On the Limits and Practice of Automatically Designing Self-Stabilization

By

Alex Peter Klinkhamer

A Dissertation

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

In Computer Science

Michigan Technological University

2016

This dissertation has been approved in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science.

Department of Computer Science

Dissertation Advisor: *Dr. Ali Ebnenasir*

Committee Member: *Dr. Melissa Keranen*

Committee Member: *Dr. Jean Mayo*

Committee Member: *Dr. Charles Wallace*

Department Chair: *Dr. Min Song*

For Dan and Luci, my loving parents,
whose realism and optimism forever guide me

Contents

List of Protocols

List of Algorithms

List of Figures

Preface

The first three chapters present concepts, both old and new, and discuss related work. Some introductory/summary text is reused from papers co-authored with Ali Ebnenasir [129, 132, 133], in paragraphs 2, 3, and 4 of Chapter 1. Roughly 70% of the original summary content was written by Dr. Ebnenasir, but I have rephrased much of it to fit the context of this work.

For all other content in this dissertation that is co-authored with Dr. Ebnenasir, I wrote the text first and he either commented on or revised it. I have rephrased his heavier revisions simply due to stylistic differences. All protocols, algorithms, and figures in this work are written by me.

Chapter 4 and Chapter 5 build upon content from a conference paper entitled "Synthesizing Self-Stabilization through Superposition and Backtracking" in the *International Symposium on Stabilization, Safety, and Security of Distributed Systems* [130] with an associated technical report [131], and most of the new content is also published as "Shadow/Puppet Synthesis: A Stepwise Method for the Design of Self-Stabilization" in *IEEE Transactions on Parallel and Distributed Systems* [133]. Specifically, Section 4.2, Section 4.3, and Section 4.4 duplicate the content of [133] with minor changes. These minor changes are due to my reformulation of the shadow/puppet concept as introduced in Section 2.2.3. I have rewritten the problem introduction (Section 4.1) and added a discussion of synthesizing probabilistic stabilization (Section 4.5). The following protocols and their descriptions are reproduced from [133]: Protocol 2.2, Protocol 5.1, Protocol 5.3, and Protocol 5.6.

Chapter 6 proves NP-completeness of the problem of designing self-stabilization and nonmasking fault tolerance. The self-stabilization result is based on content from "On the Complexity of Adding Convergence" in *Fundamentals of Software Engineering* [127] and the nonmasking fault tolerance result was published as "On the Hardness of Adding Nonmasking Fault Tolerance" in *IEEE Transactions on Dependable and Secure Computing* [132]. However, I have rewritten this entirely in order to simplify the proofs and make the results more general.

Chapter 7 proves that verifying stabilization is undecidable on unidirectional rings. This was published as "Verifying Livelock Freedom on Parameterized Rings and Chains" in the *International Symposium on Stabilization, Safety, and Security of*

Distributed Systems [129] with associated technical report [128]. Since then, I have simplified the main proof, made the results more general, and added results for synthesis under the guidance of Ali Ebnenasir.

Acknowledgements

This work owes its mathematical rigor to what I learned from Randy Schwartz and Mark Huston at Schoolcraft College. Their math and logic courses revealed the beauty of pairing intuition with proof: to understand the mechanisms at play, and to allow oneself to be humbled by rigor when intuition fails. Thank you Professor Schwartz for pushing me to be a tutor, that confidence and the people there had a huge impact on me.

To complement rigor, I thank Steve Seidel for teaching me when it is safe to spare the reader from unnecessary details. That said, any hand-waving in this work is my own misuse of his teachings.

Finally, thanks to Ali Ebnenasir, Aly Farahat, Mitch Johnson, and other friends from Michigan Tech who have engaged in endless hours of arguments to further our understanding of this world.

Abstract

A protocol is said to be self-stabilizing when the distributed system executing it is guaranteed to recover from any fault that does not cause permanent damage. Designing such protocols is hard since they must recover from all possible states, therefore we investigate how feasible it is to synthesize them automatically. We show that synthesizing stabilization on a fixed topology is NP-complete in the number of system states. When a solution is found, we further show that verifying its correctness on a general topology (with any number of processes) is undecidable, even for very simple unidirectional rings. Despite these negative results, we develop an algorithm to synthesize a self-stabilizing protocol given its desired topology, legitimate states, and behavior. By analogy to shadow puppetry, where a puppeteer may design a complex puppet to cast a desired shadow, a protocol may need to be designed in a complex way that does not even resemble its specification. Our shadow/puppet synthesis algorithm addresses this concern and, using a complete backtracking search, has automatically designed 4 new self-stabilizing protocols with minimal process space requirements: 2-state maximal matching on bidirectional rings, 5-state token passing on unidirectional rings, 3-state token passing on bidirectional chains, and 4-state orientation on daisy chains.

Chapter 1: Introduction

Distributed systems are subject to a variety of faults introduced by bad initialization, loss of coordination, and environmental factors. Due to the complexity of distributed systems, it is expected that faults will occur, therefore they must be designed to anticipate, tolerate, and recover from these faults. Since programming such a system without faults is a difficult task, we look toward automation to design the recovery mechanisms. Particularly, we consider the automatic design of self-stabilization, which provides recovery from all transient faults; i.e., all variables may be corrupted by the faults, but faults eventually stop occurring. When a protocol provides recovery from only certain transient faults, then it is called nonmasking fault-tolerant. Practically, many types of faults can be detected with timeouts or checksums and can be corrected with resets and redundancy [177]. However, not all fault classes can be resolved in such a way, and they necessarily affect the design of a protocol. If faults are not a concern, self-stabilization and nonmasking fault tolerance remain useful as they model distributed algorithms such as token rings [44, 64, 104], leader election [31, 115], and various graph decomposition schemes [30, 57, 109, 154]. Further, self-stabilization remains a theoretically interesting problem as it has applications in several fields such as network protocols , multi-agent systems [94], cloud computing [144], and equilibrium in socioeconomic systems [101].

Several researchers have investigated the problem of adding stabilization and nonmasking fault tolerance to protocols [13, 14, 16, 24, 137, 150]. For instance, Liu and Joseph [150] present a method for the transformation of a fault-intolerant protocol to a fault-tolerant version thereof by going through a set of refinement steps – where a *fault-intolerant* protocol provides no guarantees when faults occur. They model faults by state perturbation, where protocol actions are executed in an interleaving with fault actions. Arora and Gouda [13, 14] provide a unified theory for the formulation of fault tolerance in terms of closure and convergence, where *closure* means that, in the absence of faults, a fault-tolerant protocol remains in a set of legitimate states, and *convergence* specifies that the system eventually recovers to the legitimate states from other states that can be reached due to the occurrence of faults (a.k.a. from the *fault span*). Self-stabilization is formalized as a special case where the fault span is equal to all states of the system. In a shared memory model, Kulkarni and Arora [137] demonstrate that adding failsafe/nonmasking/masking fault tolerance to high atomicity protocols can be done in polynomial time in number of system states, where the

processes of a *high atomicity* protocol can read and write all variables in an atomic step. Nonetheless, they show that, for distributed systems, adding masking fault tolerance is NP-complete (in the number of system states) on an unfair scheduler. The authors of [137] model distribution in a *low atomicity* shared memory model, specifically *composite atomicity*, where each process can atomically read and write only the variables of neighboring processes. Kulkarni and Ebnenasir [139, 140] show that adding failsafe fault tolerance to distributed systems is also an NP-complete problem. However, the complexity of adding nonmasking fault tolerance has remained an open problem for more than a decade. While this problem is known to be in NP, no polynomial-time algorithms are known for efficient design of stabilization or nonmasking fault tolerance. Our work solidifies this fact with a proof of NP-completeness.

A common feature of distributed systems is that they consist of a finite but unbounded number of processes that communicate based on a specific network topology; i.e., *parameterized systems*. There are numerous methods [45, 82, 89, 93] for the verification of safety properties of parameterized systems, where safety requires that nothing bad happens in system executions (e.g., the system does not reach a deadlock). However, methods for verifying liveness properties [38, 86, 92] are often incomplete, where liveness requires that something good eventually happens (e.g., a waiting process eventually gets permission to access a shared resource). Apt and Kozen [11] illustrate that, in general, verifying Linear Temporal Logic (LTL) [77] properties for parameterized systems is Π_1^0-complete. Suzuki [169] shows that the verification problem remains Π_1^0-complete even for unidirectional rings where all processes have a similar code that is parameterized in the number of nodes. Our work solidifies this result for verifying self-stabilization with a proof of Π_1^0-completeness.

Faced with the difficult problem of designing stabilization, most existing methods are either manual [64, 100, 104, 168, 175] or use heuristics [7, 69, 76, 88] that may fail to generate a solution. Gouda and Multari [104] use a method called *convergence stairs* to prove stabilization, where the state space is divided into supersets of the legitimate states. Stomp [168] provides a method based on ranking functions for design and verification of self-stabilization. Methods for algorithmic design of convergence [7, 69, 76, 88] are mainly based on sound heuristics that search through the state space of a non-stabilizing system in order to synthesize recovery actions while ensuring non-interference with closure. Specifically, Abujarad and Kulkarni [6] present a method for algorithmic design of self-stabilization in locally-correctable protocols. Farahat and Ebnenasir [75, 88] present algorithms for the design of self-stabilization in non-locally correctable systems. They also provide a swarm method [76] to exploit computing clusters for the synthesis of self-stabilization. Nonetheless, the aforementioned methods may fail to find a solution while there exists one; i.e., they are sound but incomplete.

1.1 Motivation for Shadow/Puppet Synthesis

The main contribution of this work is the idea of shadow/puppet synthesis, which has allowed us to automatically design 4 new self-stabilizing protocols that improve upon the process space requirements of published work: 2-state maximal matching on bidirectional rings (Section 5.1), 5-state token passing on unidirectional rings (Section 5.2), 3-state token passing on bidirectional chains (Section 5.3), and (effectively 4-state) orientation on daisy chains (Section 5.4). This is partially due to an efficient synthesis algorithm, but it is also due to freedom a designer has in specifying legitimate states and behavior.

In order to illustrate this freedom, consider designing a token passing protocol on a unidirectional ring of N processes. Such a protocol p_{tok} operates correctly when there is exactly one token in the ring, and whenever a process π_i gets the token (for any process index $i \in \mathbb{Z}_N$ where $\mathbb{Z}_N \equiv \{0, \ldots, N-1\}$), it passes the token to the next process π_{i+1} (whose index is computed modulo N). We can express that one token exists using a formula $\mathcal{L}_{\text{tok}}$ where $\mathcal{L}_{\text{tok}} \equiv (\exists! i \in \mathbb{Z}_N : tok_i = 1)$ uses binary variables $tok_0, \ldots, tok_{N-1}$ to indicate which process has a token. Likewise, we can express that each process π_i passes the token to π_{i+1} using the following action for every π_i:

$$\pi_i : \ tok_i = 1 \longrightarrow tok_i := 0; \ tok_{i+1} := 1;$$

An *action* written this way declares a rule for π_i such that it may atomically perform the assignments ($tok_i := 0$ and $tok_{i+1} := 1$) when the action's *guard* ($tok_i = 1$) is satisfied. This *shadow protocol* (token ring) is very natural to specify using *shadow variables* ($tok_0, \ldots, tok_{N-1}$) to establish its legitimate states (one token exists) and legitimate behavior (pass the token).

Next one would construct a *shadow/puppet topology* by revoking read access to unrealistic shadow variables and adding *puppet variables* to the system along with their associated read/write accesses. A shadow/puppet synthesis algorithm would then try to create a shadow/puppet protocol (p'_{tok}) that stabilizes to legitimate states (one token exists) and performs the shadow protocol (pass the token). For example, consider the $tok_0, \ldots, tok_{N-1}$ variables to be unrealistic because they are written across process boundaries. By adding puppet variables $x_0, \ldots, x_{N-1} \in \mathbb{Z}_N$ to the system, we can synthesize protocol p'_{tok} that could match[1] the one originally introduced by Dijkstra [64]. Such a protocol allows π_0 to behave differently than the others, therefore for presentation[2], we give it the name Bot_0 and give each other $\pi_{i \geq 1}$ the name P_i.

[1] In this work we treat variables as having constant domains that do not vary with number of processes. Therefore, while we could synthesize Dijkstra's protocol on any specific ring size, it would not use the new x_i values at larger ring sizes.

[2] Processes in the specification also need to be given names to enforce symmetry.

The actions Dijkstra gives are as follows, with extra assignments to shadow variables.

$$\begin{aligned} Bot_0 :&\ x_{N-1} = x_0 \longrightarrow x_0 := x_0 + 1;\ tok_0 := 0;\ tok_1 := 1; \\ P_i :&\ \ x_{i-1} \neq x_i \longrightarrow x_i := x_{i-1}; \quad tok_i := 0;\ tok_{i+1} := 1; \end{aligned}$$

Herein lies the beauty of the shadow/puppet technique: We have the freedom to specify the shadow protocol p_{tok} using tok_i variables, and a synthesized shadow/puppet protocol p'_{tok} is guaranteed to change each tok_i in same way without actually using them because they are not readable!

1.2 Contributions

In this work, we discuss an algorithm for designing stabilization, the new protocols it has discovered, the hardness of designing stabilization, and the undecidability of verifying stabilization of protocols that are meant to scale to any number of processes. First, we give a simple yet powerful backtracking search algorithm that looks to add self-stabilization to a non-stabilizing protocol. Our search algorithm is an implementation of what we call *shadow/puppet synthesis*, which allows a designer to specify a protocol's desired behavior without being constrained by implementation details. This separation of specification (the shadow) from implementation (the puppet) provides expressive power that other automated methods lack, which is emphasized by 4 newly-discovered protocols that improve upon the space requirements of previously published work. However, such a backtracking search may take exponential time, therefore AI search techniques, complete heuristics, and parallelism are used to mitigate this cost. Next, we show that this exponential cost is likely unavoidable due to NP-completeness. We investigate the complexities of adding convergence, stabilization, and nonmasking fault tolerance properties to a protocol with a fixed topology and find that these problems are NP-complete under most fairness assumptions. In particular, we find that some problems like adding nonmasking fault tolerance are NP-complete even if global fairness is assumed (i.e., where all continuously reachable states are guaranteed to be reached). This sharpens the result of Kulkarni and Arora [137], whose hardness proof for masking fault tolerance utilizes safety properties to restrict the recovery paths, whereas we leverage read restrictions of the topology. Finally, we show that verifying stabilization of very simple unidirectional ring protocols is undecidable. This sharpens the result of Suzuki [169], whose proof relies on having a fixed initial state.

Outline. Chapter 2 introduces and formally defines the major concepts. Chapter 3 summarizes related work. Chapter 4 presents our shadow/puppet synthesis algorithm and its heuristics. Chapter 5 highlights the protocols discovered using this synthesis algorithm. Chapter 6 explores the hardness of adding self-stabilization to a non-stabilizing protocol. Chapter 7 explores the decidability of cycle detection in parameterized ring protocols. Chapter 8 describes our future research directions.

Chapter 2: Concepts

In this chapter, we formally define protocols and their semantics (adapted from [137]), self-stabilization, faults, and fault tolerance (adapted from [14, 64, 67, 99, 100]). We adopt a shared memory model [145] since reasoning in a shared memory setting is convenient, and several (correctness-preserving) transformations [60, 155] exist to refine shared memory fault-tolerant protocols to their message passing versions.

Briefly, a *protocol* (a.k.a. *distributed program*) p defines the behavior for a network $N \geq 1$ of processes (finite state machines). Each process π_i ($i \in \mathbb{Z}_N$, where $\mathbb{Z}_N \equiv \{0, \ldots, N-1\}$) can read or read & write certain variables determined by the network topology (e.g., rings). A *system* is the protocol running on some fixed topology (e.g., a ring of $N = 4$ processes). A *global state* or *configuration* of the system is characterized by a value for each variable. A *global transition* is a change in the system's state, which occurs during normal execution when a process reassigns its writable variables. The *state space* S of a system is the set of all possible global states. A *predicate* is a subset of S, often expressed as a formula over the system variables that evaluates to true (a.k.a. *holds*) if and only if (*iff*) the state is contained in the set.

2.1 Topology and Protocol

Formally, a protocol $p \equiv \langle \mathcal{V}, \Pi, \mathcal{W}, \mathcal{R}, \Delta \rangle$ is a tuple containing its variables ($\mathcal{V}$), its processes (Π), the variables that processes can write and read ($\mathcal{W}$ and $\mathcal{R}$), and its transitions (Δ). Each variable $v_i \subseteq \mathcal{V}$ ($i \in \mathbb{Z}_{|\mathcal{V}|}$) has a symbolic name such as $x_j \equiv v_i$ and has a finite non-empty domain $\textsc{Dom}(v_i)$. Notice that we write $\subseteq$ and $\equiv$ when referencing the variable itself, allowing us to unambiguously write $\in$ and $=$ when using its value. Each process $\pi_i \in \Pi$ ($i \in \mathbb{Z}_{|\Pi|}$) has a symbolic name such as $P_j \equiv \pi_i$, a list of variables that it can write $\mathsf{W}_i \subseteq \mathcal{W}$, a list of variables that it can read $\mathsf{R}_i \subseteq \mathcal{R}$, and a transition function $\delta_i \subseteq \Delta$. When there are no write-only variables, the local behavior of π_i is modeled by a semiautomaton with states $\Gamma_i \equiv \textsc{Dom}(\mathsf{W}_i)$ from read/write variables, an input alphabet $\Sigma_i \equiv \textsc{Dom}(\mathsf{R}_i \setminus \mathsf{W}_i)$ from read-only variables, and a transition function $\delta_i : \Gamma_i \times \Sigma_i \to \Gamma_i$. We also treat Δ directly as a relation that holds all global transitions $\Delta : S \times S$ that can occur due to some local transition function δ_i.

Generalized Topology. Processes with the same name (ignoring the subscripts)

are assumed to have the same transition function. In this way, a protocol for a fixed topology also describes behavior for other instances of the general topology (e.g., rings of different numbers of processes). Such a protocol is also called a *parameterized system* and is formally defined as p^N where N is the number of processes or some other parameter(s) that can be used to construct a fixed topology. A protocol is *generalizable* when it works as desired for all valid instances of a general topology (e.g., all rings of $N \geq 3$ processes).

Example 2.1.1 (3-Coloring)**.** *Choosing different color values on a bidirectional ring.*

As a running example throughout this section, consider a bidirectional ring protocol $p_{\text{color}} \equiv \langle \mathcal{V}_{\text{color}}, \Pi_{\text{color}}, \mathcal{W}_{\text{color}}, \mathcal{R}_{\text{color}}, \Delta_{\text{color}} \rangle$ consisting of N processes named $P_0, \ldots, P_{N-1}$, where each process P_i can read & write a variable $x_i \in \mathbb{Z}_3$ and can also read x_{i-1} and x_{i+1}, where the indices are computed modulo N. The goal of this protocol is for each process in the ring to choose a value, or "color" in the problem's context, that differs from those of its neighbors. We can represent the global states that form a valid coloring with the state predicate $\mathcal{L}_{\text{color}} \equiv \forall i \in \mathbb{Z}_N : x_{i-1} \neq x_i$.

We will use $N = 4$, therefore the variables are $\mathcal{V}_{\text{color}} \equiv (x_0, x_1, x_2, x_3)$, processes are $\Pi_{\text{color}} \equiv (\pi_0, \pi_1, \pi_2, \pi_3) \equiv (P_0, P_1, P_2, P_3)$, writable variables are $\mathcal{W}_{\text{color}} \equiv (\mathsf{W}_0, \mathsf{W}_1, \mathsf{W}_2, \mathsf{W}_3) \equiv ((x_0), (x_1), (x_2), (x_3))$ and readable variables are $\mathcal{R}_{\text{color}} \equiv (\mathsf{R}_0, \mathsf{R}_1, \mathsf{R}_2, \mathsf{R}_3) \equiv ((x_0, x_3, x_1), (x_1, x_0, x_2), (x_2, x_1, x_3), (x_3, x_2, x_0))$. The transition functions $\Delta_{\text{color}} \equiv (\delta_0, \delta_1, \delta_2, \delta_3)$ must all be identical since each process has the same name.

For this protocol, we define the transition function $\delta_i : \Gamma_i \times \Sigma_i \to \Gamma_i$ for each process P_i, where Γ_i are values of x_i and Σ_i are values of (x_{i-1}, x_{i+1}). The actual δ_i is as follows, and Figure 2.1 shows its semiautomaton, where P_i has a transition from state a to state c with label $b = (b_0, b_1)$ *iff* evaluating the transition function yields $c = \delta_i(a, b)$, which can also be written as $(a, b, c) \in \delta_i$.

$$\delta_i(x_i, (x_{i-1}, x_{i+1})) \equiv \begin{cases} (x_i + 1) \bmod 3 & \text{if } (x_{i-1} = x_i \wedge x_i = x_{i+1}) \\ (2 \cdot x_i - x_{i+1}) \bmod 3 & \text{if } (x_{i-1} = x_i \wedge x_i \neq x_{i+1}) \end{cases}$$

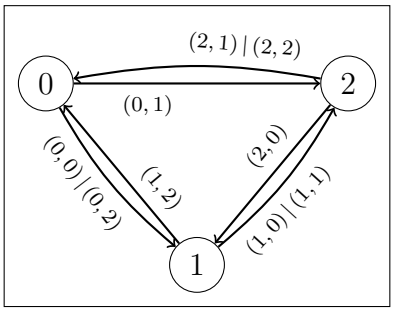


Figure 2.1: Semiautomaton for a process in the example's 3-coloring protocol.

Composition. Protocols are often used in layers to accomplish a complicated task. One way to express these layers is *parallel composition*, which is a union [46] between protocols that simply allows them to act on the same data.

Definition 2.1.2 (Parallel Composition)**.** The *parallel composition* of two protocols p_{A} and p_{B} is a protocol $p_{\mathrm{A}} \parallel p_{\mathrm{B}}$ that preserves the processes of p_{A} and p_{B} (renaming them if necessary) and merges the variables of p_{A} and p_{B} such that variables with the same name are not duplicated (and their domains are merged if necessary).

For a simple example, 3-coloring protocol p_{color} could be written as a parallel composition $p_1 \parallel p_{023}$, where p_1 defines actions of process P_1 using variables x_0, x_1, x_2, whereas p_{023} defines the actions of processes P_0, P_2, P_3 using variables x_0, x_1, x_2, x_3. When the protocols are composed to form p_{color}, their processes are simply placed in the same system.

2.1.1 Actions of a Process

Dijkstra's guarded command language [65] simplifies the presentation of the transitions δ_i of a process π_i. A *guarded command* (a.k.a. *action*) is of the form *guard* $\longrightarrow$ *statement*, where *guard* is a predicate over the readable variables R_i of a process π_i and *statement* assigns values to its writable variables W_i. An action is *enabled* in a global state s *iff* its guard evaluates to true at s. Likewise, a process π_i is *enabled* to act in global state s *iff* at least one of its actions is enabled.

For example, Protocol 2.1 shows how the transition function δ_i of each process P_i in the example 3-coloring protocol p_{color} can be represented by 2 actions, where x_i is assigned modulo its domain size (3).

Protocol 2.1 — 3-Coloring on Bidirectional Rings

$$P_i :\ x_{i-1} = x_i \wedge x_i = x_{i+1} \longrightarrow x_i := x_i + 1;$$
$$P_i :\ x_{i-1} = x_i \wedge x_i \neq x_{i+1} \longrightarrow x_i := 2 \cdot x_i - x_{i+1};$$

Legitimate States: $\mathcal{L}_{\mathrm{color}} \equiv \forall i \in \mathbb{Z}_N : x_{i-1} \neq x_i$

Minimal Actions. Actions of a process π_i can be decomposed into a set of *minimal actions*, where each action is minimal/indivisible in the sense that no other action can be written to represent a subset of its local transitions (in δ_i). Such an action has a guard that tests all readable variables for specific values (one each) and has a statement that assigns (a subset of) writable variables as specific values. When a process has no write-only variables, a minimal action of π_i can be considered to assign all writable variables, and such an action corresponds to exactly one local transition in

δ_i. If write-only variables do exist, then the choice to *not assign* a write-only variable is significant.

The 2 actions of our example 3-coloring protocol p_{color} are not minimal, otherwise we would have to write 9 of them. For example, the first action $(x_{i-1} = x_i \wedge x_i = x_{i+1} \longrightarrow x_i := x_i + 1;)$ of P_i can be decomposed into 3 minimal actions that correspond with local transitions shown in Figure 2.1: $(x_{i-1} = 0 \wedge x_i = 0 \wedge x_{i+1} = 0 \longrightarrow x_i := 1;)$, $(x_{i-1} = 1 \wedge x_i = 1 \wedge x_{i+1} = 1 \longrightarrow x_i := 2;)$, and $(x_{i-1} = 2 \wedge x_i = 2 \wedge x_{i+1} = 2 \longrightarrow x_i := 0;)$.

Since the behavior of any process π_i can be represented as a set of minimal actions, we can represent its possible behavior by enumerating all possible minimal actions. Each readable variable v_j has one of $\textsc{Dom}(v_j)$ possible values in an action's guard and, if it is writable, can be assigned $\textsc{Dom}(v_j)$ different values. Each write-only variable v_j has $\textsc{Dom}(v_j) + 1$ possible ways it can appear in an assignment statement since it has the extra possibility of not being assigned. Therefore, any process π_i has the following number of (possible) minimal actions:

$$\left(\sum_{v_j \in (\mathsf{R}_i \setminus \mathsf{W}_i)} \textsc{Dom}(v_j) \right) \cdot \left(\sum_{v_j \in (\mathsf{W}_i \cap \mathsf{R}_i)} \textsc{Dom}(v_j)^2 \right) \cdot \sum_{v_j \in (\mathsf{W}_i \setminus \mathsf{R}_i)} (\textsc{Dom}(v_j) + 1)$$

Furthermore, if we are given a topology p and are asked to create a protocol p' by giving actions to its processes, then we can do so in 2^m different ways, where m is the total number of possible minimal actions of processes with unique names (which can behave differently).

Deterministic, Self-Disabling Processes. In a later chapter, we argue (Theorem 4.1.4) that the large number of possible ways to design a protocol can be significantly reduced (without sacrificing computational power) by restricting processes to behave deterministically. A process π_i is *deterministic iff* at most one of its actions can be enabled at any time (formally: no two transitions $(w_0, a, w_1), (w_0, a, w_2) \in \delta_i$ exist such that $w_1 \neq w_2$). Some nondeterminism can also be achieved if a process can act multiple times in sequence but chooses to be slow enough for a neighboring process to use one of its intermediate values. A process π_i is *self-disabling iff* it cannot be enabled immediately after acting (formally: no two transitions $(w_0, a, w_1), (w_1, a, w_2) \in \delta_i$ exist).

Write-Only Variables and Self-Loop Removal. We often need to ensure specific values for write-only variables based on the current values to readable variables. This can be done using a transition $(w_0, a, w_1) \in \delta_i$ where only write-only variables are assigned. Logically, adding such a transition also adds a transition (w_1, a, w_1) to δ_i because a process cannot act differently based on unreadable values! Such a transition to the same state is called a *self-loop*, which is undesirable because it implies that π_i is not self-disabling. Therefore, we trivially assume that no self-loops are created by actions that assign write-only variables.

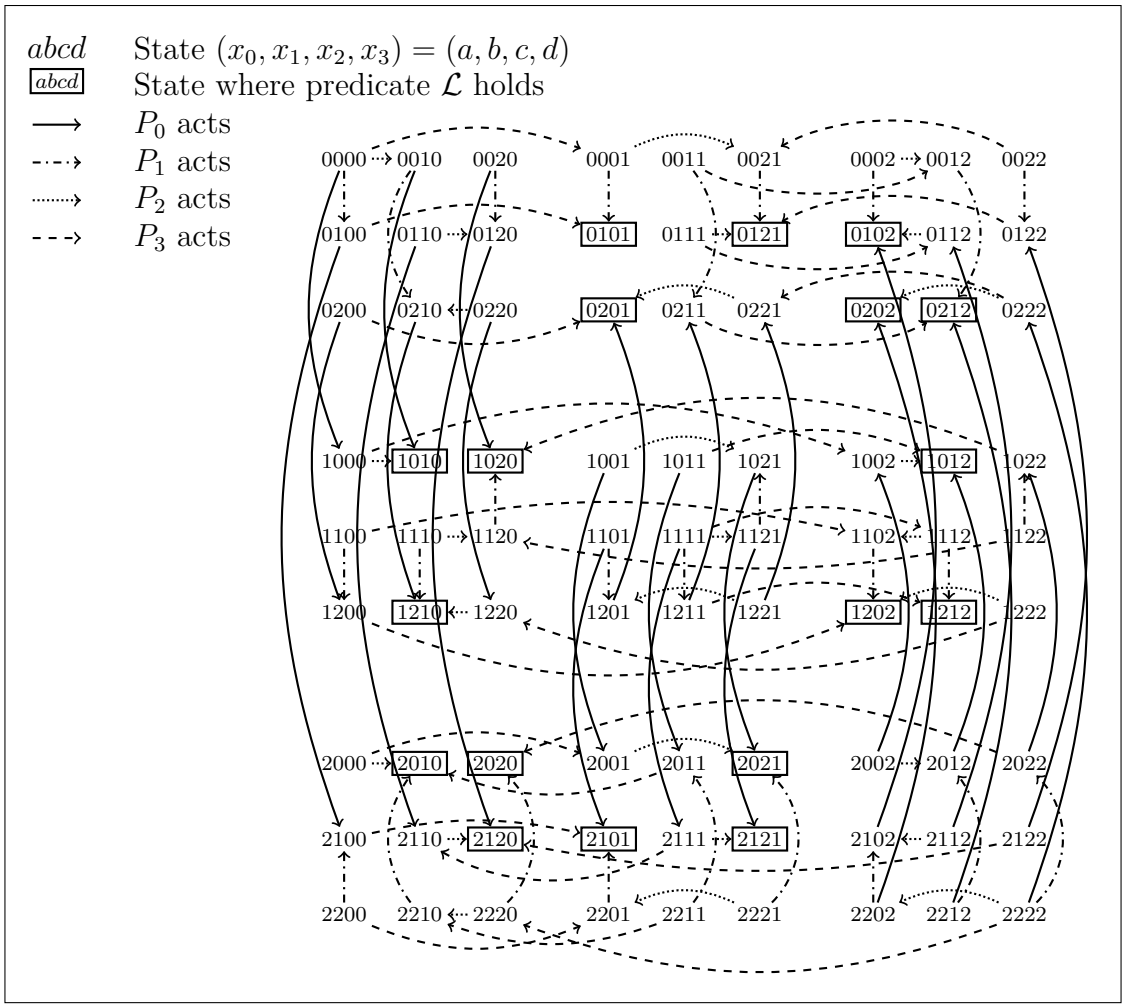


Figure 2.2: Labeled transition system (LTS) Δ_{color} of the 3-coloring protocol on a ring of size 4. Arcs are drawn with the style associated with a process P_i have label i.

2.1.2 Transitions of a Protocol

It is also useful to observe that a local transition (of a process) can correspond to multiple global transitions (of the system). Figure 2.2 depicts the *labeled transition system* Δ for the 3-coloring protocol p_{color} on a ring of size 4. For each $i \in \mathbb{Z}_4$, the set of global transitions δ_i of a process P_i are illustrated as arcs with label i (shown as line styles rather than labels). As such, each process P_i is associated with 27 global transitions. However, as Figure 2.1 shows, each process P_i has only 9 local transitions. This occurs since a process is unable to read the full system state. To see this, consider the action $(x_0 = 0 \wedge x_1 = 0 \wedge x_2 = 2 \longrightarrow x_1 := 1;)$ that corresponds to a single transition of P_1. When the system has $N = 4$ processes as shown in Figure 2.2, then this action corresponds to 3 separate global transitions $(0, 0, 2, 0) \rightarrow (0, 1, 2, 0)$,

$(0,0,2,1) \to (0,1,2,1)$, and $(0,0,2,2) \to (0,1,2,2)$, where each 4-tuple represents a global state (x_0, x_1, x_2, x_3). These global transitions correspond to the 3 possible values of x_3 that P_1 cannot read.

2.1.3 Executions of a Protocol

An *execution* σ of a protocol p is a finite ($\langle s_0, s_1, \ldots, s_k \rangle$) or infinite ($\langle s_0, s_1, \ldots \rangle$) sequence of global states such that for every two consecutive states s_i and s_{i+1} correspond to a global transition of p. Thus, an execution of p corresponds to a walk in the graph of its transition system. A *maximal execution* is either an infinite execution or a finite execution $\langle s_0, s_1, \ldots, s_k \rangle$ such that s_k is *silent*; i.e., it has no outgoing transitions. An execution $\langle s_0, s_1, \ldots, s_k \rangle$ need not end at a silent state s_k, but we will not call this a maximal or finite execution. For example, it is valid to say that all executions of a token ring protocol are infinite, but we can also reason about an execution $\langle s_0, s_1, s_2 \rangle$ where the token held by process π_0 at state s_0 is eventually passed to π_2 at state s_2.

Execution within States. We write $\Delta|Q$ (or sometimes $p|Q$ for convenience) to denote the transitions of a protocol within a state predicate Q. That is, $\Delta|Q \equiv \{(s_0, s_1) \in \Delta : (s_0 \in Q) \wedge (s_1 \in Q)\}$. In our 3-coloring protocol p_{color}, there are no transitions within $\mathcal{L}_{\text{color}}$, therefore $\Delta_{\text{color}}|\mathcal{L}_{\text{color}}$ is empty. However, $\Delta_{\text{color}} \neq \Delta_{\text{color}}|\overline{\mathcal{L}_{\text{color}}}$ because $\Delta_{\text{color}}|\overline{\mathcal{L}_{\text{color}}}$ excludes transitions from $\overline{\mathcal{L}_{\text{color}}}$ to $\mathcal{L}_{\text{color}}$.

2.2 Convergence and Stabilization

Distributed systems often run algorithms to reach certain goals, such as electing a leader or simply agreeing on a value, before they can perform their normal operation. We call such a goal for a protocol the set of *legitimate states* [64], commonly denoted as a state predicate $\mathcal{L}$ (e.g., $\mathcal{L}_{\text{color}}$ forms a coloring in Example 2.1.1). Likewise, a protocol's normal or desired operation is called *legitimate behavior*. A protocol is called *self-stabilizing* when, if started in any arbitrary state, it is guaranteed to eventually reach its legitimate states and behave correctly within those states [67]. Section 2.2.1 introduces stabilization as two components, convergence and closure, without yet considering the behavior within legitimate states. Section 2.2.2 addresses a common case where the legitimate behavior is to halt. Section 2.2.3 formalizes the general case where the legitimate behavior is specified as a protocol. Section 2.2.4 emphasizes that convergence to legitimate behavior implies stabilization to a *subset* of legitimate states, but the implication does not hold when convergence is specified without behavioral constraints. Section 2.2.5 reintroduces stabilization as a form of fault tolerance.

2.2.1 To Legitimate States

Using terms adapted from [13,14,99,100], a protocol p is *closed* [14] within legitimate states $\mathcal{L}$ *iff* it has no transition from $\mathcal{L}$ to $\overline{\mathcal{L}}$. As such, $\mathcal{L}$ is often called an *invariant* of p. A *deadlock* of p is the existence of a *silent* state in $\overline{\mathcal{L}}$; i.e., no process is enabled to act. A *livelock* of p is an infinite execution that never reaches $\mathcal{L}$. Protocol p *converges* to $\mathcal{L}$ *iff* execution from any state will eventually visit $\mathcal{L}$; i.e., when it is free of deadlocks and livelocks. Protocol p *stabilizes* to $\mathcal{L}$ *iff* all executions from all states eventually visit and remain within $\mathcal{L}$; i.e., p converges to $\mathcal{L}$ and is closed within $\mathcal{L}$.

We can show that the 3-coloring protocol p_{color} (Protocol 2.1) stabilizes to $\mathcal{L}_{\text{color}} \equiv \forall i \in \mathbb{Z}_N : x_{i-1} \neq x_i$ on any ring of size N. Closure holds since no process is enabled to change its color when $\mathcal{L}_{\text{color}}$ holds. Convergence holds since the number of instances where $x_{i-1} = x_i$ (for some $i \in \mathbb{Z}_N$) holds is decreased each time a process acts, therefore the system will reach $\mathcal{L}_{\text{color}}$ within at least N transitions. We can also visually check these statements for a ring of size $N = 4$ using Figure 2.2 by observing that (1) no transition exists from legitimate states, and (2) $\overline{\mathcal{L}_{\text{color}}}$ does not contain silent states (deadlocks) or cycles (livelocks).

Definition 2.2.1 (Closure)**.** A protocol p is *closed* within states $\mathcal{L}$ *iff* all transitions from $\mathcal{L}$ are to $\mathcal{L}$.

Definition 2.2.2 (Convergence)**.** A protocol p *converges* to states $\mathcal{L}$ *iff* no deadlocks or livelocks exist in $\overline{\mathcal{L}}$.

Definition 2.2.3 (Stabilization)**.** A protocol p *stabilizes* (a.k.a. *self-stabilizes*) to states $\mathcal{L}$ *iff* it converges to $\mathcal{L}$ and is closed within $\mathcal{L}$.

Unlike the original definition of convergence [14] to some $\mathcal{L}$, our definition does not require closure within $\mathcal{L}$. This matches other work [47, 83] and is particularly convenient when using a nuanced version of closure [43, 99, 103, 130] that is incompatible with the well-established version of closure implied by self-stabilization [14, 44, 64]. Convergence to $\mathcal{L}$ may also be specified *from* a set of states Q, where our definition again deviates from [14] yet again because Q need not be closed. Stabilization to $\mathcal{L}$ from Q is defined similarly but requires closure within Q.

Definition 2.2.4 (Convergence from States)**.** A protocol p converges to states $\mathcal{L}$ from states Q *iff* it is deadlock-free and livelock-free within $Q \setminus \mathcal{L}$ and has no transition from there to $\overline{Q} \setminus \mathcal{L}$.

Definition 2.2.5 (Stabilization from States)**.** A protocol p stabilizes to states $\mathcal{L}$ from states Q *iff* it converges to $\mathcal{L}$ from Q and is closed within Q and $\mathcal{L}$.

Definition 2.2.4 can be used to reason about the behavior of a protocol p as convergence from Q_k to Q_{k+1} for each Q_k in a sequence of state predicates $Q_0, \ldots, Q_n$.

Clearly any execution from a state in $Q_0 \cup \ldots \cup Q_n$ must eventually reach Q_n, therefore we can say that $Q_0 \cup \ldots \cup Q_n$ converges to Q_n. Gouda and Multari use this concept of *convergence stairs* with p closed within Q_n in order to prove stabilization [104].

