0	2	4	$\underline{1}$	3	3	1

Corollary 3.7. *Lemma 3.4 holds under both synchronous and asynchronous execution semantics.*

Proof. In Figure 3, even if the transitions from one row to another are synchronous, the proof of Lemma 3.4 would still hold. Moreover, under asynchronous executions, there are collision-free interleavings that can get us from each row to its successor row. It is straightforward to verify this claim in the context of the *Sum-Not-Odd* example and Figure 3. Moreover, a formal proof of this claim appears in our previous work [24]. □

4 Tiling

With our new characterization of livelocks in a unidirectional ring protocol from Lemma 3.4, we can explore the difficulty of livelock detection. We use the notion of action graphs as an intuitive bridge between problems. To complete the bridge, we re-state a well-studied undecidable problem, the domino problem, and reduce a variant of this problem to the problem of livelock detection.

4.1 The Domino Problem and Its Periodic Variant

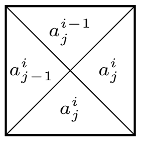
Problem 4.1 (The Domino Problem)**.**

- **Input:** A set of square tiles with a color (label) on each edge. All tiles are the same size.
- **Question:** Can copies of these tiles cover the infinite plane by placing them side-by-side, without changing tile orientations, such that edge colors match where tiles meet? In other words, can the following be satisfied for each tile $T[i,j]$ at row i and column j on the plane?

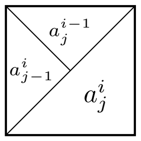
$$(T[i{-}1,j].S{=}T[i,j].N) \wedge (T[i,j{-}1].E{=}T[i,j].W)$$

  where $T[i,j].N$ is the color on the north edge of tile $T[i,j]$. Similarly, the $.S$, $.W$, and $.E$ suffixes refer to south, west, and east edge colors of their respective tiles.

The domino problem was introduced by Wang [67], and the square tiles are commonly referred to as Wang tiles. Berger showed the problem to be undecidable [5]. Specifically, the problem is co-semi-decidable, also written as Π^0_1-complete using the arithmetical hierarchy notation of Rogers [58]. A tile set is *NW-deterministic* when each tile in the set can be identified uniquely by its north and west edge colors. In this



(a) Wang Tile



(b) SE-Identical Tile

Figure 4: Tile for action $(a_{j-1}^i, a_j^{i-1}, a_j^i)$.

case, if a tile meets another at its southwest (resp. northeast) corner, then the tile to its south (resp east) side is uniquely determined. Kari proved that the domino problem remains undecidable for NW-deterministic tile sets [40].

Problem 4.2 (The Periodic Domino Problem)**.** This domino problem asks whether an infinite plane can be covered by placing copies of a fixed rectangular arrangement of tiles side-by-side such that a *repeating pattern* forms. In other words, can Problem 4.1 be solved such that there exist m and n where the following is satisfied for each tile $T[i, j]$ on the plane?

$$(T[i, j] = T[i + m, j]) \wedge (T[i, j] = T[i, j + n])$$

Problem 4.2 is equivalent to asking whether a torus can be completely covered using the same tiling rules. Gurevich and Koriakov [34] give a semi-algorithm that terminates if the given tile set can periodically tile the plane or cannot tile the plane at all, but it does not halt when the plane can only be tiled aperiodically. It follows that this problem is semi-decidable, also written as Σ_1^0-complete using notation of Rogers [58].

Action tiles. A tile is *SE-identical* when it has identical south and east edge colors. For such sets, we refer to the south and east edge colors of a tile $T[i, j]$ as $T[i, j].SE$. We write (a, b, c) to denote such a tile with colors a, b, c, and c on its west, north, east, and south edges respectively. A set of SE-identical tiles is *W-disabling* when no two tiles that have the same west color have matching north and south colors respectively. In other words, a SE-identical tile set is W-disabling if and only if for every tile (a, b, c) in the set, no color d exists such that the tile (a, c, d) is also in the set. We use the term *action tile* strictly to denote tiles in a SE-identical W-disabling tile set and *action tile set* to denote the set itself.

4.2 Reduction to Livelock Detection

The triples that we use to represent tiles in an action tile set are subject to the same constraints as actions in a unidirectional ring protocol of symmetric, self-disabling processes. That is, the W-disabling constraint for tiles is equivalent to the self-disabling constraint for actions. As such, we have a bijection between these kinds of tile sets and protocols.

Lemma 4.3. *There is a bijective function that maps an action tile set to a unidirectional ring protocol of self-disabling processes such that the tile set admits a periodic tiling if and only if the protocol contains a livelock. The mapping preserves determinism (resp. NW-determinism) in the protocol (resp. tile set).*

Proof. Given a set of triples such that no two triples (a, b, c) and (a, c, d) coexist, the set can represent an action tile set or the actions of self-disabling processes in a unidirectional ring. Likewise, if the set of triples also ensures that no two triples (a, b, c) and (a, b, d) coexist (where $c \neq d$), then the corresponding tile set is NW-deterministic and the corresponding protocol is deterministic. Thus, our mapping function (the identity) is bijective and preserves determinism.

We are left to show that the conditions that a set of triples must meet in order to form a periodic tiling are the same conditions that must be met to form a livelock. Recall that a livelock can be characterized by a list of m periodic propagations of length n where each propagation leads the next one in the list (and the last leads the first). From Lemma 3.2, we know that this is equivalent to finding a_j^i values, with indices ranging over $i \in \mathbb{Z}_m$ and $j \in \mathbb{Z}_n$, such that each triple $(a_{j-1}^i, a_j^{i-1}, a_j^i)$ forms an action in the protocol

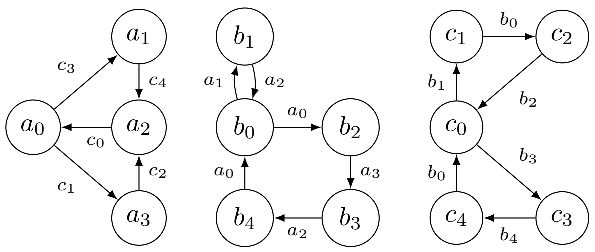


Figure 5: "A-b-C" protocol graph.

(with indices computed modulo m and n). Southeast tile colors in a periodic tiling T behave the same as a^i_j since each triple $(T[i,j-1].SE, T[i-1,j].SE, T[i,j].SE)$ is a tile in the set with the same values as $T[i,j] \equiv (T[i,j].W, T[i,j].N, T[i,j].SE)$. As such, satisfying the constraints on a^i_j is equivalent to solving the periodic domino problem, where each action $(a^i_{j-1}, a^{i-1}_j, a^i_j)$ must exist as a tile in the action tile set as shown in Figure 4. □

Lemma 4.4. *An action tile set admits a periodic tiling* iff *in the protocol obtained by Lemma 4.3 there are some m propagations with some period n, where the $(i-1)$-th propagation leads the i-th propagation for each index i modulo m. (Proof follows from Lemmas 3.4 and 4.3.)*

Example 4.5. *Fictional "A-b-C" protocol with a livelock.*

Figure 5 shows the action graph of our example unidirectional ring protocol where each arc corresponds to an action. Note that the labels $a_0, \ldots, c_4$ are constants that could equivalently be changed to numbers $0, \ldots, 13$. The protocol has an interesting livelock. First, propagations that characterize the livelock do not correspond to simple cycles in the action graph. Second, 3 of these 6 propagations correspond to walks that are unique regardless of the starting node.

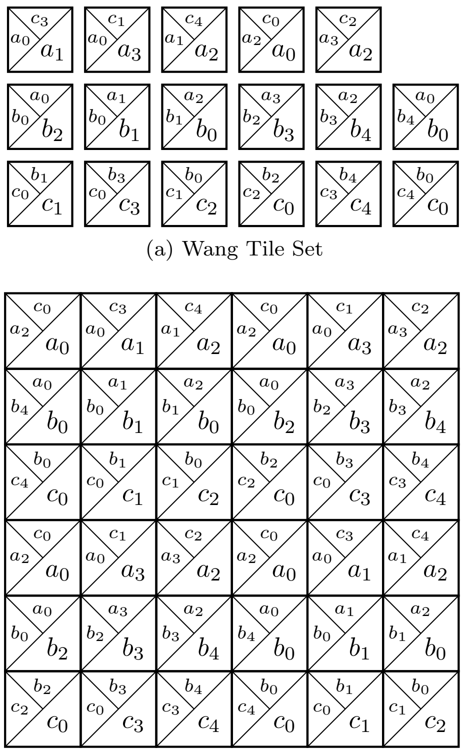
A livelock can be found by taking a walk through the graph. We start by choosing a closed walk starting with node c_0 and visiting nodes c_3, c_4, c_0, c_1, c_2, and c_0 without considering which arcs were taken.

1. Using the previous nodes as arc labels c_0, c_3, c_4, c_0, c_1, and c_2, start from node a_2 to form a closed walk visiting nodes a_0, a_1, a_2, a_0, a_3, and a_2. This corresponds to the periodic propagation: $\langle(a_2,c_0,a_0),(a_0,c_3,a_1),(a_1,c_4,a_2),(a_2,c_0,a_0),(a_0,c_1,a_3),(a_3,c_2,a_2)\rangle$
2. Using the previous nodes as arc labels a_0, a_1, a_2, a_0, a_3, and a_2, start from node b_4 to form a closed walk visiting nodes b_0, b_1, b_0, b_2, b_3, and b_4. This corresponds to the periodic propagation: $\langle(b_4,a_0,b_0),(b_0,a_1,b_1),(b_1,a_2,b_0),(b_0,a_0,b_2),(b_2,a_3,b_3),(b_3,a_2,b_4)\rangle$
3. Using the previous nodes as arc labels b_0, b_1, b_0, b_2, b_3, and b_4, start from node c_4 to form a closed walk visiting nodes c_0, c_1, c_2, c_0, c_3, and c_4. This corresponds to the periodic propagation: $\langle(c_4,b_0,c_0),(c_0,b_1,c_1),(c_1,b_0,c_2),(c_2,b_2,c_0),(c_0,b_3,c_3),(c_3,b_4,c_4)\rangle$
4. Use previous nodes as arc labels to form a closed walk through nodes a_2, a_0, a_3, a_2, a_0, a_1, and a_2.
5. Use previous nodes as arc labels to form a closed walk through nodes b_0, b_2, b_3, b_4, b_0, b_1, and b_0.
6. Use previous nodes as arc labels to form a closed walk through nodes c_2, c_0, c_3, c_4, c_0, c_1, and c_2. We started with this same sequence of nodes, therefore we are done and have found the last periodic propagation to be: $\langle(c_2,b_2,c_0),(c_0,b_3,c_3),(c_3,b_4,c_4),(c_4,b_0,c_0),(c_0,b_1,c_1),(c_1,b_0,c_2)\rangle$

Each of the 6 propagations is compactly illustrated as a row in Figure 6b, which is a periodic block formed by action tiles in Figure 6a. Since copies of the periodic block can be placed beside themselves to tile the infinite plane, this is a solution to the periodic domino problem.

Lemma 4.6. *A periodic tiling of action tiles exists* iff *the protocol obtained by Lemma 4.3 has a synchronous livelock with all processes enabled for some ring size.*

Proof. Synchronous semantics ensure that every enabled process acts at every step, which is not allowed under the interleaving semantics. For example, consider the protocol of Example 4.5 on a ring of size 6



(a) Wang Tile Set

(b) Periodic Block

Figure 6: Instance and solution for the periodic domino problem that corresponds to finding a livelock in the "A-b-C" protocol.

beginning at state $(c_0, b_3, a_2, c_0, b_3, a_2)$. This state is constructed from SE colors of tiles along the antidiagonal of the periodic block in Figure 6b. In this state, every process is enabled to act, and they all will act under synchronous semantics, making the subsequent state $(a_0, c_3, b_4, a_0, c_3, b_4)$. In the figure, these values correspond to the SE tile colors that are one tile below the antidiagonal. The execution can continue with $(b_0, a_1, c_4, b_0, a_1, c_4)$, and will eventually reach the initial state after 6 total steps. This livelock is no coincidence, but rather can always be constructed from the antidiagonal of a square periodic block of action tiles. Indeed, it is by definition that every two consecutive values along the antidiagonal (or shifted version) will form some action $(T[i, j-1].SE, T[i-1, j].SE, \mathtt{X})$ where $\mathtt{X}$ is a wildcard value. Furthermore, we can always construct a square $N \times N$ periodic block (for some N) by appropriately placing copies of a non-square block beside itself. We can of course continue this pattern to form a $kN \times kN$ periodic block for any integer $k > 0$. Any periodic block of action tiles therefore implies a synchronous livelock where all processes are enabled on some ring of size N, and likewise for every ring of size kN.

