Automated design of self-stabilization

Aly M. Farahat
Michigan Technological University

AUTOMATED DESIGN OF SELF-STABILIZATION

By

Aly M. Farahat

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

(Computer Science)

MICHIGAN TECHNOLOGICAL UNIVERSITY

2012

This dissertation, "Automated Design of Self-Stabilization," is hereby approved in partial fulfillment of the requirements for the Degree of DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE.

Department of Computer Science

Signatures:

Dissertation Advisor	Dr. Ali Ebnenasir
Committee Member	Dr. Steven Seidel
Committee Member	Dr. Donald Kreher
Committee Member	Dr. Jean Mayo
Committee Member	Dr. Charles Wallace
Interim Department Chair	Dr. Charles Wallace
Date	

To Mom, *Dad*, *Omar* and *Mr. Pierre Chaoul – my high school math and computer science professor*.

Their love, support and knowledge made this work see light. I owe them what I am now.

Contents

List of Figures . . . . **xi**

List of Tables . . . . **xv**

Acknowledgments . . . . **xvii**

Abstract . . . . **xix**

1 Introduction **1**
- 1.1 Motivation and Significance 2
- 1.2 Problem Statement 3
- 1.3 Organization 4

2 Background **5**
- 2.1 Preliminaries 5
  - 2.1.1 Protocols as non-deterministic finite-state machines 5
  - 2.1.2 Computations and execution semantics 8
  - 2.1.3 Closure, Convergence and Self-Stabilization 8
  - 2.1.4 Deadlocks and Livelocks. 9
  - 2.1.5 Temporal Logic. 10
- 2.2 Formal Problem Statement 14

3 Lightweight Methods for Automated Design of Convergence **15**
- 3.1 Introduction 15
- 3.2 Approximating Strong Convergence 20
- 3.3 Algorithmic Design of Strong Convergence 25
- 3.4 Case Studies 31
  - 3.4.1 Maximal Matching on a Bidirectional Ring 32
  - 3.4.2 Three Coloring 33
  - 3.4.3 Three-Ring Token Ring 34
- 3.5 Algorithmic Design of Strong Convergence in Symmetric Protocols 37
- 3.6 Case Studies for Symmetric Protocols 42
  - 3.6.1 Three Coloring 42
  - 3.6.2 Leader Election 43
  - 3.6.3 Agreement 43

3.7 Experimental Results 44
3.8 Discussion 48
3.9 Summary and Extensions 50

4 Swarm Synthesis of Convergence for Symmetric Protocols 53
4.1 Introduction 53
4.2 Complexity of Synthesizing Convergence 55
4.3 A Method for Swarm Synthesis 58
4.4 Case Studies 64
4.4.1 Maximal Matching 65
4.4.2 Agreement 67
4.5 Summary and Extensions 69

5 Local Reasoning for Global Convergence of Parameterized Rings 71
5.1 Introduction 71
5.2 Deadlock-Freedom 72
5.3 Livelock-Freedom 76
5.4 Application in Automated Design of Convergence 85
5.4.1 Synthesis Methodology 85
5.4.2 Further Examples 87
5.5 Summary and Extensions 90

6 An Exact Algebraic Characterization of Livelocks in Unidirectional Rings . . 91
6.1 Introduction 91
6.2 Algebraic Properties of Livelocks 93
6.2.1 Binary Relations on Local Transitions 93
6.2.2 Conditions for Livelock-Freedom in Unidirectional Rings 96
6.3 Additional Examples 102
6.4 Summary and Extensions 104

7 Application: Fault Tolerance of Wireless Sensor Nodes 107
7.1 Introduction 107
7.2 Structural Invariance of TinyOS Task Queue 109
7.3 Addition of Recovery 111
7.3.1 Correcting Bit-Flips with ECC 111
7.3.2 Adding Recovery to Task Queue 112
7.4 Summary and Extensions 118

8 Tools for Automated Synthesis of Convergence 121
8.1 STabilization Synthesizer (STSyn) 121
8.2 Parallel STabilization Synthesizer (pSTSyn) 123

9 Related Work: A Taxonomy 125

9.1 Control-Theoretic Approach 126
9.1.1 Design 126
9.1.2 Analysis 127
9.2 Game-Theoretic Approach 129
9.2.1 Synthesis 129
9.2.1.1 Synthesis from Specification 130
9.2.1.2 Program Revision 131
9.2.2 Verification 132
9.3 Distributed Algorithms Approach 132
9.3.1 Manual Design 133
9.3.1.1 Convergence Stairs/Ranking Functions 133
9.3.1.2 Local Checking and Correction, Snapshots and Global Reset 134
9.3.2 Automated Synthesis from Specification 135
9.3.3 Automated Program Revision 136
9.3.4 Automated Verification of Self-Stabilization 137
9.3.5 Parameterized Verification of Convergence 138
9.4 Discussion 141

10 Concluding Remarks 145
10.1 Summary 145
10.2 Extensions 146

References 149

List of Figures

1.1 Partitioning of the Set of Configurations of a Dynamic System into Legitimate and Illegitimate Configurations 2
1.2 Algorithmic Addition of Self-Stabilization 3