Lemma 2.2.6 (Convergence Stairs)**.** *A protocol p converges to states $\mathcal{L}$ from states Q iff some n predicates $Q_0, \ldots, Q_{n-1}$ exist such that $Q = \bigcup_{i=0}^{n-1} Q_i$ and for every $i \in \mathbb{Z}_n$, p converges from Q_i to some Q_j where $i < j \leq n$, letting $Q_n = \mathcal{L}$ for convenience.*

2.2.2 To Silent States

When we are tasked with constructing a self-stabilizing protocol, its behavior within the legitimate states $\mathcal{L}$ also becomes a constraint. In Example 2.1.1, we implicitly assumed that no color should change when a valid coloring is reached ($\mathcal{L}_{\text{color}} \equiv \forall i \in \mathbb{Z}_N : x_{i-1} \neq x_i$). This behavior is reasonable since the particular colors may have a meaning that should not change. For example, if a higher layer of the system involves bulk data transfer (and also has a clock), each color x_i could indicate a time slice that process P_i can safely send data in bulk to its neighbors. Furthermore, each P_i would also know when to listen for incoming data from P_{i-1} and P_{i+1} based on their colors x_{i-1} and x_{i+1}. A bulk transmission may therefore be lost if a color changes within $\mathcal{L}$, which is clearly illegitimate behavior but is allowed by Definition 2.2.3. We therefore use the term *silent stabilization* [68] for stabilizing protocols that should have no transitions within legitimate states.

Definition 2.2.7 (Silent Stabilization)**.** A protocol p *silent-stabilizes* to legitimate states $\mathcal{L}$ *iff* it converges to $\mathcal{L}$ and all states in $\mathcal{L}$ are silent.

With the problem of coloring still in mind ($\mathcal{L}_{\text{color}} \equiv \forall i \in \mathbb{Z}_N : x_{i-1} \neq x_i$), suppose we design a protocol p that silent-stabilizes to $\mathcal{L}' \equiv \forall i : (x_i = \min(\mathbb{Z}_3 \setminus \{x_{i-1}, x_{i+1}\}))$, where each color x_i is the smallest possible value that differs from x_{i-1} and x_{i+1}. Is p a self-stabilizing coloring protocol? Every state in $\mathcal{L}' \subset \mathcal{L}_{\text{color}}$ is a valid coloring, therefore p is silent-stabilizing to a set of colorings. However, p is not silent-stabilizing to the set of *all* colorings because it has transitions within $\mathcal{L}_{\text{color}}$. Furthermore, p may not even be closed within $\mathcal{L}_{\text{color}}$, in which case p would not even be stabilizing to $\mathcal{L}_{\text{color}}$! These technicalities are inconsequential when the coloring is used to resolve resource contention, where we indeed only need to silent-stabilize to a *subset* of legitimate states [103, 137]. For such protocols, we use the term *silent convergence*.

Definition 2.2.8 (Silent Convergence)**.** A protocol p *silent-converges* to states $\mathcal{L}$ *iff* it converges to some $\mathcal{L}' \subseteq \mathcal{L}$ where all states in $\mathcal{L}'$ are silent.

2.2.3 To a Shadow Protocol

In the example of Section 1.1, we saw that the specification of legitimate behavior can itself be a protocol with some legitimate states where it operates correctly. In

that example, we used a token ring p_{tok} for behavior and legitimate states $\mathcal{L}_{\text{tok}}$. For the synthesized protocol shadow/puppet protocol p'_{tok}, we removed read access to tok_i variables and added x_i variables to the system. This section begins by defining in general how such protocols p_{tok} and p'_{tok} must be related in order for p'_{tok} to make sense as an implementation for p_{tok}. Namely, a projection function $\mathcal{H}$ must exist such that, given a state of p'_{tok}, it can compute a state of p_{tok} by removing the x_i values. Then, the concepts convergence, closure, and stabilization to behavior (written as $p_{\text{tok}}|\mathcal{L}_{\text{tok}}$) are defined. Lastly, we give an example showing how the shadow vs puppet idea resembles that of superposition [46], but rather than being a composition where the shadow is layer of functionality assumed by the puppet, the puppet exists only to enhance the shadow with stabilization properties.

Definition 2.2.9 (Shadow/Puppet Protocol)**.** A *shadow/puppet protocol* $p' \equiv \langle \mathcal{V}', \Pi', \mathcal{W}', \mathcal{R}', \Delta' \rangle$ with projection function $\mathcal{H}$ has a topology and actions that are compatible with a *shadow protocol* $p \equiv \langle \mathcal{V}, \Pi, \mathcal{W}, \mathcal{R}, \Delta \rangle$ that operates within legitimate states $\mathcal{L}$ *iff* it satisfies the following constraints:

1. Preserve shadow variables (may add puppet variables): $\mathcal{V} \subseteq \mathcal{V}'$
   - $\mathcal{H}$ maps states of p' to states of p by removing puppet variables
2. Preserve process names (may add new processes): $\Pi \subseteq \Pi'$
3. Preserve write access to shadow variables: $\forall \mathsf{W}'_i \in \mathcal{W}' : ((\mathsf{W}'_i \cap \mathcal{V}) = \mathsf{W}_i)$
4. Limit read access to shadow variables (may revoke): $\forall \mathsf{R}'_i \in \mathcal{R}' : ((\mathsf{R}'_i \cap \mathcal{V}) \subseteq \mathsf{R}_i)$
5. Respect shadow actions: For each minimal action in p', either
   No write-only shadow variable is assigned
   or The same shadow variable assignment is used by the same process of p
   or All write-only shadow variables are assigned such that the resulting state may be a silent state of p in $\mathcal{L}$

The last constraint of Definition 2.2.9 ensures that the relation is clear between actions of the puppet protocol and actions/states of the shadow protocol. For example, even though an action $(x_{i-1} = 0 \wedge x_i = 1 \longrightarrow x_i := 0;\ tok_i := 0;)$ of P_i in p'_{tok} may make sense if it is only used when converging to $\mathcal{L}_{\text{tok}}$, we require it to either not assign write-only shadow variables at all or to match the shadow action $(tok_i = 0 \longrightarrow tok_i := 0;\ tok_{i+1} := 1;)$ by assigning both $tok_i := 0$ and $tok_{i+1} := 1$ (see Section 5.3 for an example). If the shadow protocol is silent in some legitimate states (usually it is all or none), then we also allow actions that assign all write-only variables. Effectively, this gives us a mapping from the readable values to the write-only values (see Section 5.1 for an example).

Definition 2.2.10 (Shadow Convergence)**.** A protocol $p' \equiv \langle \mathcal{V}', \Pi', \mathcal{W}', \mathcal{R}', \Delta' \rangle$ converges to $p|\mathcal{L}$ (a.k.a. $\Delta|\mathcal{L}$), where $p \equiv \langle \mathcal{V}, \Pi, \mathcal{W}, \mathcal{R}, \Delta \rangle$ operates within legitimate states $\mathcal{L}$ and is a shadow protocol for p' using projection function $\mathcal{H}$, *iff* p' satisfies the following conditions:

1. Recovery: p' converges to some states $\mathcal{L}' \subseteq \mathcal{H}^{-1}[\mathcal{L}]$
2. Progress: $\forall s_0 \in \mathcal{L}$ with $n \geq 1$ outgoing transitions $(s_0, s_1), \ldots, (s_0, s_n) \in \Delta|\mathcal{L}$:
$$p' \text{ converges from } \mathcal{L}' \cap \mathcal{H}^{-1}[\{s_0\}] \text{ to } \bigcup_{j=1}^{n} (\mathcal{L}' \cap \mathcal{H}^{-1}[\{s_j\}])$$
3. Fixpoint: $\forall s_0 \in \mathcal{L}$ with no outgoing transition in $\Delta|\mathcal{L} \setminus \{(s_0, s_0)\}$:
$$p' \text{ is closed within } \mathcal{L}' \cap \mathcal{H}^{-1}[\{s_0\}]$$

In short, convergence to shadow behavior $p|\mathcal{L}$ provides stabilization to a subset $\mathcal{L}'$ of $\mathcal{H}^{-1}[\mathcal{L}]$, where the shadow variables will change as they can in p. Convergence to $p|\mathcal{L}$ also guarantees that within $\mathcal{L}'$, p' changes shadow variables as they are changed by transitions of p within $\mathcal{L}$ (allowing intermediate self-loops), but there is no guarantee that all transitions of p are preserved. This has not been an issue in our work because the shadow protocols we consider either have at most one transition from any legitimate state (e.g., silent protocols and token rings). The closure property defined below aims to require this in its behavioral (second) constraint, but since we have not used shadow protocols where multiple transitions can be taken from a legitimate state, we give a simple "placeholder" constraint.

Definition 2.2.11 (Shadow Closure)**.** A protocol $p' \equiv \langle \mathcal{V}', \Pi', \mathcal{W}', \mathcal{R}', \Delta' \rangle$ is closed within $p|\mathcal{L}$ (a.k.a. $\Delta|\mathcal{L}$), where $p \equiv \langle \mathcal{V}, \Pi, \mathcal{W}, \mathcal{R}, \Delta \rangle$ operates within legitimate states $\mathcal{L}$ and is a shadow protocol for p' using projection function $\mathcal{H}$, *iff* p' satisfies the following conditions:

1. Preserve shadow states: p' is closed within some states $\mathcal{L}'$ such that $\mathcal{L} = \mathcal{H}[\mathcal{L}']$
2. Preserve shadow transitions: $\Delta|\mathcal{L} = \{(s_0, s_1) \in \mathcal{H}[\Delta'|\mathcal{L}'] : s_0 \neq s_1\}$
   - A stricter version would preserve all execution paths: $\forall (s_0, s_1), (s_1, s_3) \in \Delta|\mathcal{L} : \forall (s'_0, s'_1) \in \Delta'|\mathcal{L}' :$ $\Big($if $\mathcal{H}(s'_0) = s_0$ and $\mathcal{H}(s'_1) = s_1$, then there exists a state $s'_3 \in \mathcal{H}^{-1}[\{s_3\}]$ that can be reached by an execution of p' from s'_1 such that $\mathcal{H}(s'_2) = s_1$ holds for each intermediate state $s'_2\Big)$

Definition 2.2.12 (Shadow Stabilization)**.** A protocol $p' \equiv \langle \mathcal{V}', \Pi', \mathcal{W}', \mathcal{R}', \Delta' \rangle$ stabilizes to $p|\mathcal{L}$ *iff* p' converges to, and is closed within, $p|\mathcal{L}$.

Example 2.2.13 (4-State Token Ring)**.** *Constructing a stabilizing 4-state token ring protocol from a 2-state token ring protocol by adding a binary variable to each process.*

The shadow variables of a system do not necessarily need to be write-only. Such a case may be easier to analyze because the shadow variables are still present, therefore we do not have to extract the meaning of puppet variables. Since both shadow

and puppet variables are present in the system, it more closely matches the idea of superposition from Chandy and Misra [46]. However, superposition in [46] is a composition technique that allows a higher protocol layer (our puppet variables) to be built upon a lower protocol layer (our shadow variables). In our work, the shadow protocol indeed operates correctly within legitimate states, but we use the puppet layer to add convergence functionality.

In this example, we specify a token ring using a non-stabilizing 2-state token ring p_{tok2}. Each process π_i owns a binary variable t_i and can read t_{i-1}. The first process π_0 is distinguished as Bot_0, in that it acts differently from the others, which are named P_i (for $i > 0$). Bot_0 has a token when $t_{N-1} = t_0$ and each other process $P_{i>0}$ is said to have a token when $t_{i-1} \neq t_i$. Bot_0 has action $(t_{N-1} = t_0 \longrightarrow t_0 := 1 - t_0;)$, and each other process $P_{i>0}$ has action $(t_{i-1} \neq t_i \longrightarrow t_i := t_{i-1};)$.

$$\begin{aligned} Bot_0 &: (x_{N-1} = x_0) \longrightarrow x_0 := 1 - x_0; \\ P_i &: (x_{i-1} \neq x_i) \longrightarrow x_i := 1 - x_i; \end{aligned}$$

Let $\mathcal{L}_{\text{tok2}}$ denote the legitimate states where exactly one process has a token:

$$\mathcal{L}_{\text{tok2}} \equiv \exists! i \in \mathbb{Z}_N : ((i = 0 \wedge t_{i-1} = t_i) \vee (i \neq 0 \wedge t_{i-1} \neq t_i))$$

To transform this protocol to a self-stabilizing version p_{tok4}, we add a binary puppet variable x_i to each process π_i. Each process π_i can also read its predecessor's variable x_{i-1}. Let $\mathcal{L}_{\text{tok4}} \equiv \mathcal{H}^{-1}[\mathcal{L}_{\text{tok2}}]$ be the legitimate states of this transformed protocol. Let p_{tok4} be defined as Protocol 2.2, which gives the actions for Bot_0 and every other process $P_{i>0}$.

Protocol 2.2 — 4-State Token Ring (Not Generalizable)

$$\begin{aligned} Bot_0 &: (x_{N-1} = x_0) \wedge (t_{N-1} \neq t_0) \longrightarrow x_0 := 1 - x_0; \\ Bot_0 &: (x_{N-1} = x_0) \wedge (t_{N-1} = t_0) \longrightarrow x_0 := 1 - x_0;\ t_0 := x_{N-1}; \\ P_i &: (x_{i-1} \neq x_i) \wedge (t_{i-1} = t_i) \longrightarrow x_i := 1 - x_i; \\ P_i &: (x_{i-1} \neq x_i) \wedge (t_{i-1} \neq t_i) \longrightarrow x_i := 1 - x_i;\ t_i := x_{i-1}; \end{aligned}$$

Shadow Variables: $t_0 \ldots t_{N-1} \in \mathbb{Z}_2$ (read & write)
Puppet Variables: $x_0 \ldots x_{N-1} \in \mathbb{Z}_2$
Legitimate States: $\mathcal{L}_{\text{tok4}} \equiv \mathcal{H}^{-1}[\mathcal{L}_{\text{tok2}}] \equiv \exists! i \in \mathbb{Z}_N : (\quad i = 0 \wedge t_{i-1} = t_i \vee i \neq 0 \wedge t_{i-1} \neq t_i)$

This protocol is self-stabilizing for all rings of size $N \in \{2, \ldots, 7\}$ but contains a livelock when $N = 8$. In fact, using the complete search algorithm in Chapter 4, we will find that no such protocol is stabilizing for all $N \in \{2, \ldots, 8\}$. Gouda and

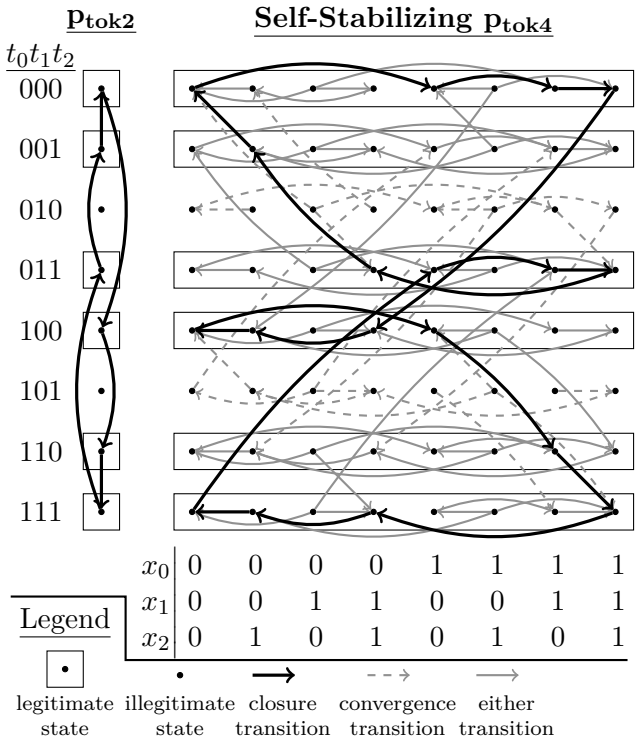


Figure 2.3: Transition systems of the non-stabilizing 2-state and self-stabilizing 4-state token rings of size $N = 3$. © 2016 IEEE [133].

Haddix [102] give a similar token ring protocol that stabilizes for all ring sizes. They introduce another binary variable $ready_i$ to each process.

Let us check that the superposition preserves the 2-state token ring protocol for a ring of size $N = 3$. Figure 2.3 shows the transition system of the non-stabilizing 2-state protocol p_{tok2} within $\mathcal{L}_{\text{tok2}}$ and the self-stabilizing 4-state protocol p_{tok4}, where each state is a node and each arc is a transition. Legitimate states are boxed and reoccurring transitions within these states are black. Recovery transitions are drawn with dashed gray lines. Solid gray lines denote transitions within our maximal choice of $\mathcal{L}_{\text{tok4}}$ but would serve as recovery transitions for smaller choices of $\mathcal{L}_{\text{tok4}}$. Verifying the conditions for closure (Definition 2.2.11), we find: (1) shadow states are preserved because $\mathcal{L}_{\text{tok4}} = \mathcal{H}^{-1}[\mathcal{L}_{\text{tok2}}]$ (even though we only really need $\mathcal{L}_{\text{tok2}} = \mathcal{H}[\mathcal{L}_{\text{tok4}}]$), and (2) transitions are preserved because each transition of p_{tok2} within $\mathcal{L}_{\text{tok2}}$ corresponds to a (vertical) transition of p_{tok4} within $\mathcal{L}_{\text{tok4}}$. Verifying the conditions for convergence (Definition 2.2.10), we find: (1) convergence to $\mathcal{L}_{\text{tok4}}$ holds because there are no deadlocks or livelocks in $\overline{\mathcal{L}_{\text{tok4}}}$, (2) progress is preserved within $\mathcal{L}_{\text{tok4}}$ since each boxed row of p_{tok4} converges to the boxed row corresponding to the next state of p_{tok2}, and (3) the fixpoint case is satisfied vacuously since the shadow variables will always change.

2.2.4 To a Subset of States

In the previous two sections, we saw how stabilization to a subset of legitimate states $\mathcal{L}$ could be a useful property. In Section 2.2.2, silent convergence to $\mathcal{L}$ is equivalent to silent stabilization to some subset of $\mathcal{L}$. Likewise in Section 2.2.3, convergence to shadow behavior $p|\mathcal{L}$ is equivalent to stabilization to some subset of transitions of p within a subset of $\mathcal{L}$ (though each non-silent state must retain *some* transition). We would like to complete this pattern for Section 2.2.1 by having a property that denotes stabilization to a subset of states $\mathcal{L}$ [99, 103] without constraining behavior. In terms of temporal logic, this type of stabilization is a fundamentally simple property written as $\Diamond \Box \mathcal{L}$; i.e., *eventually* it is *always* the case that $\mathcal{L}$ holds.

Definition 2.2.14 (Eventually Always)**.** Given a protocol p, a state predicate $\mathcal{L}$ *eventually always* holds from every state *iff* p stabilizes to a subset of $\mathcal{L}$.

In Terms of Shadow Convergence. Notice that silent convergence to $\mathcal{L}$ can be rephrased as convergence to $\emptyset^2|\mathcal{L}$, where the projection function $\mathcal{H}$ is assumed to not remove any variables. Likewise, the eventually always property can be rephrased as convergence to $\mathcal{L}^2|\mathcal{L}$. Since $\mathcal{L}^2$ contains all transitions (including self-loops), convergence to $\mathcal{L}^2|\mathcal{L}$ allows all behavior (including silent states). It may therefore be convenient to think of convergence to behavior as a general way to express stabilization to a subset of legitimate states/behavior. However, for clarity, we will not use these alternative notations for silent convergence to $\mathcal{L}$ or stabilization to a subset of $\mathcal{L}$.

2.2.5 From Transient Faults

Due to the nature of distributed programs, which have many points of failure, we must anticipate and tolerate *faults* that bring the system to these bad states. For example, consider a compute cluster simulating the temperature of a building throughout a day, where each process updates values for certain elements on a mesh at each time step. If one process suddenly crashes and restarts using an old time step, it may be simulating nighttime while all other processes are simulating the sun shining overhead. To tolerate this kind of process crash and prevent an abnormally cold spot in the simulation's result, a process could occasionally consult its neighbors to ensure that its time step is consistent. Since this process crash does not cause permanent damage, it can be modeled as a *transient fault*, which merely perturbs the state of the system. Transition systems can model many types of faults by adding fault transitions. Chen and Kulkarni classify these faults and explore modeling techniques [48].

Fault tolerance is a general framework where the system need only recover from a given fault class f_{trans}. For the specific kind of fault tolerance we consider, f_{trans} represents the anticipated transient faults, which can be represented as a set of transitions. Like the transitions of a protocol, a set of faults can be specified with guarded commands.

Given legitimate states $\mathcal{L}$ and a protocol p with transitions Δ, the *fault span* for fault class f_{trans} is the set of states reachable from $\mathcal{L}$ by the transitions $\Delta \cup f_{\text{trans}}$.

Definition 2.2.15 (Nonmasking Fault Tolerance)**.** Given a class of faults f_{trans}, a protocol p is *nonmasking fault-tolerant* to legitimate states $\mathcal{L}$ *iff* p stabilizes from its fault span to $\mathcal{L}$.

Notice that stabilization is a special case of fault tolerance, where the fault class f_{trans} is the set of all transitions $f_{\text{trans}} \equiv (\mathcal{L} \times \overline{\mathcal{L}})$ and the fault span contains all states. Stabilization can therefore be harder or impossible to design in some cases, but there is no risk of unanticipated transient faults.

Fault Tolerance with Safety Constraints. It is often desirable to enforce or check safety conditions, which are assertions whose counterexamples take the form of a finite execution. For example, the safety properties of stabilization are deadlock freedom and closure. In contrast, livelock freedom is a liveness property since a counterexample is an infinite execution. Many safety properties can be specified by a set of forbidden transitions [141]. Using this idea, the concept of convergence can be extended to *safe convergence* [122], where certain transitions are disallowed during recovery. When this idea is applied to fault tolerance, it is called *masking fault tolerance*, since the safety specification is said to "mask" faults during recovery. Similarly, *failsafe fault tolerance* [120, 141] guarantees that a system avoids bad behavior in the presence of faults, but recovery is not guaranteed. In cases where bad behavior could be catastrophic, we may just prefer the corrupted process or system to halt/crash [55].

2.3 Scheduling Daemon

As introduced in Section 2.1.3, an execution of a protocol corresponds to a walk in the graph of the protocol's transition system. While we can often assume that all walks, including infinite ones, correspond to valid executions, it is not always realistic. For example, imagine that a ring protocol p_{service} relies on a coloring in order to operate correctly. Composing a coloring protocol p_{color} such as the one from Example 2.1.1 should provide such a coloring. However, the transition system of $p_{\text{service}} \parallel p_{\text{color}}$ may contain cycles where only processes of p_{service} execute even though some process of p_{color} is enabled to act. This is an issue of fairness, which is an additional constraint $\mathcal{F}$ of a system. We often assume *no fairness* ($\mathcal{F} =$ unfair), where all infinite walks are valid, because it simplifies analysis [47].

Theorem 2.3.1 (Unfair Cycle)**.** *An infinite execution of a protocol p under no fairness ($\mathcal{F} =$ unfair) is characterized by an execution $\langle s_0, \ldots, s_{k-1}, s_0 \rangle$ that revisits a state s_0 after some $k \geq 1$ steps.*

Proof. Since we model protocols as a finite number of finite-state processes, p has a finite number of global states. Thus, an infinite execution must eventually revisit

some state s_0. The ability to revisit s_0 corresponds to a cycle in the transition system of p, therefore a valid execution under no fairness can repeat such a cycle forever. □

2.3.1 Fairness

However, $p_{\text{service}} \parallel p_{\text{color}}$ requires some (any) amount of fairness to guarantee that p_{color} will eventually assign the color variables as needed. *Weak fairness* ($\mathcal{F} = \text{weak}$) is a natural assumption that guarantees that if a process is continuously enabled, then it will eventually act. *Local fairness* ($\mathcal{F} = \text{local}$) guarantees that if a process does not become continuously disabled, then it will eventually act. In some contexts, weak and local fairness are applied to actions or local transitions, which weakens weak fairness and strengthens local fairness. *Global fairness* ($\mathcal{F} = \text{global}$) guarantees that if any global transition does not become continuously disabled, then it will eventually occur in the execution. This definition of global fairness corresponds to Gouda's "strong fairness" [100]. We make this explicit since some others use the term "strong fairness" to describe local fairness [123].

Definition 2.3.2 (Weak Fairness)**.** An infinite execution is valid under weak fairness ($\mathcal{F} = \text{weak}$) *iff* every process that becomes enabled eventually acts or becomes disabled in some future state.

Definition 2.3.3 (Local Fairness)**.** An infinite execution is valid under local fairness ($\mathcal{F} = \text{local}$) *iff* every process that is enabled infinitely often also acts infinitely often.

Definition 2.3.4 (Global Fairness)**.** An infinite execution is valid under global fairness ($\mathcal{F} = \text{global}$) *iff* every global transition that is enabled infinitely often is executed infinitely often.

Global fairness ensures that if a global state remains reachable during an execution, then it will eventually be reached. This can greatly simplify reasoning about convergence and stabilization in particular [47, 88], where illegitimate state need only provide reachability to a legitimate state. Stabilization under global fairness is called *weak stabilization* due to Gouda [100].

2.3.2 Execution Semantics

Unless otherwise specified, each state change corresponds with an action of a single process. This assumption is called *interleaving semantics* (a.k.a. execution under a *central daemon*). In contrast, *synchronous semantics* (a.k.a. execution under a *synchronous daemon*) forces all enabled processes to act simultaneously. Further, *subset semantics* (a.k.a. execution under a *distributed daemon*) allows any non-empty subset of processes to simultaneously act.

Since the synchronous case resembles a scheduling policy, we treat it as a type of "fairness" corresponding to $\mathcal{F} = \text{sync}$. As with any other $\mathcal{F} \in \{\text{unfair}, \text{weak}, \text{local}\}$

that is not global, if a process has more than one local transition enabled at the same time under the synchronous scheduler, then it chooses nondeterministically (i.e., unfairly).

Definition 2.3.5 (Synchronous Scheduler)**.** An execution is valid under a synchronous scheduler ($\mathcal{F} = \text{sync}$) *iff* every enabled process acts atomically at each step.

2.3.3 Probabilistic Processes

When a process has multiple actions enabled, every fairness model except for global fairness treat the nondeterministic choice between these actions as the worst possible choice. However, global fairness is a much stronger assumption [63], therefore we would like a different way for a process to choose fairly between actions. *Probabilistic stabilization* [110] captures this idea by associating each action that is enabled with a probability to be chosen. Such a protocol is said to *almost surely* stabilize *iff* the probability of stabilization equals 1 as time goes to infinity. Since this idea only affects fairness within a single process, it can be used with any of the other fairness models that place no restriction on nondeterministic choice ($\mathcal{F} \in \{\text{unfair}, \text{weak}, \text{local}, \text{sync}\}$), and it is redundant when used with global fairness except when analyzing recovery time [85].

Most authors treat the concept of random choice as a scheduler extension (a.k.a. *probabilistic daemon*), but we do not. Instead, we grant a process π_i the ability of random choice by giving it read-only access to a randomized variable rng_i, which we treat as taking a new (equally likely) random value at every execution step of the system. In this way, π_i can reference this source of entropy in its action's guards while essentially remaining deterministic! Section 4.5 gives insight into why we take this approach.

Chapter 3: Related Work

In this chapter, we first look at the reductions to and from our chosen model of computation. Then we review work related to the verification and design of self-stabilizing protocols on fixed topologies. Lastly, we consider the verification and design of parameterized self-stabilizing protocols.

3.1 System Model

Processes of a system could be threads in a program, programs on a single computer, computers on the Internet, sensors communicating via an ad-hoc wireless network, or hardware components communicating via a bus. Our preferred method of modeling from Chapter 2 is one of many well-studied formalisms [145], and this section shows how it relates and reduces to other formalisms.

3.1.1 Communication

Processes in Chapter 2 communicate by atomically reading and writing variables. Specifically, we use the shared-variable (shared memory) communication model under the central daemon using composite atomicity.

A special case of the shared-variable model is the *state-reading model*, which provides a conceptually simpler view of network topologies. In the state-reading model, each process P_i owns a single variable x_i that defines its state, and the topology defines which other x_j variables P_i can read. This distinction from a shared-variable model is trivial in some cases, but it can force processes to access more information than necessary. Further, the state-reading model is strictly less powerful than the shared-variable model since it introduces impossibility for ring orientation [112, 117].

Another special case of the shared-variable model is the *link-register model* that stipulates that any pair of adjacent processes P_i and P_j communicate via a link where P_i has an output register that can be read by P_j and vice versa. If a process P_i wants to share a value with two neighbors, it must use two output registers. Contrast this with the shared-variable model, which allows us to simply grant the adjacent processes of P_i access to read P_i's variable.

Contrasting with the central daemon, the *distributed daemon* loosens the atomicity restrictions on adjacent processes. Rather than one process acting, multiple

processes can act simultaneously. However, this daemon makes several deterministic self-stabilizing protocols impossible, but this can be remedied with randomization [105, 117, 163] or unbounded memory [23, 25]. Goddard, Hedetniemi, Jacobs, and Srimani [97] show how randomization can be used to convert any self-stabilizing protocol under the central daemon to a self-stabilizing version under the distributed daemon.

To model delay, the *fully distributed daemon* introduces the possibility of delaying a state change. That is, when a process π_i acts, it reads all variables atomically and writes its variables atomically, but the reads and writes may occur at different steps. While the write operation of a process is delayed, the process itself is blocked from performing any further actions. When the updates do occur for the variables of π_i, all values are updated atomically and π_i is no longer blocked.

Finally, the most restrictive daemon is the *read/write daemon*, which chooses a single processor to read or write exactly one variable. This is also called *read/write atomicity*, as opposed to composite atomicity. This models both delay and inconsistent viewpoints found in real shared memory systems, but it complicates both automatic and manual reasoning. Fortunately, we can leverage the methods of Dolev [67] and Nesterenko and Arora [155] to automatically transform a stabilizing protocol using composite atomicity into a stabilizing protocol using read/write atomicity.

3.1.1.1 Message Passing

A natural model for communication is message passing, where a process sends messages to others, but the messages take some time to arrive at their destinations. Along the way, a message may be lost, take a longer route than others, or have its contents corrupted. Corrupted messages are always possible, but are avoided with high probability by using checksums and error correction. The User Datagram Protocol (UDP) provides a minimal abstraction to communicate via message passing, but the programmer must account for message loss and reordering. The Transmission Control Protocol (TCP) is similar but tolerates these faults using sequence numbers, acknowledgment packets, and retries, but repeated failures present themselves as timeouts to the programmer. Faults and system assumptions complicate the analysis of self-stabilizing protocols built upon message passing. Fortunately, methods exist to transform a self-stabilizing protocol in the shared memory model to use message passing without sacrificing stabilization properties [60, 114, 155].

3.1.1.2 Point-to-Multipoint

Some systems use a shared physical medium or channel for communication such as Wi-Fi or an internal bus. For generality, let us assume that processes corrupt their messages if they broadcast to the channel simultaneously (e.g., Ethernet in half-duplex mode). These collisions can be avoided with the help of clocks and randomization, where processes can randomly contend for channel access and coordinate access times

with each other. Kulkarni and Arumugam [138] give a stabilization-preserving transformation from the read/write atomicity model to this write-all-with-collision model (shared channel) where processes have some timing mechanism. In other words, given a self-stabilizing protocol that uses shared memory, we can transform it to use read/write atomicity (as discussed earlier) and then apply their transformation to obtain and deploy the resulting self-stabilizing protocol on a system in which processes communicate via a shared physical medium.

3.1.2 Fairness

In a theoretical sense, if we have randomization, the fairness assumption does not impact our ability to find a self-stabilizing protocol for any given problem. This result is due to Devismes, Tixeuil, and Yamashita [63] who show how to use randomization to transform a deterministic weakly stabilizing protocol into a randomized protocol that is stabilizing under any scheduler. For this method to work without a physical timing device, the topology must be connected for a phase clock [39, 40]. Practically however, such a protocol that relies on global fairness may take an unreasonable amount of time to stabilize.

The algorithms in this work only use no fairness or global fairness. These extremes make stabilization easier to verify than other assumptions such as weak fairness [47]. We do not lose much in terms of generality by ignoring weak and local fairness since stabilization-preserving transformations exist. Kosowski and Kuszner [135] show how to transform a stabilizing protocol under weak fairness into a stabilizing protocol under no fairness. Karaata [123] gives a similar transformation from local fairness to weak fairness. These transformations work similarly to the cross-over composition technique of Beauquier, Gradinariu, and Johnen [26, 27], which allows one to impose a certain type of fairness on a protocol by composing it with another protocol that simulates the desired type of fairness.

3.1.3 Faults

In this work, we focus on the concepts of closure an convergence to provide recovery from transient faults [14]. However, our focus on self-stabilization is not motivated purely by fault recovery; rather, we often expect the system to be initialized randomly. Transient faults and others can usually be detected with checksums or timeouts and can be corrected with resets and redundancy [177]. Even so, unless checkpointing or resets are available, some faults require recovery to be part of the protocol.

Chen and Kulkarni [48] show that 20 out of 31 categories of faults classified by Avizienis, Laprie, Randell, and Landwehr [22] can be modeled by state perturbation in a transition system. For example, Liu and Joseph [150] use state perturbation to model fail-stop failures (process crash).

In the case of a process crash, the fault must be detected (with a timeout) and the

process should be removed from the topology. We know that a self-stabilizing protocol will recover from the topology change if the new topology is valid for the protocol, but we would also like a fast recovery. Dolev and Herman [70] call this *superstabilization*. Superstabilization is particularly useful in peer-to-peer networks where processes are frequently joining and parting.

Byzantine faults are persistent faults where a process is permanently corrupted or has become malicious [146]. Such a process may even report different values to its neighbors. If the process is corrupted rather than malicious, we would prefer to have it crash [55]. It is straightforward to model a Byzantine process as reporting arbitrary values, and synthesis procedures have been made to take these faults into account during stabilization [66]. However, Daliot and Dolev [56] note that Byzantine fault tolerance and self-stabilization are difficult properties to have simultaneously in a protocol, which motivates their method to use distributed reset [15] to add stabilization to existing Byzantine fault-tolerant protocols.

3.2 Verification

Verification, or model checking, is the problem of deciding if a parallel or serial program meets its specification. The specification is given in a temporal logic such as Linear Temporal Logic (LTL) [77,159], Computation Tree Logic (CTL) [78], the combination of the two (CTL*), or CTL* without the next-time operator (CTL$^*\backslash$X). For example, convergence to a state predicate $\mathcal{L}$ be specified using the Eventually (a.k.a. Finally) operator $\Diamond$ as $\Diamond\,\mathcal{L}$, and closure within $\mathcal{L}$ can be specified using the Always (a.k.a. Globally) operator $\Box$ as $\Box(\mathcal{L} \implies \Box\,\mathcal{L})$.

3.2.1 Hardness

Verifying whether a temporal formula is true for a given system is a computationally-intensive task. Sistla and Clarke [165] prove that verification using LTL is PSPACE-complete in the size of the input (which includes all states of the system). Likewise, verification of CTL* is shown to be PSPACE-complete by Clarke, Emerson, and Sistla [52]. They, along with Arnold and Crubille [12], prove that CTL verification is P-complete. Self-stabilization and fault tolerance are such problems that can be verified in linear time with respect to the size of the transition system.

3.2.2 Implementation

The Spin model checker [113] has been extremely helpful with regard to investigating protocols. It is an explicit-state model checker that uses a C-like syntax with non-determinism to express concurrent programs. The temporal properties to be checked are specified as a formula in linear temporal logic. The complement of the temporal formula compiled to a Büchi automaton, and program verification reduces to

checking that the automaton never reaches an accepting state during any program execution [173].

Meseguer, Palomino, and Martí-Oliet [153] represent actions of processes as rewrite rules within the Maude tool. Using properties of term rewriting systems, they are able to automatically prove termination in special cases. Given the generality of term rewriting systems, Maude can also model and reason about algorithms and data structures.

Symbolic Representation. A transition system can also be represented as a boolean formula over unprimed and primed variables. For example, consider the 3-coloring protocol of Example 2.1.1. Particularly, consider the minimal action

$$(x_0 = 0 \wedge x_1 = 0 \wedge x_2 = 2) \longrightarrow x_1 := 1;$$

of P_1 on a ring of size 4. This can be represented with the following formula

$$x_0 = 0 \wedge x_1 = 1 \wedge x_2 = 2 \wedge x_1' = 1 \wedge x_0' = x_0 \wedge x_2' = x_2 \wedge x_3' = x_3$$

which uses $x_1' = 1$ to represent the assignment $x_1 := 1$ and where $x_0' = x_0 \wedge x_2' = x_2 \wedge x_3' = x_3$ forbids the x_0, x_2, x_3 variables from changing. The complete transition system can be built by taking the disjunction of all such formulas of all processes.

Binary decision diagrams (BDDs) [41] and multi-valued decision diagrams (MDDs) can be used to represent the boolean formula of a transition system. The popular NuSMV model checker [50] and PVS proof assistant [157] uses this data structure. The size of a BDD can be exponentially larger than the number of variables, and its size depends largely on the variable ordering and nature of the function, but they are much better than explicit state representations in practice. Many MDD and BDD libraries exist to manipulate transition systems; we use the GLU and CUDD libraries [166].

BDD-based cycle detection can still be infeasible due to size. Biere, Cimatti, Clarke, Strichman, and Zhu [29] introduce bounded model checking as a way to avoid using a data structure to fully represent the transition system. They show that the use of SAT solvers and abstraction can find counterexamples where BDD-based methods cannot handle the input model. However, BDD-based verification can also outperform their SAT-based method. Further, their method is incomplete, therefore one cannot assume a system is correct if no counterexample is found.

Unbounded Variables. When domains of variables are unbounded, Boigelot and Wolper [32] show that finite automata can be represent values and constraints. Borowsky and Edelkamp [36] apply this to the planning problem where variables are unbounded. They explain the problem representations and algorithms in detail and discuss which problem conditions can prevent the algorithms from terminating. The planning problem is similar to verification and can also be solved using the same techniques [51].

3.2.3 Symbolic Cycle Detection

Livelock detection is a fundamental step in model checking self-stabilizing algorithms. These correspond to cycles in the transition system being checked. Gentilini, Piazza, and Policriti [96] give a cycle detection algorithm that is linear in the size of the BDD representing the transition system. In practice, algorithms that compute strongly connected components (SCCs) outperform explicit algorithms, even though they have a higher worst-case complexity [91]. Emerson and Lei [80] give on such fixpoint algorithm for which several variations exist [91].