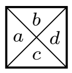
Given a synchronous livelock where all processes are enabled, it is also straightforward to see that a periodic block of action tiles exists. Let the livelock occur on a ring of size n. Since the system is finite-state, a synchronous livelock will have some state C that it can revisit after some m steps. We would like m and n to be the same, therefore consider a ring of size $N = mn$ instead whose initial state is created by repeating C as $C = (x_0, \ldots, x_{n-1}) = (x_n, \ldots, x_{2n-1}) = \cdots = (x_{N-n}, \ldots, x_{N-1})$. The system will still revisit this state after m steps, but it can also be said to revisit the state after N steps. Thus, the initial state of N values can be used to form a tiling by using the initial x_j values as the SE colors of the antidiagonal in a periodic block. Each tile $T[-j, j]$ below the antidiagonal can now have its north and west colors filled in to match the antidiagonal's SE colors. That is, each north and west color of tile $T[-j, j]$ are the initial values of x_{j-1} and x_j respectively. Therefore, the *SE* color of $T[-j, j]$ is the value of x_j after one step, and $T[-j, j]$ indeed represents an action from the tile set. Repeating this for all N steps, we fill a periodic block. □

Lemma 4.7. *A unidirectional ring protocol has a synchronous livelock with all processes enabled for some ring size* iff *there are some m propagations with some period n, where the $(i-1)$-th propagation leads the i-th propagation for each index i modulo m. (Proof follows from Lemmas 4.6 and 4.4.)*

4.3 Equivalent Tile Sets

The remainder of this section shows how to transform a NW-deterministic Wang tile set into a NW-deterministic action tile set that is equivalent with respect to the domino problems. This gives us the tools to reduce the periodic domino problem to livelock detection in the next section which proves that livelock detection is undecidable for unidirectional ring protocols of symmetric, deterministic, self-disabling processes.

Lemma 4.8. *For any set of Wang tiles, a W-disabling set of SE-identical tiles (i.e., an action tile set) exists that gives the same result to Problem 4.1 and Problem 4.2 and preserves NW-determinism.*

Proof. Let (a, b, c, d) denote a Wang tile 

by listing its edge colors in order of W, N, S, and E. From any Wang tile set, our new action tile set has colors: $a_{\rightarrow}$ and $a_{\uparrow}$ for every color a in the original tile set, $abcd$ for every tile (a, b, c, d) in the original set, and a new color \$. The new set has tiles $(a_{\rightarrow}, b_{\uparrow}, abcd)$, $(\$, abcd, c_{\uparrow})$, $(abcd, \$, d_{\rightarrow})$, and $(c_{\uparrow}, d_{\rightarrow}, \$)$ for each tile (a, b, c, d) in the original set. Figure Figure 7 illustrates this reduction.
Tiling correspondence. Observe that if a tile with a color of the form $abcd$ is placed on the plane, we can determine three other tiles that must be placed near it to form the 2×2 arrangement shown in Figure 7. Two of these are determined since the color $abcd$ appears on exactly three tiles in the set (for the W, N, and SE edges). The third tile $(c_{\uparrow}, d_{\rightarrow}, \$)$ is determined since any tile with $\mathtt{X}_{\uparrow}$ on its W edge or $\mathtt{X}_{\rightarrow}$ on its N edge has a SE edge color of \$. Conversely, if a tile of the form $(c_{\uparrow}, d_{\rightarrow}, \$)$ is placed on the plane, its west neighbor must have the form $(\$, abcd, c_{\uparrow})$ for some a and b corresponding to the original set of colors. After knowing these a and b, the two tiles to the north are determined due to the reasoning in the previous paragraph.

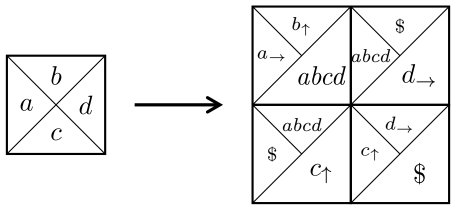


Figure 7: Transform 1 Wang tile to 4 action tiles.

Thus, any valid tiling T' using the new tile set consists of 2×2 blocks corresponding to the tiles in the original set. Further, since the \$ colors must match across these 2×2 blocks, the blocks must be aligned.

For correspondence, it remains to show that a valid tiling T exists using the original tile set if and only if a valid tiling T' exists using the new set. This is easy to see since two tiles (a, b, c, d) and (w, x, y, z) in the original set can border each other if and only if their corresponding 2×2 blocks in the new set can border each other.

The new tile set is a W-disabling set of SE-identical tiles. All tiles in the new set are obviously SE-identical, therefore we only need to show that the set is W-disabling. That is, for every tile (a, b, c) in the new set, there does not exist another tile (a, c, e) in the set for any e. Observe in Figure 7 that the action tiles each use 3 distinct forms of colors: $\mathtt{X}_\rightarrow$, $\mathtt{X}_\uparrow$, $\mathtt{XXXX}$, and \$. Furthermore, the form of the west color determines the forms of the other two colors. Thus, for any two action tiles with e the same west color, we know that their north and southeast colors will not match. The tile set is therefore W-disabling.

The new tile set preserves NW-determinism. Recall that a tile set is NW-deterministic when for every tile (a, b, c, d), there does not exist another tile (a, b, e, f) in the set for any $e \neq c$ and $d \neq f$. If this is the case in the original set, then any tile in the new set with a west color of $a_\rightarrow$ and a north color of $b_\uparrow$ for any a and b has a uniquely determined southeast color $abcd$.

Each other tile in the new set (those with \$ on some edge) can be uniquely identified by its west or north color. For any $abcd$, a tile whose west color is $abcd$ uniquely has the form $(abcd, \$, d_\rightarrow)$. Similarly, a tile whose north color is $abcd$ uniquely has the form $(\$, abcd, c_\uparrow)$. Lastly, for any c, a tile whose west color is $c_\uparrow$ uniquely has the form $(c_\uparrow, d_\rightarrow, \$)$. This covers all forms of tiles in the new set, therefore the new set preserves NW-determinism. □

5 Unecidability of Verification

This section begins with a theorem of undecidability as a direct consequence of our mappings between livelock detection and the periodic domino problem. Section 5.1 presents a corollary about the undecidability of verifying stabilization. Section 5.2 extends these undecidability results to other execution semantics and fairness assumptions.

Processes are symmetric, constant-space, deterministic, and self-disabling. For brevity, we avoid restating our usual restrictions on processes within this section. While these restrictions strengthen our undecidability results, they are also used at every step of reasoning.

Theorem 5.1 (Σ_1^0-completeness of livelock detection for unidirectional rings)**.** *Livelock detection for unidirectional ring protocols is undecidable under no fairness and interleaving semantics (Σ_1^0-complete).*

Proof. By Lemma 4.8 and Lemma 4.3, we can construct such a protocol from any NW-deterministic Wang tile set such that the protocol has a livelock if and only if the set can form a periodic tiling. Since this is a mapping reduction from tile sets to protocols, livelock detection is at least as hard as the periodic domino problem. Livelock detection is also no harder than the periodic domino problem since Lemma 4.3 gives a mapping reduction in the opposite direction. Therefore, like the periodic domino problem, livelock detection is Σ_1^0-complete. □

Corollary 5.2 (Σ_1^0-completeness of livelock detection for bidirectional rings). *Livelock detection for bidirectional rings of deterministic, constant-space and self-disabling processes is undecidable under no fairness and interleaving semantics.*

Proof. A unidirectional ring is a special case of a bidirectional ring since a bidirectional ring can behave like a unidirectional ring. □

Corollary 5.3 (Σ_1^0-completeness of livelock detection for cyclic topologies). *Let p be a parameterized protocol consisting of deterministic, constant-space and self-disabling processes. If the underlying topology graph of p includes a ring, then detecting livelocks of p is undecidable under no fairness and interleaving semantics.*

Proof. Similar to the proof of Corollary 5.2, unidirectional ring is a special case of protocols with a cyclic topology graph. □

Theorem 5.4 (Undecidability under synchronous semantics for rings). *Livelock detection for both unidirectional and bidirectional ring protocols is undecidable under synchronous semantics.*

Proof. Proof follows from Lemmas 3.4, 4.6 and 4.7, and Corollary 5.2. □

Corollary 5.5 (Undecidability under synchronous semantics for cyclic topologies). *Let p be a parameterized protocol consisting of deterministic, constant-space and self-disabling processes. If the underlying topology graph of p includes a ring, then detecting livelocks of p is undecidable under synchronous semantics. (Proof straightforward; hence omitted.)*

5.1 Verifying Stabilization

As the complementary problem of livelock detection, verifying livelock freedom is Π_1^0-complete. This livelock freedom problem can be posed as a problem of verifying *silent* stabilization (see Section 2), which is a special case of self-stabilization. Closure and deadlock freedom are satisfied by design, therefore verifying stabilization is Π_1^0-hard. We first present a lemma that shows Π_1^0 membership. Then, we prove Π_1^0-completetess.

Lemma 5.6 (Membership in Σ_1^0 and Π_1^0). *Livelock detection is in Σ_1^0 for protocols whose valid topologies and states can be enumerated and whose executions can only reach a finite number of states. Likewise, verifying livelock freedom and stabilization are in Π_1^0.*

Proof. In concrete terms, unidirectional rings can be enumerated by ring size. Each ring of fixed size has a finite number of states, therefore the states can be enumerated and any execution will only reach a finite number of states. For each ring size, the detection of livelocks, deadlocks, and closure violations (from any state) is decidable since only a finite number of states are reachable. Furthermore, we can enumerate (ith topology, jth state) pairs in order of increasing $i + j$, which is the natural way to enumerate $\mathbb{Z}^+ \times \mathbb{Z}^+$. As such, we can write a semi-algorithm for detecting livelocks or non-stabilization that checks every state of every valid topology in order of enumeration, halting when it detects invalid protocol behavior. Thus, detecting livelocks or non-stabilization is in Σ_1^0, and its complement problem is in Π_1^0. □

Theorem 5.7 (Π_1^0-completeness of verifying stabilization for unidirectional rings). *Verifying stabilization for unidirectional ring protocols is undecidable (Π_1^0-complete).*

Proof. Verifying stabilization requires livelock-freedom. Theorem 5.1 shows the Σ_1^0-hardness of the complement problem; hence Π_1^0-hardness of livelock-freedom. The membership of Π_1^0 follows from Lemma 5.6; hence Π_1^0-completeness. □

Corollary 5.8 (Π_1^0-completeness of verifying stabilization for cyclic topologies). *Let p be a parameterized protocol consisting of deterministic, constant-space and self-disabling processes. If the underlying topology graph of p includes a ring, then verifying self-stabilization of p is undecidable (Π_1^0-complete) under no fairness, interleaving and synchronous semantics.*

Proof. Proof follows from Corollaries 5.3 and 5.5, and Theorem 5.7. □

5.2 Effects of Consistency Model and Scheduling Policy

In Lemma 4.7, we observed that livelock existence is preserved in the extreme case where all processes are enabled and act synchronously. Synchronous execution can be viewed as a result of a relaxed memory consistency model or as a very strict kind of process scheduling. We extend the idea of livelock equivalence to other consistency models that allow communication delay between processes. However, proving the undecidability of livelock detection in uni-rings under strong fairness cannot be performed using the technique we use in the proof of Lemma 4.7 because the proof of Lemma 4.7 is based on a correspondence between the existence of periodic tiling in action tiles sets versus the existence of synchronous livelocks in uni-rings (Lemmas 4.4 and 4.6). Strong fairness is relevant only if we are concerned with the interleaving/asynchronous semantics. Instead, we prove the undecidability of livelock detection under any scheduling policy by designing a reduction to uni-ring protocols where there is exactly one process that is enabled to execute at any moment, called *deterministic livelocks*. First, we show that detecting deterministic livelocks is also undecidable. Then, we prove that the detection of deterministic livelocks (Lemma 5.13) is a precondition for livelock detection under any scheduling policy, which results in the undecidability of detecting livelocks in uni-rings under any scheduling policy. This is a counterintuitive result because the common understanding [30] is that assuming strong fairness ensures livelock-freedom.