2.1 Transition Graph of Mutual Exclusion Example 13

3.1 The proposed lightweight method for automated design of convergence. 16
3.2 Automated design of weak and strong convergence. 18
3.3 Adding weak convergence. 20
3.4 Partitioning of $\neg I$. 22
3.5 Computing ranks and partitioning the set of illegitimate states $\neg I$. 22
3.6 Proposed heuristic for adding strong convergence. 27
3.7 Pass_Template is invoked three times by the proposed heuristic with different inputs. . . 28
3.8 Add convergence from a state predicate *From* to another state predicate *To*. 30
3.9 The Three-Ring Token Ring (TR^2) protocol. 34
3.10 Convergence synthesis for symmetric protocols 39
3.11 Summary of case studies. 44
3.12 Time spent for adding convergence to matching versus the number of processes. 45
3.13 Space usage for adding convergence to matching versus the number of processes. 46
3.14 Time spent for adding convergence to 3-Coloring versus the number of processes. 46
3.15 Space usage for adding convergence to 3-Coloring versus the number of processes. 47
3.16 Time spent for adding convergence to Token Ring versus the number of processes. 47
3.17 Space usage for adding convergence to Token Ring versus the number of processes. . . . 48
3.18 Time spent for adding convergence to Agreement versus the number of processes. 48
3.19 Space usage for adding convergence to Agreement versus the number of processes. 49
3.20 Time spent for adding convergence to Agreement versus the domain size. 49
3.21 Space usage for adding convergence to Agreement versus the domain size. 49

4.1 Ranking and approximating self-stabilization. 55
4.2 Actions of the Token Ring protocol where $j = 1, 2$. 56
4.3 State transition graph of the non-stabilizing Token Ring protocol 56
4.4 Transition groups included in the TR protocol for deadlock resolution. . . . 57
4.5 Inclusion of actions A_1, A_2, A_3, B_1, B_2 and B_3 creates a non-progress cycle in $\neg I_{TR}$. 58

4.6 Inclusion of actions A_4, A_2, A_3, B_4, B_2 and B_3 results in a self-stabilizing version of TR. 59
4.7 Overview of swarm synthesis of convergence. 60
4.8 The Main routine. 61
4.9 The Looper routine. 62
4.10 The AddConvergence routine. 63
4.11 Time spent for adding convergence to matching versus the number of processes 66
4.12 Space usage for adding convergence to matching versus the number of processes 66
4.13 Time spent for adding convergence to agreement versus the size of the variable domain . . 68
4.14 Space usage for adding convergence to agreement versus the size of the variable domain . 68
4.15 Time spent for adding convergence to agreement versus the number of processes 69
4.16 Space usage for adding convergence to agreement versus the number of processes 69

5.1 Continuation relation over all local states of Maximal Matching 74
5.2 Continuation Relation over local deadlocks of Example 5.2.2 75
5.3 Continuation Relation over local deadlocks of Example 5.2.3 76
5.4 RCG$_p$ and LTG$_p$ of the Agreement protocol 78
5.5 Precedence relation for local transitions in Example 5.3.2 82
5.6 Two precedence-preserving livelocks for Example 5.3.2. The starting global state is marked by "I" 82
5.7 LTG$_p$ of 3-coloring example 87
5.8 RCG$_p$ and LTG$_p$ of Agreement Example 88
5.9 LTG$_p$ of the Two Coloring Example 88
5.10 LTG$_p$ of the Sum-Not-Two example including every candidate t-arc. To the right, we demonstrate a s-cycle for each individual local deadlock that we resolved. 90

6.1 Pseudolivelocks Over Local Transitions of a 3-Agreement Protocol 95
6.2 Feasible Propagations Over Local Transitions of a 3-Agreement Protocol . . 95
6.3 The source state of t_i^r and $t_j^{r\ominus 1}$ share the value of $x_{r\ominus 1}$, that is why $\kappa_r(t_i^r) = t_j^{r\ominus 1}$ for any $|E| > 0$ 98
6.4 ϵ_r preserves the cyclic structure of η_r and $\eta_{r\oplus 1}$. 98

7.1 Example states of the task queue. 109
7.2 Excerpt of the Tiny OS Scheduler. 110
7.3 Valid and invalid task queue structures. 110
7.4 Adding recovery to the task queue. 112
7.5 Data structures. 113
7.6 Detect and correct `m_head`. 114
7.7 Corruption of `m_head`. 114
7.8 Cyclic corruption of the task queue. 115
7.9 Detect and correct queue size. 116
7.10 Corruption of `m_tail`. 116

7.11 Corrupted acyclic structure. 117
7.12 Detect and correct acyclic structures. 117
7.13 Space/Time cost of correction of BFs. 118
7.14 Scope of correction for Single Bit-Flips (SBFs) and Multiple Bit-Flips (MBFs). 118

8.1 Modules Included in the Implementation of STSyn 123

9.1 Taxonomy and key authors 126
9.2 Synthesize in Small Scale and Generalize 142

List of Tables

9.1 Contribution of approaches to design and verification of self-stabilization . 141
9.2 Summary of Complexity Results 141

Acknowledgments

Numerous people enter one's life, but very few have ever lasting impact. I consider myself blessed having met exceptional people during my short Ph.D. journey.