Algorithm 3.1 shows the version of the Emerson-Lei algorithm that we use for detecting unfair cycles. It is only notable in that we avoid unnecessary computation during the main fixpoint iteration (Line 5) since we assume that the protocol is closed within the initial set of states (called `span`). The algorithm is written using set notation, but recall that a set of states or transitions is efficiently represented as a BDD (boolean formula) that evaluates to true for states/transitions in the set.

Algorithm 3.1 Check for unfair cycles in a transition system.

CycleCheck(`&span`: closed set of initial states (also a return value),
Δ: transitions of protocol)

Output: Whether a cycle exists.

1: **let** `next` := `span`
2: {Fixpoint iteration using image}
3: **repeat**
4: `span` := `next`
5: `next` := Δ[`span`]
6: **until** `span` = `next`
7: {Fixpoint iteration using preimage to make `span` resemble the SCCs more closely}
8: **repeat**
9: `span` := `next`
10: `next` := `span` $\cap\ \Delta^{-1}$[`span`]
11: **until** `span` = `next`
12: {`span` is now all states that can be visited after arbitrarily many steps in an infinite execution, but unlike in an SCC, `span` may contain some states that cannot be visited infinitely often}
13: **return** (`span` $\neq \emptyset$)

Chen, Abujarad, and Kulkarni [47] investigate how fairness impacts the cost of verifying stabilization. Weak fairness is found to substantially increase verification cost, whereas an assumption of global or no fairness allows faster verification. Particularly, since global fairness alleviates the need for cycle detection, it admits the fastest verification times.

3.3 Design

This section discusses existing work in the design of self-stabilization and fault tolerance. We first give an overview of NP-hardness results that are related to the addition of convergence. Then, we discuss the techniques for adding convergence, both manual and automatic.

3.3.1 Hardness

Kulkarni and Arora [137] present a family of polynomial-time algorithms for the addition of different levels of fault tolerance in the high atomicity distribution model, while demonstrating that adding masking fault tolerance in the low atomicity model is NP-complete. Lin, Bonakdarpour, and Kulkarni [148] give a similar polynomial-time result for the high atomicity model under a synchronous scheduler. Kulkarni and Ebnenasir [140] show that adding failsafe fault tolerance is NP-complete. Bonakdarpour and Kulkarni [35] similarly prove the NP-hardness of designing progress from one state predicate to another. Farahat and Ebnenasir [88] show that weak stabilization can be added in polynomial time.

3.3.2 Manual Techniques

Manual techniques for designing convergence are mainly based on the approach of *design and verify*, where one designs a fault-tolerant system and then verifies the correctness of (1) functional requirements in the absence of faults, and (2) fault tolerance requirements in the presence of faults. For example, Liu and Joseph [150] provide a method for augmenting fault-intolerant systems with a set of new actions that implement fault tolerance functionalities. Gouda and Herman [103] show how to compose protocols where one relies on the other for stabilization. They rely on the idea of stabilization to a subset of states [99], which our work emphasizes quite heavily (Section 2.2.4). Katz and Perry [125] present a general method for adding convergence to non-stabilizing protocols by taking global snapshots and resetting to a legitimate state when an illegitimate one is detected. Varghese [175] also proposes a counter flushing method for detection and correction of global predicates. Arora, Gouda, and Varghese [16] design nonmasking fault tolerance by creating a dependency graph of the local constraints of program processes, and by illustrating how these constraints should be satisfied so global recovery is achieved. Arora and Kulkarni [18,136] decompose a system into components that are the fault-intolerant protocol, components that detect when faults occur, and components that correct the system state after faults occur. Arora and Kulkarni [17] also give a method to design masking fault tolerance by first adding nonmasking fault tolerance and then enforcing safety specifications afterward. Jhumka, Freiling, Fetzer, and Suri [120, 121] investigate the addition of failsafe fault tolerance while accounting for efficiency.

Nesterenkol and Tixeuil [156] employ a mapping to define all system states as legitimate state of an abstract specification. This effectively removes the need for convergence, otherwise known as snap-stabilization [42], but it is not always possible or may require human ingenuity in the specification. Demirbas and Arora [61] use mappings as a tool to work from abstract specification to a self-stabilizing implementation. This is a good manual tool seems too open-ended to fully automate.

3.3.3 Automated Techniques

Kulkarni and Arora [137] consider the high atomicity distribution model where each process can read all variables. They demonstrate that adding failsafe, masking, or nonmasking fault tolerance to high atomicity protocols can be done in polynomial time in the size of the state space. Lin, Bonakdarpour, and Kulkarni [148] give a similar polynomial-time algorithm for the case of a synchronous scheduler. Ebnenasir [74] establishes a foundation for the addition of fault tolerance in the composite atomicity model using efficient heuristics and component-based methods. Attie, Emerson, and Arora [19,20] tackle the state explosion problem by considering only pairs of processes, though the resulting protocols impose high atomicity. Attie and Emerson [21] give a method to practically implement a protocol that assumes high atomicity, where they only assume the ability to atomically test and set a single variable at any time.

Bonakdarpour and Kulkarni [34] exploit symbolic techniques to increase the scalability of the addition of fault tolerance. A similar choice is utilized by Bonakdarpour and Kulkarni [33], where progress properties are added to distributed protocols. Abujarad and Kulkarni [7] consider the program invariant as a conjunction of a set of local constraints, each representing the set of local legitimate states of a process. Then, they synthesize convergence actions for correcting the local constraints. Nonetheless, they do not explicitly address cases where local constraints have cyclic dependencies (e.g., maximal matching on a ring), and their case studies include only acyclic topologies. They are able to improve their algorithm's efficiency with multi-threading [5]. Farahat and Ebnenasir [87, 88] give a polynomial-time heuristic to add self-stabilization that essentially (1) finds a weakly stabilizing protocol, (2) ranks illegitimate states by shortest path to a legitimate state, (3) ranks actions by the ranks of deadlock states they resolve, and (4) add the actions in order of their rank, discarding those that create livelocks. They give a parallelized version of this algorithm where independent tasks shuffle the actions within each rank [76]. This allows the tasks to add actions in different orders, increasing the probability of finding a self-stabilizing protocol. Zhu and Kulkarni [179] give a genetic programming approach for the design of fault tolerance, using a fitness function to quantify how close a randomly-generated protocol is to being fault-tolerant.

Faghih and Bonakdarpour [84] formulate stabilization as a set of constraints that an SMT solver such as Z3 [58] can solve. This yields a complete approach to synthesis that can exploit future advancements in SMT solvers. The authors specify

behavioral constraints explicitly as transitions, but there is nothing fundamentally preventing such a technique from using LTL properties or our shadow/puppet technique (Definition 2.2.12). For example, the *bounded synthesis* method Finkbeiner and Schewe [90] uses a similar SMT encoding and allows behavior to be specified in LTL. Each process is considered to be a FSM, with some initial state(s), whose current state can be read by neighboring processes (i.e., the state-reading model). Their synthesis method has the freedom to add new states to the FSM, though the search must be given a bound so it eventually terminates. A bound is required since Pnueli and Rosner [160] have shown that for a given LTL formula and environment process, synthesis of two FSMs that satisfy a given LTL formula is undecidable. Bounded synthesis is an elegant idea since it could be seen as our shadow/puppet technique where the puppet variables are synthesized, rather than having manually-specified domains. Both works show promise, but for this synthesis problem, SMT solvers do not yet match the performance of the algorithm used in Chapter 4 since the generic solver does not optimize its encoding of individual states, whereas we can use MDDs (Section 3.2.2).

3.4 Parameterized Systems

It is usually desirable for a protocol to give correct behavior for an arbitrary number of processes. Consider how troublesome it would be to reprogram all of the processes if one were added or removed! Generally, proving stabilization of a parameterized protocol is done manually. Gouda and Multari [104] use convergence stairs to prove stabilization (Lemma 2.2.6). This technique can be used inductively to prove stabilization of a parameterized system, where the number of layers grows with the number of processes. Stomp [168] provides a method based on ranking functions for design and verification of self-stabilization.

When global properties can be expressed as constraints local to each process, manual proof of generalization is simplified. Varghese [174] and Afek, Kutten, and Yung [8] provide a method based on local checking for global recovery of locally correctable protocols. Similarly, the Arora, Gouda, and Varghese [16] reason locally about nonmasking fault tolerance by creating a dependency graph of the local constraints of program processes, and by illustrating how these constraints should be satisfied so global recovery is achieved.

3.4.1 Decidability of Verification

When problems become unbounded, they often become undecidable; verifying parameterized systems is no exception. Over time, increasingly restricted systems have been shown to be undecidable, or specifically, co-semi-decidable, written as Π^0_1-complete using the arithmetical hierarchy notation of Rogers [161]. Apt and Kozen [11] prove that verifying an LTL formula holds for a parameterized system is undecidable.

Suzuki [169] builds on this result, showing that the problem remains undecidable on symmetric unidirectional ring protocols where only the number of processes is parameterized. Emerson and Namjoshi [82] show that the result holds even when a single token, which can take two different values, is passed around such a ring.

Abello and Dolev [3] show that any Turing machine can be simulated on a bidirectional chain topology in a self-stabilizing manner. Among other variables in their protocol, each process has variables to represent an input tape cell, a working tape cell, and an output. When the Turing machine accepts, rejects, or fails to compute a result (due to cycles or insufficient tape cells) for the given input, the output value of each process will eventually be 1, 0, or $\perp$ respectively. Once a simulation of the Turing machine finishes, it begins again in case some fault corrupts an output value or the input is changed.

3.4.2 Decidable Restrictions

Some classes of parameterized systems can be automatically verified since the kernel of undecidability is removed.

3.4.2.1 Safety

If the topology allows, safety properties such as deadlock freedom can be checked completely. Farahat and Ebnenasir [89] show that deadlocks be detected in a parameterized ring protocol by observing the states where individual processes are disabled, constructing a graph, and performing cycle detection in the graph. Fribourg and Olsén [93] represent sets of states in a ring or chain with regular languages (represented by deterministic finite automata). They can build the sets of silent and invariant states of a protocol and check for deadlocks by subtracting the invariant from the silent states and then checking for non-emptiness. This forms a basis for regular model checking, which we discuss in detail in Section 3.4.3. Clarke, Grumberg, and Jha [53] similarly use regular languages to represent sets of states within the network invariant context. Cachera and Morin-Allory [45] use polyhedra (as opposed to DFAs) to verify safety properties. They are able to classify/identify sufficient conditions for cases where their verification technique is complete.

3.4.2.2 Global Computations

Some protocols perform a computation over the topology. For example, (1) given two nodes, find the shortest path between them, or (2) given one node to use as the root, find a spanning tree. Tel [170] shows how some global computations can be expressed by defining an infimum operator for processes to apply to the inputs from adjacent processes. Ducourthial, Tixeuil, and Delaet [59, 71–73] investigate r-operators as an extension of infimum operators that ensure a global computation is self-stabilizing. It is interesting to note the CALM (consistency and logical monotonicity) theorem [10]

in distributed computing takes a similar approach, where monotonic functions do not require explicit coordination between processes.

3.4.2.3 Token Ring Systems

Emerson and Namjoshi [82] consider unidirectional rings of symmetric processes that only communicate by passing a single token (the ability to act) in a fixed direction. They show that when processes cannot communicate any information, many LTL (CTL$^*\backslash$X) properties can be checked by explicitly checking all rings below a certain size. They further show that if processes are allowed to communicate when passing the token, then the problem becomes undecidable.

Khalimov, Jacobs, and Bloem [119,126] combine these cutoff theorems with bounded synthesis [90] to synthesize protocols for token ring systems from an LTL specification.

3.4.2.4 Symmetric Guards

Emerson and Kahlon [79] also give cutoff theorems for protocols involving arbitrarily many processes of different types. Their model assumes that every action of a process has a guard that quantifies (using $\forall$ or $\exists$) over all other process states using the same state predicate for all processes of the same type. In other words, a process can communicate with all or none of the processes of a certain type, and its guards can only test if a predicate holds for "at least one" or "exactly all" of those processes. Emerson and Kahlon give cutoff theorems for these systems, where the maximum cutoff size is dependent on the number of states of each process template.

3.4.3 Regular Model Checking

In regular model checking [2, 37, 178], system states are represented by strings of arbitrary length, and a protocol is represented by a transducer that can transform the strings. The topology must permit a finite representation of the states of arbitrarily many processes. Fribourg and Olsén [93] show how sets of states of a parameterized ring or chain can be represented by a regular language. Abdulla, Jonsson, Mahata, and d'Orso [1] show how to use tree automata and transducers to apply regular model checking to tree topologies. Touili [172] shows how to use regular hedge automata and transducers to apply regular model checking to arbitrary width tree-like structures.

Early work in regular model checking focused on safety and reachability properties, which are more straightforward than liveness properties such as cycle detection. Notably, Habermehl and Vojnar [108] give an inference method for computing the reachable states of a parameterized system, where the reachable states (or an acceptable over-approximation) for a generalized system are inferred from the reachable states of small explicit systems. Bouajjani, Legay, and Wolper [38] investigate liveness properties such as cycle detection and give experimental results. Fisman, Kupferman, and

Lustig [92] use regular model checking to verify fault tolerance, where they consider fail-stop, Byzantine, and transient faults as well as the special case of self-stabilization.

Deadlock Detection. Let $\mathcal{L}$ be the legitimate states of a parameterized system, and let R be the relation of the protocol's transducer. A deadlock exists *iff* $\mathcal{L} \setminus \textsc{Pre}(R)$ is non-empty. Note that $\textsc{Pre}(R)$ or its complement can be constructed directly, much like the continuation relation in [89] is constructed to characterize the silent states in a ring.

Livelock Detection. Let R be the relation of this transducer and let R^+ be its transitive closure. A cycle can then be detected by checking if R^+ maps some string to itself, or in other words, checking if $R^+ \cap R_{id}$ is non-empty, where R_{id} is the identity relation. Of course no algorithm computes R^+ in all cases, therefore heuristic acceleration techniques are used such as widening [171].

3.5 Protocols

It is useful to see some examples of well-studied problems and the techniques required to make self-stabilizing protocols possible or yield better behavior. Such techniques include randomization, large variable domains, variable domains that exactly correspond to the number of processes, restrictions on the number of processes (prime or odd), synchronous execution, and assumptions about the scheduler. Some of these techniques are unreasonable or impossible in practice, but the resulting protocols give insight to the problem nonetheless.

3.5.1 Coloring

For the coloring problem, processes that can communicate with each other wish to select distinct "colors". This is useful for breaking symmetry or giving adjacent processes different priorities. A 3-coloring ring protocol was given in Example 2.1.1, but livelocks can occur when adjacent processes can act synchronously. Shukla, Rosenkrantz, and Ravi [163] show that, without randomization, a 3-coloring protocol cannot exist on a ring of symmetric processes under the distributed daemon. However, using randomization, they show that such a 3-coloring protocol is possible on a unidirectional (or bidirectional) ring. They give similar results and protocols for other specialized graph topologies in [164]. Coloring algorithms exist for general graphs as well. Hedetniemi, Jacobs, and Srimani [109] give a $(D+1)$-coloring protocol that works on arbitrary graphs of maximum degree D.

3.5.2 Orientation

If anonymous processes have some method of finding each other, it is easy for them to form a ring topology. However, a problem arises when they must decide on a common direction around the ring. This is the problem of *orientation*.

3.5.2.1 Rings

Israeli and Jalfon [117] give a randomized ring orientation protocol that is self-stabilizing under the distributed daemon. Under the central daemon, this protocol can be deterministic since randomization is only needed for coloring under the distributed daemon. The orientation works by passing tokens around the ring. All tokens eventually circulate in the same direction, which gives processes a basis for a common orientation. There eventually may become no tokens in the system in order to reach a silent state, but this is not guaranteed. (In Section 5.4, we give an orientation protocol that is silent but is not known to stabilize under the distributed daemon.)

A key component of this protocol is the ability for a process P_i to determine which, if any, of its neighbors P_{i-1} and P_{i+1} are "pointing" towards it. This is not possible when using the state-reading model. In fact, Israeli and Jalfon show that no stabilizing ring orientation protocol exists using the state-reading model, where processes read the entire state of neighboring processes. This prevents any kind kind of directionality information from being shared since a process without orientation cannot use a single value (its state) to communicate that it is pointing to one process and not the other.

3.5.2.2 Rings of Odd Size

Hoepman [112] introduces a simpler orientation protocol in the state-reading model that is stabilizing when the ring is guaranteed to be of odd size. The protocol arranges processes in a bidirectional ring and uses token circulation to determine a common direction around the ring. Whenever a process propagates a token, it sets its chosen direction as the direction that the token is traveling. The system is free of deadlocks since number of tokens can be forced to be odd, which is made possible by discarding rings of even size. Further, all tokens will eventually circulate in the same direction since any tokens traveling in different directions will eventually collide and cancel each other.

3.5.3 Token Passing

In a token ring protocol, processes are oriented in a ring topology and circulate single token. A token can be defined in any way, but a process must know whether it has the token. For example, a process π_i might be defined to have a token when it is enabled to act. The token itself is used to signify an exclusive privilege that the process has, which could be access to a lock or some shared resource.

Burns an Pachl [44] show that no stabilizing token ring exists under no fairness where all processes are symmetric and deterministic, but they give such a protocol for rings of prime size.

3.5.3.1 Dijkstra's Token Ring

Dijkstra [64] first introduced self-stabilization in the context of a unidirectional token ring. We saw this protocol defined in Section 1.1 when introducing the separation shadow vs puppet protocols. In short, each process π_i has a variable $x_i \in \mathbb{Z}_N$ that can hold at least as many values as there are processes in the ring, of which we say there are N. The first process π_0 is distinguished as Bot_0 since it acts differently from each other process $P_{i\geq 1}$. The protocol is very simple, where Bot_0 increments its x_0 value (modulo its domain) when x_{N-1} has the same value. Each P_i essentially does the opposite by assigning x_i to match x_{i-1} whenever the values differ. A process in this protocol is considered to have a token whenever it is enabled to act.

Along with being simple and quick to converge (given that its topology is a ring), Dijkstra's protocol is also practical to implement. While we may not know the exact ring size N when deploying such a system, it is easy to choose a domain size for x_i variables that exceeds any expected ring size (perhaps 2^{32}). However, if x_i values d only have a finite state space, livelocks occur. The non-stabilizing 2-state token ring used in Example 2.2.13 is an extreme example of this, since it matches Dijkstra's token ring exactly when x_i variables are binary.

3.5.3.2 Token Ring of Three Bits

Gouda and Haddix [102] give a protocol for a token ring using only 3 bits per process, regardless of the ring size N. Like other token rings, there is a distinguished process Bot_0 and $N-1$ other processes $P_1, \ldots, P_{N-1}$. Each process π_i owns 3 binary variables t_i, e_i, and $ready_i$ and can read 2 of its predecessor's variables t_{i-1} and e_{i-1}.

Processes are defined to have tokens under the same conditions as the non-stabilizing 2-state token ring p_{tok2} introduced in Example 2.2.13. Therefore, the legitimate states can be written as the state predicate $\mathcal{L} = \exists! i \in \mathbb{Z}_N : ((i = 0 \wedge t_{N-1} = t_0) \vee (i \neq 0 \wedge t_{i-1} \neq t_i))$. Note also that the protocol only changes the t_i variables in exactly the same manner as the non-stabilizing version. Using language from Chandy and Misra [46], we can call the non-stabilizing token ring of one bit the *underlying protocol* (a.k.a.*shadow protocol* in our terminology) that was transformed to the token ring of three bits by adding 2 superposed variables e_i and $ready_i$ to each process.

The token ring of three bits is designed to be run on a synchronous system (e.g., components sharing a clock), rather than on a distributed system. Multiple processes can be enabled without multiple tokens existing. In fact, if e processes act synchronously each step, then it takes an average of $N/(2e)$ steps for a process to pass a token to its successor in the ring. When the system is asynchronous like a distributed system, e is likely to eventually decrease and may decrease to 1, making it very costly to pass the token on average.

3.5.3.3 Bidirectional Token Passing

Dijkstra [64] also provided two protocols that stabilize to passing a single token back and forth between two end processes (across all other processes). One of these protocols uses 4 states per process arranged in a linear topology, which we call a *chain*. The other protocol uses 3 states per process, but it requires the two end processes to be able to communicate, making it a bidirectional ring topology. Chernoy et al. [49] give a similar 3-state ring protocol with an improved worst-case convergence time. In Section 5.3, we show that a 3-state protocol is also possible when the topology is a chain.

3.5.4 Leader Election

A distinguished process can greatly simplify a protocol or, as is the case with token rings, may affect whether a self-stabilizing version exists. As such, another well-studied problem is that of determining a unique process to distinguish or act as a *leader*. In fact, Mayer, Ostrovsky, Ofek, and Yung [151] have shown that a stabilizing token ring protocol is an equivalent to stabilizing leader election on a bidirectional ring. Given the impossibility results of Burns an Pachl [44] for token rings, a stabilizing leader election protocol cannot exist where processes are symmetric and deterministic.

3.5.4.1 Rings of Prime Size

Huang [115] gives a leader election protocol that works on bidirectional rings of prime size. Each process owns a variable whose domain exactly matches the ring size N. When the protocol is finished, each process's variable will hold a unique value, and a process can determine itself as the leader if its value is 0. Desel, Kindler, Vesper, and Walteret [62] intuitively reformulate this as an agreement protocol on any ring size between humans sitting around a table with a deck of cards, rather than in terms of processes in a ring with variables. Even though the reformulated version focuses on agreement, it is the same protocol and still elects a leader when the ring size is prime.

Itkis, Lin, and Simon [118] give another leader election protocol that also requires a prime ring size, but processes require only constant space. This protocol relies on a bidirectional ring topology. In the same work, they also give a protocol that works on a unidirectional ring of prime size N that requires $O(\lg N)$ space per process, and they prove that no constant-space version exists for unidirectional rings.

3.5.4.2 Ring using Logarithmic Space

Blin and Tixeuil [31] give a deterministic leader election protocol on a bidirectional ring under weak fairness. This protocol is not silent and requires each of the N processes in the ring to have $O(\lg N)$ states (or in their words, $O(\lg \lg N)$ bits). Note

that even though process domains must grow with the ring size, the processes do not need to know the ring size. This is in the same spirit as Dijkstra's token ring [64].

Chapter 4:
A Backtracking Algorithm for Shadow/Puppet Synthesis

Section 2.2.3 introduced the idea of stabilization to a *shadow protocol* that operates within some set of legitimate states. Recall that a shadow protocol encapsulates the desired behavior of a protocol without the constraints associated with a realistic topology. For example, in Section 1.1, we described how to specify a token ring protocol p_{tok} of some N processes, operating within states where exactly one token exists ($\exists! i \in \mathbb{Z}_N : tok_i = 1$), where each π_i passes the token with an action ($tok_i = 1 \longrightarrow tok_i := 0;\ tok_{i+1} := 1;$). Such a token ring is certainly not self-stabilizing, and it is likely unrealistic because processes write directly to each other's memories. Therefore, these tok_i *shadow variables* should not be used in a self-stabilizing protocol, and we instead add *puppet variables* (named x_i in this example) to achieve behavior in a realistic way. To ensure that a synthesized protocol p'_{tok} preserves the behavior of p_{tok}, we look at how the tok_i variables change, but they are treated as write-only in p'_{tok} because they are unrealistic and should not affect decision-making.

This chapter proposes an automated synthesis algorithm that respects this notion of using a shadow protocol for specification and puppet variables for implementation. The algorithm itself is a backtracking search [162], which is a class of search algorithms that is easy to implement and can yield very good results. It is also a complete search, a property that several automated techniques forgo [4, 5, 76, 88, 179] due to the inherent complexity of designing stabilization [64, 100, 132]. Other complete methods do exist, particularly reducing the problem of synthesis to the language of an SMT solver [84, 90]. However, we find that backtracking with some (completeness-preserving) heuristics performs better, primarily because we can guide the search to make decisions at logical points (choosing actions) rather than getting lost in the protocol representation (states and transitions).

Contributions. The contributions of this work are multi-fold. First, we devise a two-step design method that separates the concerns of closure and convergence for the designer, and enables designers to intuitively specify functional behaviors and

Sections 4.2–4.4 contain material from A. P. Klinkhamer and A. Ebnenasir. Shadow/Puppet Synthesis: A Stepwise Method for the Design of Self-Stabilization. *IEEE Transactions on Parallel and Distributed Systems*, 2016.

systematically include computational redundancy. Second, we propose a *parallel* and *complete* backtracking search that finds an SS solution if one exists. If a solution does not exist in the current state space of the program, then designers can include additional puppet variables or alternatively increase the domain size of the existing puppet variables and rerun the backtracking search. Third, we present three different implementations of the proposed method as a software toolset, called Protocon (`http://asd.cs.mtu.edu/projects/protocon/`), where we provide a sequential implementation and two parallel implementations; one multi-threaded and the other an MPI-based implementation. We also demonstrate the power of the proposed method by synthesizing several new network protocols that all existing heuristics fail to synthesize. These case studies include 2-state maximal matching on bidirectional rings, 5-state token passing on unidirectional rings, 3-state token passing on a bidirectional chains, and orientation on daisy chains.

Organization. Section 4.1 defines the problem of synthesis and gives the essential ideas used by our search algorithm. Section 4.2 presents a flowchart overview of our synthesis algorithm. Section 4.3 gives algorithm details, including pseudocode. Section 4.4 discusses the optimizations we use to minimize the cost of our backtracking search, which is exponential in the worst case. Chapter 5 is a continuation of this chapter that showcases 4 new protocols found using this synthesis algorithm, including the search time for synthesizing these and other related protocols. It also gives some insight into the techniques involved in specification and synthesis.

4.1 Synthesis Problem

This section begins with a formal description of the synthesis problem, though most problem constraints were introduced in Section 2.2.3. We then give a brief explanation of backtracking search, the assumptions made about protocols, and some consequences that simplify the design of self-stabilization.

Synthesis. Let p be a non-stabilizing shadow protocol and $\mathcal{L}$ be the legitimate states of p. We *manually* expand the state space of p to create a shadow/puppet topology p' by adding puppet variables and making unrealistic shadow variables write-only. A synthesis algorithm then searches for actions to add to processes of p' in order to achieve stabilization. Our experience shows that better performance can be achieved if variables with small domains are included initially. If the synthesis fails, then designers can incrementally increase variable domains or include additional puppet variables. This way designers can manage the growth of the state space and keep the synthesis time/space costs under control. Such an increase in state space could be trivially automated in order to achieve a similar effect as bounded synthesis [90].

Problem 4.1.1 (Shadow/Puppet Synthesis)**.**

- INPUT: A shadow protocol p, its legitimate states $\mathcal{L}$, a shadow/puppet topology, and a projection function $\mathcal{H}$

  - Processes of p are assumed to be deterministic and self-disabling
- OUTPUT: A shadow/puppet protocol p' that stabilizes to $p|\mathcal{L}$ (Definition 2.2.12)
  - The scheduler is assumed to be unfair

Backtracking Search. Like any other backtracking search, our algorithm incrementally builds upon a guess, or a partial solution, until it either finds a complete solution or finds that the guess is inconsistent. We decompose the *partial solution* into two pieces: (1) an *under-approximation* formed by making well-defined decisions about the form of a solution, and (2) an *over-approximation* that is the set of remaining possible solutions (given the current under-approximation).

Minimal Actions. Since a protocol's behavior can be represented as a set of minimal actions of different types of processes, our under-approximation is a list named `delegates` that contains all minimal actions that will be present in a solution if our guesses so far are correct. Likewise, an over-approximation is a set of all minimal actions that could possibly be included in a solution. A list named `candidates` contains these "possibly included" actions, but it excludes the already-selected `delegates`, therefore `delegates` $\cup$ `candidates` constitutes the over-approximation.

Decision Tree. A backtracking search works by choosing actions from the over-approximation to definitely include in the under-approximation. Each time a choice is made to build upon the under-approximation, the current partial solution is saved at decision level j and a copy that incorporates the new choice is placed at level $j+1$. If the guess at level $j+1$ is inconsistent, we move back to level j and discard the choice that brought us to level $j+1$. If the guess at level 0 is found to be inconsistent, then enough guesses have been tested to determine that no solution exists. A partial solution is inconsistent when (1) the under-approximation causes a *conflict* in the problem constraints, or (2) the over-approximation cannot possibly contain a solution.

No Fairness. We assume that the scheduler is unfair. This allows us to characterize the two kinds of inconsistencies as (1) actions in `delegates` creating a livelock, and (2) actions in `delegates` $\cup$ `candidates` do not provide enough transitions to provide stabilization. Notice that under weaker fairnesses, we cannot consider a livelock in `delegates` to cause a conflict because some livelocks can be resolved by adding more actions to the protocol!

Lemma 4.1.2 (Inconsistency)**.** *Given* `delegates` *and* `candidates`*, let states* $\mathcal{L}'$ *and transitions* $\Delta'|\mathcal{L}'$ *be over-approximations. No subset of actions in* `delegates` $\cup$ `candidates` *can form a stabilizing protocol if an inconsistency breaks one of the following constraints:*

1. *The transitions of* `delegates` *do not form livelocks in* $\overline{\mathcal{L}'}$ *(Constraint 1 in Definition 2.2.10)*
2. *The transitions of* `delegates` $\cup$ `candidates` *provide convergence to* $\mathcal{L}'$ *under global fairness (Constraint 1 in Definition 2.2.10)*

3. *$\Delta'|\mathcal{L}'$ preserve all transitions of the shadow protocol (Constraint 2 in Definition 2.2.11)*

Proof. Since $\mathcal{L}'$ and $\Delta'|\mathcal{L}'$ are over-approximations, a stabilizing solution protocol will use some subset of legitimate states $\mathcal{L}'$ and subset of legitimate transitions $\Delta'|\mathcal{L}'$. Thus, if the under-approximation causes a livelock within $\overline{\mathcal{L}'}$, this inconsistency cannot be fixed by adding transitions. Likewise, if the over-approximation does not provide reachability from $\overline{\mathcal{L}'}$ to $\mathcal{L}'$, this inconsistency cannot be fixed by removing transitions. Finally, if the over-approximated $\Delta'|\mathcal{L}'$ does not preserve shadow transitions, this inconsistency cannot be fixed by removing transitions. □

Lemma 4.1.3 (Legitimate States and Behavior)**.** *Given* `delegates` *and* `candidates`*, a maximal set of states $\mathcal{L}'$ and transitions $\Delta'|\mathcal{L}'$ can be constructed such that the following constraints are satisfied:*

- $\mathcal{L}' \subseteq \mathcal{H}^{-1}[\mathcal{L}]$
- *The transitions of* `delegates` *within $\mathcal{L}'$ do not violate progress due to livelocks*
- *$\Delta'|\mathcal{L}'$ is a subset of transitions of* `delegates` $\cup$ `candidates`
- *$\Delta'|\mathcal{L}'$ contains all transitions of* `delegates` *that begin in $\mathcal{L}'$*
- *$\Delta'|\mathcal{L}'$ satisfies progress and fixpoint convergence constraints under global fairness*

Proof. Perform the following steps.

1. Initialize $\mathcal{L}' := \mathcal{H}^{-1}[\mathcal{L}]$ using legitimate shadow states of the shadow protocol p
2. For each state $s \in \mathcal{L}$ from which p requires progress to a different state, if the transitions of `delegates` form a cycle within $\mathcal{H}^{-1}[\{s\}]$, then remove $\mathcal{H}^{-1}[s]$ from $\mathcal{L}'$
3. Initialize $\Delta'|\mathcal{L}'$ as the set of each (s'_0, s'_1) of `delegates` $\cup$ `candidates` where $s'_0, s'_1 \in \mathcal{L}'$ and either $\mathcal{H}(s'_0) = \mathcal{H}(s'_1)$ or $(\mathcal{H}(s'_0), \mathcal{H}(s'_1))$ is a transition of $p|\mathcal{L}$
4. Iteratively remove any state s'_0 from $\mathcal{L}'$ and $\Delta'|\mathcal{L}'$ that satisfies either:
   (a) (s'_0, s'_1) is some transition of `delegates` that does not exist in $\Delta'|\mathcal{L}'$
   (b) $p|\mathcal{L}$ requires progress from $\mathcal{H}(s'_0)$ to a different state, and no execution of $\Delta'|\mathcal{L}'$ from s'_0 contains a state s'_1 where $\mathcal{H}(s'_0) \neq \mathcal{H}(s'_1)$

These steps correspond to the constraints of the lemma in order. Following the steps, it should be clear that states and transitions are only removed if they violate a constraint of the lemma, therefore the computed $\mathcal{L}'$ and $\Delta'|\mathcal{L}'$ are maximal. □

Parameterized Systems. It is usually the case that we want to design a protocol that is correct/stabilizing for all topologies of a certain class (e.g., rings). Therefore, we actually consider multiple topologies at the same time (e.g., rings of sizes $N \in \{2, \ldots, 7\}$). This is easy to implement since it just involves checking consistency of the protocol on each topology, rather than just one specific topology. We omit this idea of multiple systems from the algorithm pseudocode, but we are explicit about this aspect in the case studies of Chapter 5.

Deterministic, Self-Disabling Processes. Processes are restricted to be deterministic and self-disabling. This allows us to remove many possibilities from `candidates`, which prunes many unnecessary decision subtrees. Under no fairness (and also weak fairness), no computational power is lost using this restriction.

Theorem 4.1.4 (Deterministic, Self-Disabling Processes)**.** *If some protocol p converges to states $\mathcal{L}$ from states Q under fairness $\mathcal{F} \in \{\text{unfair}, \text{weak}\}$ such that no single process acting twice can cause an execution $\langle s_0, s_1, s_2 \rangle$ where $s_0 \in Q \setminus \mathcal{L}$, $s_1 \in \mathcal{L}$, and $s_2 \in \overline{Q} \setminus \mathcal{L}$, then the same can be achieved by a protocol p_{det} whose processes are deterministic and self-disabling.*

Proof. Assuming p converges to $\mathcal{L}$ from Q using the constraints specified, we want to achieve the same convergence using a protocol p_{det} whose processes are deterministic and self-disabling. One such p_{det} can be constructed in 3 phases of transformations to the transition function δ_i of the finite state machine of each process π_i of p: (1) While transitions $(w_0, a, w_1), (w_0, a, w_2) \in \delta_i$ exist such that $w_1 \neq w_2$, arbitrarily remove one of these nondeterministic transitions. (2) While transitions $(w_0, a, w_1), (w_1, a, w_2) \in \delta_i$ exists such that $w_0 \neq w_1$, replace (w_0, u, w_1) with (w_0, u, w_2) so that it no longer enables (w_1, a, w_2). (3) While a transition $(w_0, a, w_0) \in \delta_i$ exists, remove this self-loop transition. Phase 1 makes processes of p_{det} deterministic, and phases 2 & 3 make processes of p_{det} self-disabling. We are left to show that these transformations preserve convergence to $\mathcal{L}$ from Q under fairness $\mathcal{F}$. By Definition 2.2.4, this means we want to prove that each transformation phase preserves deadlock and livelock freedom within $Q \setminus \mathcal{L}$ and does not introduce any transitions from $Q \setminus \mathcal{L}$ directly to $\overline{Q} \setminus \mathcal{L}$.

Phase 1. This phase only removes the opportunity for a process to nondeterministically choose between two or more enabled actions. By definition, each fairness $\mathcal{F} \in \{\text{unfair}, \text{weak}\}$ only applies to a process's ability to act (this is true for $\mathcal{F} \in \{\text{local}, \text{sync}\}$ as well). That is, if a process has multiple actions enabled, there is no notion of fairness when choosing between which action to take. Therefore, in executions of p under $\mathcal{F}$, it is always valid for processes to deterministically choose which action to take if multiple actions are enabled. This shows that, at this phase of construction, all maximal executions of p_{det} correspond to maximal executions of p that are valid under fairness $\mathcal{F}$. Thus, no deadlocks, livelocks, or invalid transitions are introduced, proving that phase 1 preserves convergence to $\mathcal{L}$ from Q.

Phase 2. This phase makes each process of p_{det} act in such a way that it skips intermediate states where it would otherwise be enabled, defaulting to be a self-loop action if all "intermediate" states cause the process to be enabled. Fairness $\mathcal{F} \in \{\text{unfair}, \text{weak}\}$ does not guarantee that a process π_j will act between any two actions of another process π_i. This shows that, at this phase of construction, all maximal executions of p_{det} under $\mathcal{F}$ correspond to maximal executions of p under $\mathcal{F}$, but some intermediate states are skipped. Thus, no deadlocks or livelocks are introduced.

Convergence to $\mathcal{L}$ from Q also implies a safety constraint that no transition exists from $Q \setminus \mathcal{L}$ to $\overline{Q} \setminus \mathcal{L}$. However, we have assumed that no single process of p can cause an execution $\langle s_0, s_1, s_2 \rangle$ where $s_0 \in Q \setminus \mathcal{L}$, $s_1 \in \mathcal{L}$, and $s_2 \in \overline{Q} \setminus \mathcal{L}$, therefore phase 2 will not introduce any transitions directly from $Q \setminus \mathcal{L}$ to $\overline{Q} \setminus \mathcal{L}$. In total, this proves that phase 2 preserves convergence to $\mathcal{L}$ from Q.

Phase 3. Self-loops do not change the state of a system, therefore they can be removed without affecting convergence to $\mathcal{L}$ from Q. Phase 3 does this exactly, therefore all 3 phases preserve convergence, completing the proof that p_{det} converges to $\mathcal{L}$ from Q under fairness $\mathcal{F}$ using deterministic and self-disabling processes. □

Theorem 4.1.5. *Let p be a shadow protocol operating within legitimate states $\mathcal{L}$ that uses deterministic and self-disabling processes. If a shadow protocol p' stabilizes to $p|\mathcal{L}$ under fairness $\mathcal{F} \in \{\text{unfair}, \text{weak}\}$, then p' can be modified to have deterministic and self-disabling processes without sacrificing its stabilization property.*

Proof. Let $\mathcal{H}$ be the projection function that maps a state of p' to a state of p. Let $\mathcal{L}' \subseteq \mathcal{L}$ be a set of legitimate states satisfying the behavioral (Definition 2.2.10) and closure (Definition 2.2.11) properties. Let ${p'}_{\text{det}}$ be a version of p' that has been modified using the technique in Theorem 4.1.4 such that its "phase 1" prefers to remove actions that do not change shadow variables. We want to show that the convergence and closure constraints that are satisfied by p' are also satisfied by ${p'}_{\text{det}}$.