Definition 5.9 (FIFO Consistency)**.** In the context of stabilization on unidirectional rings, the weakest consistency model allows arbitrarily delayed (but ordered) communication between processes.

FIFO consistency on interleaving and no fairness. *FIFO consistency* [51] adds arbitrary communication delay between processes while preserving order of communication. Using the common structure and logic of FIFO queues, we can define FIFO consistency as a simple extension to our usual interleaving semantics. In a uni-ring, consider an unbounded FIFO queue q_i between P_i and P_{i+1}, and give each process P_{i+1} access to a variable x'_i instead of x_i. Finally, connect each x_i and x'_i with q_i that has the following actions:

$$\begin{array}{rl}
\mathsf{Empty}(q_i) \land x_i \neq x'_i & \longrightarrow \mathsf{PushLast}(q_i, x_i); \\
\neg\mathsf{Empty}(q_i) \land x_i \neq \mathsf{Last}(q_i) & \longrightarrow \mathsf{PushLast}(q_i, x_i); \\
\neg\mathsf{Empty}(q_i) & \longrightarrow x'_i := \mathsf{PopFirst}(q_i);
\end{array}$$

While the second and the third actions introduce non-determinism in queue processes (when they are non-empty), this non-determinism is constrained in each queue q_i between P_i and P_{i+1}. Consider a scenario where P_i updates x_i and becomes disabled, and the queue q_i is non-empty. In this case, both the second and the third actions of q_i are enabled. First, notice that the second action is self-disabling; once a $\mathsf{PushLast}$ operation is done, the condition $x_i \neq \mathsf{Last}(q_i)$ becomes false and will not hold again until P_i updates x_i again. Updating x_i can only occur when P_i is enabled again by P_{i-1}, which requires P_{i+1} to propagate the enabledness by the third action of q_i. Before the second action executes, the third action may execute several times depending on the number of values in the queue. Due to non-determinism between the second and the third actions, the second action may execute anytime as long as the queue is non-empty. However, no matter when the second action executes, the set of states we can reach will not change because the second action just adds a new value to the end of the queue and gets disabled. As such, the non-determinism between the second and the third actions will not have a global impact.

Spectrum of consistency models. The strongest consistency model is *strict consistency*, where processes see updates from each other instantly. At the other extreme, we consider FIFO consistency to be the weakest consistency model in the context of stabilization on unidirectional rings. Our eventual goal is to show that the choice of consistency model does not impact whether a particular unidirectional ring protocol is stabilizing for all ring sizes. In the taxonomy of shared memory consistency models given by Steinke and Nutt [63], the local consistency and slow consistency models are weaker than FIFO consistency. These traditionally weaker models become equivalent to FIFO consistency by two implicit assumptions, that (1) each P_i reads exactly one variable of P_{i-1}, and (2) each P_i sees updated values from P_{i-1} in order. Without the first assumption, *slow consistency* would allow changes to different variables be seen at different times, which could introduce new livelocks. Given our choice of a unidirectional ring, a process can always restrict itself to owning a single variable because multiple variables can be encoded by a single one. This would not be feasible in a more general context where multiple processes can write to the same variables. Without the second assumption, *local consistency* would allow changes to a single variable to be seen out of order, which could introduce new livelocks. We forbid message reordering since stabilization assumes that faults are transient and eventually stop occurring.

Simulating strict consistency and synchrony. Since strict consistency is the strongest consistency model, an execution under strict consistency can be simulated under other consistency models. For example, an action of any process P_i under strict consistency can be simulated under FIFO consistency in 3 steps from a state with empty queues: (1) P_i acts, (2) q_i performs PushLast to copy the new x_i value, and (3) q_i performs PopFirst to assign the new value to x'_i.

Lemma 5.10. *If a unidirectional ring protocol p has a livelock under interleaving and no fairness, then p has a livelock under FIFO consistency and no fairness as well.*

Proof. Proof follows from the fact that FIFO consistency model can simulate strict consistency and interleaving semantics, and exhibit a livelock. □

FIFO consistency can also simulate synchronous actions under any consistency model. For example, the synchronous actions of k processes under strict consistency can be simulated under FIFO consistency in $3k$ steps from a state with empty queues: (1) k processes act, (2) the k queues of those processes perform PushLast to copy the new x values, and (3) the queues perform PopFirst to assign the new x' values.

Propagation count. Until now, we have considered the number of propagations in a state to be the number of enabled processes. This formula changes when delays are introduced. That is, the propagation count is computed as the number of pending actions of processes and queues, which is the sum of: (1) number of enabled processes, (2) number of queues enabled to call PushLast, and (3) number of values in queues.

Lemma 5.11 (Limited Propagations)**.** *Given N processes in a unidirectional ring executing under any scheduler and consistency model, the number of propagations cannot increase and the number of reachable states is therefore finite for a ring of size N.*

Proof. In Section 3, we noticed that the number of propagations (i.e., enabled processes under strict consistency) cannot increase during an execution when processes are self-disabling. This result is preserved for FIFO consistency and therefore all other consistency models. To see this, consider the 3 types of actions that can occur: (1) the action of a process P_i, (2) a queue q_i calling PushLast, and (3) a queue q_i calling PopFirst. In the first scenario, P_i will disable itself but may enable q_i. In the second scenario, q_i will disable itself from calling PushLast but will increase its number of stored values. In the third scenario, q_i will reduce its number of stored values but may enable P_{i+1}. Thus, each of the 3 cases will either decrease the number of propagations or keep it constant. Given that the number of propagations cannot increase, we have an upper bound on the number of values in queues for any execution, meaning that the number of reachable states is finite. □

Lemma 5.12. *A unidirectional ring protocol has a livelock under interleaving and no fairness iff it also has a livelock under FIFO consistency with unbounded queues in any state. The livelocks may appear for different ring sizes.*

Proof. The proof of left-to-right follows from Lemma 5.10. Thus, we are left to show that a livelock of p under interleaving semantics is implied by a livelock of p under FIFO consistency (allowing unbounded queues in any state). Assume protocol p has a livelock under FIFO consistency for some ring size. By Lemma 5.11, the number of propagations cannot increase during an execution but obviously must eventually stop decreasing, therefore some fixed m propagations eventually exists in an infinite execution.

We are left to show that these m propagations respect the properties given in Section 3 and that they are sufficient to form an interleaved livelock. Indeed, if a process P_k performs an action (a_{j-1}, b_j, a_j), then a_j is pushed onto q_i, and eventually we have $x'_k = a_j$, allowing P_{k+1} to perform an action (a_j, b_{j+1}, a_{j+1}). Thus, the propagations match the definition given in Section 3. Furthermore, the action (a_{j-1}, b_j, a_j) must lead the next action (d_{j-1}, e_j, d_j) that P_k takes, making $e_j = a_j$. We therefore have m propagations that lead each other. Using the same proof idea as in Lemma 3.4, we can show that the propagations are periodic. Only a finite number of states exist with m propagations by Lemma 5.11, therefore the system must revisit some state C by Lemma 3.3. Consider a fixed execution from C to itself that involves n process actions. After revisiting C a total of m times using this fixed execution, each propagation is located exactly where it was initially. Since each of the m propagations has reached the same position in the same state after nm total process actions, they can repeat with period n. Finally by Lemma 3.4, m propagations of period n lead each other, implying that protocol p has a livelock under interleaving semantics. □

Lemma 5.13 (Deterministic Livelock)**.** *If a unidirectional ring exhibits a livelock where exactly one enabled action exists, then a livelock exists under every scheduler and consistency model.*

Proof. If such a *deterministic livelock* exists, there is only one choice for each subsequent state in the execution because only one process or queue is enabled. Thus, the livelock is not affected by the scheduler choice. Furthermore, adding or removing delay between processes simply changes whether a process enables its successor in the ring immediately. Thus, the livelock is not affected by the consistency model. □

Lemma 5.14. *There is a ring protocol that has a livelock for some ring size N iff there is a ring protocol that has a deterministic livelock for the ring size N.*

Proof. We show that there is a function that maps any unidirectional ring protocol p to another such protocol p' such that if p has a synchronous livelock with all processes enabled, then p' has a deterministic livelock for the same ring size. Otherwise, both protocols are livelock-free. Without loss of generality, we can assume each process P_i acting under p has a single variable x_i. Let each P_i have transition function ξ, and recall from Definition 2.1 that we can define all actions of P_i as:

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

In the new protocol p', give each process P_i variables x_i and x'_i and give it read access to x'_{i-1}. The new protocol p' is defined by giving each process P_i the action:

$$(x'_{i-1}, x_i) \in \mathsf{Pre}(\xi) \longrightarrow x'_i := \begin{cases} \xi(x'_{i-1}, x_i); & \text{if } (x'_i = x_i) \\ x_i; & \text{otherwise} \end{cases}$$
$$x_i := \xi(x'_{i-1}, x_i);$$

⇐: Notice that in a livelock, p' performs the p protocol using x'_{i-1} instead of x_{i-1}, and x'_{i-1} is eventually updated to x_{i-1}. Therefore we can safely say that if p does not have a livelock under FIFO consistency, then p' does not have a livelock. By Lemma 5.12 and Lemma 4.7, a livelock exists under FIFO consistency *iff* a synchronous livelock exists with all processes enabled. Thus, livelock freedom of p implies livelock freedom of p'.

⇒: We are left to show that if p has a livelock, then p' has a deterministic livelock. Assume p has a livelock where all processes are enabled. By Lemma 3.3, an execution exists that revisits some state $C = (c_0, \ldots, c_{N-1})$. Let C' be a state of a system executing p' such that $x_i = c_i$ for all i and $x'_0 = c_0$. Our goal is to make only P_1 enabled, therefore for all $i \neq 1$, choose x'_{i-1} such that $(x'_{i-1}, c_i) \notin \mathsf{Pre}(\xi)$. Finding such values is of course possible because processes are self-disabling. Now only P_1 is enabled in C'. When P_1 acts, it assigns x'_1 as c_1 and assigns x_1 as $\xi(c_0, c_1)$.

Now only P_2 is enabled, and it assigns x'_2 as c_2 and assigns x_2 as $\xi(c_1, c_2)$. At the Nth step, P_0 acts to assign both x'_0 and x_0 as $\xi(c_{N-1}, c_0)$. In this state, we have $x_i = \xi(c_{i-1}, c_i)$ for all i, $x'_0 = x_0$, and $x_i = c_i$ for all $i > 0$. On the x_i values, this is the same state that would be visited after one synchronous step of p from state C. Likewise, the x'_i values meet the same constraints as in C', where $x'_0 = x_0$ and the other x'_i values (where $i > 0$) make P_{i+1} disabled. Since $x'_0 = \xi(c_{N-1}, c_0)$ and $x_1 = \xi(c_0, c_1)$, process P_1 is once again enabled. Since P_1 is enabled again, this execution of p' can continue to use one propagation to simulate the synchronous livelock of p.

This is indeed a deterministic livelock since one process is enabled at all times, actions are deterministic and self-disabling in both p and p', and an action of P_i in the livelock only changes values read by P_{i+1}. We have therefore shown a mapping from p to p' such that p has a livelock if and only if p' has a deterministic livelocks. □

Theorem 5.15. *Livelock detection for unidirectional ring protocols is undecidable (Σ_1^0-complete) for any choice of scheduler and consistency model.*

Proof. Given such a protocol p, we can create a similar protocol p' using the method of Lemma 5.14. That is, if p has a livelock, then p' has a deterministic livelock and otherwise, both protocols are livelock-free. In the case of a p livelock, the deterministic p' livelock exists under every scheduler and consistency model by Lemma 5.13. In the case that p is livelock-free, no livelock exists in p' under FIFO consistency by Lemma 5.12, which means p' is livelock-free for every scheduler and consistency model because FIFO model is the weakest consistency model in the context of our work. A livelock in p has been shown to exist under interleaving semantics if and only if a livelock exists in p' under any particular choice of scheduler and consistency model. Thus, the Σ_1^0-hardness of livelock detection shown in Theorem 5.1 is preserved for any particular choice of scheduler and consistency model.

We can enumerate the valid topologies for a protocol p by number of processes, and we can enumerate states by their number of propagations. By Lemma 5.11, a state of p can only reach a finite number of other states under any scheduler and consistency model. Thus, the problem of livelock detection remains Σ_1^0-complete. □

Lemma 5.16. *If a unidirectional ring is stabilizing under FIFO consistency and no fairness, then it is stabilizing under every scheduler and consistency model.*

Proof. Consider a protocol p that is stabilizing to legitimate states where no process or queue action is enabled. Since FIFO consistency is the most relaxed model in the context of our work, it can simulate any scheduling policy and any consistency model. Such a simulation will not have an impact on the form of individual execution steps (be it process action or queue action). Thus, the choice of scheduler and consistency model will not break closure, nor will it create livelocks in illegitimate states. Likewise, no deadlocks will be created outside legitimate states because the choice of scheduler and consistency model will not disable any action outside the legitimate states. Therefore, the ring remains stabilizing under every scheduler and consistency model. □

Corollary 5.17. *Verifying stabilization for unidirectional ring protocols is undecidable (Π_1^0-complete) for any choice of scheduler and consistency model.*

Proof. Since we have considered livelock detection in the entire state space, even for arbitrarily filled queues in the case of FIFO consistency (Lemma 5.12), we can reduce the problem of verifying livelock freedom to that of verifying stabilization to states where all processes and queues are disabled. Using this set of legitimate states and the Σ_1^0-completeness result of Theorem 5.15, the proof of Theorem 5.7 applies for any particular scheduler and consistency model, showing that verifying stabilization is Π_1^0-complete. □

6 A Semi-Algorithm for Livelock Detection

In this section, we address the following question: *Given the action graph of a protocol, can we verify livelock freedom up to a finite scope*? We show that the answer to this question is affirmative. Specifically, we present a semi-algorithm that can detect the existence of livelocks in rings of specific sizes. We then scale up the scope of the search to larger periodic propagations (in terms of their period and the number of propagations). Our semi-algorithm is an implication of Lemma 3.4 that provides a different approach for verification of uni-rings where, instead of state space exploration, we search for periodic propagations that lead each other circularly. We show that this approach significantly outperforms state-of-the-art model checkers (see Section 7). Thus, we focus on the following problem:

Problem 6.1 (Find Periodic Propagations)**.**

- INPUT: A protocol p in terms of its actions or its action graph; positive integers $m, n > 1$; domain size M.

- OUTPUT: m propagations with period n that lead each other circularly. An empty set, if no such propagations exists.

In the context of action graphs, Problem 6.1 poses the following question: *Are there m closed walks with length n that lead each other circularly*? Recall that a closed walk is a walk that starts from some vertex and traverses the action graph (potentially with repeated vertices/arcs) and returns to the starting vertex. To compute walks of different length, one could start from small lengths and check if a set of closed walks that form circular periodic propagations exist. For each length, we will have a finite (but potentially exponential) number of closed walks. This approach is incomplete because if we do not find periodic propagations up to some upper bound, then we cannot say that no livelocks exist for larger sizes. Is there a specific length where we should stop the search and conclude that there are no such periodic propagations that lead each other circularly for any ring size? In general, it is impossible to answer this question due to the undecidability of the problem. As a result, we propose a finite-scope search for specific values of m and n that can exhaustively search the problem space for m propagations with period n that lead each other circularly. To solve Problem 6.1, we present Algorithm 1

Algorithm 1. *FindPropagations($G = (V, A)$: action graph, m, n: positive integers greater than 1; M: domain size)*

```
Output: A set of sets of $m$ propagations with period $n$ that circularly lead each other.
1: Represent the action graph $G = (V, A)$ as a matrix $D$, where each entry $d_{ij}$ in $D$ contains the set of
   arc labels between $v_i$ and $v_j$.
2: for $len := 2$ to $n$ do
3:     Compute $D^{len}$, where each entry $d_{ij}$ of $D \cdot D$ is computed by $\cup_{k=0}^{M-1}\{d_{ik}\} \times \{d_{kj}\}$ for $0 \leq i, j \leq M-1$,
       and $\{d_{ik}\} \times \{d_{kj}\}$ denotes the Cartesian product of the set of labels of $d_{ik}$ by the set of labels of $d_{kj}$.
       That is, each entry of matrix $D^2$ includes a set of ordered pairs of arc labels, each entry of matrix
       $D^3$ includes a set of ordered triples of arc labels, and so forth. Thus, each entry of the matrix $D^{len}$
       contains a set of tuples of length len, each representing the labels of a walk of size len in the action
       graph. As such, each entry on the diagonal of $D^{len}$ includes a set of (labels of) closed walks of length
       len.
4: end for
5: Let $C$ denote the set of closed walks of length $n$ extracted from the diagonal entries of $D^n$.
6: Let $S^n$ denote another $M \times M$ matrix derived from $D^n$, where each entry $s_{ij}$ of $S^n$ is a set of tuples
   each obtained by including the labels of the destination vertices of a closed walk in the corresponding
   entry of $D^n$.
7: Create a directed graph $G_W = (V_W, A_W)$, where each vertex $w \in V_W$ represents a closed walk with
   length $n$ in $C$. For any pairs of vertices $w_1, w_2 \in V_W$, if $w_1$ appears in some entry on the diagonal of
   $S^n$ and $w_2$ appears in some entry on the diagonal of $D^n$, then include an arc $(w_1, w_2) \in A_W$.
8: Return the simple cycles of $G_W$ with length $m$. Each cycle of length $m$ represents a set of $m$ propaga-
   tions with period $n$ that successively lead each other.
end
```

Algorithm 1 starts with the action graph $G = (V, A)$ of a protocol, where V denotes the set of vertices and A represents the set of arcs of G. The graph G may be a multigraph because P_i may execute an action where $x_{i-1} = v_1$, but x_i could have one of several values in $\mathbb{Z}_M$. Due to determinism of processes, the outgoing arcs from each vertex v have distinct labels. Since there is a one-to-one correspondence between periodic propagations and closed walks in G, Algorithm 1 actually searches for sets of closed walks that successively lead each other. One can compute the set of closed walks with a specific length n from a vertex v by representing G as a matrix D, where each entry d_{ij} in D contains the set of labels of arcs from v_i to v_j. Then, the set of closed walks with length n from each vertex v is captured by the entry d_{vv} in D^n (Steps 2 to 4 of Algorithm 1). Step 5 makes a union of all diagonal entries of D^n in a set C. In other words, each element

w of the set C represents the arc labels of a unique closed walk. To find leading closed walks, we need to determine the values that are assigned to x_i in each closed walk $w_1 \in C$ (i.e., labels of the vertices reached in w_1) because such values will be the labels of another walk w_2 led by w_1. Thus, Algorithm 1 computes another matrix S^n, where each entry s_{ii} contains the set of vertex labels corresponding to each closed walk in the entry d_{ii} of D^n. To find the closed walks that lead each other in a circular way, we construct another directed graph $G_W = (V_W, A_W)$, where each V_W captures the set of vertices and A_W denotes the set of arcs of G_W. There is a one-to-one correspondence between the elements of C and vertices of V_W; i.e., each vertex of V_W represents a closed walk in C. We draw an arc (w_1, w_2) between two vertices of V_W (i.e., closed walks) *iff* w_1 appears in some entry on the diagonal of S^n and w_2 appears in some entry on the diagonal of D^n. Each simple cycle of length m in G_W captures m propagations with period n that lead each other circularly. Based on Lemma 3.4, such propagations represent a livelock.

For example, consider the action graph of the Sum-Not-Two protocol in Figure 1 (where $M = 3$ and $\mathbb{Z}_M = \{0, 1, 2\}$). The first step of Algorithm 1 creates a matrix D as follows where each entry d_{ij} (for $0 \le i, j \le 2$) contains the arc label from vertex i to vertex j.

$$D =$$

$$\begin{bmatrix} \{\} & \{2\} & \{\} \\ \{\} & \{\} & \{1\} \\ \{\} & \{0\} & \{\} \end{bmatrix}$$

Let $n = 2$ and $m = 2$. Then, in Steps 2 and 3, Algorithm 1 computes D^2.

$$D^2 =$$

$$\begin{bmatrix} \{\} & \{\} & \{(2,1)\} \\ \{\} & \{(1,0)\} & \{\} \\ \{\} & \{\} & \{(0,1)\} \end{bmatrix}$$

Observe that, entry d_{11} (respectively, d_{22}) is equal to $(1, 0)$ (respectively, $(0, 1)$), which means a closed walk of length two that starts and ends at vertex 1 (respectively, 2) first takes an arc with label 1 (respectively, 0) and then an arc with label 0 (respectively, 1) in Figure 1. In Step 5, we compute the set C as the union of all sets on the diagonal of D^2, which is equal to $\{(1,0),(0,1)\}$. In Step 6, we calculate the matrix S^2 as follows:

$$S^2 =$$

$$\begin{bmatrix} \{\} & \{\} & \{\} \\ \{\} & \{(2,1)\} & \{\} \\ \{\} & \{\} & \{(1,2)\} \end{bmatrix}$$

The entry $(2, 1)$ is a tuple of vertex labels that are reached in the closed walk $(1, 0)$. Likewise, entry $(1, 2)$ captures the vertices met in the closed walk $(0, 1)$. Notice that, in Step 7, the graph G_W has no arcs because none of the tuples on the diagonal of S^2 appears in set C. As such, there are no two periodic propagations of length 2 in the Sum-Not-Two protocol that lead each other circularly. In search of livelocks, we can re-invoke Algorithm 1 for larger values of n and m as long as we have computational resources.

Lemma 6.2. *Each simple cycle in G_W (constructed in Algorithm 1) corresponds to a set of periodic propagations.*

Proof. Each vertex $w \in G_W$ represents a closed walk (with length n) from some vertex v in the action graph. Let w_1 and w_2 be any pair of vertices in G_W, where w_1 denotes a closed walk $v_1, l_1, v_2, l_2, \cdots, v_n, l_n, v_1$, where l_i values represent the arc labels and v_i values denote vertices in action graph. Likewise, w_2 represents another closed walk $X, v_2, X, v_3, \cdots, X, v_n, X, v_1, X$, where X denotes a placeholder for any value in $\mathbb{Z}_M$. Based on the definition of the *leads* relation in Section 2, w_1 denotes a propagation that leads the propagation represented by w_2. Thus, a simple cycle in G_W represents a set of propagations with length n that lead each other successively. $\square$

Theorem 6.3 (Soundness)**.** *Algorithm 1 is sound.*

Proof. Soundness follows from Lemma 6.2 as any return value of Algorithm 1 does indeed represent a set of periodic propagations that lead each other circularly. □

Theorem 6.4 (Complexity)**.** *The asymptotic time complexity of Algorithm 1 is* $O((M \cdot b^n)^{\lfloor m/2 \rfloor} \cdot k^{\lceil m/2 \rceil})$, *where* M *denotes the domain size,* m *and* n *capture the block size of periodic propagations,* b *represents the branching factor of the action graph and* k *is the degeneracy of* G_W *constructed in Step 7.*

Proof. In the worst case, Steps 2 to 4 take $O(n \cdot M^3)$. Let b denote the branching factor of the action graph. Then, Step 5 has an asymptotic complexity in $M \cdot b^n$, which is the maximum number of closed walks of length n. Manoussakis [53] shows that listing all cycles of a fixed size m in a digraph G with V vertices has time complexity of $O((|V|)^{\lfloor m/2 \rfloor} \cdot k^{\lceil m/2 \rceil})$, where k is the degeneracy of G. Thus, Step 7 has time complexity $O((M \cdot b^n)^{\lfloor m/2 \rfloor} \cdot k^{\lceil m/2 \rceil})$. (The degeneracy of a graph G is the smallest value k for which every subgraph of G has the max degree k.) □

While the result of Theorem 6.4 may seem a little disheartening, we would like to note that the focus of this paper is on constant-space parameterized protocols with small domains (e.g., up to $M = 5$). As such, we conjecture that, in practice, the worst case complexity would be manageable if one can guarantee that variable domains remain small. Moreover, we plan to exploit the computational power of Graphics Processing Units (GPUs) for the verification of livelock-freedom in parameterized protocols.