Our new ${p'}_{\text{det}}$ preserves closure within $\mathcal{L}'$ (Constraint 1 in Definition 2.2.11) since any execution of ${p'}_{\text{det}}$ is also an execution of p', possibly skipping some intermittent states. By the same logic, no execution of ${p'}_{\text{det}}$ within $\mathcal{L}'$ changes shadow variables unless they are changed by a similar execution of p'. This satisfies the fixpoint constraint of convergence (Constraint 3 in Definition 2.2.10). Since $\mathcal{L}'$ is closed, we also know that ${p'}_{\text{det}}$ converges to $\mathcal{L}'$ by Theorem 4.1.4. This satisfies the recovery constraint of convergence (Constraint 1 in Definition 2.2.10).

Since processes of p are deterministic, each transition of p from a state $s_0 \in \mathcal{L}$ corresponds to a different process, implying that each transition of p' that changes shadow variables from a state in $\mathcal{H}^{-1}[\{s_0\}] \cap \mathcal{L}'$ corresponds to a different process. Furthermore, since no transitions (s_0, s_1) and (s_1, s_2) of p belong to the same process, no single process of p' transitions both from $\mathcal{H}^{-1}[\{s_0\}] \cap \mathcal{L}'$ to $\mathcal{H}^{-1}[\{s_1\}] \cap \mathcal{L}'$ and from $\mathcal{H}^{-1}[\{s_1\}] \cap \mathcal{L}'$ to $\mathcal{H}^{-1}[\{s_2\}] \cap \mathcal{L}'$. Thus, the construction of ${p'}_{\text{det}}$ preserves such transitions of p' within $\mathcal{L}'$ that change shadow variables. This satisfies the transition closure constraint (Constraint 2 in Definition 2.2.11), though it would not satisfy the stricter constraint of preserving all execution paths (which we believe may be useful, but it provides no extra constraints for any of the protocols that we investigate in this work).

Furthermore, the construction of ${p'}_{\text{det}}$ preserves convergence from $\mathcal{H}^{-1}[\{s_0\}] \cap \mathcal{L}'$ to $\bigcup_{i=j}^{n}(\mathcal{L}' \cap \mathcal{H}^{-1}[\{s_j\}])$ for any such $n \geq 1$ transitions $(s_0, s_1), \ldots, (s_0, \ldots, s_n)$ because

processes of p' meet the conditions of Theorem 4.1.4. This satisfies the progress constraint of convergence (Constraint 2 in Definition 2.2.10). Thus, we have shown that a ${p'}_{\text{det}}$ stabilizes to $p|\mathcal{L}$. □

4.2 Overview of the Search Algorithm

Figure 4.1 illustrates an abstract flowchart of the proposed backtracking algorithm. We start with the non-stabilizing protocol p, its legitimate states $\mathcal{L}$, the topology of p', and the projection function $\mathcal{H}^{-1}$ that maps states of p' to states of p. The algorithm in Figure 4.1 starts by computing all *valid* candidate actions (in the expanded state space) that adhere to the read/write permissions of all processes. The initial value of `delegates` is often the empty set unless there are specific actions that must be in the solution (e.g., to ensure the reachability of particular states). The algorithm in Figure 4.1 then calls ReviseActions to remove self-loops from `candidates` (since they violate convergence), and checks for inconsistencies in the partial solution. The designer may give additional constraints that forbid certain actions.

In general, ReviseActions (see the bottom dashed box in Figure 4.1) is invoked whenever we strengthen the partial solution by adding to the under-approximation or removing from the over-approximation. It may further remove from the over-approximation by enforcing the determinism and self-disablement constraints (see Theorem 4.1.4). Then ReviseActions computes the largest possible invariant $\mathcal{L}'$ that could be used by the current partial solution. That is, it finds the weakest predicate $\mathcal{L}'$ for which the constraints of Problem 4.1.1 can be satisfied using some set of transitions Δ' permissible by the partial solution. The partial solution requires Δ' to include all transitions corresponding to actions in `delegates`. Additionally, Δ' can include any subset of transitions corresponding to actions in `candidates`. For example, the first constraint of convergence to shadow behavior (Definition 2.2.10) stipulates that the transitions of `delegates` are cycle-free outside of $\mathcal{L}'$ and that the transitions of `delegates` $\cup$ `candidates` provide reachability (weak convergence) to $\mathcal{L}'$. If such an $\mathcal{L}'$ does not exist, then the partial solution is inconsistent.

If our initialized `delegates` and `candidates` give a consistent partial solution, then we invoke the AddStabilizationRec routine. The objective of AddStabilizationRec (see the top dashed box in Figure 4.1) is to go through all actions in `candidates` and check their eligibility for inclusion in the self-stabilizing solution. In particular, AddStabilizationRec has a loop that iterates through all actions of `candidates` until it becomes empty or an inconsistency is found. In each iteration, AddStabilizationRec picks a candidate action to resolve some remaining deadlock at the next decision level. In general, the candidate action can be randomly selected. However, to limit the possible choices, we use an intelligent method for picking candidate actions described in Section 4.2. After picking a new action A, we invoke ReviseActions to add action A to a copy of the current partial solution by

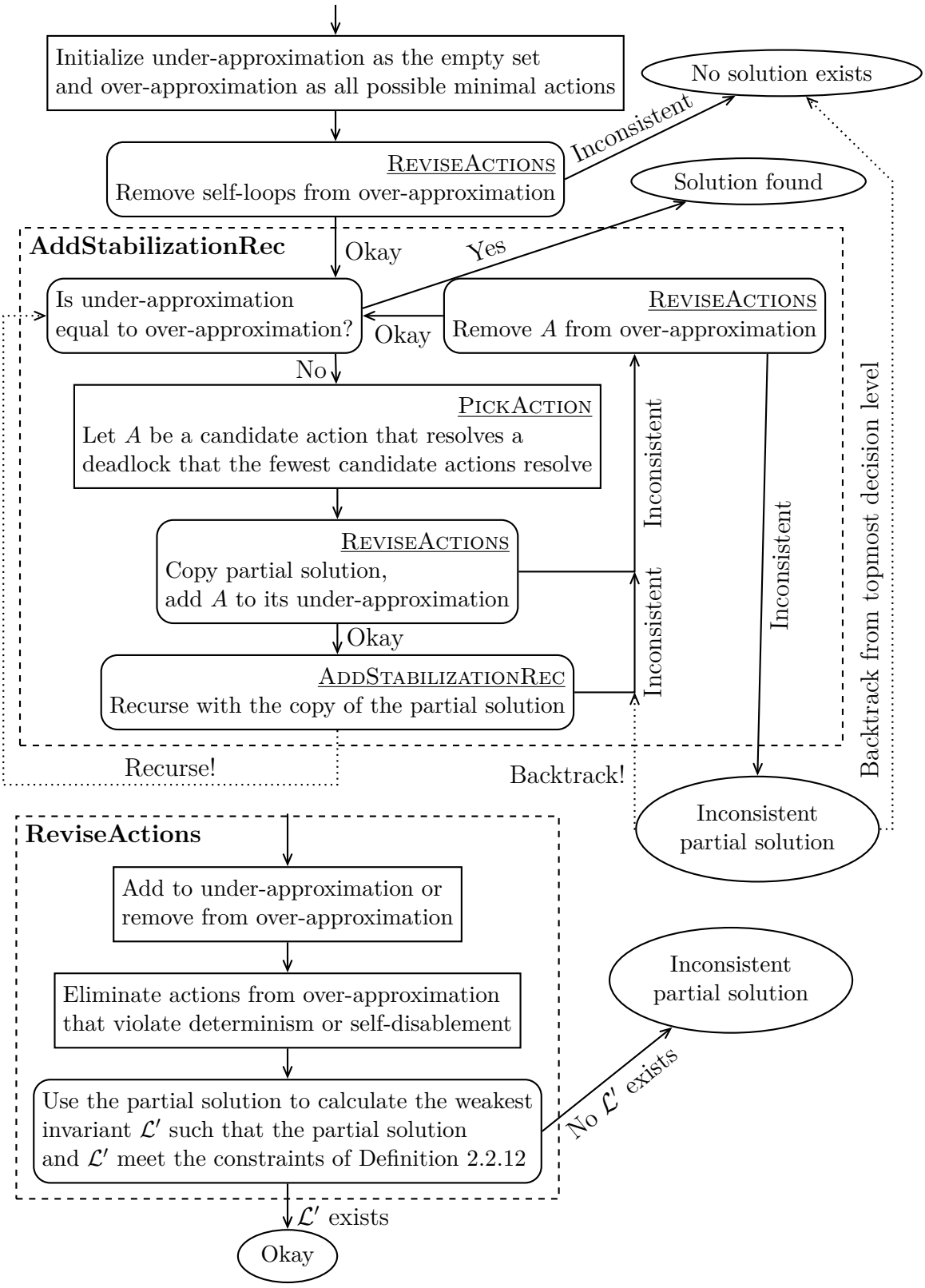


Figure 4.1: Overview of the backtracking algorithm. © 2016 IEEE [133].

including A in the copy of `delegates` and removing it from the copy of `candidates`. If the copied partial solution is consistent, then AddStabilizationRec makes a recursive call to itself, using the copied partial solution for the next decision level. If the copied partial solution is found to be inconsistent (either by a call to ReviseActions or by the exhaustive search in the call to AddStabilizationRec), then we remove action A from `candidates` using ReviseActions. If after removal of A the partial solution is consistent, then we continue in the loop. Otherwise, we backtrack since no stabilizing protocol exists with the current under-approximation.

4.3 Algorithm Details

This section presents the details of the proposed backtracking method. Notice that we assume that the input to this algorithm is a non-stabilizing shadow protocol already superposed with some new finite-domain puppet variables. Misusing C/C++ notation, we prefix a function parameter with an ampersand (`&`) if modifications to it will affect its value in the caller's scope (i.e., it is a return parameter).

AddStabilization. Algorithm 4.1 is the entry point of our backtracking algorithm. The AddStabilization function returns **true** *iff* a self-stabilizing protocol is found which will then be formed by the actions in `delegates`. Initially, the function determines all possible candidate minimal actions. Next, the function determines which actions are explicitly required (Line 2) or disallowed by additional constraints (Line 4). We invoke ReviseActions to include `adds` in the under-approximation and remove `dels` from the over-approximation on Line 6. If the resulting partial solution is consistent, then the recursive version of this function (AddStabilizationRec) is called. Otherwise, a solution does not exist.

AddStabilizationRec. Algorithm 4.2 defines the main recursive search. Like AddStabilization, it returns **true** *iff* a self-stabilizing protocol is found that is formed by the actions in `delegates`. This function continuously adds candidate actions to the under-approximation `delegates` as long as candidate actions exist. If no candidates remain, then `delegates` and the over-approximation `delegates` $\cup$ `candidates` of the protocol are identical. If ReviseActions does not find anything wrong, then `delegates` is self-stabilizing, hence the successful return on Line 16.

On Line 2 of AddStabilizationRec, a candidate action A is chosen by calling PickAction (Algorithm 4.4). Any candidate action may be picked without affecting the search algorithm's correctness, but the next section explains a heuristic we use to pick certain candidate actions over others to improve search efficiency. After picking an action, we copy the current partial solution into `delegates`$'$ and `candidates`$'$, and add the action A on Line 6. If the resulting partial solution is consistent, then we recurse by calling AddStabilizationRec. If that recursive call finds a self-stabilizing protocol, then it will store its actions in `delegates` and return successfully. Otherwise, if action A does not yield a solution, we will remove it from the candidates

Algorithm 4.1 Entry point of the backtracking algorithm for solving Problem 4.1.1.

AddStabilization(p: shadow protocol, $\mathcal{L}$: legitimate states,
$\&p'$: shadow/puppet topology, $\mathcal{H}$: mapping $S' \to S$,
`delegates`: forced actions, `forbidden`: forbidden actions)

Output: Return **true** when a solution `delegates` can be found. Otherwise, **false**.

1: **let** `candidates` be all local transitions that are valid for the process types in p'
2: **let** `adds` := `delegates` {Forced actions, if any}
3: `delegates` := $\emptyset$
4: **let** `dels` := `candidates` $\cap$ `forbidden`
5: **let** $\mathcal{L}' := \emptyset$
6: **if** ReviseActions(p, $\mathcal{L}$, $\mathcal{H}$, `&delegates`, `&candidates`, $\&\mathcal{L}'$, `adds`, `dels`) **then**
7: **if** AddStabilizationRec(p, $\mathcal{L}$, $\mathcal{H}$, `&delegates`, `candidates`, $\mathcal{L}'$) **then**
8: Modify p' to use the actions in `delegates`
9: **return true**
10: **end if**
11: **end if**
12: **return false**

Algorithm 4.2 Recursive backtracking function to add stabilization.

AddStabilizationRec(p, $\mathcal{L}$, $\mathcal{H}$, `&delegates`, `candidates`, $\mathcal{L}'$)

Output: Return **true** if `delegates` contains the solution. Otherwise, return **false**.

1: **while** `candidates` $\neq \emptyset$ **do**
2: **let** A := PickAction(p, $\mathcal{H}$, `delegates`, `candidates`, $\mathcal{L}'$)
3: **let** `delegates`$'$:= `delegates`
4: **let** `candidates`$'$:= `candidates`
5: **let** $\mathcal{L}'' := \emptyset$
6: **if** ReviseActions(p, $\mathcal{L}$, $\mathcal{H}$, `&delegates`$'$, `&candidates`$'$, $\&\mathcal{L}''$, $\{A\}$, $\emptyset$) **then**
7: **if** AddStabilizationRec(p, $\mathcal{L}$, $\mathcal{H}$, `&delegates`$'$, `candidates`$'$, $\mathcal{L}''$) **then**
8: `delegates` := `delegates`$'$ {Assign the actions to be returned}
9: **return true**
10: **end if**
11: **end if**
12: **if not** ReviseActions(p, $\mathcal{L}$, $\mathcal{H}$, `&delegates`, `&candidates`, $\&\mathcal{L}'$, $\emptyset$, $\{A\}$) **then**
13: **return false**
14: **end if**
15: **end while**
16: **return true**

on Line 12. If this removal creates a non-stabilizing protocol, then return in failure; otherwise, continue the loop.

ReviseActions. Algorithm 4.3 is a key component of the backtracking search.

Algorithm 4.3 Add `adds` to the under-approximation and remove `dels` from the over-approximation.

ReviseActions(p, $\mathcal{L}$, $\mathcal{H}$, `&delegates`, `&candidates`, `&`$\mathcal{L}'$, `adds`, `dels`)

Output: Return **true** if `adds` can be added to `delegates` and `dels` can be removed from `candidates`, and $\mathcal{L}'$ can be revised accordingly. Otherwise, return **false**.

1: `delegates` := `delegates` $\cup$ `adds`
2: `candidates` := `candidates` $\setminus$ `adds`
3: **for** $A \in$ `adds` **do**
4: Add each action $B \in$ `candidates` to `dels` if it belongs to the same process as A and satisfies one of the following conditions:
- A enables B (enforce self-disabling process)
- B enables A (enforce self-disabling process)
- A and B are enabled at the same time (enforce determinism)

{Find candidate actions that are now trivially unnecessary for stabilization}
5: **end for**
6: `candidates` := `candidates` $\setminus$ `dels`
7: Compute the maximal $\mathcal{L}'$ and $\Delta'|\mathcal{L}'$ using the method of Lemma 4.1.3
8: Check for inconsistencies as per Lemma 4.1.2
9: **if** no inconsistency found **then**
10: `adds` := $\emptyset$
11: `dels` := $\emptyset$
12: **if** CheckForward(p, $\mathcal{L}$, $\mathcal{H}$, `delegates`, `candidates`, $\mathcal{L}'$, `&adds`, `&dels`) **then**
13: **if** `adds` $\neq \emptyset$ **or** `dels` $\neq \emptyset$ **then**
14: **return** ReviseActions(p, $\mathcal{L}$, $\mathcal{H}$, , `&delegates`, `&candidates`, `&`$\mathcal{L}'$, `adds`, `dels`)
15: **end if**
16: **return true**
17: **end if**
18: **end if**
19: **return false**

ReviseActions performs five tasks: (1) Add actions to the under-approximated protocol by moving the `adds` set from `candidates` to `delegates`. (2) Remove forbidden actions from the over-approximated protocol by removing the `dels` set from `candidates`. (3) Enforce self-disablement and determinism (Theorem 4.1.4), which results in removing more actions from the over-approximated protocol. (4) Compute the maximal invariant $\mathcal{L}'$ and transitions $\Delta'|\mathcal{L}'$ in the expanded state space such that the shadow behavior may be achieved given the current under/over-approximations. (5) Check that there are no inconsistencies by ensuring that the under-approximation is livelock-free within $\overline{\mathcal{L}'}$, the over-approximation provides reachability to $\mathcal{L}'$ from any state, and closure properties are preserved.

If the check finds an inconsistency, then ReviseActions returns **false**. Finally,

ReviseActions invokes the CheckForward function to infer actions that must be added to the under-approximation or removed from the over-approximation, and will return **false** only if it infers that the current partial solution cannot be used to form a self-stabilizing protocol. A trivial version of CheckForward can just return **true**.

A good ReviseActions implementation should provide early detection for when `delegates` and `candidates` cannot be used to form a self-stabilizing protocol. At the same time, since the function is called whenever converting candidates to delegates or removing candidates, it cannot have a high cost. Thus, we ensure that actions in `delegates` do not form a livelock and that actions in `delegates` $\cup$ `candidates` provide weak stabilization.

A good CheckForward implementation should at least remove candidate actions that are not needed to resolve deadlocks. This can be performed quickly and allows the AddStabilizationRec function to immediately return a solution when all deadlocks are resolved.

Theorem 4.3.1 (Completeness)**.** *The* AddStabilization *algorithm is complete.*

Proof. Assuming that AddStabilization returns **false**, we want to prove that no solution exists. Since each candidate action is minimal and we consider all such actions as candidates, a subset of the candidate actions form a stabilizing protocol *iff* such a protocol exists. Observe that AddStabilizationRec follows the standard backtracking [134] procedure where we (1) add a candidate action to the under-approximation in a new decision level, and (2) backtrack and remove that action from the candidates if an inconsistency (which cannot be fixed due to Lemma 4.1.2) is discovered by ReviseActions at that new decision level. Even though ReviseActions removes candidate actions in order to enforce deterministic and self-disabling processes, we know by Theorem 4.1.5 that this will not affect the existence of a self-stabilizing protocol. Thus, since we follow the general template of backtracking [134], the search will test every consistent subset of the initial list of candidate actions where processes are deterministic and self-disabling. Therefore, if our search fails, then no solution exists. □

Theorem 4.3.2 (Soundness)**.** *The* AddStabilization *algorithm is sound.*

Proof. We show that if AddStabilization returns **true**, then it has found a self-stabilizing protocol formed by the actions in `delegates`. Notice that when AddStabilizationRec returns **true**, the AddStabilization or AddStabilizationRec function that called it simply returns **true** with the same `delegates` list. The only other case where AddStabilization returns **true** is when `candidates` is empty in AddStabilizationRec (Line 16). Notice that to get to this point, ReviseActions must have been called and must have returned **true** after emptying the `candidates`

list. By inspection of Lemma 4.1.3 and Lemma 4.1.2, verifying the constraints of Definition 2.2.12 is equivalent to building $\mathcal{L}'$ and $\Delta'|\mathcal{L}'$ using an empty `candidates` list and then finding no inconsistencies equivalent to verifying. Therefore, when AddStabilization returns **true** the actions of `delegates` form a self-stabilizing protocol. □

4.4 Optimizing the Decision Tree

This section presents the techniques that we use to improve the efficiency of our backtracking algorithm.

Picking Actions via the Minimum Remaining Values Heuristic. The worst-case complexity of a depth-first backtracking search is determined by the branching factor b and depth d of its decision tree, evaluating to $O(b^d)$. We can tackle this complexity by reducing the branching factor. To do this, we use a minimum remaining values (MRV) method in PickAction. MRV is classically applied to constraint satisfaction problems [162] by assigning a value to a variable that has the minimal remaining candidate values. In our setting, we pick an action that resolves a deadlock with the minimal number of remaining actions available to resolve it.

Algorithm 4.4 shows the details of PickAction that keeps an array `deadlock_sets`, where each element `deadlock_sets[`i`]` contains all the deadlocks that are resolved by exactly i candidate actions. We initially start with array size |`deadlock_sets`| = 1 and with `deadlock_sets[0]` containing all unresolved deadlocks. We then shift deadlocks to the next highest element in the array (bubbling up) for each candidate action that resolves them. After building the array, we find the lowest index i for which the deadlock set `deadlock_sets[`i`]` is nonempty, and then return an action that can resolve some deadlock in that set. Line 21 can only be reached if either the remaining deadlocks cannot be resolved (but ReviseActions catches this earlier) or all deadlocks are resolved (but CheckForward can catch this earlier).

When multiple topologies are being considered, PickAction is performed on all topologies, each corresponding to its own `deadlock_sets` array. The check of Line 17 is performed for each topology. In this way, if all deadlocks are resolved by 6 different actions on a ring of size $N = 3$, but there is a deadlock that only be resolved by 2 actions on a ring of size $N = 5$, then we will choose from the 2 actions rather than the 6.

Conflicts. Every time a new candidate action is included in `delegates`, ReviseActions checks for inconsistencies, which involves cycle detection and reachability analysis. These procedures become very costly as the complexity of the transition system grows. To mitigate this problem, whenever an inconsistency is found, we record a minimal set of decisions (subset of `delegates`) that causes it. We reference these *conflict sets* [162] in CheckForward to remove candidate actions that would cause an inconsistency.

Algorithm 4.4 Pick an action using the minimum remaining values (MRV) method.

PickAction(p, $\mathcal{H}$, `delegates`, `candidates`, I')

Output: Next candidate action to pick.

```
1: let deadlock_sets be a single-element array, where deadlock_sets[0] holds a
   set of deadlocks in ℒ' ∪ ℋ^{-1}[Pre(Δ)] that actions in delegates do not resolve
2: for all action ∈ candidates do
3:   let i := |deadlock_sets|
4:   while i > 0 do
5:     i := i − 1
6:     let resolved := deadlock_sets[i] ∩ Pre(action)
7:     if resolved ≠ ∅ then
8:       if i = |deadlock_sets| − 1 then
9:         let deadlock_sets[i + 1] := ∅ {Grow array by one element}
10:      end if
11:      deadlock_sets[i] := deadlock_sets[i] \ resolved
12:      deadlock_sets[i + 1] := deadlock_sets[i + 1] ∪ resolved
13:    end if
14:  end while
15: end for
16: for i = 1, ..., |deadlock_sets| − 1 do
17:   if deadlock_sets[i] ≠ ∅ then
18:     return Any action from candidates that resolves a deadlock in
        deadlock_sets[i]
19:   end if
20: end for
21: return An action from candidates {Edge case}
```

Randomization and Restarts. When using the standard control flow of a depth-first search, a bad choice near the top of the decision tree can lead to infeasible runtime. This is the case since the bad decision exists in the partial solution until the search backtracks up the tree sufficiently to change the decision. To limit the search time in these branches, we employ a method outlined by Gomes et al. [98] that combines randomization with restarts. In short, we limit the amount of backtracking to a certain height (we use 3). If the search backtracks past the height limit, it forgets the current decision tree and restarts from the root. To avoid trying the same unfruitful decisions after a restart, PickAction randomly selects a candidate action permissible by the MRV method. This approach remains complete since a conflict is recorded for any set of decisions that causes a restart.

Parallel Search. In order to increase the chance of finding a solution, we instantiate several parallel executions of the algorithm in Figure 4.1; i.e., *search diversification*. As noted in [98], parallel tasks will generally avoid overlapping computations due to the randomization used in PickAction. We have observed linear speedup when

the search tasks are expected to restart several times before finding a solution [131]. The parallel tasks share conflicts with each other to prevent the re-exploration of branches that contain no solutions. In our MPI implementation, conflict dissemination occurs between tasks using a virtual network topology formed by a generalized Kautz graph [116] of degree 4. This topology has a diameter logarithmic in the number of nodes and is fault-tolerant in that multiple paths between two nodes ensure message delivery. That is, even if some nodes are performing costly cycle detection and do not check for incoming messages, they will not slow the dissemination of new conflicts.

4.5 Probabilistic Stabilization

It may seem that probabilistic stabilization is fundamentally incompatible with our synthesis algorithm since we assume processes are deterministic (and self-disabling). However as discussed in Section 2.3.3, we can grant processes the power of random choice by granting them access to their own read-only variables that change randomly whenever any process acts. In this way, a processes can behave deterministically based on random values they read.

Since a randomized variable can hold a specific value for arbitrarily many steps, we only consider a process to be involved in a livelock if its actions within that livelock are fully determined (for all values of its randomized variables). Therefore, we can say that if a livelock exists, no action can be added to resolve it. This makes our notion of a partial solution valid in backtracking search.

Cycle Detection. Though we are still using an unfair scheduler, our usual cycle detection algorithm (Algorithm 3.1) cannot be used. Algorithm 4.5 shows the new algorithm that respects ability of processes to break cycles based on random choice. It does not rely on our use of randomized variables because all process nondeterminism as random choice. However, since our synthesis algorithm enforces deterministic processes, synthesized protocols limit nondeterminism to changes in random variables. Such an algorithm is not particularly novel, because symbolic model checkers such as PRISM exist that can verify probabilistic stabilization [110, 143] and many other properties, but it may be useful nonetheless due to its simplicity.

The cycle detection algorithm operates on a simple premise, that if every enabled process in a state can transition out of a livelock, then a livelock will almost surely not revisit that state infinitely often. This is the idea of the fixpoint iteration starting on Line 5. During each iteration, we find states for which each process is either disabled or has a transition to leave `span`, which is the current set of states that could be involved in cycles. These resolved states will not be in `span` in the next iteration.

When no more states can be removed by the preimage fixpoint iteration, we invoke an image fixpoint iteration on Line 15. This removes the remaining states that cannot be

Algorithm 4.5 Check for cycles in a protocol where processes have random choice.

SureCycleCheck(`&span`: closed set of initial states (also a return value),
$\delta_0,\ldots,\delta_{N-1}$: transitions of each process,
`progress`: progress transitions)

Output: Whether the protocol contains a cycle.

1: {Note: `progress` is either (1) transitions that directly recover to the maximal closed set of legitimate states, or (2) transitions corresponding to the shadow protocol, but not both}
2: **let** $\Delta := \bigcup_{i=0}^{N-1} \delta_i$ {Global transitions}
3: {Fixpoint iteration using preimage}
4: **let** `next` := Pre(Δ) {Contains the next value of `span`, states that could be involved in cycles, but also includes states outside of `span` for efficiency (these states are ultimately ignored)}
5: **repeat**
6: `span` := `span` $\cap$ `next`
7: `next` := $\emptyset$
8: **for** $i = 0, \ldots, N-1$ **do**
9: **let** `resolved` := Pre($\delta_i \cap$ `progress`) $\cup$ $\delta_i^{-1}[\overline{\texttt{span}}]$
10: `next` := `next` $\cup$ (Pre(δ_i) $\setminus$ `resolved`)
11: **end for**
12: **until** `span` $\subseteq$ `next`
13: {Fixpoint iteration using image to make `span` resemble the SCCs more closely}
14: `next` := Img(Δ)
15: **repeat**
16: `span` := `span` $\cap$ `next`
17: `next` := Δ[`span`]
18: **until** `span` $\subseteq$ `next`
19: {`span` is now all states that can be visited after arbitrarily many steps in an infinite execution, but unlike in an SCC, `span` may contain some states that cannot be visited infinitely often}
20: **return** (`span` $\neq \emptyset$)

visited after arbitrarily many steps in an infinite execution. Algorithm 3.1 similarly has two fixpoint iterations, but it performs image before preimage.

Chapter 5: Case Studies

In this chapter, we look for exact lower bounds for constant-space maximal matching and token passing protocols using shadow/puppet synthesis. Section 5.1 presents an intuitive way to synthesize maximal matching with shadow variables. Section 5.2

Protocol		Procs	Verified Procs	MPI Procs	Synthesis Time
2-State Ring Matching	⋆†	2–7	8–100	1	0.54 secs
3-State Ring Matching [76]	†	2–7	8–30	1	0.40 secs
4-State Token Ring	⋆†	2–8	N/A	4	1.50 hrs
4-State Token Ring	⋆	2–8	N/A	64	100 hours
5-State Token Ring	⋆†	2–9	10–30	4	23.75 mins
4-State Token Ring		2–8	N/A	4	7.46 mins
6-State Token Ring		2–9	10–30	4	20.30 mins
8-State Token Ring [102]		2–9	10–25	4	17.29 mins
3-State Token Chain	⋆†	2–5	N/A	4	5.29 secs
3-State Token Chain	⋆	2–5	6–30	4	48.93 secs
4-State Token Chain [64]	⋆†	2–4	5–25	4	10.11 secs
4-State Token Chain [64]	⋆	2–4	5–15	4	1.14 mins
3-State Token Ring [64]	⋆†	2–5	6–15	4	1.02 mins
3-State Token Ring [64]	⋆	2–5	6–15	4	5.22 mins
Daisy Chain Orientation	†	2–6	7–24	1	2.02 mins
Odd-Sized Ring Orientation [112]		3,5,7	9,11	4	34.09 mins
⋆: Shadow variables are write-only			N/A: No solution exists		
†: Shadow self-loops are forbidden			Others assumed generalizable		

Figure 5.1: Synthesis runtimes for case studies.

Discussion of the main 3 protocols in Sections 5.1–5.3 contains material from A. P. Klinkhamer and A. Ebnenasir. Shadow/Puppet Synthesis: A Stepwise Method for the Design of Self-Stabilization. *IEEE Transactions on Parallel and Distributed Systems*, 2016.

Superior, a high performance computing cluster at Michigan Technological University, was used in obtaining results presented in this chapter.

explores the unidirectional token ring with a distinguished process. Section 5.3 explores bidirectional token passing protocols. Section 5.4 introduces the problem of orientation on a topology that could either be a ring or chain. Each section gives a new self-stabilizing protocol that we conjecture uses the minimal number of states per process to achieve stabilization.

Figure 5.1 provides the synthesis runtimes of all case studies. Each problem may have a $\star$ or † symbol. The $\star$ symbol indicates whether shadow variables are write-only, which allows the specified behavior to be more expressive (e.g., the token ring in Section 1.1 vs the one in Example 2.2.13). The † symbol indicates whether actions within the $\mathcal{L}'$ invariant may leave shadow variables unchanged. This is achieved by modifying Line 7 of Algorithm 4.3 to enforce that $(\Delta'|\mathcal{L}') \subseteq \mathcal{H}^{-1}[\Delta]$. Forbidding these actions makes every action of a token passing protocol actually pass a token.

The *Procs* column indicates the system sizes (numbers of processes) that are simultaneously considered during synthesis. For example, the maximal matching case shows 2–7, which means that we are resolving deadlocks without introducing livelocks for 6 different systems of various sizes. These ranges are chosen to increase the chance of synthesizing a generalizable protocol. The *Verified Procs* column shows the system sizes that were verified after synthesis. We believe that the protocols that we verified are generalizable. The *MPI Procs* column indicates the number of MPI processes used for synthesis.

5.1 2-State Maximal Matching on Rings

A matching for a graph is a set of edges that do not share any common vertices. A matching is *maximal iff* adding any new edge would violate the matching property. In a ring, a set of edges is a matching as long as at least 1 of every 2 consecutive edges is excluded from the set. For the set to be a maximal matching, at least 1 out of every 3 consecutive edges must be included. To see that the matching is maximal, consider selecting another edge to create a new matching. The edge itself cannot be in the current matching nor can either of the two adjacent edges, but we have already enforced that one of those three is selected, therefore the new edge cannot be added!

To specify this problem, use a binary shadow variable e_i to denote whether the edge between adjacent processes P_{i-1} and P_i is in the matching ($e_i = 1$ means to include the edge, otherwise exclude the edge). The legitimate states are therefore the states where at least 1 of every 3 consecutive e values equals 1, but at least 1 of every 2 consecutive e values equals 0.

$$\mathcal{L}_{\text{match}} \equiv \forall i \in \mathbb{Z}_N : ((e_{i-1} = 1 \vee e_i = 1 \vee e_{i+1} = 1) \wedge (e_i = 0 \vee e_{i+1} = 0))$$

We would like processes to determine whether their neighboring links are included in a matching, therefore we give each P_i write access to e_i and e_{i+1}. Each e_i is a

shadow variable since it exists only for specification, therefore processes have write-only access. Since a matching should not change, there are no shadow actions; i.e., *silent* stabilization. We give each process P_i a binary puppet variable x_i to read and write along with read access to the x_{i-1} and x_{i+1} variables of its neighbors in the ring. We only are guessing that an x_i domain size of 2 is large enough to achieve stabilization and represent (e_i, e_{i+1}) values with (x_{i-1}, x_i, x_{i+1}) values in some way. Synthesis gives the protocol in Protocol 5.1, which we believe to be generalizable after verification of rings up to size $N = 100$. Notice that the e values are fully determined based on the x values.

Protocol 5.1 — 2-State Maximal Matching on Bidirectional Rings

$$
\begin{aligned}
&P_i :\ x_{i-1} = 1 \wedge x_i = 1 \wedge x_{i+1} = 1 \longrightarrow e_i := 1;\ x_i := 0;\ e_{i+1} := 0;\\
&P_i :\ x_{i-1} = 0 \wedge x_i = 1 \wedge x_{i+1} = 1 \longrightarrow e_i := 0;\ x_i := 0;\ e_{i+1} := 0;\\
&P_i :\ x_{i-1} = 0 \wedge x_i = 0 \wedge x_{i+1} = 0 \longrightarrow e_i := 0;\ x_i := 1;\ e_{i+1} := 1;\\
&P_i :\ x_{i-1} = 1 \wedge x_i = 0 \longrightarrow e_i := 1;\ e_{i+1} := 0;\\
&P_i :\ x_{i-1} = 0 \wedge x_i = 0 \wedge x_{i+1} = 1 \longrightarrow e_i := 0;\ e_{i+1} := 0;\\
&P_i :\ x_{i-1} = 0 \wedge x_i = 1 \wedge x_{i+1} = 0 \longrightarrow e_i := 0;\ e_{i+1} := 1;
\end{aligned}
$$

Shadow Variables: $e_0 \ldots e_{N-1} \in \mathbb{Z}_2$ (write-only)
Puppet Variables: $x_0 \ldots x_{N-1} \in \mathbb{Z}_2$
Legitimate Shadow States: $\mathcal{L}_{\text{match}} \equiv \forall i \in \mathbb{Z}_N :\ (\quad (e_{i-1} = 1 \vee e_i = 1 \vee e_{i+1} = 1) \wedge (e_i = 0 \vee e_{i+1} = 0))$

Optimization. Since the number of possible protocols is relatively small, this particular protocol is the result of an exhaustive search that optimizes the worst-case number of steps to achieve convergence. This exhaustive search only considered rings of size 2–7 and took less than 30 seconds. We have found that this kind of optimization can yield protocols that are easier to verify.

Removing Shadow Variables. An implementation of this matching protocol consists only of puppet variables, and therefore discards the e variables. Of the 6 actions above, only the first 3 modify x values. From these 3, we discard the e variables and combine the first 2 actions. This leaves us with the puppet protocol that would be used for implementation, where each P_i has the following actions:

$$
\begin{aligned}
&P_i :\ x_i = 1 \wedge x_{i+1} = 1 \longrightarrow x_i := 0;\\
&P_i :\ x_{i-1} = 0 \wedge x_i = 0 \wedge x_{i+1} = 0 \longrightarrow x_i := 1;
\end{aligned}
$$

Further, we can derive the meaning of the puppet variables by observing how the value of (e_i, e_{i+1}) is assigned for each particular value of (x_{i-1}, x_i, x_{i+1}). We could

do this by observing all 6 actions, but we only need to observe the first 3 since they subsume the last 3. The last 3 actions respectively assign $(1,0)$, $(0,0)$, and $(0,1)$ to (e_i, e_{i+1}) to denote that P_i is matched with (i) P_{i-1}, (ii) nothing, and (iii) P_{i+1}. We use (x_{i-1}, x_i, x_{i+1}) values to represent these cases:

$$\begin{aligned}
&x_{i-1} = 1 \wedge x_i = 0 && (P_i \text{ matched with } P_{i-1})\\
&x_{i-1} = 0 \wedge x_i = 0 \wedge x_{i+1} = 1 && (P_i \text{ not matched})\\
&x_{i-1} = 0 \wedge x_i = 1 \wedge x_{i+1} = 0 && (P_i \text{ matched with } P_{i+1})
\end{aligned}$$

3-State Version

Gouda and Acharya [101] give a stabilizing matching protocol for rings and treat the legitimate states as a Nash equilibrium, using elements of game theory to reason about convergence. Their version uses 4 states per process. Ebnenasir and Farahat [76] automatically synthesize a maximal matching using 3 states per process. Each process P_i has a variable $m_i \in \mathbb{Z}_3$ whose values indicate whether P_i is matched with P_{i-1}, not matched, or matched with P_{i+1}. Using symbols to indicate these respective scenarios $\mathtt{L} = 0$, $\mathtt{S} = 1$, and $\mathtt{R} = 2$, the legitimate states are those that satisfy the following for each P_i:

$$\begin{aligned}
&m_{i-1} = \mathtt{R} \wedge m_i = \mathtt{L} && (P_i \text{ matched with } P_{i-1})\\
&m_{i-1} = \mathtt{L} \wedge m_i = \mathtt{S} \wedge m_{i+1} = \mathtt{R} && (P_i \text{ not matched})\\
&m_i = \mathtt{R} \wedge m_{i+1} = \mathtt{L} && (P_i \text{ matched with } P_{i+1})
\end{aligned}$$

Using the same specification, we synthesized the protocol in Protocol 5.2 using an exhaustive search that optimizes worst-case convergence time in the number of asynchronous steps. This exhaustive search only considered rings of size 2–7 and took less than 10 seconds.