Application. An important application of Algorithm 1 is in bounded scope verification where we incrementally increase the values of m and n for each given protocol. While the idea of scope-based verification is not new (see [38]), our verification method is novel in that we search for periodic propagations instead of searching the state space of protocols (often represented as a reachability graph, a decision diagram or a constraint satisfaction problem). We argue that Algorithm 1 can verify livelock-freedom in large-scale parameterized systems in a more efficient way. As an example, consider a protocol on a uni-ring whose action graph is illustrated in Figure 8 (for $M = 5$).

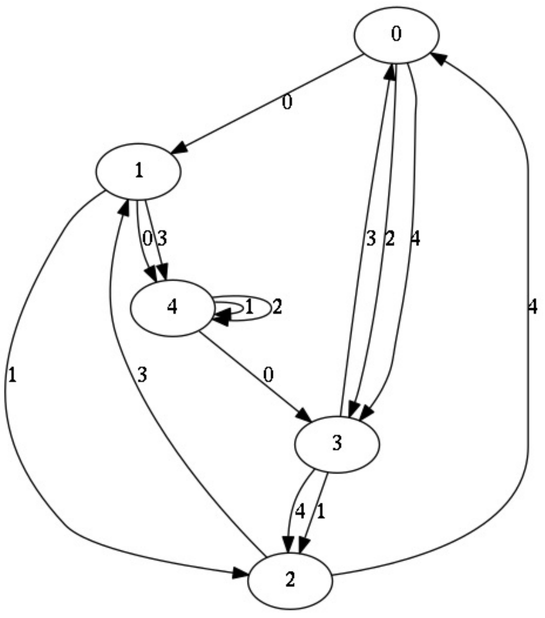


Figure 8: Action graph of an example protocol for $M = 5$.

We have implemented Algorithm 1 as a bounded scope model-checker, called *PROP*, that incrementally searches for periodic propagations. Our current sequential implementation of *PROP* symbolically represents periodic propagations in memory; nonetheless, we are currently developing a matrix-based implementation of *PROP* that can exploit the computational power of GPUs towards the verification of livelock-freedom in parameterized rings. The first set of propagations that *PROP* finds is for $m = n = 21$ illustrated as an instance of tiling problem in Figure 9. These propagations represent a livelock in a ring of 441 processes that can be constructed using the technique presented in Section 3. The search for the periodic propagations of Figure 9 took a fraction of a second on a MacBook Air laptop with 2.2 GHz Intel Core i7 and 8GB RAM, and OS X 10.11.6.

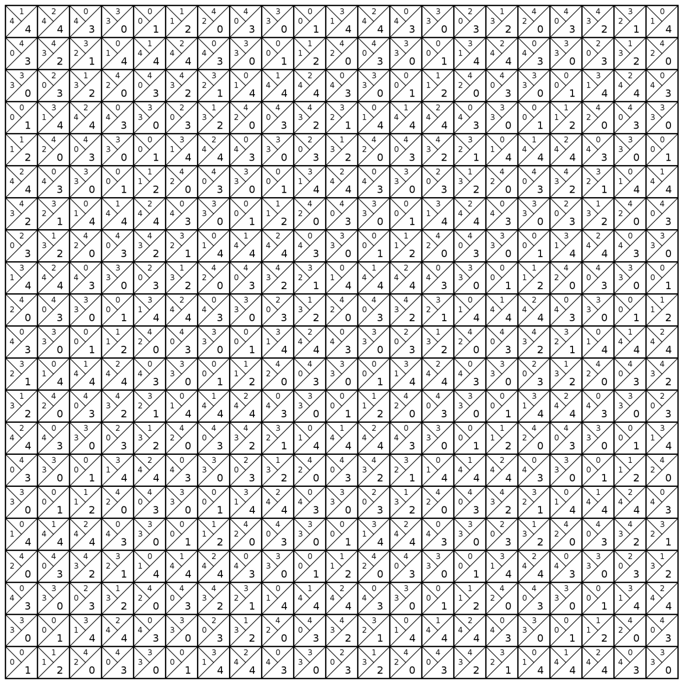


Figure 9: 21 propagations with period 21 illustrated as periodic tiles. Using the approach of Section 3, one can build a ring of size 441 processes that has a livelock represented by these propagations.

Theorem 6.5. *Assume a unidirectional ring protocol of symmetric, self-disabling processes. Let* $\mathcal{P} = \{(m_1, n_1), \cdots, (m_k, n_k)\}$ *be the set of all periodic propagations found by Algorithm 1 in successive invocations of Algorithm 1 for different values of* m_i, n_i*, where* $m_i \leq m_k$ *and* $n_i \leq n_k$ *for* $1 \leq i \leq k$*. Let* $N = m_k \times n_k$ *denote the maximum value among all multiplications* $m_i \times n_i$*, for* $1 \leq i \leq k$*. Then, for rings of size up to* N*, any ring whose size is not equal to some value* $(m_i \times n_i)$ *for some* $(1 \leq i \leq k)$ *is livelock-free.*

Proof. By contradiction, consider that there is some ring size $N' < N$, where $\forall i : 1 \leq i \leq k : N' \neq m_i \times n_i$, but the protocol has a livelock. Then, Lemma 3.4 implies that there must exist m' propagations with some period n' such that these propagations successively lead each other modulo m'. Since $N' < N$, either $m' < m_k$ or $n' < n_k$ or both. Thus, during successive invocations of Algorithm 1 for incremental values of m_i and n_i we would have reached m' and n'. That is, $(m', n') \in \mathcal{P}$, which is a contradiction. Therefore, N' does not exist. □

Notice that Theorem 6.5 has significant applications in the design and verification of distributed programs (e.g., network protocols) whose underlying communication topology includes rings. More specifically, Theorem 6.5 along with Algorithm 1 enable designers to predict the ring sizes at which livelock failures may occur. Such predictions can result in saving time, energy and costs by ensuring that the system is in a *safe scale* where no livelocks can occur.

7 Experimental Results

In this section, we present the experimental results we obtained through a sequential implementation of *Prop* (introduced in Section 6). The platform used for these experiments is a regular MacBook Air laptop with a 2.2 GHz Intel Core i7 with 8GB of memory. The timing results are averaged over 100 runs for each example protocol and each domain size. Our objective in these experiments is to evaluate the effectiveness of *Prop* in finding livelocks when the domain size of variables and the length of the periodic propagations (i.e., length of livelocks) grow. We would like to note that we have also developed an enumerator (i.e., Turing recognizer) [43] that searches through all possible protocols for each specific domain size and prints out the protocols that have livelocks. Our experimental results with this enumerator show that this is a very inefficient way for verification and the average search time is exponentially larger than the results we report in this section on the performance of *Prop*. Sections 7.1 to 7.5 present five case studies, and Section 7.6 discusses relevant verifier tools and compares their capabilities with *Prop*.

7.1 Leader Election

Leader election is an important protocol in distributed systems and is used in a variety of contexts such as wireless networks resistant to jamming [44], mobile ad-hoc networks [14], locking, synchronization and load balancing [42, 52]. While it is known that there is no leader election protocol on symmetric uni-rings [4], our objective here is to evaluate the power of *Prop* in finding livelocks that prevent stabilization to states where a unique leader is elected. We consider a leader election protocol on a symmetric uni-ring of $N > 1$ processes, where the objective is to assign unique values modulo N to processes. That is, the protocol should self-stabilize to legitimate states where processes have unique values. In such states, the process with value 0 can be declared as the leader. For each process of the Leader Election (LE) protocol, we consider the following action:

$$(x_{i-1} = x_i) \rightarrow x_i := x_i \oplus 1$$

where $\oplus$ denotes addition modulo M, and M represents the domain size of variable x_i of process P_i. Stabilization is impeded by livelocks that we detect using *Prop*. The input to *Prop* includes the Protocol Under Verification (PUV), the domain size of x_i and an upper bound on the period of propagations that may create a livelock. That is, an upper bound on the scope of the search for leading closed walks.

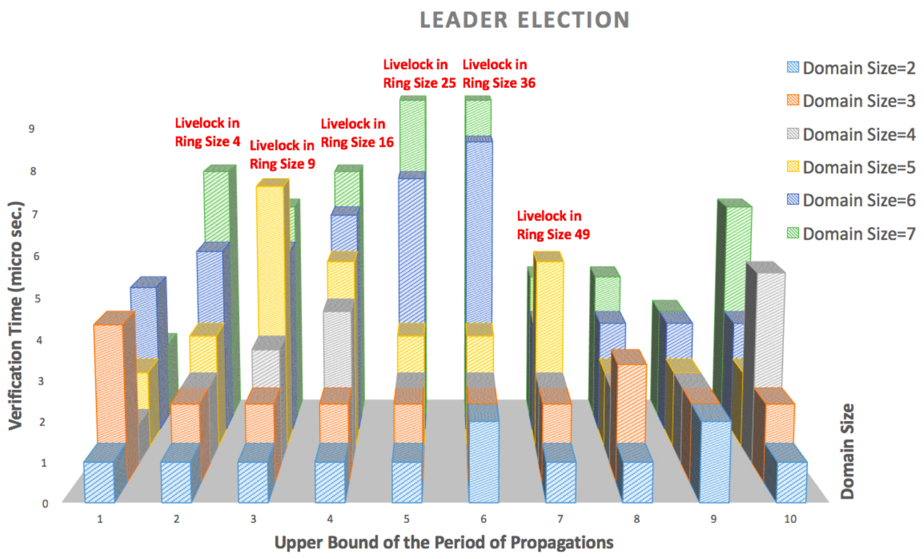


Figure 10: Verifying livelock-freedom in leader election on a ring.

Figure 10 illustrates the results of verifying the LE protocol introduced above. We have verified this protocol for domain sizes of 2 to 7 and for propagations with periods 1 to 10. The vertical axis captures the average verification time in micro seconds. Notice that for period 2, we find a livelock that manifests itself in a ring size 4, which is the result of a 2×2 periodic propagation (see Figure 11). This pattern continues as we increase the period length of propagations to livelocks in ring sizes $9, 16, 25, \cdots$. As the domain size increases (i.e., the depth dimension), the average verification time slowly increase to 9 microsecond, which is not significant.

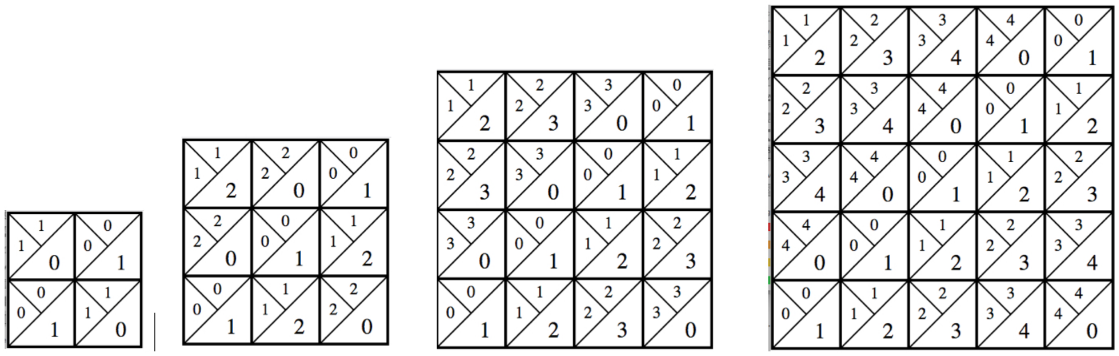


Figure 11: Periodic propagations found in Leader Election, each representing a livelock in ring sizes 4, 9, 16 and 25.

7.2 Token Ring

Token Ring (TR) is another important protocol in distributed computing used to implement mutual exclusion [13] on rings. We verify a candidate TR protocol where each process in the ring is privileged to take an action when its x_i value is one unit less than its predecessor's modulo domain size M. The objective is to design a self-stabilizing TR protocol that self-stabilizes to the set of legitimate states where there is at most one token in the entire ring. While Dijkstra has already solved this problem [13], it took him a long time to prove the correctness of his design and prove that it is impossible to solve this problem on a fully symmetric ring. Our proposed method implemented in *Prop* provides an automatic approach for showing why a symmetric uni-ring does not self-stabilize to legitimate states by finding its livelocks in a few microseconds. Each process in a symmetric uni-ring includes the following parameterized action:

$$(x_{i-1} = x_i \oplus 1) \rightarrow x_i := x_{i-1}$$

Figure 12 illustrates our experimental results for verifying livelock-freedom of this protocol. We found livelocks for ring sizes 2 to 7 as we increased the domain size from 2 to 7. We observe that, in the case of TR, the existence of livelocks depends on domain size. That is, as we increase the domain, we detect a set of periodic propagations whose period is equal to the domain size. Figure 13 illustrates a few periodic propagations we found for ring sizes 2, 3, 4 and 5. For larger ring sizes, we find similar propagations that have a larger dimension of $k \times 1$ for ring sizes $k > 5$. Similar to leader election, the average verification time remains in the scope of a few micro seconds.

7.3 Agreement

When processes in a distributed system decide on a value (i.e., output of a function, vote, etc.), some sort of collective agreement must be reached. As such, reaching agreement is an important primitive and a classic problem in distributed computing. Designing a self-stabilizing agreement protocol requires recovery to the set of legitimate states $\forall i : i \in \mathbb{Z}_N : x_{i \ominus 1} = x_i$, where N represents the number of processes in the ring and $\ominus$ denotes subtraction modulo N. We verify the following agreement protocol where each process copies the value of its predecessor if $x_{i-1} \neq x_i$ holds.

$$x_{i-1} \neq x_i \rightarrow x_i := x_{i-1}$$

Figure 14 shows how average verification time increases when domain size and period of propagations grow. Up to the scope we searched, we just found a livelock in a ring size 2. This livelock could occur

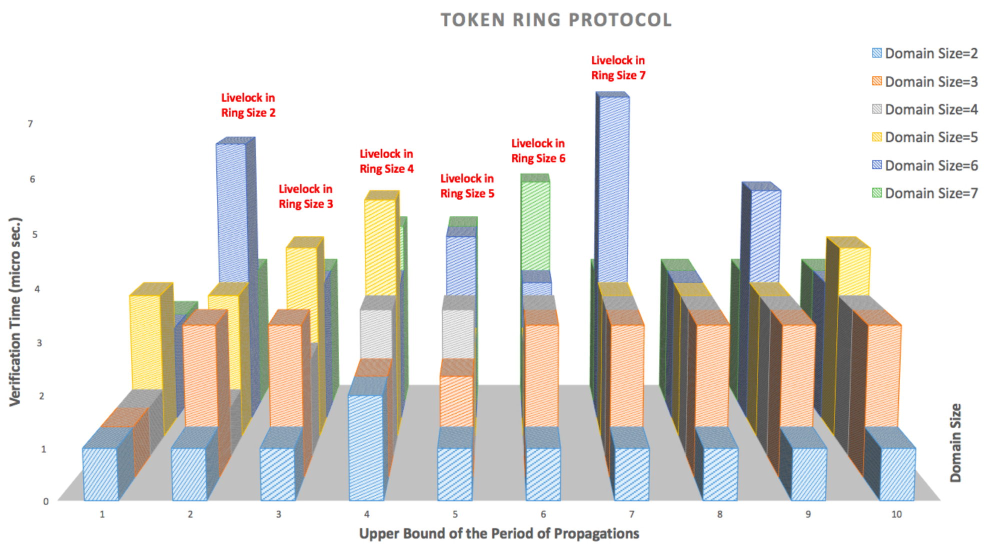


Figure 12: Verifying token passing on a uni-ring.

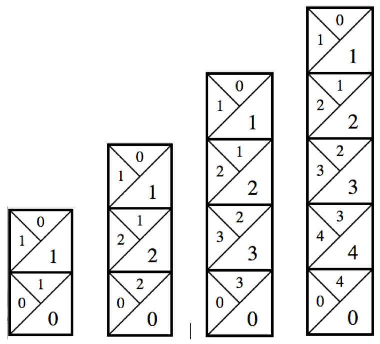


Figure 13: Periodic propagations found in Token Ring, each representing a livelock in ring sizes 2, 3, 4 and 5.

regardless of the domain size, where two processes (in a ring size 2) alternatively execute the actions $\langle 1,0,1\rangle$ and $\langle 0,1,0\rangle$ generating two propagations of period 1 that lead each other circularly.

7.4 Coloring

The graph M-coloring problem (where $M > 1$) asks for the coloring of a graph using M colors in such a way that no two neighboring vertices have the same color. The coloring problem has significant applications in several domains such as scheduling, register allocation, frequency band allocation, etc. We have verified a candidate coloring protocol on uni-rings with the following parameterized action for each process. The candidate coloring protocol should stabilize to legitimate states $\forall i : i \in \mathbb{Z}_N : (x_{i\ominus 1} \neq x_i) \wedge (x_{i\oplus 1} \neq x_i)$, where $\ominus$ and $\oplus$ respectively denote subtraction and addition modulo N. While it is known [6] that no self-stabilizing coloring protocol exists on uni-rings, we are interested in verifying the source of such impossibility results in an automated fashion. For example, we have verified the following coloring protocol and have identified its livelocks.

$$(x_{i-1} = x_i) \rightarrow x_i := \mathit{other}(x_{i-1}, x_i)$$

The increase in the size of the livelocks found in the coloring protocol is proportional to the domain size. More specifically, starting from domain size 2 and upper bound 2 for the period of propagations, we find periodic propagations of size $M \times M$, which would constitute a livelock in a ring size M^2. The average verification time remains below 11 micro seconds.

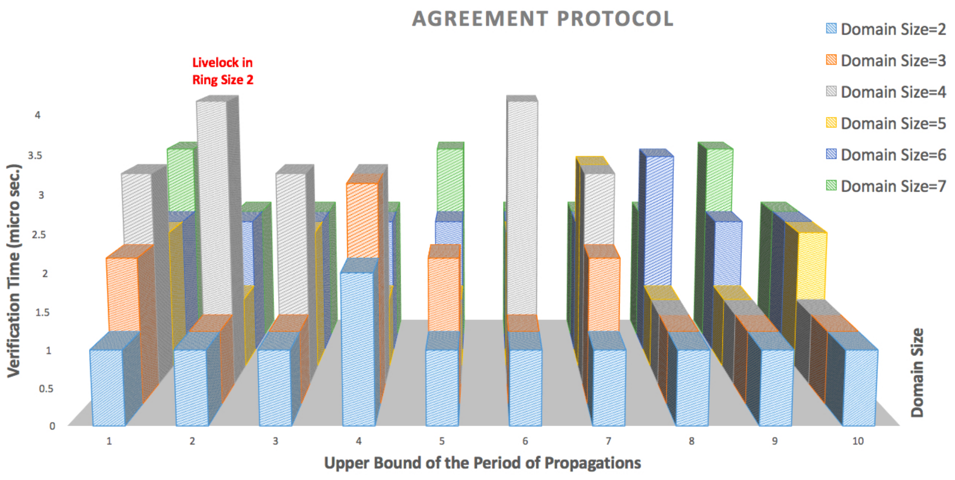


Figure 14: Verifying agreement on a uni-ring.

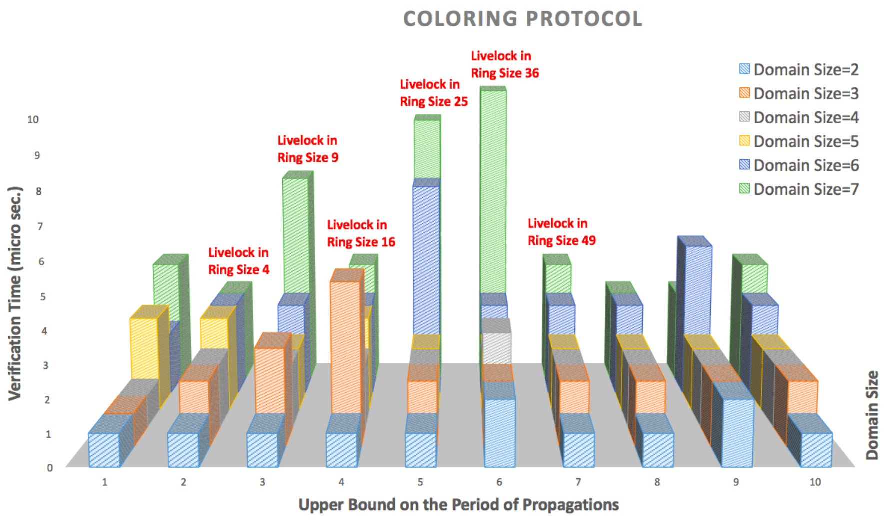


Figure 15: Verifying M-coloring on a uni-ring.

7.5 Parity

The Parity protocol requires the entire ring to stabilize to a set of legitimate states where all x_i values are even or odd. If there is an even (respectively, odd) value in the ring, then all values should be even (respectively, odd) in a legitimate state. Thus, from any state, Parity 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. The set of legitimate states includes states where $\forall i : i \in \mathbb{Z}_N : ((\mid x_{i-1} - x_i \mid) \texttt{ mod } 2) = 0$ holds. A candidate parameterized action for the symmetric uni-ring is as follows, where $\oplus$ and $\ominus$ respectively denote addition and subtraction modulo domain size M.

$$(\mid x_{i-1} \ominus x_i \mid \texttt{ mod } 2) \neq 0 \rightarrow x_i := x_{i-1} \oplus 2$$

We have verified the livelock-freedom of Parity for domain sizes of 2 to 8 and propagations with periods 1 to 10 (see Figure 16). We observe that this protocol is livelock-free for odd-size domains (i.e., when M is odd) because in an odd domain the number of odd values is one less than the number of even values. This will break the symmetry that forms periodic propagations created by the above action.

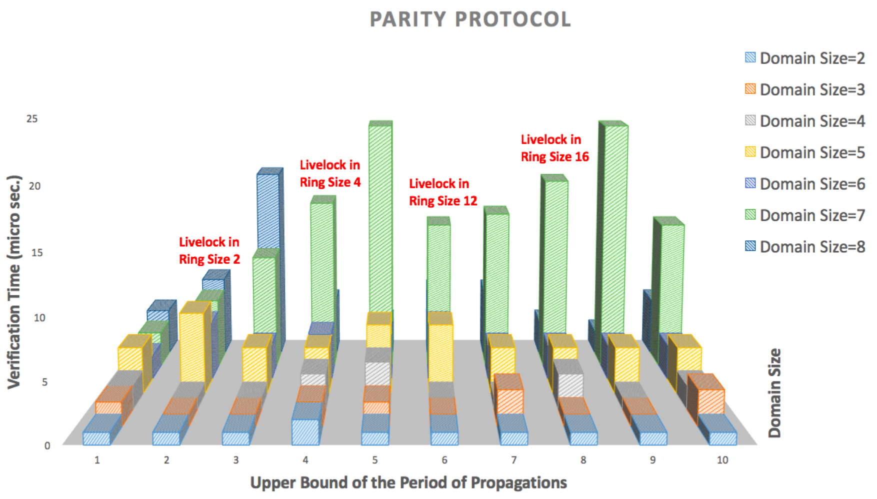


Figure 16: Verifying the parity protocol on a uni-ring.