Protocol 5.2 — 3-State Maximal Matching on Bidirectional Rings

$$\begin{aligned}
&P_i :: m_{i-1} \neq \mathtt{R} \wedge m_i \neq \mathtt{R} \wedge m_{i+1} \neq \mathtt{R} \longrightarrow x_i := \mathtt{R};\\
&P_i :: m_{i-1} \neq \mathtt{R} \wedge m_i \neq \mathtt{S} \wedge m_{i+1} = \mathtt{R} \longrightarrow x_i := \mathtt{S};\\
&P_i :: m_{i-1} = \mathtt{R} \wedge m_i \neq \mathtt{L} \wedge m_{i+1} \neq \mathtt{L} \longrightarrow x_i := \mathtt{L};
\end{aligned}$$

Shadow Variables: $m_0 \ldots m_{N-1} \in \mathbb{Z}_3$ (read & write)

Legitimate States: $\mathcal{L}_{\text{match3}} \equiv \forall i \in \mathbb{Z}_N : (\quad m_{i-1} = \mathtt{R} \wedge m_i = \mathtt{L}$
$\vee\ m_{i-1} = \mathtt{L} \wedge m_i = \mathtt{S} \wedge m_{i+1} = \mathtt{R}$
$\vee\ \quad m_i = \mathtt{R} \wedge m_{i+1} = \mathtt{L})$

5.2 5-State Token Ring

In the token ring shadow specification p_{tok} discussed in Section 1.1, each process π_i is given a binary shadow variable tok_i that denotes whether the process has a token. The legitimate states are all states where exactly one token exists ($\exists! i \in \mathbb{Z}_N : tok_i = 1$), where N is the number of processes, and each process should eventually pass the token within the legitimate states ($tok_i = 1 \longrightarrow tok_i := 0;\ tok_{i+1} := 1;$). For synthesis, we give each π_i a puppet variable x_i and also let it read x_{i-1}. Like in Dijkstra's token ring [64], we distinguish π_0 as Bot_0 to allow its actions to differ from the other processes, and every other $\pi_{i>0}$ is named P_i. We also force each action in the legitimate states to pass a token by forbidding shadow self-loops. With this restriction, we found that no protocol using 4 states per process is stabilizing for all rings of size $N \in \{2, \ldots, 8\}$. Without this restriction, we also found that in the general case, no unidirectional token ring exists using 4 states per process.

Protocol 5.3 — 5-State Token Ring

$$
\begin{array}{rll}
Bot_0: & x_{N-1} = 0 \wedge x_0 = 0 \longrightarrow x_0 := 1; & tok_0 := 0;\ tok_1 := 1; \\
Bot_0: & x_{N-1} = 1 \wedge x_0 \leq 1 \longrightarrow x_0 := 2; & tok_0 := 0;\ tok_1 := 1; \\
Bot_0: & x_{N-1} > 1 \wedge x_0 > 1 \longrightarrow x_0 := 0; & tok_0 := 0;\ tok_1 := 1; \\
P_i: & x_{i-1} = 0 \wedge x_i > 1 \longrightarrow x_i := \lfloor x_i/4 \rfloor; & tok_i := 0;\ tok_{i+1} := 1; \\
P_i: & x_{i-1} = 1 \wedge x_i \neq 1 \longrightarrow x_i := 1; & tok_i := 0;\ tok_{i+1} := 1; \\
P_i: & x_{i-1} = 2 \wedge x_i \leq 1 \longrightarrow x_i := 2 + x_i; & tok_i := 0;\ tok_{i+1} := 1; \\
P_i: & x_{i-1} \geq 3 \wedge x_i \leq 1 \longrightarrow x_i := 4; & tok_i := 0;\ tok_{i+1} := 1;
\end{array}
$$

Shadow Variables: $tok_0 \ldots tok_{N-1} \in \mathbb{Z}_2$ (write-only)
Puppet Variables: $x_0 \ldots x_{N-1} \in \mathbb{Z}_5$
Legitimate Shadow States: $\mathcal{L}_{\text{tok}} \equiv \exists! i \in \mathbb{Z}_N : (tok_i = 1)$

Using 5 states per process (i.e., each x_i has domain $\mathbb{Z}_5$), the synthesized protocol is not always generalizable, even when we synthesize for all rings of size $N \in \{2, \ldots, 9\}$. However, we can increase our chances of finding a generalizable version by allowing the search to record many solutions and verify correctness for larger ring sizes. After sufficient synthesize-and-verify, we are left with Protocol 5.3, which we think is generalizable after verification of rings up to size $N = 30$.

6-State Token Ring

As shown by Figure 5.1, we also considered on 4-state, 6-state, and 8-state token rings designed using superposition rather than using write-only shadow variables. The 4-state version is from Example 2.2.13, where binary t_i variables defines legitimate

behavior and binary x_i variables help to provide stabilization. Like the other 4-state token ring considered above, we could not find a version that stabilizes for all rings of size $N \in \{2, \ldots, 8\}$. The 6-state token ring uses the same specification as the 4-state version, but the x_i variables are ternary rather than binary. One such protocol is shown in Protocol 5.4, and we believe it is generalizable after verifying it up to rings of size $N = 30$. The 8-state token ring again uses the 4-state specification, but rather than increasing the x_i domains, we a third binary variable is added to each process that no other process can read. Gouda and Haddix [102] such a 3-bit token ring protocol.

Protocol 5.4 — 6-State Token Ring

$$
\begin{array}{rlll}
Bot_0: & x_{N-1} \neq 2 \wedge x_0 \neq 2 \wedge t_{N-1} \leq t_0 & \longrightarrow x_0 := 2 - t_{N-1}; & t_0 := 1 - t_{N-1}; \\
Bot_0: & x_{N-1} = 2 \wedge x_0 = 2 \wedge t_{N-1} \leq t_0 & \longrightarrow x_0 := 1 + t_{N-1}; & t_0 := 1 - t_{N-1}; \\
P_i: & x_{i-1} \neq 2 \wedge & t_{i-1} < t_i \longrightarrow x_i := t_{i-1}; & t_i := t_{i-1}; \\
P_i: & x_{i-1} \neq 2 \wedge x_i = 2 \wedge & t_{i-1} \geq t_i \longrightarrow x_i := t_{i-1}; & \\
P_i: & x_{i-1} = 2 \wedge & t_{i-1} < t_i \longrightarrow x_i := 2; & t_i := t_{i-1}; \\
P_i: & x_{i-1} = 2 \wedge x_i = 0 \wedge & t_{i-1} \geq t_i \longrightarrow x_i := 2; & \\
P_i: & x_{i-1} = 2 \wedge x_i = 1 \wedge & t_{i-1} \geq t_i \longrightarrow x_i := 2; & t_i := t_{i-1};
\end{array}
$$

Shadow Variables: $t_0 \ldots t_{N-1} \in \mathbb{Z}_2$ (read & write)
Puppet Variables: $x_0 \ldots x_{N-1} \in \mathbb{Z}_2$
Legitimate States: $\mathcal{L}_{\text{tok6}} \equiv \mathcal{H}^{-1}[\mathcal{L}_{\text{tok2}}] \equiv \exists! i \in \mathbb{Z}_N : (\quad i = 0 \wedge t_{i-1} = t_i \vee i \neq 0 \wedge t_{i-1} \neq t_i)$

This 6-state protocol p_{tok6} preserves the legitimate states of the shadow protocol p_{tok2} exactly. We saw the same phenomenon in Example 2.2.13 with the 4-state protocol p_{tok4}. The 8-state (3-bit) protocol p_{tok8} of Gouda and Haddix [102] also preserves these legitimate states.

$$\mathcal{H}^{-1}[\mathcal{L}_{\text{tok2}}] \equiv \mathcal{L}_{\text{tok4}} \equiv \mathcal{L}_{\text{tok6}} \equiv \mathcal{L}_{\text{tok8}} \equiv \exists! i \in \mathbb{Z}_N : ((i = 0 \wedge t_{i-1} = t_i) \vee (i \neq 0 \wedge t_{i-1} \neq t_i))$$

Stabilization Time

In analyzing these token rings, we found an interesting efficiency trade-off between process memory, convergence speed, and adherence to the shadow protocol. The 6-state protocol in Protocol 5.4 is a prime example to demonstrate this trade-off since it actually converges to having one enabled process, but it still contains shadow self-loops. This means that we can either consider a process to have a token when it is enabled, or we can continue using the shadow variables (t_i) to determine which processes have tokens. If we choose to consider any enabled process to have a token,

then the token is passed with every action within the legitimate states, which makes normal operation efficient, but it also makes convergence time slower.

Token Ring Protocol	**Number of Processes**												
	2	3	4	5	6	7	8	9	10	11	12	13	13⋆
4-State	0	4	12	20	41	54							
5-State †	0	2	44	75	111	175	227	315	384	495	581	715	140
6-State †	0	2	45	75	112	175	227	315	384	495	581	715	140
6-State	0	2	10	22	37	64	83	127	153	210	243	313	22
8-State [102]	0	3	12	18	46	55	115	128	212	229	337	358	27
N-State [64] †	0	2	13	24	38	55	75	98	124	153	185	220	23
	Worst-Case Number of Steps to Stabilize												

†: Enabled *iff* has token ⋆: Synchronous scheduler

Figure 5.2: Comparison of Token Ring Efficiencies

Figure 5.2 shows how the stabilization times (in number execution steps) changes between the various token ring protocols we have mentioned. In particular, we see that the 6-state protocol converges almost exactly as fast as the 5-state version when every process is considered to have a token. When the 6-state protocol instead uses the original definition of a token (using t_i variables), it converges even faster than the 8-state version. In this case, the 8-state protocol is still arguably better because it can retain many enabled processes (using a synchronous scheduler), which guarantees that that a token can be passed every few steps in a legitimate execution (roughly every $N/(2e)$ synchronous steps where e out of N processes are enabled) [102]. Dijkstra's N-state token ring [64] gives the best performance overall since it converges quickly under any scheduler and passes the token with every step within legitimate states[1].

The above analysis gives a strong argument for future work that focuses on synthesizing protocols where processes can have large domains that are "big enough" with respect to the number of processes. It is a very reasonable assumption that modern programmers make without question (though not without peril), that various integer types can hold values that are theoretically unbounded (e.g., IP addresses, memory locations, or the current time in milliseconds).

Randomized Token Ring

Another practical option to consider when balancing state space, stabilization time, and performance is to use randomization. Herman [111] gives such a token ring using 3 states per process and allows Bot_0 to have random choice.

To design this protocol, start from the shadow protocol p_{tok} with legitimate states $\mathcal{L}_{\text{tok}}$ that were used as a basis for the 5-state token ring. Give each process π_i a ternary

[1] In a practical setting, it might actually be better to minimize token passing during convergence since those actions could have user code that accesses a shared resource.

puppet variable $x_i \in \mathbb{Z}_3$ to read & write, and also let it read x_{i-1}. As before, the first process π_0 is named Bot_0 to distinguish its behavior from each other $\pi_{i>0}$ named P_i. To allow Bot_0 the ability to make random choice, we give it a binary read-only variable $rng_0 \in \mathbb{Z}_2$ that is randomized at every execution step. Protocol 5.5 shows a solution that can be synthesized in roughly 10 seconds using the same parameters as the 5-state token ring. This version is somewhat different from Herman's since Bot_0 is self-disabling, which more closely matches the version given by Alari [9].

Protocol 5.5 — Randomized Token Ring

$$\begin{aligned} Bot_0 :&\ x_{N-1} = x_0 \longrightarrow x_0 := 1 + x_{N-1} + rng_0;\ tok_0 := 0;\ tok_1 := 1; \\ P_i :&\ \ x_{i-1} \neq x_i \longrightarrow x_i := x_{i-1}; \qquad\qquad\quad tok_i := 0;\ tok_{i+1} := 1; \end{aligned}$$

Shadow Variables: $tok_0 \ldots tok_{N-1} \in \mathbb{Z}_2$ (write-only)
Puppet Variables: $x_0 \ldots x_{N-1} \in \mathbb{Z}_3$, $rng_0 \in \mathbb{Z}_2$
Legitimate Shadow States: $\mathcal{L}_{\text{tok}} \equiv \exists! i \in \mathbb{Z}_N : (tok_i = 1)$

5.3 3-State Token Chain

Rather than around a ring, we can also pass a token back-and-forth along a linear (chain) topology. The end processes π_0 and π_{N-1} of the chain topology are distinguished with different names Bot_0 and Top_{N-1}, and all middle processes $\pi_{0<i<N-1}$ are named P_i. As with the ring, let there be N processes, where each process π_i can read & write a binary shadow variable tok_i that denotes whether it has a token. Additionally, processes can read the token variable of its neighbors, meaning that Bot_0 reads tok_1, Top_{N-1} reads tok_{N-2} and each P_i can read both tok_{i-1} and tok_{i+1}. Additionally, we add a single binary shadow variable fwd that denotes the direction the token is moving (1 means up, 0 means down). That is, a process $P_{0<i<N-1}$ passes the token to P_{i+1} when $fwd = 1$ and passes it to P_{i-1} when $fwd = 0$. All processes can read this variable, and the end processes Bot_0 and Top_{N-1} can write it since they of course change the direction of the token. The full shadow protocol p_{tokc} is given by the following actions (where $0 < i < N-1$).

$$\begin{aligned} Bot_0 :&\ tok_0 = 1 && \longrightarrow fwd := 1; && tok_0 := 0; && tok_1 := 1; \\ P_i :&\ tok_i = 1 \wedge fwd = 1 && \longrightarrow && tok_i := 0; && tok_{i+1} := 1; \\ P_i :&\ tok_i = 1 \wedge fwd = 0 && \longrightarrow tok_{i-1} := 1; && tok_i := 0; && \\ Top_{N-1} :&\ tok_{N-1} = 1 && \longrightarrow tok_{N-2} := 1; && tok_{N-1} := 0; && fwd := 0; \end{aligned}$$

For synthesis, give each process a ternary puppet variable x_i. Each $P_{0<i<N-1}$ can also read x_{i-1} and x_{i+1}. Likewise, Bot_0 can read x_1 and Top_{N-1} can read x_{N-2}. One of

the synthesized protocols p_{tokc3} is given in Protocol 5.6, which we conjecture to be generalizable after verification of chains up to size $N = 30$.

Protocol 5.6 — 3-State Token Chain

$$
\begin{array}{rlll}
Bot_0: & x_0{\neq}1 \wedge x_1{=}2 & \longrightarrow x_0{:=}1; \quad fwd{:=}1; & tok_0{:=}0;\ tok_1{:=}1; \\
Bot_0: & x_0{\neq}0 \wedge x_1{\neq}2 & \longrightarrow x_0{:=}0; & \\
P_i: & x_{i-1}{=}1 \ \wedge x_i{\neq}1 & \longrightarrow x_i{:=}1; & tok_i{:=}0;\ tok_{i+1}{:=}1; \\
P_i: & x_{i-1}{=}0 \ \wedge x_i{=}1 \wedge x_{i+1}{=}1 & \longrightarrow x_i{:=}0; & \\
P_i: & x_{i-1}{=}0 \ \wedge x_i{=}0 \wedge x_{i+1}{=}2 & \longrightarrow x_i{:=}2; \quad tok_{i-1}{:=}1; & tok_i{:=}0; \\
Top_{N-1}: & x_{N-2}{=}1 \wedge x_{N-1}{\neq}1 & \longrightarrow x_{N-1}{:=}1; & \\
Top_{N-1}: & x_{N-2}{\neq}1 \wedge x_{N-1}{\neq}2 & \longrightarrow x_{N-1}{:=}2;\ tok_{N-2}{:=}1; & tok_{N-1}{:=}0;\ fwd{:=}0;
\end{array}
$$

Shadow Variables: $tok_0 \ldots tok_{N-1} \in \mathbb{Z}_2$ (write-only), $fwd \in \mathbb{Z}_2$ (write-only)
Puppet Variables: $x_0 \ldots x_{N-1} \in \mathbb{Z}_3$
Legitimate Shadow States: $\mathcal{L}_{\text{tok}} \equiv \exists! i \in \mathbb{Z}_N : (tok_i = 1)$

This protocol does not always pass the token within the invariant, however we found that no such protocol exists. As shown by Dijkstra [64], we can obtain a stabilizing protocol with this behavior by either using 4 states per process or allowing Bot_0 and Top_{N-1} to communicate. Using the same shadow specification and the puppet topologies from [64], we synthesized 4-state token chains and 3-state token rings. Both cases appear to give generalizable protocols (verified up to $N = 15$), regardless of whether we force each action in the invariant to pass a token.

5.4 Daisy Chain Orientation

If anonymous processes have some method of finding each other, it is easy for them to form a ring or chain topology. This ambiguous topology is called a *daisy chain*. When processes do form links, they should agree on a common direction to call "left" or "right".

Bidirectional Ring. Let us focus only on ring topologies in order to introduce the problem. Each process is named P_i and must therefore act the same as other processes. Each P_i has two neighbors P_j and P_k, which are technically P_{i-1} and P_{i+1}, but giving symbolic indices helps intuition.

The direction of P_i is held by two binary variables w_{ij} and w_{ik} that it can read & write. We assume that $w_{ij} \neq w_{ik}$ because it is trivial for P_i to enforce, and it allows us to say that P_i is pointing to P_j *iff* $w_{ij} = 1$, and in the other direction, P_i is pointing to P_k *iff* $w_{ik} = 1$. Each P_i can also read the directions of its neighbors w_{ji} and w_{ki}.

In the overall system, variables w_{ij} and w_{ik} of P_i have indices $w_{2\cdot i}$ and $w_{2\cdot i+1}$. We can phrase the legitimate states as the states where no two processes are pointing at each other or both pointing away from each other.

$$\mathcal{L}_{\text{orient}} \equiv \forall i \in \mathbb{Z}_{N-1} : w_{2\cdot i+1} \neq w_{2\cdot i+2}$$

For stabilization, we give each P_i a binary variable b_i that it can read & write. Furthermore, a process can read its neighbors' b_j and b_k variables.

The processes in our model have an implicit orientation because their readable and writable variables have an ordering (Section 2.1). Rather than change our formalism to somehow hide orientation information, we can instead ensure that processes in the model do not *use* the information. Such information is rendered useless if we mirror all actions. That is, for every action of a process P_i:

$$b_j{=}a_0 \wedge w_{ji}{=}a_1 \wedge w_{ij}{=}a_2 \wedge b_i{=}a_3 \wedge w_{ik}{=}a_4 \wedge w_{ki}{=}a_5 \wedge b_k{=}a_6 \longrightarrow w_{ij}{:=}a_7;\ b_i{:=}a_8;\ w_{ik}{:=}a_9;$$

A mirrored action must be included:

$$b_j{=}a_6 \wedge w_{ji}{=}a_5 \wedge w_{ij}{=}a_4 \wedge b_i{=}a_3 \wedge w_{ik}{=}a_2 \wedge w_{ki}{=}a_1 \wedge b_k{=}a_0 \longrightarrow w_{ij}{:=}a_9;\ b_i{:=}a_8;\ w_{ik}{:=}a_7;$$

In this way, processes must negotiate with each other to agree on a common direction.

Daisy Chain. If we allow the topology to be a chain, let end processes π_0 and π_{N-1} not be connected. These two processes read fewer variables, therefore they must act differently from the others and we call them End_0 and End_{N-1}. Protocol 5.7 is found by performing synthesis for all rings of sizes $N \in \{2, \ldots, 6\}$ and all chains of sizes $N \in \{2, \ldots, 6\}$ simultaneously. We believe it is generalizable after verifying it for both rings and chains of up to $N = 24$.

Protocol 5.7 — Daisy Chain Orientation

$$\begin{array}{rll}
End_i: & \textbf{true} & \longmapsto b_i := 1; \quad w_{ij} := w_{ji} - b_j; \\
P_i: & w_{ji} = 0 \wedge w_{ki} = 0 & \longmapsto b_i := 0; \\
P_i: & w_{ji} = 1 \wedge w_{ki} = 1 \wedge b_j = b_k & \longmapsto b_i := 1 - b_j; \\
P_i: & w_{ji} = 1 \wedge w_{ki} = 1 \wedge b_j \neq b_k & \longmapsto b_i := 1; \quad w_{ij} := b_k; \quad w_{ik} := b_j; \\
P_i: & w_{ji} \neq w_{ki} & \longmapsto b_i := b_j \cdot b_k;\ w_{ij} := w_{ki};\ w_{ik} := w_{ji};
\end{array}$$

Shadow Variables: $w_0 \ldots w_{2\cdot N-1} \in \mathbb{Z}_2$ (read & write)
Puppet Variables: $b_0 \ldots b_{N-1} \in \mathbb{Z}_2$
Legitimate Shadow States: $\mathcal{L}_{\text{orient}} \equiv \forall i \in \mathbb{Z}_{N-1} : (w_{2\cdot i+1} \neq w_{2\cdot i+2})$

For convenience, the actions are written assuming that self-loops do not exist. To make this clear, we use the symbol $\longmapsto$ rather than $\longrightarrow$. The actions can be written

without this notation by conjuncting the result each action's assignment to its guard. For example, the guard of the End_i action could be rewritten as $\textbf{true} \wedge \neg(b_i = 1 \wedge w_{ij} = w_{ji} - b_j)$.

Related

Israeli and Jalfon [117] give general solution for the ring protocol, but none have considered the case of daisy chains. Furthermore, the solution in [117] allows neighboring processes to act synchronously without causing livelocks. However, their solution is not guaranteed to be silent within the legitimate states. Another such protocol is due to Hoepman [112] and only works on odd-sized rings. Hoepman's protocol uses token circulation to determine a common direction around the ring. By only considering rings of odd size, the number of tokens can be forced to be odd. Eventually, tokens of opposing directions will cancel and leave at least one token circulating.

5.5 Specifying Various Topologies

Thus far, we have focused on fairly simple topologies such as rings and chains. This section demonstrates more interesting regular topologies can be specified. These include bands, Möbius ladders, Kautz graphs, trees, and meshes.

The trouble with our current model is that process types have a fixed input alphabet. That is, processes of the same type must read and write the same number of variables with the same domain sizes (in the same order). In Protocon, this is enforced by defining all processes of the same type to read write the same variable names, differing only by their indices. The parameters of a protocol are used to determine the number of processes of each type and, along with the process indices, determine which variables are accessible.

For example, rings are easy to specify based on a process count parameter N. There is N symmetric processes $P_0, \ldots, P_{N-1}$ (or $N-1$ of those if the first is distinguished as Bot_0), and each process π_i accesses variables indexed by $i-1$, i, and $i+1$.

Unoriented daisy chains take an additional binary parameter *Chain* that is 1 *iff* the topology should be a chain instead of a ring. In this case, there are $2 \cdot Chain$ processes named End_i where $i \in \{0, N-1\}$ when they exist, and there are $N-2 \cdot Chain$ processes named P_i where $i \in \{Chain, \ldots, N-1-Chain\}$. The neighboring processes of a P_i are $j = i-1$ and $k = i+1$, and the neighbor of a End_i is computed conditionally as $j = 1$ or $j = N-2$ based on whether $i = 0$ or $i = N-1$.

Band

A band is a simple extension of a ring, where essentially copy of a ring is placed above itself and the nodes above are connected with the nodes below. We can construct a band of $2 \cdot N$ processes in the same way, where each process P_i has a variable x_i.

Two rings are formed by giving each P_i read access to x_{i-2} and x_{i+2}, making rings of even-indexed and odd-indexed processes. Processes are then connected pairwise by giving each P_i read access to $x_{i+1-2\cdot(i \bmod 2)}$, which connects processes P_i and P_j whose index divided by 2 are equal ($\lfloor \frac{i}{2} \rfloor = \lfloor \frac{j}{2} \rfloor$).

Möbius Ladder

A Möbius ladder [107], much like a Möbius strip, adds a single twist to a band topology. This twist can be formed by adding special cases for P_0, P_1, P_{N-2}, and P_{N-1} so that P_0 and $P_{2 \cdot N-1}$ read from each other and P_1 and $P_{2 \cdot N-2}$ read from each other. This is certainly the most efficient approach since processes are numbered close to each other (it helps with BDD variable ordering). A simpler and less efficient way to connect processes is to give each of the $2 \cdot N$ processes P_i read access to x_{i-1}, x_{i+N}, and x_{i+1}.

Kautz Graph

Kautz graphs have many applications in peer-to-peer networks due to their constant degree, optimal diameter, and low congestion [147]. In fact, our parallel search algorithm from Chapter 4 uses a generalized Kautz graph of degree 4 to disseminate conflicting sets of actions. For simplicity, we focus on Kautz graphs of degree 2, where each process P_i reads from 2 processes and its x_i variable is read by 2 other processes.

From [116], a generalized Kautz graph of N vertices with degree d has an arc from vertex j to vertex i *iff* $i = -(2 \cdot j + q + 1) \bmod N$ for some $q \in \mathbb{Z}_d$. This is easy to compute in reverse for $d = 2$, where we let each process P_i read x_j and x_k where $j = -(2 \cdot i + 1) \bmod N$ and $k = -(2 \cdot i + 2) \bmod N$.

To compute Kautz graphs of degree 2 in the proper direction, we must compute which j and k vertices have arcs to each vertex i (for any q values). Let $j = \lfloor \frac{N-1-i}{2} \rfloor$ and let $k = N - 1 - \lfloor \frac{i}{2} \rfloor$. This gives $i = -(2 \cdot j + q + 1) \bmod N$ for $q = (i + 1 + N) \bmod 2$ and gives $i = -(2 \cdot k + q + 1) \bmod N$ for $q = (i + 1) \bmod 2$.

Kautz graphs may contain self-loop arcs, which can be undesirable. We detect such cases where $j = i$ or $k = i$ and instead use $j = k \neq i$. This keeps the number of variables read by each P_i the same, which is important for our model.

Tree

Trees are a common network topology since a lack of cycles in the topology removes many possibilities for livelocks in a protocol. A complete binary tree of $L \geq 2$ levels ($2^L - 1$ total processes) is easy to specify with 1 root process $Root_0$, $2^{L-1} - 2$ inner processes $P_1, \ldots, P_{2^{L-1}-2}$, and 2^{L-1} leaf processes $Leaf_{2^{L-1}-1}, \ldots, Leaf_{2^L-2}$. Each π_i has a variable x_i. To form the connections, each process π_i with a parent ($i \geq 1$) can read x_j where $j = \lfloor \frac{i-1}{2} \rfloor$, and each process π_i with children ($i \leq 2^{L-1} - 2$) can read $x_{2 \cdot i+1}$ and $x_{2 \cdot i+2}$.

A more likely scenario is that we have a tree that is not complete and may not even have a defined root. For this case, we have 3 types of processes End_i, Duo_i, and Tri_i with 1, 2, and 3 neighbors respectively. It is not easy to enumerate instances of trees, therefore we explicitly encode the connections using the system's parameters. Let 3 parameters *NEnds*, *NDuos*, and *NTris* denote the numbers of each process. Furthermore, let 3 parameters *EndIdcs*, *DuoIdcs*, and *TriIdcs* be *arrays* of the indices that processes can read, where the arrays have *NEnds*, $2 \cdot$*NDuos*, and $3 \cdot$*NTris* elements respectively. It is also probably desirable to use unoriented processes (Duo_i and Tri_i) as we did in Section 5.4 so that the order of neighbors does not matter.

Chapter 6:
Adding Convergence is Hard

In this chapter, we investigate the complexity of adding actions to a protocol in order to achieve convergence, and related properties, to a set of legitimate states. We find that for most fairness assumptions, designing convergence or stabilization is NP-complete in the number of system states. This is not the case for global fairness, where convergence and stabilization can be achieved by simply giving all processes all local transitions that do not break closure [75, 88, 100]. However, we find that stabilization to a *subset* of legitimate states is a hard property to add, even under global fairness. The same complexity holds for adding nonmasking fault tolerance (see Figure 6.1). Our hardness proofs are based on reductions from the 3-SAT problem [95] to the various problems that involve adding actions to a protocol in order to achieve convergence. Since stabilization is a special case of nonmasking fault tolerance, it follows that, in general, it is unlikely that adding nonmasking fault tolerance to low atomicity programs can be done efficiently (unless P = NP).

	No Fairness	Weak/Local/Sync Fairness	Global Fairness
Convergence	NP-complete*	NP-complete*	P
Eventually Always	NP-complete*	NP-complete*	NP-complete*
Self-Stabilization	NP-complete*	NP-complete*	P
Nonmasking Fault Tolerance	NP-complete*	NP-complete*	NP-complete*

Figure 6.1: The complexity of adding convergence and various related problems. (* denotes the contributions of this chapter).

Organization. In the following sections, we define the various problems under consideration (Section 6.1), give a simple mapping from 3-SAT (Section 6.1), prove NP-

To obtain stronger results, this chapter is a completely rewritten version of A. P. Klinkhamer and A. Ebnenasir. On the Hardness of Adding Nonmasking Fault Tolerance. *IEEE Transactions on Dependable and Secure Computing*, 2015.

completeness for adding the properties of convergence (Section 6.3), self-stabilization (Section 6.4), and nonmasking fault tolerance (Section 6.5).

6.1 Problem Statement

In this section, we introduce the various problems discussed in this chapter. The first problem is 3-SAT, which is a well-known NP-complete problem that asks whether a given boolean formula ϕ, having a very simple form, can ever evaluate to **true**. Our definition of 3-SAT places a trivial restriction on the problem by forbidding ϕ from having a clause that only references one propositional variable.

Problem 6.1.1 (3-SAT Decision Problem)**.**

- INSTANCE: A set V of n propositional variables $(v_0, \cdots, v_{n-1})$ and m clauses $(C_0, \cdots, C_{m-1})$ over V such that each clause C_i is of the form $(v_q = b_0^i \vee v_r = b_1^i \vee v_s = b_2^i)$ where $b_0^i, b_1^i, b_2^i \in \mathbb{Z}_2$ are binary truth values and $q, r, s \in \mathbb{Z}_n$ are indices such that $\neg(q = r = s)$.
- QUESTION: Is there a satisfying truth-value assignment for the variables in V such that each C_i evaluates to **true**, for all $i \in \mathbb{Z}_m$?

Below, we define decision problems that ask whether the various properties described in Section 2.2 can be achieved by adding actions to a protocol: AddConvergence for convergence to states (Definition 2.2.2), AddEventuallyAlways for stabilization to some subset of states (Definition 2.2.14), AddStabilization for stabilization to an exact set of states (Definition 2.2.3), and AddFaultTolerance for nonmasking fault tolerance (Definition 2.2.15). We also use some related problems but omit their obvious definitions: AddSilentConvergence (Definition 2.2.8), AddSilentStabilization (Definition 2.2.8), and AddShadowConvergence (Definition 2.2.10).

Problem 6.1.2 (AddConvergence Decision Problem)**.**

- CONSTANT: A fairness $\mathcal{F}$.
- INSTANCE: A protocol p with no transitions (a.k.a. a topology) and states $\mathcal{L}$.
- QUESTION: Can actions be added to p to create a protocol p' that converges to $\mathcal{L}$ under fairness $\mathcal{F}$? (Definition 2.2.2)

Problem 6.1.3 (AddEventuallyAlways Decision Problem)**.**

- CONSTANT: A fairness $\mathcal{F}$.
- INSTANCE: A protocol p with no transitions (a.k.a. a topology) and states $\mathcal{L}$.
- QUESTION: Can actions be added to p to create a protocol p' that stabilizes to a subset of $\mathcal{L}$ under fairness $\mathcal{F}$? (Definition 2.2.14)

Problem 6.1.4 (AddStabilization Decision Problem)**.**

- CONSTANT: A fairness $\mathcal{F}$.
- INSTANCE: A protocol p with no transitions (a.k.a. a topology) and states $\mathcal{L}$.

- QUESTION: Can actions be added to p to create a protocol p' that stabilizes to $\mathcal{L}$ under fairness $\mathcal{F}$? (Definition 2.2.3)

Problem 6.1.5 (AddFaultTolerance Decision Problem)**.**
- CONSTANT: A fairness $\mathcal{F}$.
- INSTANCE: A protocol p with no transitions (a.k.a. a topology), states $\mathcal{L}$, and transient fault transitions f_{trans}.
- QUESTION: Can actions be added to p to create a protocol p' that tolerates f_{trans} from $\mathcal{L}$ under fairness $\mathcal{F}$? (Definition 2.2.15)

Lemma 6.1.6. *All closure, convergence, stabilization, and nonmasking fault tolerance properties discussed in Chapter 2 can be verified in polynomial time with respect to the number of global states.*

Proof. These properties can all be verified using standard polynomial-time graph algorithms such as checking whether edges exist (closure and deadlock freedom), cycle detection (livelocks under no fairness), and reachability (convergence under global fairness). Thus, the problems of adding/synthesizing convergence, stabilization, and nonmasking fault tolerance are in NP [88, 137, 140]. □

6.2 Polynomial-Time Mapping

In this section, we present a polynomial-time mapping from an instance of 3-SAT to an instance of the problem of adding convergence (Problem 6.1.2). That is, given an instance of 3-SAT as a boolean formula ϕ, we construct an instance of AddConvergence as a tuple $\langle p_{\text{SAT}}, \mathcal{L}_{\text{SAT}} \rangle$, where $p_{\text{SAT}} \equiv \langle \mathcal{V}_{\text{SAT}}, \Pi_{\text{SAT}}, \mathcal{W}_{\text{SAT}}, \mathcal{R}_{\text{SAT}}, \Delta_{\text{SAT}} \rangle$ is a topology (i.e., Δ_{SAT} contains no transitions) and $\mathcal{L}_{\text{SAT}}$ is a set of legitimate states. This mapping shall be used throughout the rest of this chapter to demonstrate that ϕ is satisfiable *iff* we can add actions to p_{SAT} to create a protocol p'_{SAT} that converges to $\mathcal{L}_{\text{SAT}}$. Lemma 6.2.2 concludes this section with a formal correctness proof for this mapping.

Topology. The instance topology p_{SAT} consists of three processes: π_0, π_1, and π_2. Each process π_i ($i \in \mathbb{Z}_3$) has two variables $x_i \in \mathbb{Z}_n$ and $y_i \in \mathbb{Z}_2$ that it can read, but only y_i can be written. The domain of each x_i matches the number of propositional variables (of the 3-SAT instance ϕ). Thus, the topology has variables $\mathcal{V}_{\text{SAT}} \equiv \{x_0, y_0, x_1, y_1, x_2, y_2\}$, processes $\Pi_{\text{SAT}} \equiv \{\pi_0, \pi_1, \pi_2\}$, write access $\mathsf{W}_i \equiv \{y_i\}$ for each $\mathsf{W}_i \in \mathcal{W}_{\text{SAT}}$, and read access $\mathsf{R}_i \equiv \{x_i, y_i\}$ for each $\mathsf{R}_i \in \mathcal{R}_{\text{SAT}}$.

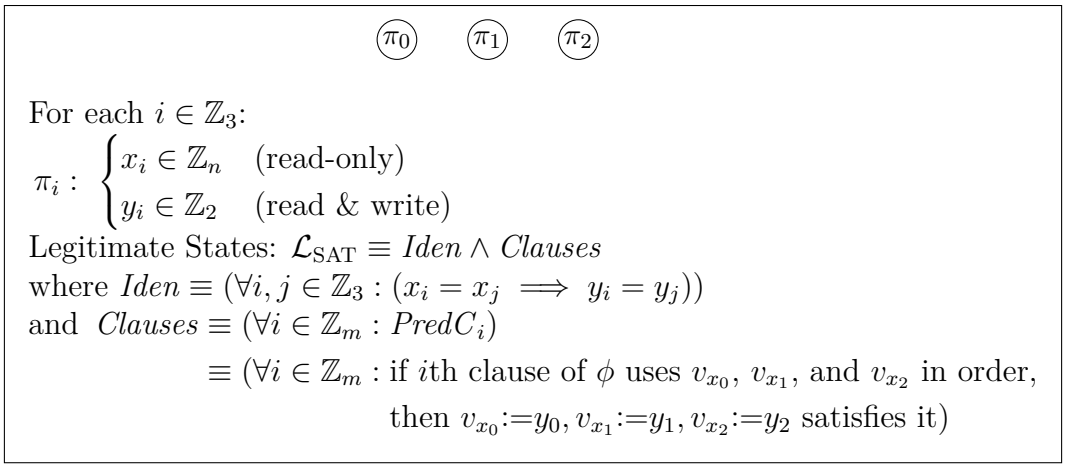


Figure 6.2: Reduction from 3-SAT (formula ϕ with n variables and m clauses) to AddConvergence (topology p_{SAT} and legitimate states $\mathcal{L}_{\text{SAT}}$).

Certificate. For every $r \in \mathbb{Z}_n$, we would like each process π_j of a solution protocol p'_{SAT} to have exactly one action that changes y_j to a binary value a_r when $x_j = r$. Our reduction proofs interpret these $a_0, \ldots, a_{n-1}$ values as a truth-value assignment ($v_r := a_r$ for all $r \in \mathbb{Z}_m$) that satisfies ϕ.

Legitimate States. Inspired by the form of the 3-SAT instance, we define a state predicate $\mathcal{L}_{\text{SAT}} \equiv \mathit{Iden} \wedge \mathit{Clauses}$ that serves as the legitimate states.

- *Iden*: Since the value of y_j when $x_j = r$ should correspond to the binary value we assign to v_r to satisfy ϕ, we must disallow any processes π_i and π_j from choosing different values for y_i and y_j when $x_i = x_j$. Thus, one component of $\mathcal{L}_{\text{SAT}}$ is $\mathit{Iden} \equiv (\forall i, j \in \mathbb{Z}_3 : (x_i = x_j \implies y_i = y_j))$.
- *Clauses*: Corresponding to each clause $C_i = (v_q = b_0^i \vee v_r = b_1^i \vee v_s = b_2^i)$, we construct a state predicate $\mathit{PredC}_i \equiv (x_0 = q \implies y_0 = b_0^i) \vee (x_1 = r \implies y_1 = b_1^i) \vee (x_2 = s \implies y_2 = b_2^i)$ where **true** and **false** values of b_j^i are treated as integers 1 and 0. In other words, we have $\mathit{PredC}_i = (((x_0 = q) \wedge (x_1 = r) \wedge (x_2 = s)) \implies ((y_0 = b_0^i) \vee (y_1 = b_1^i) \vee (y_2 = b_2^i)))$. This way, we construct a state predicate $\mathit{Clauses} \equiv (\forall i \in \mathbb{Z}_m : \mathit{PredC}_i)$. Notice that we check the value of each x_j with respect to the index of the variable appearing in position j in C_i, where $j \in \mathbb{Z}_3$. This is due to the fact that the domain of x_j is equal to the range of the indices of propositional variables (i.e., $\mathbb{Z}_n$).

Polynomial Time. Notice that p_{SAT} has $|S_{\text{SAT}}| = 2(2n)^3$ states, which is polynomial in the size of the 3-SAT instance. Furthermore, we are able to construct $\mathcal{L}_{\text{SAT}}$ in polynomial time, therefore we have a polynomial-time mapping.