Notice that the average verification time hiked to around 25 micro seconds for some instances of this protocol. However, such an average verification time is still significantly better than the state-of-the-art verifiers such as Cubicle [12, 11] and ByMC (i.e., parameterized Promela/SPIN) [49].

7.6 Comparison With Existing Automatic Verifiers

In this section, we discuss the related research on the verification of parameterized systems that has culminated in the development of verifier tools. Most existing automatic verifiers for parameterized systems focus on safety properties and can be classified into SMT-based, symbolic and explicit-state verifiers. We first introduce some of these verifiers and then compare them with respect to a set of criteria in Table 1.

Cubicle. *Cubicle* is a Parameterized Model Checker (PMC) [12, 11] for the verification of safety properties of concurrent array-based systems, where system state is modeled as an array whose each memory cell represents the local state of a process. *Cubicle* formulates the verification problem as a backward reachability problem and utilizes SMT solvers to search for violations of safety properties.

ByMC. *ByMC* model checker [49] extends Promela/SPIN [36] for threshold-guided distributed algorithms, which form a special family of parameterized systems that have Threshold-Guarded Actions (TGA). Each TGA includes conditions specified on the number of messages sent/received from a portion of all processes (called the action *threshold*) without the need for knowing the identity of those processes. *ByMC* targets the verification of threshold-guarded Fault-tolerant Distributed Protocols (FTDP), which covers a subset of distributed protocols with TGAs. Protocols are specified in an extended version of Promela, which is the specification language of the SPIN model checker. Then, *ByMC* automatically extracts Threshold-Automata (TA) from the input specifications. The TA is then transformed into an abstract counter system that captures the behaviors of the input protocol in terms of the number of processes that are in a specific location of their code at any system state; hence the term *counter system*. *ByMC* uses BMC and SMT solving to conduct verification on the counter system.

Sally. *Sally* provides a language and a verification framework for the specification and verification of synchronous FTDP. *Sally* utilizes several state-of-the-art verification methods (e.g., Property-Directed k-induction [39]) to enable Bounded Model Checking (BMC) using SMT solvers.

IronFleet. *IronFleet* [35] provides a framework for formal specification, verification and the generation of correct code for distributed protocols. At its verification core, *IronFleet* uses BMC and SMT solving in order to verify he correctness of safety and liveness properties of distributed protocols. After verification, *IronFleet* can generate verified code as well.

Ivy. *Ivy* [65] provides a novel method for the verification of safety properties in infinite-state protocols. *Ivy*

Verifier	**Average Verification Time**	**System Topology**	**Safety (S) Liveness (L)**	**Verification Method**	**System Scope**	**Fault Tolerance**	**Fairness**
Cubicle [12, 11]	Tens of milli seconds to minutes	General	S	Symbolic and SMT	Array-based systems	No	N/A
ByMC [47, 48, 46, 49]	Seconds to hours	Clique	S and L	Abstraction, SMT, Explicit state	Threshold Automata	Yes	Yes
Sally [15]	Tens of milli seconds to minutes	General	S	SMT, k-induction, BMC	Synchronous array-based	Yes	N/A
IronFleet [35]	Minutes to hours	General	S and L	SMT	Distributed Protocols	Yes	Yes
Ivy [56, 65]	Minutes	General	S	Inductive invariants and BMC	Distributed Protocols	No	N/A
BIP [45]	Tens of milli seconds	General	S	Model checking and SMT	Multi-Party synchronous	No	N/A
Prop	Micro seconds	Specific	L	Periodic Propagations	Distributed Protocols	Yes	No

Table 1: Comparison of state-of-the-art verifiers for distributed parameterized protocols.

employs BMC and a new bounded debugging method for the generation of inductive invariants used to prove safety properties. *Ivy* also benefits from a graphical user interface that facilitates user interactions as *Ivy* is not a fully automatic verifier.

Behavior-Interaction-Priority (BIP). Parameterized *BIP* [45] provides a framework for the verification of distributed protocols specified in BIP as a parameterized system. The scope of parameterized BIP includes multi-party synchronous systems, where transitions of a subset of components that take part in an interaction are executed synchronously. The BIP framework determines which existing method for the verification of parameterized systems is applicable to the extracted model of the system. The verification engine of BIP uses temporal model checking and SMT solving to conduct the verification of safety properties.

Comparison. Table 1 positions the aforementioned verifiers with respect to a set of criteria such as *average verification time, topology of the system under verification, safety and liveness properties, verification method used, scope of the systems that can be verified, support for verification of fault-tolerant systems and fairness* assumptions for the verification of liveness properties. As such, fairness is irrelevant to verifiers that focus on the verification of safety properties. Moreover, the issue of fairness brings up a trade off for protocol designers and protocol developers. Assuming fairness would simplify the task of designers and verifiers because resolving some livelocks under fairness becomes simplified. However, implementing a particular fairness policy in a distributed system is by itself a daunting task. As such, if a verifier is able to verify liveness properties under no fairness assumptions, then it would make life easier for designers and programmers of distributed systems. Notice that, *Prop*'s average verification time is in the scale of micro seconds while all other tools have a timing in the range of milli seconds to minutes/hours. Moreover, *Prop* can verify livelock-freedom and self-stabilization under no fairness for both synchronous and asynchronous systems. The verification method of *Prop* is totally different from most extant methods, where we detect periodic propagations and construct livelocks as well. While *Prop* benefits from a significant speed up in average verification time, it is a topology-specific verification method. *Prop* is the first step in our ongoing work on a paradigm of *topology-specific verification*, where we will develop a repository of efficient topology-specific verifiers along with property-preserving compositions that can create more complicated topologies out of elementary ones such as ring, tree, mesh, etc. The other two verifiers (*ByMC* and *IronFleet*) that can verify liveness properties each have their own limitations. For example, both of them assume some sort of fairness policies, whereas we make no such assumptions in *Prop*. Livelock detection in *ByMC* focuses on threshold automata and counter systems, where processes cannot refer to the local state of specific neighbors (which is what we need in uni-rings). Instead of weak fairness, IronFleet [35] implements always-enabled actions where processes are not blocked when a condition is false. Always-enabled actions ensure fairness for proving local liveness properties – where a process should eventually respond to some request – but cannot help for proving global liveness properties (e.g., convergence) where all processes should collectively make some global condition true.

8 Related Work

This section discusses related work regarding necessary and/or sufficient conditions for livelock freedom and decidability of livelock freedom in Parameterized Systems (PSs). Specifically, our previous work [24] investigates sufficient conditions for livelock freedom in symmetric unidirectional ring protocols of self-disabling processes. We also present necessary and sufficient conditions for deadlock detection in symmetric unidirectional and bidirectional ring protocols. This paper complements our previous work by showing that, even when assuming deterministic and self-disabling properties, livelock freedom on ring topologies is undecidable in general.

Decidability. In [3], Apt and Kozen prove that verifying an LTL formula holds for a parameterized system is Π^0_1-complete. Suzuki [64] builds on this result, showing that the problem remains Π^0_1-complete for unidirectional ring protocols of symmetric processes. Emerson and Namjoshi [20] show that the result holds even when a token that can take two different values is passed around such a ring. Abello and Dolev [2] present a reduction from any Turing machine to a self-stabilizing bidirectional chain protocol. Fabret and Petit [22] use the domino problem to show that reaching a deadlock from an initial state is undecidable for planar grid topologies. We strengthen existing results by considering constant space, deterministic and self-disabling processes. Moreover, it is unclear how Suzuki's proof [64] can be used for proving the undecidability of livelock-freedom and verification of self-stabilization on *fully symmetric* distributed rings for several reasons.

- First, his proof relies on constructing a Turing machine that simulates an *asymmetric* unidirectional ring (a.k.a. uni-ring), where there is a distinguished machine, denoted M_0, that has a specific functionality different from other machines in the ring (and has buffers of different sizes). By contrast, we investigate fully symmetric uni-rings.

- Second, his proof is based on the interleaving execution semantics where at any moment at most one machine executes, whereas our proof of undecidability of livelock-freedom is not limited to the interleaving semantics; even if we assume the synchronous execution semantics – where all processes are executed at the same time – our proof still remains valid (Corollary 3.7). Suzuki's proof does not address synchronous protocols.

- Third, Suzuki's proof relies on constructing a Turing machine that starts from a specific initial configuration/state; the distinguished machine M_0 starts executing by observing a special symbol % just acceptable to M_0. To address this issue for self-stabilization, one has to devise a proof that works no matter what symbols are on the tape when the uni-ring starts executing, which would make it a tedious proof. Instead of doing that, we reduce the Wang Tiles problem to livelock-freedom and self-stabilization because (1) tiles are intuitively similar to the self-disabling actions, and (2) the proof becomes applicable for both the interleaving and the synchronous executions.

Abstraction methods [7, 37, 57, 19] generate a finite-state model of a PS and then reduce the verification of the PS to the verification of its finite model. SMT-based verification [29, 12] is an example of such abstraction methods where SMT solvers are used to verify safety and inclusion properties in a reachability analysis phase. Parameterized Visual Diagrams (PVDs) [62] model a PS and its required properties in terms of visual abstractions (e.g., predicate automata); however, they assume weak fairness and generate a large number of verification conditions that should be verified by model checking. Esperza *et al.* [21] show the decidability of verifying whether a population protocol is well-specified; i.e., stabilizes to a predicate from any initial configuration. There are several differences between our work and Esperza *et al.*'s [21]. First, as an example, Esperza *et al.* [21] assume strong fairness, which means they focus on weak stabilization, where only *reachability* of legitimate states is required; i.e., livelock detection is not required in their work. By contrast, we verify stabilization under no fairness, where livelock-freedom is an important property to be verified. Second, they consider the clique topology where pairs of processes can read/sense each other's state and take an action. As a result, in the context of IOPP there are no topological constraints similar to what we have in uni-rings where processes receive information only from their predecessor. Third, in population protocols processes are anonymous and process identifiers (IDs) are not explicitly specified, whereas in our context, processes can refer to a specific neighbor using process IDs. This issue also impacts the kind of predicates that can specify the set of legitimate states. Fourth, the kind of constraints to which population protocols stabilize include only counting constraints, which are disjunctions of conjunctive counting constraints of the form $(x_1 \leq u) \wedge \cdots \wedge (x_k \geq v)$, where x_i, for $1 \leq i \leq k$, denotes the number of processes that are in some state i. As a result, population protocols compute counting predicates. By contrast, the specifications of the set of legitimate states that we can capture can be significantly more complicated and general (see Section 7 for examples). Fifth, our undecidability results are not limited to interleaving semantics; rather we show that our results hold even for synchronous systems and FIFO consistency model.

Network invariant approaches [68, 41, 33] find a process that satisfies the property of interest and is invariant to parallel composition; i.e., composing it with itself for an arbitrary number of times will create a system that still satisfies the property of interest. The network invariant method is mostly used for the verification of safety properties, whereas self-stabilization includes a global liveness property, namely convergence. **Neo** [54] uses network invariants to identify architectures [54] with special topologies (e.g., trees) for which safety properties are verifiable. Neo's topology-specific verification has similarities to our topology-specific method.

Methods for **compositional model checking** of PSs (e.g., cache coherence [55]) use abstract interpretation to reduce the verification of unbounded systems to finite-state model checking of a set of local temporal properties. Such abstractions are too coarse for synthesizing self-stabilization because a self-stabilizing system must guarantee convergence from each concrete state. **Logic program transformations** and inductive verification methods [59, 60, 61, 26] encode the verification of a PS as a constraint logic program and verify the equivalence of goals in the logic program. **Proof spaces** [25] enable a novel method for automated extraction of Hoare triples for unbounded multi-threaded programs, where these verification conditions are used in a deductive reasoning system.

In **regular model checking** [27, 8, 1], states of parameterized rings are represented by strings of arbitrary length, and a protocol is represented by a finite state transducer. Let R be the relation of this transducer and let I be the regular language of legitimate states (including impossible states like the empty string). All states are reachable in the context of stabilization, therefore closure is expressed as the closure of I under the operation R; i.e., $R(I) \subseteq I$. Deadlock freedom is also decidable by checking whether R produces output for every state in $\neg I$; i.e., $\neg I \subseteq \mathsf{Pre}(R)$. For livelock detection, we must check if the iterative application of R to some string $w \in \neg I$ can yield w. With R^+ as the transitive closure of R, livelock freedom is therefore equivalent to checking $(\forall w \in \neg I : w \notin R^+(\{w\}))$. R^+ is uncomputable in general, but sometimes it can be computed using techniques such as widening [66].

Emerson *et al.* [18, 17] present **cutoff theorems** for the verification of temporal logic properties in parameterized systems, where a property $\mathcal{P}$ holds for a parameterized protocol p if and only if $\mathcal{P}$ holds for an instantiation of p with a fixed number of processes k, called the *cutoff*. This method is mainly applicable for properties that are specified in terms of the locality of each process.

Our reasoning strongly resembles that of regular model checking. We can interpret the graph of a protocol p as a transducer, where arc and node labels denote input and output symbols respectively (i.e., a Moore machine). To represent periodic propagations, we must ensure that the initial and final nodes are the same. Thus, the actual transducer is a union of all machines formed by marking a single node as both initial and accepting, where the initial node does not output a symbol. Livelock freedom can be verified directly using regular model checking, with the detail that $\neg I$ is all nonempty strings because strings represent propagations instead of states.

9 Conclusion and Future Work

We investigated the problem of verifying livelock-freedom and self-stabilization on parameterized symmetric rings of constant space, deterministic and self-disabling processes. We presented three major results. First, we strengthened Apt and Kozen's [3] results in showing that even for constant space and self-disabling processes verifying livelock freedom and self-stabilization for parameterized rings remain undecidable. We also extended this result for any system with a cyclic topology. Moreover, we showed that our undecidability results hold under (1) synchronous and asynchronous execution semantics; (2) the FIFO consistency model, and (3) any scheduling policy. Then, we presented a transformative method for scope-based model checking of parameterized symmetric rings, where one could detect and construct livelocks in very large rings. We have implemented this method as a scope-based model checker, called *Prop*, that can verify livelock-freedom for parameterized symmetric rings by totally circumventing state space exploration. *Prop* can also construct livelocks in rings of large sizes when it finds them in the given scope, in addition to determining the ring sizes for which that livelock will manifest itself. This is a significant achievement as most existing model checkers of parameterized systems focus on safety and local liveness properties, whereas self-stabilization includes a global liveness property that must be achieved by the collective actions of all processes. We plan to extend this work for asymmetric parameterized rings, where a ring contains several families of symmetric processes. Moreover, we would like to extend our work for the verification of hyperproperties [10].

References

[1] P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. A survey of regular model checking. In *International Conference on Concurrency Theory*, pages 35–48, 2004.

[2] J. Abello and S. Dolev. On the computational power of self-stabilizing systems. *Theoretical Computer Science*, 182(1-2):159–170, 1997.

[3] K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.

[4] H. Attiya, M. Snir, and M. K. Warmuth. Computing on an anonymous ring. *Journal of the ACM (JACM)*, 35(4):845–875, 1988.

[5] R. Berger. *The Undecidability of the Domino Problem.* Memoirs ; No 1/66. American Mathematical Society, 1966.

[6] S. Bernard, S. Devismes, M. G. Potop-Butucaru, and S. Tixeuil. Optimal deterministic self-stabilizing vertex coloring in unidirectional anonymous networks. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[7] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for ctl*. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 388–397, 1995.

[8] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *International Conference on Computer-Aided Verification*, pages 403–418, 2000.

[9] D. Cachera and K. Morin-Allory. Verification of safety properties for parameterized regular systems. *ACM Transactions on Embedded Computing Systems*, 4(2):228–266, 2005.

[10] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[11] S. Conchon, D. Declerck, and F. Zaidi. Cubicle-W: Parameterized model checking on weak memory. In *International Joint Conference on Automated Reasoning*, pages 152–160. Springer, 2018.

[12] S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaıdi. Cubicle: A parallel smt-based model checker for parameterized systems. In *CAV*, pages 718–724. Springer, 2012.

[13] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, Nov 1974.

[14] S. Dulman, P. Havinga, and J. Hurink. Leader election protocol for energy efficient mobile sensor networks (eyes). 2002.

[15] B. Dutertre, D. Jovanović, and J. A. Navas. Verification of fault-tolerant protocols with sally. In *NASA Formal Methods Symposium*, pages 113–120. Springer, 2018.

[16] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. Elsevier, 1990.

[17] E. A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *International Conference on Automated Deduction*, pages 236–254, 2000.

[18] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–94, 1995.

[19] E. A. Emerson and K. S. Namjoshi. On reasoning about rings. *International Journal of Foundations of Computer Science*, 14(4):527–550, 2003.

[20] E. A. Emerson and K. S. Namjoshi. On reasoning about rings. *International Journal of Foundations of Computer Science*, 14(4):527–550, 2003.

[21] J. Esparza, P. Ganty, R. Majumdar, and C. Weil-Kennedy. Verification of immediate observation population protocols. *arXiv preprint arXiv:1807.06071*, 2018.

[22] A. Fabret and A. Petit. On the undecidability of deadlock detection in families of nets. In E. Mayr and C. Puech, editors, *STACS 95*, volume 900 of *Lecture Notes in Computer Science*, pages 479–490. Springer Berlin Heidelberg, 1995.

[23] A. Farahat. *Automated Design of Self-Stabilization*. PhD thesis, Michigan Technological University, 2012.

[24] A. Farahat and A. Ebnenasir. Local reasoning for global convergence of parameterized rings. In *IEEE International Conference on Distributed Computing Systems*, pages 496–505, 2012.

[25] A. Farzan, Z. Kincaid, and A. Podelski. Proving liveness of parameterized programs. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 185–196. ACM, 2016.

[26] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. *TPLP*, 13(2):175–199, 2013.

[27] L. Fribourg and H. Olsén. Reachability sets of parameterized rings as regular languages. *Electronic Notes in Theoretical Computer Science*, 9:40, 1997.

[28] P. Funk and I. Zinnikus. Self-stabilization as multiagent systems property. In *AAMAS*, pages 1413–1414. ACM, 2002.

[29] S. Ghilardi and S. Ranise. *MCMT: A Model Checker Modulo Theories*, pages 22–29. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[30] M. G. Gouda. The theory of weak stabilization. In A. K. Datta and T. Herman, editors, *WSS*, volume 2194 of *Lecture Notes in Computer Science*, pages 114–123. Springer, 2001.

[31] M. G. Gouda and H. B. Acharya. Nash equilibria in stabilizing systems. *Theoretical Computer Science*, 412(33):4325–4335, 2011.

[32] M. G. Gouda and N. J. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.

[33] O. Grinchtein, M. Leucker, and N. Piterman. Inferring network invariants automatically. In *Automated Reasoning*, pages 483–497. Springer, 2006.

[34] Y. Gurevich and I. O. Koriakov. A remark on Berger's paper on the domino problem. *Siberian Mathematical Journal*, 13(2):319–321, 1972.

[35] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17. ACM, 2015.

[36] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[37] C. N. Ip and D. L. Dill. Verifying systems with replicated components in murphi. *Formal Methods in System Design*, 14(3):273–310, 1999.

[38] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[39] D. Jovanović and B. Dutertre. Property-directed k-induction. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, pages 85–92. FMCAD Inc, 2016.

[40] J. Kari. The nilpotency problem of one-dimensional cellular automata. *SIAM Journal on Computing*, 21(3):571–586, 1992.

[41] Y. Kesten, A. Pnueli, E. Shahar, and L. Zuck. Network invariants in action. In *Concurrency Theory*, pages 101–115. Springer, 2002.

[42] C.-T. King, T. B. Gendreau, and L. M. Ni. Reliable election in broadcast networks. *Journal of Parallel and Distributed Computing*, 7(3):521–540, 1989.

[43] A. Klinkhamer and A. Ebnenasir. A Model Checker for Lovelock-Freedom in Symmetric Rings. https://github.com/grencez/protocon/tree/master/src/uni.

[44] M. Klonowski and D. Pajak. Electing a leader in wireless networks quickly despite jamming. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 304–312. ACM, 2015.

[45] I. Konnov, T. Kotek, Q. Wang, H. Veith, S. Bliudze, and J. Sifakis. Parameterized systems in bip: design and model checking. In *Proceedings of the 27th International Conference on Concurrency Theory (CONCUR 2016)*, number EPFL-CONF-221300, pages 30–1. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

[46] I. Konnov, M. Lazić, H. Veith, and J. Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. *ACM SIGPLAN Notices*, 52(1):719–734, 2017.

[47] I. Konnov, H. Veith, and J. Widder. Smt and por beat counter abstraction: parameterized model checking of threshold-based distributed algorithms. In *International Conference on Computer Aided Verification*, pages 85–102. Springer, 2015.

[48] I. Konnov, H. Veith, and J. Widder. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. *Information and Computation*, 252:95–109, 2017.

[49] I. Konnov and J. Widder. Bymc: Byzantine model checker. In *International Symposium on Leveraging Applications of Formal Methods*, pages 327–342. Springer, 2018.

[50] H. J. La and S. D. Kim. A self-stabilizing process for mobile cloud computing. In *IEEE International Symposium on Service-Oriented System Engineering*, pages 454–462, 2013.

[51] R. J. Lipton and J. S. Sandberg. PRAM : a scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Sept. 1988.

[52] C. Malloth and A. Schiper. View synchronous communication in large scale networks. In *2nd Open Workshop of the ESPRIT project BROADCAST*, number 6360. Citeseer, 1995.

[53] G. Manoussakis. Listing all fixed-length simple cycles in sparse graphs in optimal time. In *International Symposium on Fundamentals of Computation Theory*, pages 355–366. Springer, 2017.

[54] O. Matthews, J. Bingham, and D. J. Sorin. Verifiable hierarchical protocols with network invariants on parametric systems. In *Formal Methods in Computer-Aided Design (FMCAD), 2016*, pages 101–108. IEEE, 2016.

[55] K. L. McMillan. Parameterized verification of the flash cache coherence protocol by compositional model checking. In *Correct hardware design and verification methods*, pages 179–195. Springer, 2001.

[56] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham. Ivy: safety verification by interactive generalization. *ACM SIGPLAN Notices*, 51(6):614–630, 2016.

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

[58] H. Rogers. *Theory of recursive functions and effective computability (Reprint from 1967).* MIT Press, 1987.

[59] A. Roychoudhury, K. N. Kumar, C. Ramakrishnan, I. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program transformations. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 172–187. Springer, 2000.

[60] A. Roychoudhury and I. Ramakrishnan. Automated inductive verification of parameterized protocols? In *Computer Aided Verification*, pages 25–37. Springer, 2001.

[61] A. Roychoudhury and I. Ramakrishnan. Inductively verifying invariant properties of parameterized systems. *Automated Software Engineering*, 11(2):101–139, 2004.

[62] A. Sánchez and C. Sánchez. Parametrized verification diagrams. In *Temporal Representation and Reasoning (TIME), 2014 21st International Symposium on*, pages 132–141. IEEE, 2014.

[63] R. C. Steinke and G. J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM*, 51(5):800–849, Sept. 2004.

[64] I. Suzuki. Proving properties of a ring of finite-state machines. *Information Processing Letters*, 28(4):213–214, July 1988.

[65] M. Taube, G. Losa, K. L. McMillan, O. Padon, M. Sagiv, S. Shoham, J. R. Wilcox, and D. Woos. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 662–677. ACM, 2018.

[66] T. Touili. Regular model checking using widening techniques. *Electronic Notes in Theoretical Computer Science*, 50(4):342–356, 2001.

[67] H. Wang. Proving theorems by pattern recognition II. *Bell System Technical Journal*, 40:1–42, 1961.

[68] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *International Workshop on Automatic Verification Methods for Finite State Systems*, pages 68–80, 1989.