Example 6.2.1. *Consider the following instance of* 3-SAT*:*

$$\begin{aligned}\phi = \quad & (\ v_0 \vee \ v_1 \vee \ v_2) \\ & \wedge (\neg v_1 \vee \neg v_1 \vee \neg v_2) \\ & \wedge (\neg v_1 \vee \neg v_1 \vee \ v_2) \\ & \wedge (\ v_1 \vee \neg v_2 \vee \neg v_0)\end{aligned}$$

Since there are three propositional variables and four clauses, we have $n = 3$ and $m = 4$. Moreover, based on the mapping described before, we have $C_0 = (v_0 \vee v_1 \vee v_2)$, $C_1 = (\neg v_1 \vee \neg v_1 \vee \neg v_2)$, $C_2 = (\neg v_1 \vee \neg v_1 \vee v_2)$, and $C_3 = (v_1 \vee \neg v_2 \vee \neg v_0)$. Thus, we have $(b_0^0, b_1^0, b_2^0) = (1, 1, 1)$, $(b_0^1, b_1^1, b_2^1) = (0, 0, 0)$, $(b_0^2, b_1^2, b_2^2) = (0, 0, 1)$, and $(b_0^3, b_1^3, b_2^3) = (1, 0, 0)$. The predicates $PredC_i$ $(i \in \mathbb{Z}_4)$ have the following form:

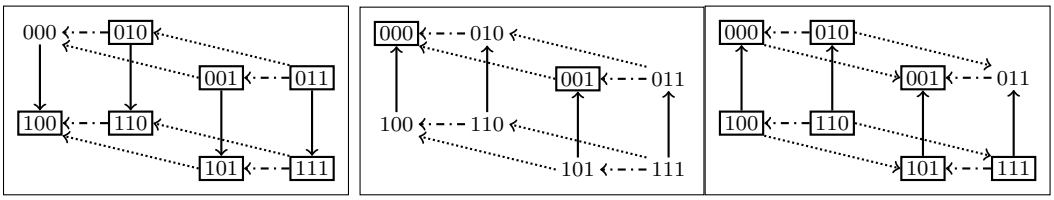
$$\begin{aligned}PredC_0 &= ((x_0 = 0 \wedge x_1 = 1 \wedge x_2 = 2) \implies (y_0 = 1 \vee y_1 = 1 \vee y_2 = 1)) \\ PredC_1 &= ((x_0 = 1 \wedge x_1 = 1 \wedge x_2 = 2) \implies (y_0 = 0 \vee y_1 = 0 \vee y_2 = 0)) \\ PredC_2 &= ((x_0 = 1 \wedge x_1 = 1 \wedge x_2 = 2) \implies (y_0 = 0 \vee y_1 = 0 \vee y_2 = 1)) \\ PredC_3 &= ((x_0 = 1 \wedge x_1 = 2 \wedge x_2 = 0) \implies (y_0 = 1 \vee y_1 = 0 \vee y_2 = 0))\end{aligned}$$

The state predicate *Iden* is as defined before.

We can satisfy ϕ by assigning $v_0 := \textbf{true}$, $v_1 := \textbf{false}$, $v_2 := \textbf{false}$. Using this value assignment, we can construct p'_{SAT} by giving each process π_j the following actions:

$$\begin{aligned}&\pi_j : \ x_j = 0 \wedge y_j = 0 \longrightarrow y_j := 1; \\ &\pi_j : \ x_j = 1 \wedge y_j = 1 \longrightarrow y_j := 0; \\ &\pi_j : \ x_j = 2 \wedge y_j = 1 \longrightarrow y_j := 0;\end{aligned}$$

Figure 6.3 shows the transitions of p'_{SAT} for certain fixed values of x_0, x_1, x_2 chosen to correspond with the clauses of ϕ. Each state is represented by three bits that signify the values of y_0, y_1, and y_2. States within $\mathcal{L}_{\text{SAT}}$ are outlined by rectangles, and transitions of different processes are drawn differently (π_0 has vertical transitions, π_1 has horizontal transitions, and π_2 has diagonal transitions). In particular, consider how Figure 6.3b reflects how clauses C_1 and C_2 force v_1 to be **false**. In the figure, where $x_0 = 1$, $x_1 = 1$, and $x_2 = 2$, the predicates $PredC_1$ and $PredC_2$ force either y_0 or y_1 to equal 0. Furthermore, *Iden* forces $y_0 = y_1$ to hold within legitimate states, therefore $\mathcal{L}$ only holds in the two states where y_0 and y_1 both equal 0.



(a) $x_0 = 0$, $x_1 = 1$, $x_2 = 2$ **(b)** $x_0 = 1$, $x_1 = 1$, $x_2 = 2$ **(c)** $x_0 = 1$, $x_1 = 2$, $x_2 = 0$

Figure 6.3: Changes to y_0, y_1, y_2 values for fixed x_0, x_1, x_2 values.

Lemma 6.2.2 (Mapping)**.** *Given an instance ϕ of* 3-SAT*, let p_{SAT} and $\mathcal{L}_{SAT}$ be defined by the mapping in Section 6.2. Let $a_0, \ldots, a_{n-1}$ be any binary values. Let protocol p'_{SAT} be constructed by giving each process π_j the following action for each $r \in \mathbb{Z}_n$:*

$$x_j = r \wedge y_j \neq a_r \longrightarrow y_j := a_r;$$

All of the following are true for any fairness $\mathcal{F} \in \{\text{unfair}, \text{weak}, \text{local}, \text{global}, \text{sync}\}$:

1. *This p'_{SAT} is silent-stabilizing to $(\forall j \in \mathbb{Z}_3 : \forall r \in \mathbb{Z}_n : (x_j = r \implies y_j = a_r))$ under fairness $\mathcal{F}$*
2. *This p'_{SAT} converges to the Iden component of $\mathcal{L}_{SAT}$ under fairness $\mathcal{F}$*
3. *Formula ϕ is satisfied by the assigning $v_r := a_r$ for each $r \in \mathbb{Z}_n$ iff p'_{SAT} converges to $\mathcal{L}_{SAT}$ under fairness $\mathcal{F}$*

Proof. Each of the three claims is proven with a bold heading. The last claim is an "if and only if", and therefore is broken up into sufficient and necessary cases.

1. Silent Stabilization. We want to show that this p'_{SAT} is silent-stabilizing to $\mathcal{L}'_{\text{SAT}} \equiv (\forall j \in \mathbb{Z}_3 : \forall r \in \mathbb{Z}_n : (x_j = r \implies y_j = a_r))$ under any fairness $\mathcal{F} \in \{\text{unfair}, \text{weak}, \text{local}, \text{global}, \text{sync}\}$. This is obvious because each process π_j of p'_{SAT} is enabled to assign $y_j := a_r$ if $y_j \neq a_r$ for any $x_j = r$. Each process becomes disabled after acting, and it remains disabled forever because it does not read the variables of any other process. Therefore, each process will act at most once under any fairness $\mathcal{F}$, reaching a silent state satisfying $\mathcal{L}'_{\text{SAT}}$:

$$y_0 = a_{x_0} \wedge y_1 = a_{x_1} \wedge y_2 = a_{x_2}$$

2. Guaranteed Convergence to *Iden***.** Since we know that p'_{SAT} silent-stabilizes to some $\mathcal{L}'_{\text{SAT}}$ by assigning each $y_j := a_{x_j}$, we can show that p'_{SAT} converges to *Iden* by replacing each y_j with a_{x_j}. By doing this substitution, $\mathit{Iden} \equiv (\forall i, j \in \mathbb{Z}_3 : (x_i = x_j \implies y_i = y_j))$ is satisfied as $(\forall i, j \in \mathbb{Z}_3 : (x_i = x_j \implies a_{x_i} = a_{x_j}))$ in a silent state. Therefore, p'_{SAT} converges to *Iden*.

3. Satisfiability Implies Convergence. Assuming that $\phi \equiv (\forall i \in \mathbb{Z}_m : C_i)$ is satisfied by the given $a_0, \ldots, a_{n-1}$ values, we want to show that p'_{SAT} converges to

$\mathcal{L}_{\text{SAT}} \equiv \mathit{Iden} \wedge \mathit{Clauses}$. Since p'_{SAT} converges to *Iden*, we only need to prove that it converges to $\mathit{Clauses} \equiv (\forall i \in \mathbb{Z}_m : \mathit{PredC}_i)$. Each such $\mathit{PredC}_i \equiv ((x_0 = q \implies y_0 = b_0^i) \vee (x_1 = r \implies y_1 = b_1^i) \vee (x_2 = s \implies y_2 = b_2^i))$ is defined using the $q, r, s \in \mathbb{Z}_n$ and $b_0^i, b_1^i, b_2^i \in \mathbb{Z}_2$ values from clause $C_i \equiv (v_q = b_0^i \vee v_r = b_1^i \vee v_s = b_2^i)$ of ϕ. Such a PredC_i can only be **false** when $x_0 = q$, $x_1 = r$, and $x_2 = s$, which we know will silent-stabilize to a state with $y_0 = a_q$, $y_1 = a_r$, and $y_2 = a_s$. Thus, p'_{SAT} converges to PredC_i only if our silent state satisfies $(a_q = b_0^i \vee a_r = b_1^i \vee a_s = b_2^i)$. This is equivalent to the a_q, a_r, a_s values satisfying C_i, which established by the actions of each process, therefore p'_{SAT} converges to each PredC_i, and it converges to $\mathcal{L}_{\text{SAT}}$ overall.

3. Convergence Implies Satisfiability. Assuming that p'_{SAT} converges to $\mathcal{L}_{\text{SAT}} \equiv \mathit{Iden} \wedge \mathit{Clauses}$, we want to show that ϕ is satisfied by the given $a_0, \ldots, a_{n-1}$ values. We know that p'_{SAT} converges to each $\mathit{PredC}_i \equiv ((x_0 = q \implies y_0 = b_0^i) \vee (x_1 = r \implies y_1 = b_1^i) \vee (x_2 = s \implies y_2 = b_2^i))$, which is defined using the $q, r, s \in \mathbb{Z}_n$ and $b_0^i, b_1^i, b_2^i \in \mathbb{Z}_2$ values from clause $C_i \equiv (v_q = b_0^i \vee v_r = b_1^i \vee v_s = b_2^i)$ of ϕ. Since p'_{SAT} converges to PredC_i when $x_0 = q$, $x_1 = r$, and $x_2 = s$, we know that the silent state with $y_0 = a_q$, $y_1 = a_r$, and $y_2 = a_s$ satisfies PredC_i. Thus, $(a_q = b_0^i \vee a_r = b_1^i \vee a_s = b_2^i)$ is true, which implies that assigning $v_q := a_q$, $v_r := a_r$, and $v_s := a_s$ will also satisfy clause C_i. This argument holds for every clause C_i, therefore ϕ is satisfied. □

6.3 Adding Convergence

In this section, we prove NP-completeness of various problems relating to protocol design, where the desired protocol converges to some legitimate states.

Lemma 6.3.1 (No Fairness)**.** *Let p_{SAT} and $\mathcal{L}_{SAT}$ be defined by the mapping in Section 6.2 for any* 3-SAT *instance. If actions can be added to p_{SAT} to form a protocol p'_{SAT} that converges to $\mathcal{L}_{SAT}$ under no fairness ($\mathcal{F} =$* unfair*), then p'_{SAT} has the same form as in Lemma 6.2.2.*

Proof. Assuming that a protocol p'_{SAT} converges to $\mathcal{L}_{\text{SAT}}$, we want to show that some $a_0, \ldots, a_{n-1}$ values exist such that each process π_i is defined by having the following action for each $r \in \mathbb{Z}_n$:

$$x_i = r \wedge y_i \neq a_r \longrightarrow y_i{:=}a_r;$$

For every value of $x_i = r \in \mathbb{Z}_n$, there exist only 4 possible actions for π_i:

1. $x_i = r \wedge y_i = 0 \longrightarrow y_i := 0;$
2. $x_i = r \wedge y_i = 1 \longrightarrow y_i := 1;$
3. $x_i = r \wedge y_i = 0 \longrightarrow y_i := 1;$
4. $x_i = r \wedge y_i = 1 \longrightarrow y_i := 0;$

Consider any state where $x_i = x_j = x_k = r$, $y_i = y_j$, and $y_i \neq y_k$ where i, j, k are any indices of different processes ($\{i, j, k\} = \mathbb{Z}_3$). Clearly this state does not satisfy

$\mathcal{L}_{\text{SAT}}$ due to its *Iden* constraint. One such state exists for $y_i = 0$ and also for $y_i = 1$, therefore process π_i cannot have either of the first 2 self-loop actions, otherwise an infinite execution would exist in $\overline{\mathcal{L}_{\text{SAT}}}$. Likewise, π_i cannot have both of the last 2 actions, otherwise it would cause an infinite execution in $\overline{\mathcal{L}_{\text{SAT}}}$ where π_i repeatedly toggles y_i between the value of y_k and the value of y_j. Thus, we can always find some $a_r \in \mathbb{Z}_n$ such that $x_i = x_j = x_k = r$, $y_i = y_j$, $y_i \neq y_k$, and $y_k = a_r$ is a state where π_k is disabled. In order to achieve convergence, π_i (and similarly π_j) must have an action $(x_i = r \wedge y_i \neq a_r \longrightarrow y_i := a_r;)$. We chose $r \in \mathbb{Z}_n$ and $i \in \mathbb{Z}_3$ arbitrarily, therefore some $a_0, \ldots, a_{n-1}$ values exist such that each process π_i of p'_{SAT} has exactly n actions, being $(x_i = r \wedge y_i \neq a_r \longrightarrow y_i := a_r;)$ for each $r \in \mathbb{Z}_n$. □

Lemma 6.3.2 (Global Fairness)**.** *Let p_{SAT} and $\mathcal{L}_{SAT}$ be defined by the mapping in Section 6.2 for any* 3-SAT *instance. If actions can be added to p_{SAT} to form a protocol p'_{SAT} that stabilizes to a subset of $\mathcal{L}_{SAT}$ under global fairness ($\mathcal{F}$ =* global*), then a protocol with the same form as in Lemma 6.2.2 can be constructed.*

Proof. Assuming that a protocol p'_{SAT} converges to a subset $\mathcal{L}'_{\text{SAT}}$ of $\mathcal{L}_{\text{SAT}}$ under global fairness, we will show how to transform p'_{SAT} to have the same form as in Lemma 6.2.2. This transformation performed by removing self-loop actions from each process of p'_{SAT}, which trivially does not affect convergence, therefore let p'_{SAT} actually be this transformed version for the sake of simplicity. We want to show that some $a_0, \ldots, a_{n-1}$ values exist such that each process π_i of the transformed p'_{SAT} is defined by having the following action for each $r \in \mathbb{Z}_n$:

$$x_i = r \wedge y_i \neq a_r \longrightarrow y_i := a_r;$$

To see this, consider the behavior of p'_{SAT} in states where $x_0 = x_1 = x_2 = r$ for any $r \in \mathbb{Z}_n$. By the *Iden* constraint of $\mathcal{L}_{\text{SAT}}$, the system must converge to a state where $y_0 = y_1 = y_2$. Out of the 2 states that satisfy $\mathcal{L}_{\text{SAT}}$, at least 1 of them must satisfy $\mathcal{L}'_{\text{SAT}}$. Let a_r be a binary value such that $y_0 = y_1 = y_2 = a_r$ satisfies $\mathcal{L}'_{\text{SAT}}$. Due to read & write restrictions and our removal of self-loops, there are only 2 possible actions for each process π_i: $(x_i = r \wedge y_i = a_r \longrightarrow y_i := 1 - a_r;)$, and $(x_i = r \wedge y_i \neq a_r \longrightarrow y_i := a_r;)$. The first action assigns some $y_i := 1 - a_r$ from a state where $y_0 = y_1 = y_2 = a_r$, therefore this action breaks closure of $\mathcal{L}'_{\text{SAT}}$ and cannot exist in p'_{SAT}. We are left with only one possible action $(x_i = r \wedge y_i \neq a_r \longrightarrow y_i := a_r;)$, therefore it must be used to converge from a state where $y_i \neq a_r$ and $y_j = a_r$ for some other $j \neq i$ because we just showed that no process π_i is enabled when $y_i = a_r$. It is now clear that the actions of this p'_{SAT} without self-loops matches the form of the protocol used in Lemma 6.2.2. □

Lemma 6.3.3 (Synchronous Scheduler)**.** *Let p_{SAT} and $\mathcal{L}_{SAT}$ be defined by the mapping in Section 6.2 for any* 3-SAT *instance. If actions can be added to p_{SAT} to form a protocol p'_{SAT} that stabilizes to a subset of $\mathcal{L}_{SAT}$ under a synchronous scheduler ($\mathcal{F}$ =* sync*), then we can construct a protocol with the same form as in Lemma 6.2.2 that stabilizes exactly to $\mathcal{L}_{SAT}$ under a synchronous scheduler.*

Proof. Assuming that a protocol p'_{SAT} converges to a subset $\mathcal{L}'_{\text{SAT}}$ of $\mathcal{L}_{\text{SAT}}$ under the synchronous scheduler, we will show how to transform p'_{SAT} to have the same form as in Lemma 6.2.2. This transformation performed by removing self-loop actions from each process of p'_{SAT}, which trivially does not affect convergence, therefore let p'_{SAT} actually be this transformed version for the sake of simplicity. We want to show that some $a_0, \ldots, a_{n-1}$ values exist such that each process π_i of the transformed p'_{SAT} is defined by having the following action for each $r \in \mathbb{Z}_n$:

$$x_i = r \wedge y_i \neq a_r \longrightarrow y_i := a_r;$$

First, we claim that if some process π_i has an action $(x_i = r \wedge y_i \neq a \longrightarrow y_i := a;)$ for any $r \in \mathbb{Z}_n$ and $a \in \mathbb{Z}_2$, then every process π_j has that action $(x_j = r \wedge y_j \neq a \longrightarrow y_j := a;)$. To see this for any r and a, consider π_i having such an action enabled in a state where $x_0 = x_1 = x_2 = r$ and $y_0 = y_1 = y_2 = 1 - a$. Only 2 states satisfy $\mathcal{L}_{\text{SAT}}$ when $x_0 = x_1 = x_2 = r$: (1) a state where $y_0 = y_1 = y_2 = 1 - a$ holds, and (2) a state where $y_0 = y_1 = y_2 = a$ holds. Since at least one of these states must satisfy $\mathcal{L}'_{\text{SAT}}$, every other π_j also needs to be enabled to assign $y_j := a$, otherwise (1) closure would be broken from $y_0 = y_1 = y_2 = 1 - a$ and (2) convergence to $y_0 = y_1 = y_2 = a$ would be impossible.

All processes change their y_i values identically, giving 3 possibilities for every $x_i = r \in \mathbb{Z}_n$ value: (1) No process π_i changes y_i when $x_i = r$. (2) Each process π_i toggles y_i between 0 and 1 when $x_i = r$. (3) Each process π_i changes y_i to equal some a_r when $x_i = r$. From a state $x_0 = x_1 = x_2 = r$ and $y_0 \neq y_1 \wedge y_1 \neq y_2$, which does not satisfy $\mathcal{L}_{\text{SAT}}$, the first possibility causes a deadlock and the second possibility causes a livelock. Therefore it is clear that each π_i has exactly one action $(x_i = r \wedge y_i \neq a_r \longrightarrow y_i := a_r;)$ that changes y_i for each possible $x_i = r \in \mathbb{Z}_n$ value. Since we removed self-loops from p'_{SAT}, it exactly matches the form used in Lemma 6.2.2.

Stabilization to $\mathcal{L}_{\text{SAT}}$. Under a synchronous scheduler, all enabled processes of p'_{SAT} act in one synchronous step to converge to $\mathcal{L}_{\text{SAT}}$. Since p'_{SAT} converges in one synchronous step, it cannot possibly break closure and it therefore stabilizes to $\mathcal{L}_{\text{SAT}}$. □

Theorem 6.3.4 (NP-completeness of AddEventuallyAlways)**.** *The problem of adding stabilization to a subset of legitimate states is NP-complete with respect to the number of global states for any particular choice of fairness $\mathcal{F}$ in* {unfair, weak, local, global, sync}.

Proof. Let f denote the polynomial-time mapping of Section 6.2 that transforms any instance ϕ of 3-SAT to an instance $f(\phi) \equiv \langle p_{\text{SAT}}, \mathcal{L}_{\text{SAT}} \rangle$ of adding stabilization to a subset of $\mathcal{L}_{\text{SAT}}$ under fairness $\mathcal{F}$ (i.e., AddEventuallyAlways for any particular choice of fairness $\mathcal{F} \in \{\text{unfair}, \text{weak}, \text{local}, \text{global}, \text{sync}\}$). We want to show that f is a mapping reduction, meaning that for every instance ϕ of 3-SAT, ϕ is satisfiable

iff a the corresponding instance $f(\phi) \equiv \langle p_{\text{SAT}}, \mathcal{L}_{\text{SAT}} \rangle$ of AddEventuallyAlways has a solution protocol under fairness $\mathcal{F}$. Lemma 6.2.2 proves the *sufficient case*, that a satisfiable ϕ implies the existence of a protocol p'_{SAT} that silent-converges to $\mathcal{L}_{\text{SAT}}$ under any fairness $\mathcal{F} \in \{\text{unfair}, \text{weak}, \text{local}, \text{global}, \text{sync}\}$. In the opposite direction, Lemma 6.2.2 proves the *necessary case* with the help of Lemma 6.3.2 for asynchronous schedulers ($\mathcal{F} \in \{\text{unfair}, \text{weak}, \text{local}, \text{global}\}$) and Lemma 6.3.3 for the synchronous scheduler ($\mathcal{F} = \text{sync}$). That is, Lemma 6.3.2 and Lemma 6.3.3 show that given any protocol p'_{SAT} that stabilizes to a subset of $\mathcal{L}_{\text{SAT}}$ under any fairness $\mathcal{F} \in \{\text{unfair}, \text{weak}, \text{local}, \text{global}, \text{sync}\}$, another protocol exists that matches the form in Lemma 6.2.2, which therefore implies that the corresponding ϕ is satisfiable. We have shown that f is a polynomial-time mapping reduction from 3-SAT to AddEventuallyAlways for any particular choice of fairness $\mathcal{F} \in \{\text{unfair}, \text{weak}, \text{local}, \text{global}, \text{sync}\}$, which proves that adding stabilization to a subset of legitimate states under fairness $\mathcal{F}$ is NP-hard. Lemma 6.1.6 establishes NP membership, therefore the problem is NP-complete. □

Corollary 6.3.5 (NP-completeness of AddSilentConvergence)**.** *The problem of adding silent convergence is NP-complete with respect to the number of global states for any particular choice of fairness $\mathcal{F}$ in* $\{\text{unfair}, \text{weak}, \text{local}, \text{global}, \text{sync}\}$.

Proof. This proof is identical to that of Theorem 6.3.4 because Lemma 6.3.2 and Lemma 6.3.2 establish that a protocol that stabilizes to a subset of $\mathcal{L}_{\text{SAT}}$ under fairness $\mathcal{F}$ can be transformed to achieve silent convergence to $\mathcal{L}_{\text{SAT}}$ under $\mathcal{F}$ by removing self-loops. □

Corollary 6.3.6 (NP-completeness of AddShadowConvergence)**.** *The problem of adding convergence to shadow behavior within legitimate states is NP-complete with respect to the number of global states for any particular choice of fairness $\mathcal{F}$ in* $\{\text{unfair}, \text{weak}, \text{local}, \text{global}, \text{sync}\}$.

Proof. By Definition 2.2.14, stabilization to a subset $\mathcal{L}'$ of legitimate states $\mathcal{L}$ is a special case of convergence to legitimate behavior $\mathcal{L}^2|\mathcal{L}$. Likewise, silent convergence to $\mathcal{L}$ can be written as convergence to $\emptyset^2|\mathcal{L}$. Since AddShadowConvergence is in NP by Lemma 6.1.6, and it is necessarily NP-hard due to its NP-hard special cases AddEventuallyAlways and AddSilentConvergence, the problem is NP-complete. □

Lemma 6.3.7 (Reducing AddEventuallyAlways to AddConvergence)**.** *Let p_{SAT} and $\mathcal{L}_{SAT}$ be defined by the mapping in Section 6.2 for any* 3-SAT *instance. Let $\mathcal{F}$ be any particular choice of fairness in* $\{\text{unfair}, \text{weak}, \text{local}, \text{sync}\}$. *A polynomial-time mapping f exists such that an instance $\langle p_{SAT}, \mathcal{L}_{SAT} \rangle$ of* AddEventuallyAlways *with fairness $\mathcal{F}$ has a solution iff an instance $f(\langle p_{SAT}, \mathcal{L}_{SAT} \rangle)$ of* AddConvergence *with fairness $\mathcal{F}$ has a solution.*

Proof. Lemma 6.3.1 already shows that the mapping of Section 6.2 works when $\mathcal{F} =$ unfair. However, the other fairnesses ($\mathcal{F} \in \{\text{weak}, \text{local}, \text{sync}\}$) require a different mapping.

Consider two protocols p_A and p_B that share no variables and have legitimate state sets $\mathcal{L}_\mathrm{A}$ and $\mathcal{L}_\mathrm{B}$ respectively. If their parallel composition $p_\mathrm{AB} \equiv p_\mathrm{A} \parallel p_\mathrm{B}$ converges to $\mathcal{L}_\mathrm{AB} \equiv \mathcal{L}_\mathrm{A} \wedge \mathcal{L}_\mathrm{B}$ under some fairness $\mathcal{F} \in \{\text{weak}, \text{local}\}$, then either p_A stabilizes to a subset of $\mathcal{L}_\mathrm{A}$ under $\mathcal{F}$ or p_B stabilizes to a subset of $\mathcal{L}_\mathrm{B}$ under $\mathcal{F}$. If this were not true, then some execution σ_A of p_A reaches $\overline{\mathcal{L}_\mathrm{A}}$ infinitely often, and some execution σ_B of p_B reaches $\overline{\mathcal{L}_\mathrm{B}}$ infinitely often, therefore we can find a *fair* execution of p_AB under $\mathcal{F}$ that interleaves σ_A and σ_B such that $\mathcal{L}_\mathrm{A} \wedge \mathcal{L}_\mathrm{B}$ is never satisfied.

The same fact holds for the synchronous scheduler ($\mathcal{F} = \text{sync}$) if all processes have 2 states. If this were not true, then some execution σ_A of p_A reaches $\overline{\mathcal{L}_\mathrm{A}}$ every 2 steps, and some execution σ_B of p_B reaches $\overline{\mathcal{L}_\mathrm{B}}$ every 2 steps. This is because all processes act synchronously and can either toggle between 2 states or stay in 1 single state, making the entire synchronous system either toggle between 2 states or stay in 1 single state. Given this behavior, we can find a synchronous execution of p_AB that leaves at least one of $\mathcal{L}_\mathrm{A}$ and $\mathcal{L}_\mathrm{B}$ at every step by offsetting the two synchronous executions σ_A and σ_B.

Mapping. Let the mapping f transform an instance $\langle p_\mathrm{SAT}, \mathcal{L}_\mathrm{SAT}\rangle$ of AddEventuallyAlways with the given fairness $\mathcal{F} \in \{\text{weak}, \text{local}, \text{sync}\}$ to an instance $f(\langle p_\mathrm{SAT}, \mathcal{L}_\mathrm{SAT}\rangle) \equiv \langle p_\mathrm{AB}, \mathcal{L}_\mathrm{AB}\rangle$ of AddConvergence with fairness $\mathcal{F}$. Let p_AB and $\mathcal{L}_\mathrm{AB}$ be defined as before by their components. Let $p_\mathrm{A} \equiv p_\mathrm{SAT}$ and $\mathcal{L}_\mathrm{A} \equiv \mathcal{L}_\mathrm{SAT}$. Let p_B and $\mathcal{L}_\mathrm{B}$ be copies of p_SAT and $\mathcal{L}_\mathrm{SAT}$ that use renamed variables.

Sufficient Case. Assuming that some p'_SAT gives a solution for AddEventuallyAlways with fairness $\mathcal{F}$, we want to show that some p'_AB gives a solution for AddConvergence with fairness $\mathcal{F}$. Simply define p'_AB as p'_SAT composed with a copy of itself that uses the renamed variables of p'_B. This means p'_A stabilizes to a subset of $\mathcal{L}_\mathrm{A}$ and p'_B stabilizes to a subset of $\mathcal{L}_\mathrm{B}$, therefore p'_AB stabilizes to a subset of $\mathcal{L}_\mathrm{AB}$ because $\mathcal{F} \in \{\text{weak}, \text{local}, \text{sync}\}$ ensures that processes that are enabled will eventually act.

Necessary Case. Assuming that some p'_AB gives a solution for AddConvergence with fairness $\mathcal{F}$, we want to show some solution p'_SAT exists for the associated instance $\langle p_\mathrm{SAT}, \mathcal{L}_\mathrm{SAT}\rangle$ of AddEventuallyAlways with fairness $\mathcal{F}$. Since all processes of p'_AB have 2 states and its component protocols p'_A and p'_B do not share any variables, we have already established that either p'_A stabilizes to a subset of $\mathcal{L}_\mathrm{A}$ under $\mathcal{F}$ or p'_B stabilizes to a subset of $\mathcal{L}_\mathrm{B}$ under $\mathcal{F}$. In polynomial time, we can figure out which of these protocols actually stabilizes, then use it to construct p'_SAT.

In total, we have shown that a polynomial-time mapping reduction exists from AddEventuallyAlways with fairness $\mathcal{F}$ to AddConvergence with fairness $\mathcal{F}$, for any particular choice of $\mathcal{F} \in \{\text{unfair}, \text{weak}, \text{local}, \text{sync}\}$. □

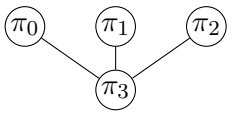
Theorem 6.3.8 (NP-completeness of AddConvergence)**.** *The problem of adding convergence to legitimate states is NP-complete with respect to the number of global states for any particular choice of fairness $\mathcal{F}$ in* $\{\text{unfair}, \text{weak}, \text{local}, \text{sync}\}$.

Proof. By the combined result of Lemma 6.3.7 and reduction used in Theorem 6.3.4, a polynomial-time mapping reduction exists from 3-SAT to AddConvergence for any $\mathcal{F}$ in $\{\text{unfair}, \text{weak}, \text{local}, \text{sync}\}$. This shows that AddConvergence is NP-hard for any such a choice of $\mathcal{F}$, and it is also NP-complete due to Lemma 6.1.6. □

6.4 Adding Self-Stabilization

In this section, we investigate the complexity of stabilization, where a protocol must be closed exactly within the legitimate states. In the previous section, we saw that designing stabilization to a subset of legitimate states (i.e., convergence to legitimate behavior) is hard even under global fairness. However, when the exact legitimate states are known, adding stabilization under global fairness (a.k.a. weak stabilization) is known to have polynomial time complexity [75, 88, 100]. This is because processes can simply use all possible actions that do not break closure.

For hardness results, we make 3 modifications to the mapping of Section 6.2: (1) we add a new binary variable *sat* that all processes can read, (2) we add a new process π_3 that can read all variables and write to *sat*, and (3) the legitimate states require that $sat = 1$. This new mapping is summarized in Figure 6.4.



For each $i \in \mathbb{Z}_3$:

$$\pi_i : \begin{cases} x_i \in \mathbb{Z}_n & \text{(read-only)} \\ y_i \in \mathbb{Z}_2 & \text{(read \& write)} \\ sat & \text{(read from } \pi_3\text{)} \end{cases}$$

$$\pi_3 : \begin{cases} sat \in \mathbb{Z}_2 & \text{(read \& write)} \\ x_i, y_i & \text{(read from } \pi_i\text{)} \end{cases}$$

Legitimate States: $\mathcal{L}_{\text{SAT-SS}} \equiv \mathcal{L}_{\text{SAT}} \wedge (sat = 1)$
where $\mathcal{L}_{\text{SAT}}$ is defined in Figure 6.2

Figure 6.4: Reduction from 3-SAT (formula ϕ with n variables and m clauses) to AddStabilization (topology $p_{\text{SAT-SS}}$ and legitimate states $\mathcal{L}_{\text{SAT-SS}}$).

Lemma 6.4.1 (Reducing 3-SAT to AddStabilization)**.** *Given any instance ϕ of* 3-SAT*, let $p_{SAT\text{-}SS}$ and $\mathcal{L}_{SAT\text{-}SS}$ be defined by the mapping in Figure 6.4. Formula ϕ is satisfiable iff actions can be added to $p_{SAT\text{-}SS}$ to form a protocol $p'_{SAT\text{-}SS}$ that stabilizes to $\mathcal{L}_{SAT\text{-}SS}$ under no fairness ($\mathcal{F} =$ unfair).*

Proof. Let p_{SAT} and $\mathcal{L}_{\text{SAT}}$ be defined by the mapping in Section 6.2. We heavily

leverage Lemma 6.3.1 and Lemma 6.2.2, which together prove that ϕ is satisfiable *iff* actions can be added to p_{SAT} in order to achieve convergence to $\mathcal{L}_{\text{SAT}}$.

Sufficient Case. Assuming that ϕ is satisfiable, we want to show that some protocol $p'_{\text{SAT-SS}}$ stabilizes to $\mathcal{L}_{\text{SAT-SS}}$. By Lemma 6.2.2, some protocol p'_{SAT} exists that silent-stabilizes to a subset $\mathcal{L}'_{\text{SAT}}$ of $\mathcal{L}_{\text{SAT}}$. Give processes π_0, π_1, π_2 of $p'_{\text{SAT-SS}}$ the same actions as in p'_{SAT}, but add the constraint $(sat = 0)$ to their guards. In this way, these processes are only enabled when $sat = 0$, and they provide convergence from $(sat = 0) \wedge \neg\mathcal{L}'_{\text{SAT}}$ to $(sat = 0) \wedge \mathcal{L}'_{\text{SAT}}$ where they become disabled. To recover from these states, we give π_3 an action $(sat = 0 \wedge \mathcal{L}_{\text{SAT}} \longrightarrow sat := 1;)$ to provide convergence from $(sat = 0 \wedge \mathcal{L}'_{\text{SAT}})$ to $(sat = 1 \wedge \mathcal{L}'_{\text{SAT}})$. The only unresolved deadlocks are $(sat = 1 \wedge \neg\mathcal{L}_{\text{SAT}})$, which we resolve by giving π_3 an action $(sat = 1 \wedge \neg\mathcal{L}_{\text{SAT}} \longrightarrow sat := 0;)$. Protocol $p'_{\text{SAT-SS}}$ now silent-stabilizes to $\mathcal{L}_{\text{SAT-SS}} \equiv \mathcal{L}_{\text{SAT}} \wedge (sat = 0)$ because no process is enabled within $\mathcal{L}_{\text{SAT-SS}}$ and we have convergence stairs (Lemma 2.2.6) to $\mathcal{L}_{\text{SAT-SS}}$ from all states:

1. From $(sat = 1 \wedge \neg\mathcal{L}_{\text{SAT}})$ to $(sat = 0 \wedge \neg\mathcal{L}_{\text{SAT}})$ via π_3
2. From $(sat = 0 \wedge \neg\mathcal{L}_{\text{SAT}})$ to $(sat = 0 \wedge \neg\mathcal{L}'_{\text{SAT}})$ since $\overline{\mathcal{L}_{\text{SAT}}} \subseteq \overline{\mathcal{L}'_{\text{SAT}}}$
3. From $(sat = 0 \wedge \neg\mathcal{L}'_{\text{SAT}})$ to $(sat = 0 \wedge \mathcal{L}'_{\text{SAT}})$ via π_0, π_1, π_2
4. From $(sat = 0 \wedge \mathcal{L}'_{\text{SAT}})$ to $(sat = 1 \wedge \mathcal{L}'_{\text{SAT}})$ via π_3
5. From $(sat = 1 \wedge \mathcal{L}'_{\text{SAT}})$ to $(sat = 1 \wedge \mathcal{L}_{\text{SAT}})$ since $\mathcal{L}'_{\text{SAT}} \subseteq \mathcal{L}_{\text{SAT}}$

Necessary Case. Assuming that some $p'_{\text{SAT-SS}}$ stabilizes to $\mathcal{L}_{\text{SAT-SS}}$, we want to show that ϕ is satisfiable. First, we claim that in such a $p'_{\text{SAT-SS}}$, only π_3 can change a value when $sat = 1$. To see this, consider any state where $(x_0 = x_1 = x_2) \wedge (y_0 = y_1 = y_2) \wedge (sat = 1)$. This state obviously satisfies $\mathcal{L}_{\text{SAT}}$, but if any π_0, π_1, π_2 changes a y_0, y_1, y_2 value, then obviously closure is broken! Such a process π_i can only read x_i, y_i, and sat, therefore it cannot act when $sat = 1$ without also including an action that breaks closure. Thus, since π_3 can only modify sat, we know that the actions of π_0, π_1, π_2 provide convergence from $(sat = 0 \wedge \neg\mathcal{L}_{\text{SAT}})$ to $(sat = 0 \wedge \mathcal{L}_{\text{SAT}})$. From these actions, we can construct a protocol p'_{SAT} that converges to $\mathcal{L}_{\text{SAT}}$. By Lemma 6.3.1 and Lemma 6.2.2, ϕ is satisfiable. □

Theorem 6.4.2 (NP-completeness of AddStabilization)**.** *Adding self-stabilization is NP-complete for any particular choice of fairness $\mathcal{F}$ in* $\{\text{unfair}, \text{weak}, \text{local}, \text{sync}\}$.

Proof. For the unfair scheduler $\mathcal{F} = \text{unfair}$, Lemma 6.4.1 establishes a polynomial-time mapping reduction from 3-SAT to AddStabilization. For the synchronous scheduler $\mathcal{F} = \text{sync}$, Lemma 6.3.3 establishes another such reduction using our earlier mapping from Section 6.2. Lemma 6.1.6 establishes NP membership, therefore clearly AddStabilization is NP-complete for any particular choice of fairness $\mathcal{F}$ in {unfair, sync}.

A reduction similar to the proof of Lemma 6.4.1 exists for the remaining fairnesses $\mathcal{F} \in \{\text{weak}, \text{local}\}$ (and also $\mathcal{F} = \text{sync}$), but the mapping from 3-SAT to AddConvergence

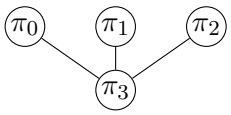
for such a fairness $\mathcal{F}$ involves the composite system constructed in the proof of Lemma 6.3.7. The idea is straightforward, where a process similar to π_3 of Figure 6.4 can read all variables and write to a binary variable *sat* that is readable by all processes. An in Lemma 6.4.1, we would find that the other processes of the composite system (there are 6) must provide convergence to their conjuncted legitimate state predicate when $sat = 0$, which would then prove that a converging protocol without *sat* exists, and would finally that ϕ is satisfiable by the mapping in Lemma 6.3.7. □

Corollary 6.4.3 (NP-completeness of AddSilentStabilization)**.** *Adding silent stabilization is NP-complete for any particular choice of fairness $\mathcal{F}$ in* $\{\text{unfair}, \text{weak}, \text{local}, \text{sync}\}$.

Proof. As shown in the sufficient case of Lemma 6.4.1, any satisfiable ϕ implies a silent-stabilizing $p'_{\text{SAT-SS}}$ to $\mathcal{L}_{\text{SAT-SS}}$. In the reduction using a composite protocol that is discussed in Theorem 6.4.2 proof (which applies to any $\mathcal{F} \in \{\text{weak}, \text{local}, \text{sync}\}$), we would also be able to obtain silent convergence for any satisfiable ϕ. The necessary cases of these reductions does not change, since a silent-stabilizing protocol is also stabilizing. □

6.5 Adding Nonmasking Fault Tolerance

In this section, we present a somewhat surprising result that adding nonmasking fault tolerance remains NP-hard under global fairness. This is surprising because adding stabilization under global fairness (a.k.a.*weak stabilization* [100]) is known to have polynomial time complexity in the number of global states [75, 88, 100]. For this proof, we reuse the reduction from the previous section since stabilization is a special case of nonmasking fault tolerance. However, though the clever use of faults, we are able to ensure that the processes used in the reduction cannot simply toggle their variables as they would otherwise be able to do under global fairness. Figure 6.5 shows our new reduction.



$\pi_0 \ldots \pi_3$: Unchanged from Figure 6.4.

New Variables : $\begin{cases} failed \in \mathbb{Z}_2 & \text{(unreadable by processes)} \\ z_0, z_1, z_2 \in \mathbb{Z}_3 & \text{(unreadable by processes)} \end{cases}$

Legitimate States: $\mathcal{L}_{\text{SAT-FT}} \equiv \mathcal{L}_{\text{SAT-SS}} \wedge (failed = 0)$
where $\mathcal{L}_{\text{SAT-SS}}$ is defined in Figure 6.4
Faults: $f_{\text{trans}} \equiv f_0 \cup f_1 \cup f_2$
where each f_i has transitions:

f_i : **true** $\longrightarrow x_i := x_i + 1;$ $\quad z_0 := 0;\ z_1 := 0;\ z_2 := 0;$

f_i : **true** $\longrightarrow y_i := y_i + 1;$ $\quad z_0 := 0;\ z_1 := 0;\ z_2 := 0;$

f_i : **true** $\longrightarrow sat := sat + 1;$ $\quad z_0 := 0;\ z_1 := 0;\ z_2 := 0;$

f_i : $sat = 0 \wedge z_i = 0 \wedge y_i = 0 \longrightarrow z_i := 1;$

f_i : $sat = 0 \wedge z_i = 1 \wedge y_i = 1 \longrightarrow z_i := 2;$

f_i : $sat = 0 \wedge z_i = 2 \wedge y_i = 0 \longrightarrow failed := 1;$

Figure 6.5: Reduction from 3-SAT (formula ϕ with n variables and m clauses) to AddFaultTolerance (topology $p_{\text{SAT-FT}}$, legitimate states $\mathcal{L}_{\text{SAT-FT}}$, and faults f_{trans}).

Lemma 6.5.1. *Let $p'_{SAT\text{-}FT}$ be a protocol whose topology $p_{SAT\text{-}FT}$, legitimate states $\mathcal{L}_{SAT\text{-}FT}$, and faults f_{trans} are defined by the mapping of Figure 6.5 for any* 3-SAT *instance. Any $p'_{SAT\text{-}FT}$ that is closed within $\mathcal{L}_{SAT\text{-}FT}$ has a state satisfying $failed = 1$ in its fault span iff some process π_i ($i \in \mathbb{Z}_3$) has two actions $(sat = 0 \wedge x_i = r \wedge y_i = 0 \longrightarrow y_i := 1;)$ and $(sat = 0 \wedge x_i = r \wedge y_i = 1 \longrightarrow y_i := 0;)$ for some $r \in \mathbb{Z}_n$ that can repeatedly toggle y_i.*

Proof. That system defined by Figure 6.5 is very similar to Figure 6.4, but it adds faults, variables $z_0, z_1, z_2, failed$ that no process can read or write, and adds the constraint $failed = 0$ to its legitimate states.

Sufficient Case. Assuming $p'_{\text{SAT-FT}}$ has two actions $(sat = 0 \wedge x_i = r \wedge y_i = 0 \longrightarrow y_i := 1;)$ and $(sat = 0 \wedge x_i = r \wedge y_i = 1 \longrightarrow y_i := 0;)$ for some $r \in \mathbb{Z}_n$ and some $i \in \mathbb{Z}_3$, we want to show that $failed = 1$ is in the fault span. Let the first 3 fault actions in Figure 6.4 occur to bring the system to a state where $x_i = r \wedge y_i = 0 \wedge sat = 0 \wedge z_i = 0$. At this point, the fourth action of fault f_i is enabled, and process π_i is enabled to assign $y_i := 1$. Assume the fault occurs because it can, bringing the system to have $z_i = 1$. Now let π_i act to assign $y_i := 1$. At this point, the fifth action of fault f_i is enabled, and process π_i is enabled to assign $y_i := 0$. Assume the fault occurs because it can, bringing the system to have $z_i = 2$. Now let π_i act to assign $y_i := 0$. At this

point, the sixth action of fault f_i is enabled, and process π_i is enabled to assign $y_i := 1$. Let the fault occur, bringing the system to have $\mathit{failed} = 1$, therefore it is in the fault span. This is depicted in Figure 6.6 if we find a path from $\mathcal{L}_{\text{SAT-FT}}$ to $\mathit{failed} := 1$.

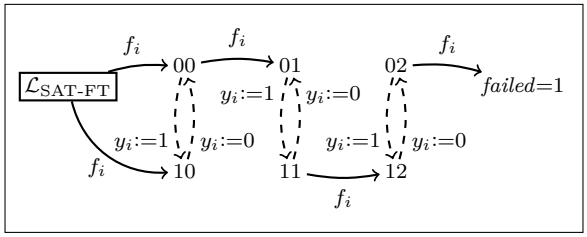


Figure 6.6: Fault class f_i prevents process π_i from toggling y_i.

Necessary Case. Assuming that a state where $\mathit{failed} = 1$ exists in the fault span of the given $p'_{\text{SAT-FT}}$, we want to show that $p'_{\text{SAT-FT}}$ necessarily has two actions $(\mathit{sat} = 0 \wedge x_i = r \wedge y_i = 0 \longrightarrow y_i := 1;)$ and $(\mathit{sat} = 0 \wedge x_i = r \wedge y_i = 1 \longrightarrow y_i := 0;)$ for some $r \in \mathbb{Z}_n$ and some $i \in \mathbb{Z}_3$. First notice that $\mathcal{L}_{\text{SAT-FT}} \equiv \mathcal{L}_{\text{SAT-SS}} \wedge (\mathit{failed} = 0)$, therefore we know that π_0, π_1, π_2 cannot act when $\mathit{sat} = 1$ without breaking closure of $\mathcal{L}_{\text{SAT-SS}}$ due to our reasoning in the necessary case proof of Lemma 6.4.1. Since the first 3 fault actions reset z_0, z_1, z_2 to 0, only a process π_i can change a y_i without resetting all z_0, z_1, z_2 values. Furthermore, *failed* can only be changed to 1 when some $z_i = 2$ and $y_i = 0$. Figure 6.6 shows this, and it is clear that an earlier state must have had $z_i = 1$ and $y_i = 1$, and still earlier we had $z_i = 0$ and $y_i = 0$. Since we have assumed that some state in the fault span satisfies $\mathit{failed} = 1$, it is obvious that some π_i has toggled its y_i value from 0 to 1 and back to 0 in order for the faults to increment z_i from 0 to 2 and finally assigning $\mathit{failed} := 1$. Since π_i can only change its y_i when $\mathit{sat} = 0$ in order to preserve closure, the only way that any π_i can toggle its y_i value is by using the two actions $(\mathit{sat} = 0 \wedge x_i = r \wedge y_i = 0 \longrightarrow y_i := 1;)$ and $(\mathit{sat} = 0 \wedge x_i = r \wedge y_i = 1 \longrightarrow y_i := 0;)$ for some $r \in \mathbb{Z}_n$. □

Lemma 6.5.2. *Given any instance ϕ of* 3-SAT*, let $p_{\mathit{SAT\text{-}FT}}$ and $\mathcal{L}_{\mathit{SAT\text{-}FT}}$ be defined by the mapping in Figure 6.5. Formula ϕ is satisfiable iff actions can be added to $p_{\mathit{SAT\text{-}FT}}$ to form a protocol $p'_{\mathit{SAT\text{-}FT}}$ that stabilizes to $\mathcal{L}_{\mathit{SAT\text{-}FT}}$ from its fault span under global fairness ($\mathcal{F} =$* global*).*

Proof. We heavily leverage the mappings between ϕ and adding stabilization (sufficient case) and convergence (necessary case).

Sufficient Case. Assuming that ϕ is satisfiable, we want to show that some protocol $p'_{\text{SAT-FT}}$ stabilizes to $\mathcal{L}_{\text{SAT-FT}}$. Let $p'_{\text{SAT-FT}}$ be the protocol $p'_{\text{SAT-SS}}$ from the sufficient case in the proof of Lemma 6.4.1 that stabilizes to $\mathcal{L}_{\text{SAT-SS}}$. Since and $p'_{\text{SAT-SS}}$ stabilizes to $\mathcal{L}_{\text{SAT-SS}}$, we only need to prove that no state in the fault span exists where $\mathit{failed} = 1$ holds. Each process π_i in the constructed protocol $p'_{\text{SAT-FT}}$ only assigns its y_i to some

fixed value given x_i, therefore by Lemma 6.5.1, no state in the fault span satisfies $\mathit{failed} = 1$.

Necessary Case. Assuming that some fault-tolerant $p'_{\text{SAT-FT}}$ exists under global fairness, we want to show that ϕ is satisfiable. Let a protocol p'_{SAT} be constructed as the actions of processes π_0, π_1, π_2 in the fault-tolerant $p'_{\text{SAT-FT}}$ (discarding references to *sat* in the actions' guards). By Lemma 6.5.1, we know that these processes do not toggle their y_i variables repeatedly, and since the fault span of $p'_{\text{SAT-FT}}$ includes all valuations of $x_0, x_1, x_2, y_0, y_1, y_2$, it is clear that $p'_{\text{SAT-FT}}$ stabilizes to a subset of states where $\mathcal{L}_{\text{SAT}}$ holds. The first 3 processes are the only ones that write y_i values, therefore p'_{SAT} also stabilizes to a subset of $\mathcal{L}_{\text{SAT}}$ under global fairness. By Lemma 6.3.2, this implies that the corresponding ϕ is satisfiable. □

Theorem 6.5.3 (NP-completeness of AddFaultTolerance)**.** *The problem of adding nonmasking fault tolerance is NP-complete with respect to the number of global states for any particular choice of fairness $\mathcal{F}$ in* $\{\text{unfair}, \text{weak}, \text{local}, \text{global}, \text{sync}\}$.

Proof. Since stabilization is a special case of nonmasking fault tolerance, Theorem 6.4.2 already proves this theorem for all $\mathcal{F} \in \{\text{unfair}, \text{weak}, \text{local}, \text{sync}\}$, therefore we need only prove it for global fairness. A polynomial-time mapping reduction between 3-SAT and AddFaultTolerance (with $\mathcal{F} = \text{global}$) has been shown in Lemma 6.5.2, and the problem is in NP due to Lemma 6.1.6, therefore AddFaultTolerance is NP-complete overall. □

In our previous work showing this result [132], we allowed a fault-tolerant protocol to stabilize to a subset of legitimate states but did not exploit this fact. In this work, we explicitly do not allow the legitimate states to "shrink" for the problem of AddFaultTolerance because otherwise, it would just be a special case of AddEventuallyAlways, which was already proven to be NP-complete for all fairness models under consideration in Theorem 6.3.4.

Corollary 6.5.4. *The problem of adding masking fault tolerance is NP-complete with respect to the number of global states for any particular choice of fairness $\mathcal{F}$ in* $\{\text{unfair}, \text{weak}, \text{local}, \text{global}, \text{sync}\}$.

Proof. Adding nonmasking fault tolerance is a special case of adding masking fault tolerance where there are no transitions forbidden by safety properties. This result matches with Kulkarni and Arora's results in [137]. □

Chapter 7:
Verifying Convergence is Undecidable on Parameterized Unidirectional Rings

Verifying stabilization is known to be a difficult task [100] due to the need to analyze execution from all states of a system. To further complicate analysis, we often want a protocol to stabilize for any instance of a general topology formed by connecting arbitrarily many copies of processes (with the same transition functions). As introduced in Section 2.1, these protocols are *parameterized* by the topology, commonly by the number of processes in the system. There are numerous methods [45,82,89,93] for 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). Apt and Kozen [11] illustrate that, in general, verifying linear temporal logic [77] properties for parameterized systems is Π_1^0-complete. Suzuki [169] shows that the verification problem remains Π_1^0-complete even for unidirectional rings where all processes have a similar code that is parameterized in the number of nodes.

Contributions. In this chapter, we extend this result for the special case where the property of interest is livelock freedom, and every system state is under consideration. Or equivalently, silent stabilization of a protocol to any set of states. Specifically, we illustrate that, even when processes are symmetric, deterministic, self-disabling, and have a finite state space, livelock detection is undecidable (Σ_1^0-complete) on unidirectional ring topologies. The proof of undecidability in our work is based on a reduction from the periodic domino problem [106]. Further, we conclude that verifying silent convergence on this simple parameterized topology is Π_1^0-complete.

Assumptions about Processes. To reiterate, we make some assumptions within this chapter in order to strengthen the undecidability results. (1) Processes are finite state, which is the usual assumption throughout this work. (2) Processes are deterministic. (3) Processes are self-disabling. (4) Processes are symmetric. Since processes are symmetric, refer to each process π_i by a name P_i. We omit the process

Sections 7.1–7.3 contain material from A. P. Klinkhamer and A. Ebnenasir. Verifying Livelock Freedom on Parameterized Rings and Chains. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, 2013.

index when referring to its transition function δ and input alphabet Σ. Each process P_i is assumed to read the entire state space of its predecessor P_{i-1} in the ring, therefore its state space $\Gamma \equiv \Sigma$ is equivalent to its input alphabet.

Organization. Section 7.1 presents some basic concepts. Section 7.2 provides a formal characterization of livelocks in unidirectional rings. Then, Section 7.3 presents a well-known undecidable problem that we will use to show the undecidability of detecting livelocks in unidirectional ring protocols. Section 7.4 gives far-reaching undecidability results for livelock detection and verifying stabilization. Section 7.5 presents a surprising result that stabilizing unidirectional ring protocols can be synthesized to work for all ring sizes, but synthesis becomes undecidable for bidirectional rings.

7.1 Concepts

This section presents the definition of unidirectional ring protocols and their propagation graphs.

Definition 7.1.1 (Propagation Function)**.** Let P_i be any process in a unidirectional ring protocol p which owns one variable x_i. We define its propagation function[1] $\delta^\top : \Sigma \times \Sigma \to \Sigma$ as a partial function such that $\delta^\top(a, b) = c$ if and only if P_i has an action $(x_{i-1} = a \wedge x_i = b \longrightarrow x_i := c;)$. In other words, $\delta^\top$ can be used to define all actions of P_i in the form of a single parametric action:

$$((x_{i-1}, x_i) \in \textsc{Pre}(\delta^\top)) \longrightarrow x_i := \delta^\top(x_{i-1}, x_i);$$

where $(x_{i-1}, x_i) \in \textsc{Pre}(\delta^\top)$ checks to see if the current x_{i-1} and x_i values are in the preimage of $\delta^\top$.

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 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 [87]. Each process P_i has a variable $x_i \in \mathbb{Z}_3$ and actions $(x_{i-1} = 0 \wedge x_i = 2 \longrightarrow x_i := 1)$, $(x_{i-1} = 1 \wedge x_i = 1 \longrightarrow x_i := 2)$, and $(x_{i-1} = 2 \wedge x_i = 0 \longrightarrow x_i := 1)$. This protocol converges to a state where the sum of each two consecutive x values does not equal 2 (i.e., the state predicate $\forall i : (x_{i-1} + x_i \neq 2)$). We represent this protocol with a graph containing arcs $(0, 2, 1)$, $(1, 1, 2)$, and $(2, 0, 1)$ as shown in Figure 7.1.

[1] In this chapter, we use a propagation function $\delta^\top$ which is based off of our definition of the transition function δ in Chapter 2. Rather, the two function arguments are swapped, giving $\delta^\top(a, b) = \delta(b, a)$, and our use of $\delta^\top$ is not applicable to general topologies. When we graph $\delta^\top$ as in Figure 7.1, it does not represent the FSM of a process, which is the case for δ as in Figure 2.1.

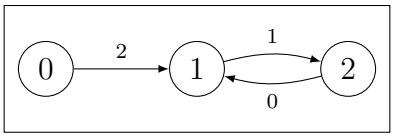


Figure 7.1: Propagation graph of the sum-not-2 protocol.

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

7.2 Livelock Characterization

This section presents a formal characterization of livelocks in parameterized rings, which is an extension of [87, 89]. This characterization is based on a notion of sequences of actions that are propagated in a ring, called *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.

Propagations. 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. This behavior is called a *propagation* and is represented by a sequence of parameterized actions. 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 its variable x_i to c 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 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 corresponds to a closed walk of length n in the graph. The sum-not-2 protocol has such a propagation of period 2: $\langle (1, 1, 2), (2, 0, 1) \rangle$.

"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 $e = c$. Similarly, a propagation leads another if and only if, for every index j, its jth action leads the jth action of the other propagation. Therefore 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 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.

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 (and the last propagation leads the first). This case can be represented succinctly.

Lemma 7.2.1 (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 the last leads the first). Such propagations exists iff we can write each jth action of each ith propagation as $(a^i_{j-1}, a^{i-1}_j, a^i_j)$ using some a^i_j values and computing superscript and subscript indices modulo m and n respectively.*

Proof. The proof is by combining the notations used in this section when defining periodic propagations and the leads relation. Using $\mathtt{X}$ as a wildcard value (i.e., any value, do not assume $\mathtt{X} = \mathtt{X}$), 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. □

Example 7.2.2. *Livelock freedom of the sum-not-2 protocol.*

Recall from Figure 7.1 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.

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 this protocol. As such, no livelock exists.

Lemma 7.2.3. *Assume a ring protocol where processes are symmetric. An execution from some state C to itself exists (i.e., a livelock exists) if and only if an execution exists from C to some C' obtained by rotating the values of C by some k positions.*

Proof. Due to a finite number of states, an infinite execution must visit some state C (i.e., C' for $k = 0$) infinitely often. Conversely, any execution from C to C can simply repeat to make an infinite execution. 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 total rotations to return the system to state C. Emerson and Namjoshi [81] similarly use this notion of rotational symmetry to reason about rings of symmetric processes. □

Lemma 7.2.4. *Assume a unidirectional ring protocol of symmetric, self-disabling processes. The protocol has a livelock for some ring size if and only if there exist some m propagations with some period n, where the $(i-1)$th propagation leads the ith propagation for each index i 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.

Sufficient Case. Assume a livelock exists on a ring of size N. By Lemma 7.2.3, 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 since, when processes in a unidirectional ring are self-disabling, the number of enabled processes will not increase or decrease over time [89] (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 nm total transitions, they can repeat with period n.

Necessary Case. 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 7.2.1, 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

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 7.2: A livelock on mn processes using m propagations of period n that lead each other in order.

an execution from some state to itself to form a livelock. Construct a ring of mn processes where the initial state is defined with $x_{((m-1-i)n+j)} = a_j^i$ 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 7.2. Conceptually, we use the first propagation to initialize the last n processes $(x_{((m-1)n)}, \dots, x_{(mn-1)}) = (a_0^0, \dots, a_{n-1}^0)$, then use the second propagation to initialize the preceding n processes $(x_{((m-2)n)}, \dots, x_{((m-1)n-1)}) = (a_0^1, \dots, a_{n-1}^1)$, and repeat this pattern until the last propagation is used to initialize the first n processes $(x_0, \dots, x_{n-1}) = (a_0^{m-1}, \dots, a_{n-1}^{m-1})$.

In the initial state, every process whose index is a multiple of n is enabled. Each such process $P_{((m-i)n)}$ has $x_{((m-i)n-1)} = a_{n-1}^i$ and $x_{((m-i)n)} = a_0^{i-1}$, and it is enabled to assign $x_{((m-i)n)} := a_0^i$ since actions have the form $(a_{j-1}^i, a_j^{i-1}, a_j^i)$. These positions are highlighted in the first row of Figure 7.2, and the second row shows the result of all enabled processes acting. Let these m processes act in any order. 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 $x_{((m-i)n+1)} = a_1^{i-1}$ is true due to our choice of initial state. Since the previous actions have assigned $x_{((m-i)n)} := a_0^i$, each $P_{((m-i)n+1)}$ is enabled to assign $x_{((m-i)n+1)} := a_1^i$. 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 7.2, 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 $x_{((m-i)n+(n-1))} = a_{n-1}^{i-1}$ is true due to our choice of initial state. Since the previous actions have assigned $x_{((m-i)n+(n-2))} := a_{n-2}^i$, each $P_{((m-i)n+(n-1))}$ is enabled to assign $x_{((m-i)n+(n-1))} := a_{n-1}^i$. After mn total steps, this execution has reached a state where each $x_{((m-i)n+j)} = a_j^i$. This state is a copy of the initial state (where $x_{((m-1-i)n+j)} = a_j^i$) that is rotated by m positions. Thus, the execution can revisit the initial state by Lemma 7.2.3, and a livelock exists. □

7.3 Tiling

With our new characterization of livelocks in a unidirectional ring protocol from Lemma 7.2.4, we can explore the difficulty of livelock detection. We use the protocol graph as an intuitive bridge between problems. To complete the bridge, we introduce a well-studied undecidable problem, the domino problem, and reduce a variant of this problem to the problem of livelock detection. The section concludes with the paper's main theorem of undecidability.

7.3.1 Variants of the Domino Problem

Problem 7.3.1 (The Domino Problem)**.**

- INSTANCE: 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 an 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 [176], and the square tiles are commonly referred to as Wang tiles. Berger showed the problem to be undecidable [28]. Specifically, the problem is co-semi-decidable, also written as Π_1^0-complete using the arithmetical hierarchy notation of Rogers [161].

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 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 [124].

Problem 7.3.2 (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 7.3.1 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 7.3.2 is equivalent to asking whether a torus can be completely covered using the same tiling rules. Gurevich and Koriakov [106] 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 [161].

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 *iff* 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. Due to the following lemma (Lemma 7.3.3), 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.

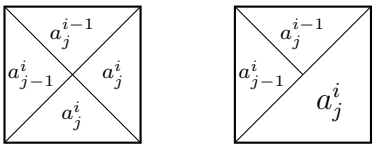
7.3.2 Equivalence to Livelock Detection

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 7.3.3. *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 iff 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 is 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 7.2.1, 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 (with indices computed modulo m and n). Southeast



(a) Wang Tile **(b)** Simplified

Figure 7.3: Tile for action $(a_{j-1}^i, a_j^{i-1}, a_j^i)$.

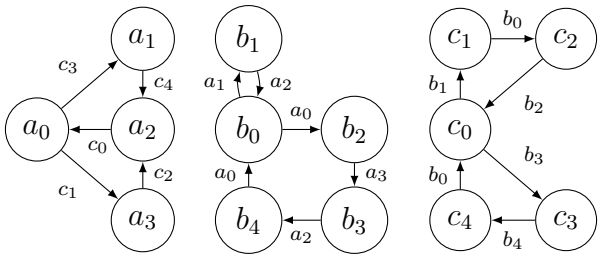


Figure 7.4: "A-b-C" protocol graph.

tile colors in a periodic tiling T behave the same as a_j^i 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_j^i is equivalent to solving the periodic domino problem, where each action $(a_{j-1}^i, a_j^{i-1}, a_j^i)$ must exist as a tile in the action tile set as shown in Figure 7.3. □

Example 7.3.4. *Fictional "A-b-C" protocol with a livelock.*

Figure 7.4 shows the 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 does not function in any meaningful way, but it does provide an interesting livelock. First, propagations that characterize the livelock do not correspond to simple cycles in the protocol's graph. Secondly, 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:
$\langle (b_4, a_0, b_0), (b_0, a_1, b_1), (b_1, a_2, b_0), (b_0, a_0, b_2), (b_2, a_3, b_3), (b_3, a_2, b_4) \rangle$

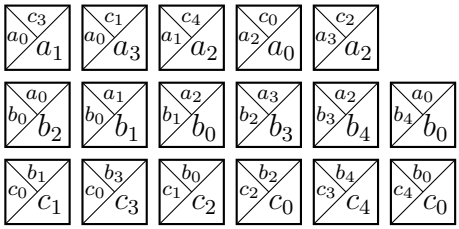
3. Using the previous nodes as arc labels b_0, b_1, b_0, b_2, b_3, and b_4, start from node c_4 to form a closed walk visiting nodes c_0, c_1, c_2, c_0, c_3, and c_4. This corresponds to the periodic propagation:
$\langle (c_4, b_0, c_0), (c_0, b_1, c_1), (c_1, b_0, c_2), (c_2, b_2, c_0), (c_0, b_3, c_3), (c_3, b_4, c_4) \rangle$
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 7.5b, which is a periodic block formed by action tiles in Figure 7.5a. 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.

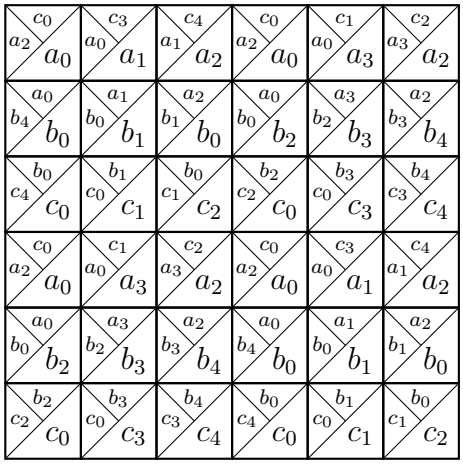
Lemma 7.3.5. *A periodic tiling of action tiles exists iff the corresponding protocol has a synchronous livelock with all processes enabled for some ring size N. Such a livelock also exists for every ring whose size is a multiple of N.*

Proof. Synchronous semantics ensure that every enabled process acts at every step, which is not allowed by our usual interleaving semantics. For example, consider the protocol of Example 7.3.4 on a ring of size 6 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 7.5b. 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 $(T[i, j-1].SE, T[i-1, j].SE, \texttt{X})$ where `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.



(a) Wang Tile Set



(b) Periodic Block

Figure 7.5: Instance and solution for the periodic domino problem that corresponds to finding a livelock in the "A-b-C" protocol.

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. □

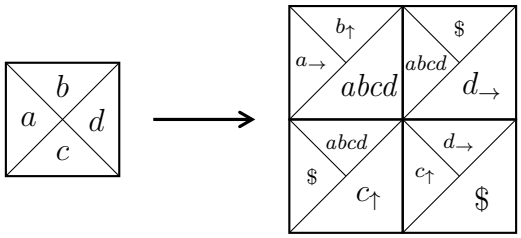


Figure 7.6: Transform 1 Wang tile to 4 action tiles.

7.3.3 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 7.3.6. *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 7.3.1 and Problem 7.3.2 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. This reduction is shown clearly in Figure 7.6.

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 7.6. 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 $\texttt{X}_\uparrow$ on its W edge or $\texttt{X}_\rightarrow$ 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. 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 7.6 that the action tiles each use 3 distinct forms of colors: $\texttt{X}_{\rightarrow}$, $\texttt{X}_{\uparrow}$, XXXX, and \$. Furthermore, the form of the west color determines the forms of the other two colors. Thus, for any two action tiles with 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 $(abcd, \$, c_{\rightarrow})$. 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 $(c_{\uparrow}, c_{\rightarrow}, \$)$. This covers all forms of tiles in the new set, therefore the new set preserves NW-determinism. □

7.4 Decidability 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 7.4.1 presents a corollary about the undecidability of verifying stabilization. Section 7.4.2 extends these undecidability results to all other execution models that we can conceive, which includes the case of a strongly fair scheduler (i.e., verifying weak stabilization).

Processes are symmetric, finite-state, 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 7.4.1 (Σ_1^0-completeness)**.** *Livelock detection for unidirectional ring protocols is undecidable (Σ_1^0-complete).*

Proof. By Lemma 7.3.6 and Lemma 7.3.3, 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 complex as the periodic domino problem. Livelock detection is also no more complex than the periodic domino problem since Lemma 7.3.3 gives a mapping reduction in the opposite direction. Therefore, like the periodic domino problem, livelock detection is Σ_1^0-complete. □

7.4.1 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 stabilization to all states where no process is enabled to act. Closure and deadlock freedom are satisfied by design, therefore verifying stabilization is Π_1^0-hard. We first claim Π_1^0-completess as a corollary to Theorem 7.4.1, followed by a lemma that is needed to show Π_1^0 membership.

Corollary 7.4.2 (Π_1^0-completeness)**.** *Verifying stabilization for unidirectional ring protocols is undecidable (Π_1^0-complete).*

Lemma 7.4.3 (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, our only topology thus far has been unidirectional rings, which 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. In this way, the current lemma is sufficient to complete the proof of Corollary 7.4.2.

In the broader context of the lemma, we can decidably detect livelocks, deadlocks, and closure violations from any state 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 is Σ_1^0, and the problem complements are in Π_1^0. □

7.4.2 Effects of Consistency and Scheduling

In Lemma 7.3.5, 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. While this equivalence

can be extended for schedulers allowing varying amounts of synchrony, it cannot be extended for strongly fair schedulers. Since our focus is undecidability, we instead find a protocol transformation such that livelocks in the resulting protocol equivalently exist under any scheduler and consistency model.

Definition 7.4.4 (FIFO Consistency)**.** In the context of stabilization on unidirectional rings, the weakest consistency model allows arbitrarily delayed (but ordered) communication between processes.

FIFO consistency [149] 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. If the topology is a unidirectional ring, 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 an unbounded FIFO queue q_i that has associated actions:

$$\begin{aligned}
\textsc{Empty}(q_i) \wedge x_i \neq x'_i &\longrightarrow \textsc{PushLast}(q_i, x_i);\\
\neg\textsc{Empty}(q_i) \wedge x_i \neq \textsc{Last}(q_i) &\longrightarrow \textsc{PushLast}(q_i, x_i);\\
\neg\textsc{Empty}(q_i) &\longrightarrow x'_i := \textsc{PopFirst}(q_i);
\end{aligned}$$

Strength of consistency models. The strongest consistency model is *strict consistency*, where processes see updates from each other instantly. Since it is the strongest, an execution under strict consistency can be simulated under every other consistency model. 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.

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 [167], 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 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 it is generally considered to be a fault, and stabilization assumes that faults are transient and eventually stop occurring.

Simulating synchrony. 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. Likewise, synchronous actions of any k processes and queues under FIFO consistency can be simulated as k individual actions by performing all PushLast actions first, process actions second, and PopFirst actions last.

Lemma 7.4.5. *If a unidirectional ring is stabilizing under FIFO consistency and no fairness, then it is stabilizing under every scheduler and consistency model.*

Proof. In the context of stabilization, we have established that FIFO consistency can simulate any execution under any other consistency model. We have also established that FIFO consistency can simulate synchronous actions under any consistency model. Since fairness does not affect the form of individual execution steps, FIFO consistency can simulate an execution under any choice of scheduler and consistency model. Thus, if a unidirectional ring of size N is stabilizing FIFO consistency and no fairness, then no other choice of scheduler and consistency model will break closure or create livelocks. Likewise, no deadlocks will be created since deadlocks are only possible when each P_i has an accurate view of x_{i-1} and clearly no deadlocks exist in these cases, where all queues are empty, under FIFO consistency. The ring is therefore stabilizing under every scheduler and consistency model. □

Propagation Count. Until now, we have considered the number of propagations in a state to be the number of enabled processes. This formula must change when delay is 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 7.4.6 (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.*

Proof. In Section 7.2, 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 7.4.7. *A unidirectional ring protocol has a livelock if and only if it also has a livelock under FIFO consistency with unbounded queues in any state. The livelocks may appear for different ring sizes.*

Proof. By Lemma 7.4.5, a livelock under interleaving semantics implies a livelock under FIFO consistency in general. 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 7.4.6, 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 7.2 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 7.2. 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 7.2.4, we can show that the propagations are periodic. Only a finite number of states exist with m propagations by Lemma 7.4.6, therefore the system must revisit some state C by Lemma 7.2.3. 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 7.2.4, m propagations of period n lead each other, implying that protocol p has a livelock under interleaving semantics. □

Lemma 7.4.8 (Deterministic Livelock)**.** *If a unidirectional ring exhibits a livelock where exactly one propagation 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 and acts

deterministically. 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 7.4.9. *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.*

Proof. 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 propagation function $\delta^\top$, and recall from Definition 7.1.1 that we can define all actions of P_i as:

$$(x_{i-1}, x_i) \in \textsc{Pre}(\delta^\top) \longrightarrow x_i := \delta^\top(x_{i-1}, x_i);$$

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:

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

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 7.4.7 and Lemma 7.3.5, 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 7.2.3, 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 $(x'_{i-1}, c_i) \notin \textsc{Pre}(\delta^\top)$. In other words, for all $i > 0$, choose x'_i such that $(x'_i, c_{i+1}) \notin \textsc{Pre}(\delta^\top)$. 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 7.4.10. *Livelock detection for unidirectional ring protocols is undecidable (Σ_1^0-complete) for any choice of scheduler and consistency model that is valid in the context of stabilization.*

Proof. Given such a protocol p, we can create a similar protocol p' using the method of Lemma 7.4.9. 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 7.4.8. In the case that p is livelock-free, no livelock exists in p' under FIFO consistency by Lemma 7.4.7, which means p' is livelock-free for every scheduler and consistency model by Lemma 7.4.5. 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 7.4.1 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 7.4.6, 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 due to Lemma 7.4.3. □

Corollary 7.4.11. *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 7.4.7), 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 7.4.10, the proof for Corollary 7.4.2 applies for any particular scheduler and consistency model, showing that verifying stabilization is Π_1^0-complete. □

7.5 Decidability of Synthesis

While verifying stabilization is undecidable for unidirectional ring protocols, we will show that it is decidable to synthesize such a protocol. Surprisingly, we find that bidirectional rings are more expressive and synthesis becomes undecidable.

Processes are symmetric, finite-state, deterministic, and self-disabling. For brevity, we only consider the case of an unfair scheduler in this section. As a consequence, any stabilizing protocol can be rewritten to use deterministic and self-disabling processes [130]. Therefore, while we retain the same constraints from previous sections, we only assume that processes are symmetric and finite-state.

Lemma 7.5.1 (Impossibility of Coloring)**.** *Let $\mathcal{L} \equiv (\forall i : L(x_{i-1}, x_i))$ be a predicate such that $\neg(\exists \gamma : L(\gamma, \gamma))$. No unidirectional ring protocol provides stabilization to $\mathcal{L}$ for all ring sizes.*

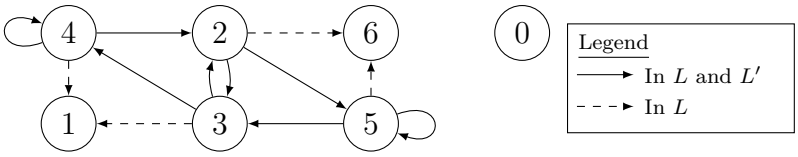
Proof. The predicate $\mathcal{L}$ implies a *coloring*, where consecutive processes must hold different values. Shukla et al. [163] have shown that no unidirectional ring protocol of deterministic, finite-state processes can stabilize to a coloring for all ring sizes. □

Theorem 7.5.2 (Decidable Synthesis for Unidirectional Rings)**.** *Given a predicate $\mathcal{L} \equiv (\forall i : L(x_{i-1}, x_i))$ and variable domain M for a unidirectional ring, $L(\gamma, \gamma)$ is true for some γ if and only if there exists a protocol that stabilizes to $\mathcal{L}$.*

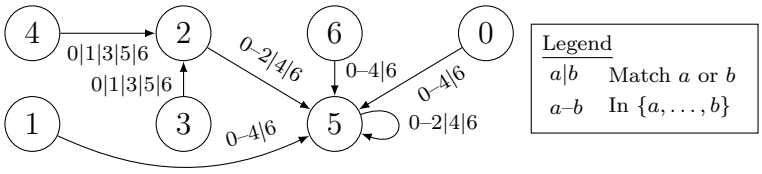
Proof. Assume that no γ exists such that $L(\gamma, \gamma)$ is true. This implies that $\forall i : x_{i-1} \neq x_i$ in the legitimate states. By Lemma 7.5.1, we have established that stabilization to $\mathcal{L}$ is impossible for all ring sizes when no γ makes $L(\gamma, \gamma)$ true. Therefore, the problem is decidable when $L(\gamma, \gamma)$ is false for all γ. We are left to show how to construct a stabilizing protocol p when some γ can make $L(\gamma, \gamma)$ true.

Find a γ such that $L(\gamma, \gamma)$ is true. Assuming such a γ exists, it is trivial to find by trying each value in $\mathbb{Z}_M$. Intuitively, we will make the stabilizing protocol p converge to $(\forall i : x_i = \gamma)$ unless it reaches some other state that satisfies $\mathcal{L}$. Figure 7.7 provides a running example where $L(x_{i-1}, x_i) \equiv ((x_{i-1}^2 + x_i^3) \bmod 7 = 3)$ and variables have domain size $M = 7$. We arbitrarily choose $\gamma = 5$ (instead of $\gamma = 4$) to satisfy $L(\gamma, \gamma)$.

Construct relation L' from arcs that form cycles in the digraph of L. The relation L can be represented as a digraph such that each arc (a, b) is in the graph *iff* $L(a, b)$ is true. Let G be this digraph (e.g., formed by both solid and dashed lines in Figure 7.7a). Closed walks in G characterize all states in $(\forall i : L(x_{i-1}, x_i))$ [89]. Create a digraph G' (and corresponding relation L') from G by removing all arcs that are not part of a cycle (e.g., arcs $(4, 1)$, $(3, 1)$, $(2, 6)$, and $(5, 6)$ in Figure 7.7a). Since closed walks of G characterize states in $\mathcal{L}$, we know that for every arc (a, b) in G that is not part of a cycle, no legitimate state contains $x_{i-1} = a \wedge x_i = b$ at any index i. All



(a) Predicates L and L' as Digraphs (i.e., G and G')



(b) Stabilizing Protocol as a Digraph

Figure 7.7: Synthesis of stabilization to $\forall i : L(x_{i-1}, x_i)$, where $L(x_{i-1}, x_i) \equiv \Big((x_{i-1}^2 + x_i^3) \bmod 7 = 3\Big)$ and $x_i \in \mathbb{Z}_7$.

closed walks of G are retained by G', therefore L' is equivalent to L for the sake of defining legitimate states $\mathcal{L} \equiv (\forall i : L'(x_{i-1}, x_i))$.

Construct a spanning tree τ with γ at the root (and has a self-loop). We can treat τ as a function where $\tau(a) = c$ means that the parent of a is c. First, let $\tau(\gamma) := \gamma$ represent the root of the tree. Next, create a tree by backwards reachability from γ in G', and assign $\tau(a) := c$ for each a that has a path $a, c, \ldots, \gamma$ in G'. Finally, let $\tau(a) := \gamma$ for each node a that has no path to γ in G'.

If these extra arcs of τ were added to G', no cycle would be created. These extra arcs are made directly to γ from nodes that cannot reach γ in G'. However, since all arcs of G' are involved in cycles, any walk in G' can find its way back to a previously visited node. Therefore, if a node cannot reach γ in G', then γ cannot reach that node. Since the extra arcs of τ would not introduce cycles in G', we know that $(\forall i : (L'(x_{i-1}, x_i) \vee \tau(x_{i-1}) = x_i))$ is yet another equivalent way to write $\mathcal{L}$.

Construct each action (a, b, c) of p by labeling each arc (a, c) of τ with all b values such that $(\neg L'(a, b) \wedge \tau(a) \neq b)$. In this way, τ defines how a process P_i in p will assign x_i when it detects an illegitimate state. Therefore, while Figure 7.7b shows the solution protocol for our example, it also illustrates τ if we ignore the arc labels. The protocol p is written succinctly by giving the following action to each process P_i.

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

This protocol p stabilizes to $\mathcal{L}$. Deadlock freedom and closure hold because each process P_i is enabled to act *iff* $(\neg L'(x_{i-1}, x_i) \wedge \tau(x_{i-1}) \neq x_i)$ holds. In other words, a process is enabled *iff* $\mathcal{L}$ is false since $\mathcal{L} = (\forall i : (L'(x_{i-1}, x_i) \vee \tau(x_{i-1}) = x_i))$. Livelock freedom holds because all periodic propagations of p consist of actions of the form

(γ, b, γ) where $L(\gamma, b)$ is false (e.g., the self-loops of node 5 in Figure 7.7b). Obviously none of these (γ, b, γ) actions lead each other since $b \neq \gamma$. Thus, our scheme has constructed a protocol p that stabilizes to $\mathcal{L}$ for any number of processes. □

Lemma 7.5.3. *Verifying livelock freedom of a unidirectional ring protocol p remains Π_1^0-complete even if a livelock of p is guaranteed to imply (1) p has a deterministic livelock that (2) involves all actions and (3) involves all values.*

Proof. Specifically, the second and third constraints stipulate that, if p has a livelock, then p has an action (a, b, c) for every value $c \in \mathbb{Z}_M$, otherwise no livelock would involve c since no action would assign using c. Additionally, if p has a livelock, then p has an action (a, b, c) for every value $a \in \mathbb{Z}_M$, otherwise no action could come after $(\mathtt{X}, \mathtt{X}, a)$ in a periodic propagation.

These 3 (numbered) constraints on p do not affect Π_1^0-completeness of verifying livelock freedom. First, by Lemma 7.4.9 and Theorem 7.4.10, deterministic livelock detection is Σ_1^0-complete on unidirectional rings. Second, deterministic livelock detection is not decidable when the livelock must involve all actions, otherwise we could detect deterministic livelocks for any protocol by checking each subset of actions. Third, deterministic livelock detection is not decidable when the livelock must involve all values, otherwise we could detect deterministic livelocks for any protocol by checking each subset of values. Thus, verifying livelock freedom for our chosen form of p remains Π_1^0-complete. □

Theorem 7.5.4 (Undecidable Synthesis for Bidirectional Rings)**.** *Given a predicate $\mathcal{L} \equiv (\forall i : L_i) \equiv (\forall i : L(x_{i-1}, x_i, x_{i+1}))$ and variable domain M (such that each $x_i \in \mathbb{Z}_M$) for a bidirectional ring, it is undecidable (Π_1^0-complete) whether a protocol can stabilize to $\mathcal{L}$ for all ring sizes such that no transitions exist within $\mathcal{L}$.*

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

Definition of closure. Our definition of closure for this proof is expanded to mean that no transitions of p' can exist within $\mathcal{L}'$. Up to this point, we have treated closure this way without loss of generality. We believe this definition is not necessary to show Π_1^0-completeness, but it certainly helps this proof.

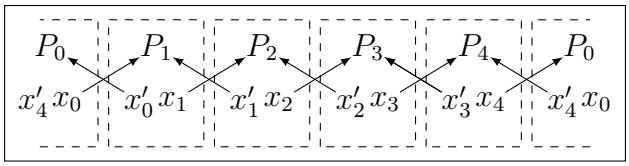


Figure 7.8: Topology for bidirectional ring protocol p' in Theorem 7.5.4. Each process P_i owns x'_{i-1} and x_i.

Assumptions about p. We assume that if p has a livelock, then it (1) has a deterministic livelock that (2) involves all actions and (3) involves all values. This does not affect Π^0_1-completeness due to Lemma 7.5.3.

Forming $\mathcal{L}'$ from p. For the bidirectional ring topology, we augment the topology of p by giving each process P_i a new variable $x'_{i-1} \in \mathbb{Z}_M$ along with its $x_i \in \mathbb{Z}_M$, making its effective domain size $M' \equiv M^2$. Since it is a bidirectional ring, P_i can read x_{i-1} and x'_{i-2} from P_{i-1} and can read x_{i+1} and x'_i from P_{i+1}. Our goal is to encode the behavior of p as a predicate $\mathcal{L}'$, therefore we need 3 distinct values to encode each action $(a, b, c) \in \xi$. To this end, we use $x_{i-1} = a$ and $x'_i = b$ to encode the precondition of a P_i action (a, b, c), and $x_i = c$ to encode its assignment. Notice that x'_i is from P_{i+1} as depicted in Figure 7.8, therefore we must ensure x'_i eventually obtains a copy of x_i when a livelock should exist. The resulting $\mathcal{L}' \equiv (\forall i : L'_i)$ looks like the p protocol with instances of x_i replaced with x'_i and a condition that x'_{i-1} is a copy of x_{i-1}. Specifically, we write L'_i as:

$$L'_i \equiv \Big((x_{i-1}, x'_i) \in \textsc{Pre}(\delta^\top) \\ \implies x'_{i-1} = x_{i-1} \wedge x_i = \delta^\top(x_{i-1}, x'_i)\Big)$$

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

$$(x_{i-1}, x'_i) \in \textsc{Pre}(\delta^\top) \wedge \Big(x'_{i-1} \neq x_{i-1} \vee x_i \neq \delta^\top(x_{i-1}, x'_i)\Big) \\ \longrightarrow \quad x'_{i-1} := x_{i-1};\ x_i := \delta^\top(x_{i-1}, x'_i);$$

Notice that p' is deadlock-free and preserves closure since a process P_i can act *iff* its L'_i is unsatisfied. In fact, we can show that this p' is the only such protocol when p has a livelock. Assuming p has a livelock, consider a ring of 5 processes executing p' where a process P_2 and its readable variables from P_1 and P_3 have arbitrary values. By our earlier assumptions about p, it has an action (a, b, c) for any given a or c (not both), and $(a, c) \notin \textsc{Pre}(\delta^\top)$ because processes of p are self-disabling. Thus, we can choose x_0 of P_0 to make $(x_0, x'_1) \notin \textsc{Pre}(\delta^\top)$ for P_1, and we can choose x'_3 of P_4 to

make $(x_2, x_3') \notin \textsc{Pre}(\delta^\top)$ for P_3. We have satisfied L_1' and L_3', and we can likewise satisfy L_0' and L_4' by choosing values of x_4 and x_4' respectively. Thus, the system is in a legitimate state *iff* L_2' is satisfied. Therefore, if L_2' is satisfied, then P_2 cannot act without adding a transition within $\mathcal{L}'$ (i.e., breaking closure). As a consequence, no other process but P_2 can act if L_2' is not satisfied, which leaves P_2 to correct L_2' itself. Since processes are symmetric, each P_i of p' must have the action above to ensure $x_{i-1}' = x_{i-1}$ and $x_i = \delta^\top(x_{i-1}, x_i')$ when $(x_{i-1}, x_i') \notin \textsc{Pre}(\delta^\top)$.

If p has a livelock, then p' has a livelock. Assume p has a livelock to show that this implies a livelock of p'. By assumption, p has a deterministic livelock from some state $C = (c_0, \ldots, c_{N-1})$ on a ring of size N where only the first process is enabled; i.e., $(c_{i-1}, c_i) \in \textsc{Pre}(\delta^\top)$ only for $i = 0$. Let $C' = (c_0', \ldots, c_{N-1}')$ be the state of this system after all processes act once. That is, $c_0' = \delta^\top(c_{N-1}, c_0)$ and $c_i' = \delta^\top(c_{i-1}', c_i)$ for all other $i > 0$. We can construct a livelock state of p' from the same $x_i = c_i$ values for all i and $x_i' = c_i$ for all $i < N-1$. The value of x_{N-1}' can be c_{N-1}, but can be anything else such that $(x_{N-2}, x_{N-1}') \notin \textsc{Pre}(\delta^\top)$. In this state of p', only P_0 is enabled since we assumed that $(c_{i-1}, c_i) \in \textsc{Pre}(\delta^\top)$ only holds for $i = 0$. P_0 then performs $x_0 := c_0'$ and $x_{N-1}' := c_{N-1}$. This does not enable P_{N-1}, but does enable P_1 to perform $x_1 := c_1'$ and $x_0' := c_0'$. The execution continues for $P_2, \ldots, P_{N-1}$ to assign $x_i := c_i'$ and $x_{i-1}' := c_{i-1}'$ for all $i > 1$. At this point the system is in a state where $x_i = c_i'$ for all i and $x_i' = c_i'$ for all $i < N-1$. The value of x_{N-1}' is c_{N-1}, which leaves it disabled. This state of p' matches state C' of p using the same constraints as we used to match the initial state C. Therefore p' can continue to simulate p, showing that it has a livelock.

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

Synthesis is Π_1^0-complete. The unidirectional ring protocol p is livelock-free *iff* a bidirectional ring protocol p' exists that stabilizes to $\mathcal{L}'$, where $\mathcal{L}'$ has a form determined by p. By Theorem 7.4.10, our reduction from verifying livelock freedom on unidirectional rings shows that synthesizing stabilization on bidirectional rings is Π_1^0-hard. The synthesis problem is also in Π_1^0 because verification is in Π_1^0 due to

Lemma 7.4.3 and there are only a finite number of possible protocols to synthesize. That is, by running the non-stabilization detection semi-algorithm on each candidate protocol, we have a semi-algorithm that returns if none of them are stabilizing. Thus, the complementary problem of synthesis is in Π_1^0, making it Π_1^0-complete. $\square$

Chapter 8:
Conclusions and Future Work

8.1 Conclusions

We have presented a shadow/puppet synthesis algorithm that, when given a specification, performs a complete backtracking search. The effectiveness of separating specification (shadow) from implementation (puppet) has been justified by 4 new self-stabilizing protocols that improve upon the process space requirements of published work. Each of these could be specified intuitively and without the risk of interfering with our ability to find a solution.

This synthesis algorithm was shown to be solving an NP-complete problem, which justifies our use of an exponential-time backtracking algorithm. However, we were able to apply many complete heuristics to limit the branching factor of the search tree. These techniques include the Minimum Remaining Values heuristic, enforcing processes to be deterministic and self-disabling, and using recorded conflicts across parallel randomized search tasks to avoid trying the same decisions twice.

Finally, we showed that verifying stabilization of parameterized protocols is undecidable (Π^0_1-complete) even on unidirectional rings with very restrictive assumptions on the processes. When designing protocols, we usually want them to work for all topologies of a certain class (e.g., rings of any size). Therefore, to design around this undecidability, we take a best-effort approach where we perform synthesis simultaneously on multiple small topologies (e.g., rings of size $N \in \{2, \ldots, 7\}$). The generated protocols may not stabilize for all topology instances, but as we verify a protocol for larger topologies, we eventually gain enough confidence to manually analyze and prove that the protocol is correct for all instances.

8.2 Future Work

Given our success in synthesizing new self-stabilizing protocols, it appears that automatic techniques have a great potential in this field. As opposed to manual techniques, our approach does not require a designer to be an expert in fault tolerance. For example, undergraduate students in computer science have successfully used the Protocon tool in coursework to analyze and design simple protocols. If our synthesis techniques can be made smarter as to handle processes with larger state spaces and

more flexible topologies, then synthesis could very well be the technology that brings self-stabilization into general use. The areas of improvement can be split into three categories: utility, speed, and generalization.

8.2.1 Utility

Current synthesis techniques leave much to be desired since they lack the constructs needed to handle large classes of useful protocols. This is understandable due to the complexity of adding convergence or general fault tolerance [132, 137, 140], therefore increasing the expressiveness of synthesis problem's input must be done carefully. Below are some features that may be compatible with our current approach to synthesis.

Weak Fairness. Weak fairness is a practical assumption, but it can be costly to verify [47] and causes our current search algorithm to be incomplete. Even so, it would allow us to consider systems where multiple processes are enabled in the legitimate states. Weak fairness may also offer advantages in reasoning since it is the most natural fairness assumption; i.e., we assume weak fairness in real programs. However, we have shown in Chapter 6 that adding stabilization under weak fairness remains NP-complete, unlike the same problem under global fairness.

Unstructured Topologies. Our current synthesis methods do not support general graph topologies. Rather, each process template can read and write fixed numbers of variables. In this way, we can enumerate all possible actions (local transitions) of the processes to consider during synthesis. There are many constant-degree topologies such as ring, chain, torus, grid, n-ary tree, and Kautz graph, but we often desire protocols to work on general graphs. In order to enumerate actions of a process that can have an arbitrary number of adjacent processes, we need to reduce its possible inputs and state changes to have a finite form. We need to further investigate how this reduction should work by considering existing protocols. One option is to synthesize an infimum operator [170] or the more general r-operator [72, 73] to reduce the values from adjacent processes. This covers operators like min, max, and $\cup$ (set union) that are useful for computing height the center of a tree, height in a tree, and graph coloring, but they may not be useful enough. Consider the Byzantine Generals' Problem [158], where the majority function is used to pick the most popular choice.

To generate connected graph topologies of a given size, we can use the *geng* tool from the Nauty software suite [152]. The Nauty software can also generate trees and planar graphs given the number of vertices. This allows us to perform automatic verification of a protocol on all topologies up to a certain number of processes. Though the number of graphs for a given number of vertices quickly becomes infeasible, our limit may be up to 5, 6, 7, or 8 unlabeled vertices that can respectively make 21, 112, 853, and 11117 unique connected graphs. We can also run these cases in parallel to ease the burden of verification.

Sufficiently Large Variables. Many problems require or benefit from a practical assumption that each process can store at least $O(\lg N)$ bits, where N is the number

of processes. For example, Dijkstra's token ring [64] can practically be implemented with 32-bit integers when it is safe to assume that the ring size will never exceed 2^{32} processes. This introduces three issues in (1) representing the transition system, (2) enumerating the possible actions of a process, and (3) enforcing decidability. The decision diagrams that we currently use do not support arbitrarily large variables, but finite automata make this feasible [32]. To enumerate the possible actions of a process, we can provide a set of operations that a process can perform on its values such as the r-operators used in [59, 72, 73]. Predetermined operations would also help with decidability since iterative application of some simple functions can yield unpredictable behavior [11, 54, 142]

8.2.2 Speed

Symmetry. Our search algorithm can achieve a substantial performance increase if we exploit symmetry of the search space. (Note that this has nothing to do with symmetry of processes or links.) That is, at a given point in the search, choosing a candidate action A may be equivalent to choosing a candidate action B. For example, consider synthesizing a 3-coloring protocol for rings. Initially, we start without any actions in the protocol. Our first decision may be to add action A that is $(x_{i-1} = 0 \wedge x_i = 0 \wedge x_{i+1} = 0 \longrightarrow x_i := 1;)$ to the under-approximated protocol. Though we could have chosen to add action B that is $(x_{i-1} = 2 \wedge x_i = 2 \wedge x_{i+1} = 2 \longrightarrow x_i := 0;)$. In fact, the effect of B is the same as A because the labeled transition systems are isomorphic. This is seen by transforming A's transition system to B's transition system by relabeling the values of x_i as $0 \to 2$, $1 \to 0$, and $2 \to 1$.

Cycle Detection. Cycle detection is the most costly operation when checking whether a protocol is self-stabilizing. This cost carries over to the synthesis algorithm when it checks for inconsistencies in a partial solution.

One technique that we have used successfully is to assume a synchronous scheduler. This kind of cycle detection is fast when processes are deterministic because the system as a whole acts deterministically. However, this cycle detection may find new cycles that are invalid in the asynchronous case, and it may miss asynchronous cycles. Some topologies, particularly unidirectional rings, guarantee that any asynchronous livelock will also appear as a synchronous livelock. We have therefore used synchronous cycle detection for our token rings in order to verify up to high process counts. Some speedups are enormous: For one of our synthesized 6-state token rings, verification on 17 processes took 23 hours under an asynchronous scheduler yet only 3 seconds using the synchronous scheduler.

8.2.3 Generalization

Regular Model Checking. In Section 3.4.3, we discussed regular model checking as a means to verify whether a protocol is generalizable. This kind of automatic

verification would be convenient to couple with synthesis, but unfortunately there is no guarantee as to whether such a verification will halt.

Deadlock Freedom. It is beneficial to further investigate the limits of the deadlock detection method of Farahat and Ebnenasir [89] for parameterized systems. For example, consider a two-dimensional torus topology where each process has 4 neighbors to form a rectangular lattice, but the opposing ends of the lattice connect with each other to form the surface of a torus. Give each process 4 color variables, and let each variable be read by a different neighboring process. Given any Wang tile set, we can create a protocol whose deadlock states correspond exactly to states where each process holds the colors of a tile in the set, and the color of tiles is the same on edges where they meet. Clearly a deadlock in the parameterized protocol corresponds to a periodic tiling, therefore deadlock detection is undecidable for this topology.

References

[1] P. A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular tree model checking. In *International Conference on Computer-Aided Verification*, pages 555–568, 2002.

[2] 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.

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

[4] F. Abujarad and S. S. Kulkarni. Constraint based automated synthesis of nonmasking and stabilizing fault-tolerance. In *IEEE Symposium on Reliable Distributed Systems*, pages 119–128, 2009.

[5] F. Abujarad and S. S. Kulkarni. Multicore constraint-based automated stabilization. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 47–61, 2009.

[6] F. Abujarad and S. S. Kulkarni. Weakest invariant generation for automated addition of fault-tolerance. *Electronic Notes in Theoretical Computer Science*, 258(2):3–15, 2009.

[7] F. Abujarad and S. S. Kulkarni. Automated constraint-based addition of nonmasking and stabilizing fault-tolerance. *Theoretical Computer Science*, 412(33):4228–4246, 2011.

[8] Y. Afek, S. Kutten, and M. Yung. The local detection paradigm and its application to self-stabilization. *Theoretical Computer Science*, 186(1-2):199–229, 1997.

[9] G. Alari. Improving the probabilistic three-state self stabilizing ring. Technical Report rr95-10, Université Catholique de Louvain, Aug. 1995. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.8312`.

[10] T. J. Ameloot, F. Neven, and J. V. den Bussche. Relational transducers for declarative networking. *Journal of the ACM*, 60(2):15, 2013.

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

[12] A. Arnold and P. Crubille. A linear algorithm to solve fixed-point equations on transition systems. *Information Processing Letters*, 29(2):57–66, 1988.

[13] A. Arora. *A foundation of fault-tolerant computing*. PhD thesis, University of Texas at Austin, 1992.

[14] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[15] A. Arora and M. G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.

[16] A. Arora, M. G. Gouda, and G. Varghese. Constraint satisfaction as a basis for designing nonmasking fault-tolerance. *Journal of High Speed Networks*, 5(3):293–306, 1996.

[17] A. Arora and S. S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Transactions on Software Engineering*, 24(6):435–450, 1998.

[18] A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *IEEE International Conference on Distributed Computing Systems*, pages 436–443, 1998.

[19] P. C. Attie, A. Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Transactions on Programming Languages and Systems*, 26(1):125–185, 2004.

[20] P. C. Attie and E. A. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Transactions on Programming Languages and Systems*, 20(1):51–115, 1998.

[21] P. C. Attie and E. A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM Transactions on Programming Languages and Systems*, 23(2):187–242, 2001.

[22] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[23] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. A time-optimal self-stabilizing synchronizer using a phase clock. *IEEE Transactions on Dependable and Secure Computing*, 4(3):180–190, 2007.

[24] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction (extended abstract). In *Symposium on Foundations of Computer Science*, pages 268–277, 1991.

[25] J. Beauquier, A. K. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. *Chicago Journal of Theoretical Computer Science*, 2002, 2002.

[26] J. Beauquier, M. Gradinariu, and C. Johnen. Cross-over composition - enforcement of fairness under unfair adversary. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 19–34, 2001.

[27] J. Beauquier, M. Gradinariu, and C. Johnen. Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings. *Distributed Computing*, 20(1):75–93, 2007.

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

[29] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[30] L. Blin, M. G. Potop-Butucaru, and S. Rovedakis. Self-stabilizing minimum degree spanning tree within one from the optimal degree. *Journal of Parallel and Distributed Computing*, 71(3):438–449, 2011.

[31] L. Blin and S. Tixeuil. Compact deterministic self-stabilizing leader election - the exponential advantage of being talkative. In *International Symposium on Distributed Computing*, pages 76–90, 2013.

[32] B. Boigelot and P. Wolper. Representing arithmetic constraints with finite automata: An overview. In *International Conference on Logic Programming*, pages 1–19, 2002.

[33] B. Bonakdarpour, A. Ebnenasir, and S. S. Kulkarni. Complexity results in revising unity programs. *ACM Transactions on Autonomous and Adaptive Systems*, 4(1), 2009.

[34] B. Bonakdarpour and S. S. Kulkarni. Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In *IEEE International Conference on Distributed Computing Systems*, page 3, 2007.

[35] B. Bonakdarpour and S. S. Kulkarni. Revising distributed unity programs is np-complete. In *International Conference on Principles of Distributed Systems*, pages 408–427, 2008.

[36] B. U. Borowsky and S. Edelkamp. Optimal metric planning with state sets in automata representation. In *AAAI Conference on Artificial Intelligence*, pages 874–879, 2008.

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

[38] A. Bouajjani, A. Legay, and P. Wolper. Handling liveness properties in (*omega*-)regular model checking. *Electronic Notes in Theoretical Computer Science*, 138(3):101–115, 2005.

[39] C. Boulinier, F. Petit, and V. Villain. When graph theory helps self-stabilization. In *ACM Symposium on Principles of Distributed Computing*, pages 150–159, 2004.

[40] C. Boulinier, F. Petit, and V. Villain. Synchronous vs. asynchronous unison. *Algorithmica*, 51(1):61–80, 2008.

[41] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[42] A. Bui, A. K. Datta, F. Petit, and V. Villain. State-optimal snap-stabilizing PIF in tree networks. In *IEEE International Conference on Distributed Computing Systems*, pages 78–85, 1999.

[43] J. E. Burns, M. G. Gouda, and R. E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1):35–42, 1993.

[44] J. E. Burns and J. K. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11(2):330–344, 1989.

[45] 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.

[46] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[47] J. Chen, F. Abujarad, and S. S. Kulkarni. Towards scalable model checking of self-stabilizing programs. *Journal of Parallel and Distributed Computing*, 73(4):400–410, 2013.

[48] J. Chen and S. S. Kulkarni. Effectiveness of transition systems to model faults. In *International Conference on Logical Aspects of Fault Tolerance*, 2011.

[49] V. Chernoy, M. Shalom, and S. Zaks. A self-stabilizing algorithm with tight bounds for mutual exclusion on a ring. In *International Symposium on Distributed Computing*, pages 63–77, 2008.

[50] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[51] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.

[52] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[53] E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. *ACM Transactions on Programming Languages and Systems*, 19(5):726–750, 1997.

[54] J. H. Conway. Unpredictable Iterations. In *Proceedings of the 1972 Number Theory Conference*, pages 49–52. University of Colorado, Boulder, 1972.

[55] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In *USENIX Annual Technical Conference*, volume 12, 2012.

[56] A. Daliot and D. Dolev. Self-stabilization of byzantine protocols. In *Self-Stabilizing Systems*, pages 48–67, 2005.

[57] A. K. Datta, L. L. Larmore, S. Devismes, K. Heurtefeux, and Y. Rivierre. Self-stabilizing small k-dominating sets. *International Journal of Networks and Communications*, 3(1):116–136, 2013.

[58] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[59] S. Delaët, B. Ducourthial, and S. Tixeuil. Self-stabilization with r-operators revisited. *Journal of Aerospace Computing, Information, and Communication*, 3(10):498–514, 2006.

[60] M. Demirbas and A. Arora. Convergence refinement. In *IEEE International Conference on Distributed Computing Systems*, pages 589–597, 2002.

[61] M. Demirbas and A. Arora. Specification-based design of self-stabilization. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):263–270, Jan 2016.

[62] J. Desel, E. Kindler, T. Vesper, and R. Walter. A simplified proof for a self-stabilizing protocol: A game of cards. *Information Processing Letters*, 54(6):327–328, 1995.

[63] S. Devismes, S. Tixeuil, and M. Yamashita. Weak vs. self vs. probabilistic stabilization. In *IEEE International Conference on Distributed Computing Systems*, pages 681–688, 2008.

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

[65] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[66] D. Dolev, J. H. Korhonen, C. Lenzen, J. Rybicki, and J. Suomela. Synchronous counting and computational algorithm design. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 237–250, 2013.

[67] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[68] S. Dolev, M. G. Gouda, and M. Schneider. Memory requirements for silent stabilization. *Acta Inf.*, 36(6):447–462, 1999.

[69] S. Dolev, Y. A. Haviv, and M. Sagiv. Self-stabilization preserving compiler. *ACM Transactions on Programming Languages and Systems*, 31(6), 2009.

[70] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997.

[71] B. Ducourthial. r-semi-groups: A generic approach for designing stabilizing silent tasks. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 281–295, 2007.

[72] B. Ducourthial and S. Tixeuil. Self-stabilization with r-operators. *Distributed Computing*, 14(3):147–162, 2001.

[73] B. Ducourthial and S. Tixeuil. Self-stabilization with path algebra. *Theoretical Computer Science*, 2003:293–1, 2003.

[74] A. Ebnenasir. *Automatic Synthesis of Fault-Tolerance*. PhD thesis, Michigan State University, May 2005.

[75] A. Ebnenasir and A. Farahat. A lightweight method for automated design of convergence. In *IEEE International Parallel and Distributed Processing Symposium*, pages 219–230, 2011.

[76] A. Ebnenasir and A. Farahat. Swarm synthesis of convergence for symmetric protocols. In *European Dependable Computing Conference*, pages 13–24, 2012.

[77] 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.

[78] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30(1):1–24, 1985.

[79] 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.

[80] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *Symposium on Logic in Computer Science*, pages 267–278, 1986.

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

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

[83] F. Faghih and B. Bonakdarpour. Smt-based synthesis of distributed self-stabilizing systems. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 165–179, 2014.

[84] F. Faghih and B. Bonakdarpour. Smt-based synthesis of distributed self-stabilizing systems. *ACM Transactions on Autonomous and Adaptive Systems*, 10(3):21, 2015.

[85] N. Fallahi, B. Bonakdarpour, and S. Tixeuil. Rigorous performance evaluation of self-stabilization using probabilistic model checking. In *IEEE Symposium on Reliable Distributed Systems*, pages 153–162, 2013.

[86] Y. Fang, N. Piterman, A. Pnueli, and L. D. Zuck. Liveness with invisible ranking. *International Journal on Software Tools for Technology Transfer*, 8(3):261–279, 2006.

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

[88] A. Farahat and A. Ebnenasir. A lightweight method for automated design of convergence in network protocols. *ACM Transactions on Autonomous and Adaptive Systems*, 7(4):38:1–38:36, Dec. 2012.

[89] 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.

[90] B. Finkbeiner and S. Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):519–539, 2013.

[91] K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 420–434, 2001.

[92] D. Fisman, O. Kupferman, and Y. Lustig. On verifying fault tolerance of distributed protocols. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 315–331, 2008.

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

[94] P. Funk and I. Zinnikus. Self-stabilization as multiagent systems property. In *International Conference on Autonomous Agents and Multi-Agent Systems*, pages 1413–1414, 2002.

[95] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[96] R. Gentilini, C. Piazza, and A. Policriti. Computing strongly connected components in a linear number of symbolic steps. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582, 2003.

[97] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Anonymous daemon conversion in self-stabilizing algorithms by randomization in constant space. In *International Conference on Distributed Computing and Networking*, pages 182–190, 2008.

[98] C. P. Gomes, B. Selman, and H. A. Kautz. Boosting combinatorial search through randomization. In *AAAI Conference on Artificial Intelligence*, pages 431–437, 1998.

[99] M. G. Gouda. The triumph and tribulation of system stabilization. In *Workshop on Distributed Algorithms*, pages 1–18, 1995.

[100] M. G. Gouda. The theory of weak stabilization. In *Workshop on Self-Stabilizing Systems*, pages 114–123, 2001.

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

[102] M. G. Gouda and F. F. Haddix. The stabilizing token ring in three bits. *Journal of Parallel and Distributed Computing*, 35(1):43–48, May 1996.

[103] M. G. Gouda and T. Herman. Adaptive programming. *IEEE Transactions on Software Engineering*, 17(9):911–921, 1991.

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

[105] M. Gradinariu and C. Johnen. Self-stabilizing neighborhood unique naming under unfair scheduler. In *International European Conference on Parallel and Distributed Computing*, pages 458–465, 2001.

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

[107] R. K. Guy and F. Harary. On the Möbius ladders. *Canadian Mathematical Bulletin*, 10(4):493–496, 1967.

[108] P. Habermehl and T. Vojnar. Regular model checking using inference of regular languages. *Electronic Notes in Theoretical Computer Science*, 138(3):21–36, 2005.

[109] S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Linear time self-stabilizing colorings. *Information Processing Letters*, 87(5):251–255, 2003.

[110] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, June 1990.

[111] T. Herman. Self-stabilization: Randomness to reduce space. *Distributed Computing*, 6:95–98, 1992.

[112] J.-H. Hoepman. Self-stabilizing ring-orientation using constant space. *Information and Computation*, 144(1):18–39, 1998.

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

[114] R. R. Howell, M. Nesterenko, and M. Mizuno. Finite-state self-stabilizing protocols in message-passing systems. *Journal of Parallel and Distributed Computing*, 62(5):792–817, 2002.

[115] S.-T. Huang. Leader election in uniform rings. *ACM Transactions on Programming Languages and Systems*, 15(3):563–573, 1993.

[116] M. Imase and M. Itoh. A design for directed graphs with minimum diameter. *IEEE Transactions on Computers*, 32(8):782–784, 1983.

[117] A. Israeli and M. Jalfon. Uniform self-stabilizing ring orientation. *Information and Computation*, 104(2):175–196, 1993.

[118] G. Itkis, C. Lin, and J. Simon. Deterministic, constant space, self-stabilizing leader election on uniform rings. In *Workshop on Distributed Algorithms*, pages 288–302, 1995.

[119] S. Jacobs and R. Bloem. Parameterized synthesis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 362–376, 2012.

[120] A. Jhumka. *Automated design of efficient fail-safe fault tolerance*. PhD thesis, Darmstadt University of Technology, 2004.

[121] A. Jhumka, F. C. Freiling, C. Fetzer, and N. Suri. An approach to synthesise safe systems. *International Journal of Security and Networks*, 1(1/2):62–74, 2006.

[122] H. Kakugawa and T. Masuzawa. A self-stabilizing minimal dominating set algorithm with safe convergence. In *IEEE International Parallel and Distributed Processing Symposium*, 2006.

[123] M. H. Karaata. Self-stabilizing strong fairness under weak fairness. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):337–345, 2001.

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

[125] S. Katz and K. J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993.

[126] A. Khalimov, S. Jacobs, and R. Bloem. Towards efficient parameterized synthesis. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 108–127, 2013.

[127] A. P. Klinkhamer and A. Ebnenasir. On the complexity of adding convergence. In *Fundamentals of Software Engineering*, pages 17–33, 2013.

[128] A. P. Klinkhamer and A. Ebnenasir. Verifying livelock freedom on parameterized rings. Technical Report CS-TR-13-01, Michigan Technological University, July 2013. `http://www.cs.mtu.edu/html/tr/13/13-01.pdf`.

[129] A. P. Klinkhamer and A. Ebnenasir. Verifying livelock freedom on parameterized rings and chains. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 163–177, 2013.

[130] A. P. Klinkhamer and A. Ebnenasir. Synthesizing self-stabilization through superposition and backtracking. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 252–267, 2014.

[131] A. P. Klinkhamer and A. Ebnenasir. Synthesizing self-stabilization through superposition and backtracking. Technical Report CS-TR-14-01, Michigan Technological University, May 2014. `http://www.mtu.edu/cs/research/papers/pdfs/CS-TR-14-01.pdf`.

[132] A. P. Klinkhamer and A. Ebnenasir. On the hardness of adding nonmasking fault tolerance. *IEEE Transactions on Dependable and Secure Computing*, 12(3):338–350, May 2015.

[133] A. P. Klinkhamer and A. Ebnenasir. Shadow/puppet synthesis: A stepwise method for the design of self-stabilization. *IEEE Transactions on Parallel and Distributed Systems*, 2016. In Press.

[134] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

[135] A. Kosowski and L. Kuszner. Energy optimisation in resilient self-stabilizing processes. In *International Conference on Parallel Computing in Electrical Engineering*, pages 105–110, 2006.

[136] S. S. Kulkarni. *Component-based design of fault-tolerance*. PhD thesis, Ohio State University, 1999.

[137] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 82–93, 2000.

[138] S. S. Kulkarni and M. Arumugam. Transformations for write-all-with-collision model. *Computer Communications*, 29(2):183–199, 2006.

[139] S. S. Kulkarni and A. Ebnenasir. The complexity of adding failsafe fault-tolerance. In *IEEE International Conference on Distributed Computing Systems*, pages 337–344, 2002.

[140] S. S. Kulkarni and A. Ebnenasir. Complexity issues in automated synthesis of failsafe fault-tolerance. *IEEE Transactions on Dependable and Secure Computing*, 2(3):201–215, 2005.

[141] S. S. Kulkarni and A. Ebnenasir. The effect of the specification model on the complexity of adding masking fault tolerance. *IEEE Transactions on Dependable and Secure Computing*, 2(4):348–355, 2005.

[142] S. A. Kurtz and J. Simon. The undecidability of the generalized collatz problem. In *Theory and Applications of Models of Computation*, pages 542–553, 2007.

[143] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic verification of herman's self-stabilisation algorithm. *Formal Aspects of Computing*, 24(4):661–670, 2012.

[144] 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.

[145] L. Lamport and N. A. Lynch. Distributed computing: Models and methods. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 1157–1199. Elsevier, 1990.

[146] L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[147] D. Li, X. Lu, and J. Wu. FISSIONE: a scalable constant degree and low congestion dht scheme based on kautz graphs. In *IEEE International Conference on Computer Communications*, pages 1677–1688, 2005.

[148] Y. Lin, B. Bonakdarpour, and S. S. Kulkarni. Automated addition of fault-tolerance under synchronous semantics. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 266–280, 2013.

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

[150] Z. Liu and M. Joseph. Transformation of programs for fault-tolerance. *Formal Aspects of Computing*, 4(5):442–469, 1992.

[151] A. J. Mayer, R. Ostrovsky, Y. Ofek, and M. Yung. Self-stabilizing symmetry breaking in constant space. *SIAM Journal on Computing*, 31(5):1571–1595, 2002.

[152] B. D. McKay and A. Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112, 2014.

[153] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. *Theor. Comput. Sci.*, 403(2-3):239–264, 2008.

[154] B. Neggazi, V. Turau, M. Haddad, and H. Kheddouci. A self-stabilizing algorithm for maximal p-star decomposition of general graphs. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 74–85, 2013.

[155] M. Nesterenko and A. Arora. Stabilization-preserving atomicity refinement. *Journal of Parallel and Distributed Computing*, 62(5):766–791, 2002.

[156] M. Nesterenko and S. Tixeuil. Ideal stabilization. *International Journal of Grid and Utility Computing*, 4(4):219–230, 2013.

[157] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752, 1992.

[158] M. C. Pease, R. E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[159] A. Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[160] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Symposium on Foundations of Computer Science*, pages 746–757, 1990.

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

[162] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[163] S. K. Shukla, D. J. Rosenkrantz, and S. S. Ravi. Developing self-stabilizing coloring algorithms via systematic randomization (extended abstract). In *Proceedings of the International Workshop on Parallel Processing*, pages 668–673, 1994.

[164] S. K. Shukla, D. J. Rosenkrantz, and S. S. Ravi. Observations on self-stabilizing graph algorithms for anonymous networks (extended abstract). In *Proceedings of the 2nd Workshop on Self-Stabilizing Systems*, pages 7–1, 1995.

[165] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[166] F. Somenzi. CUDD: Cu decision diagram package release 2.3.0, 1998.

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

[168] F. A. Stomp. Structured design of self-stabilizing programs. In *Israeli Symposium on Theory of Computing and Systems*, pages 167–176, 1993.

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

[170] G. Tel. Total algorithms. In *International Conference on Concurrency*, pages 277–291, 1988.

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

[172] T. Touili. Computing transitive closures of hedge transformations. *International Journal of Critical Computer-Based Systems*, 3(1/2):132–150, 2012.

[173] M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, 32(2):183–221, 1986.

[174] G. Varghese. *Self-stabilization by local checking and correction*. PhD thesis, MIT, Oct. 1992.

[175] G. Varghese. Self-stabilization by counter flushing. *SIAM Journal on Computing*, 30(2):486–510, 2000.

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

[177] U. Wappler and C. Fetzer. Software encoded processing: Building dependable systems with commodity hardware. In *Computer Safety, Reliability, and Security*, pages 356–369. Springer, 2007.

[178] P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *International Conference on Computer-Aided Verification*, pages 88–97, 1998.

[179] L. Zhu and S. Kulkarni. Synthesizing round based fault-tolerant programs using genetic programming. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 370–372, 2013.