I would like to express my deep gratitude to my advisor, Dr. Ali Ebnenasir, for his consistent help, continuous support, wise guidance and vivid motivation. Dr. Ebnenasir's detail-oriented methodology and punctuality developed important aspects of my work skills that, otherwise, I would have been missing. Dr. Ebnenasir has always been available when I needed help; he constructively conducted discussions that almost always lead to new contributions in my field of research. Without Dr. Ebnenasir, this dissertation would have never existed.

I am equally indebted to my committee members. Dr. Steve Seidel taught me *computation theory*, a foundational topic essential for my research. Dr. Seidel has always demonstrated support whenever I needed it. Dr. Donald Kreher's course on combinatorial algorithms solidified my understanding of my research problem. Moreover, Dr. Kreher has keenly supported me and generously answered my frequent questions about directed graphs and algorithms for manipulating them. Dr. Charles Wallace has always been there for help, Dr. Wallace consistently provided replies to my unstructured questions. Dr. Jean Mayo provided me with the necessary understanding of distributed algorithms by allowing me to audit her class, which proved to be influential for my progress.

Many thanks go to my colleagues in the department of computer science. Alex Klinkhamer, Chong Fu and Amin Alipour were always there to reply to my questions and to develop discussions that often culminated in fruitful intellectual results.

Finally, I would like to demonstrate my strong appreciation to Sandra Kalcich, the graduate program coordinator. Sandy has taken care of every administrative detail to make my road towards my Ph.D. as smooth and as pleasant as possible. Thank you, Sandy!

Abstract

Self-stabilization is a property of a distributed system such that, regardless of the legitimacy of its current state, the system behavior shall eventually reach a legitimate state and shall remain legitimate thereafter. The elegance of self-stabilization stems from the fact that it distinguishes distributed systems by a strong fault tolerance property against arbitrary state perturbations. The difficulty of designing and reasoning about self-stabilization has been witnessed by many researchers; most of the existing techniques for the verification and design of self-stabilization are either brute-force, or adopt manual approaches non-amenable to automation.

In this dissertation, we first investigate the possibility of automatically designing self-stabilization through global state space exploration. In particular, we develop a set of heuristics for automating the addition of recovery actions to distributed protocols on various network topologies. Our heuristics equally exploit the computational power of a single workstation and the available parallelism on computer clusters. We obtain existing and new stabilizing solutions for classical protocols like maximal matching, ring coloring, mutual exclusion, leader election and agreement.

Second, we consider a foundation for local reasoning about self-stabilization; i.e., study the global behavior of the distributed system by exploring the state space of just one of its components. It turns out that local reasoning about deadlocks and livelocks is possible for an interesting class of protocols whose proof of stabilization is otherwise complex. In particular, we provide necessary and sufficient conditions – verifiable in the local state space of every process – for global deadlock- and livelock-freedom of protocols on ring topologies. Local reasoning potentially circumvents two fundamental problems that complicate the automated design and verification of distributed protocols: (1) state explosion and (2) partial state information. Moreover, local proofs of convergence are independent of the number of processes in the network, thereby enabling our assertions about deadlocks and livelocks to apply on rings of arbitrary sizes without worrying about state explosion.

Chapter 1

Introduction

Software systems are ubiquitous; they take part in almost every aspect of modern technology. Examples include smart phones, personal computers, electronic watches, home appliances, land vehicles, airplanes and weapon systems. The complexity of software-based systems tends to increase drastically due to two main reasons: (1) software subsystems integrate to build complex systems through inter-networking and, (2) the proliferation of computerized systems into many facets of our daily routine places increasingly stringent dependability requirements on software. Moreover, faults can randomly perturb a distributed system resulting in a corrupted global configuration that can only be corrected by a careful coordination among spatially distant administrators.

Transient faults are external perturbations whose effect on a dynamic system is reversible during the system execution. Transient faults are very common in computing systems and they may be caused by software (e.g., bad initialization, reconfiguration, loss of coordination), hardware (e.g., hardware decay/aging, power fluctuations) and the environment (e.g., cosmic rays, electromagnetic interference). Without causing any permanent damage, transient faults manifest themselves in the writable memory of computing systems as bit-flips. Such bit-flips perturb communication protocols to either *deadlock* configurations, where no node of a distributed system can take any actions, or *non-progress cycles/livelocks*, where the nodes of a non-deadlocked distributed system repeatedly perform a set of actions that render the entire protocol unresponsive forever. An example in [1] reports an unexpected configuration of the ARPANET protocol where three incorrect sequence numbers were injected by transient faults. The protocol cycled indefinitely among these configurations until tedious manual intervention solved the problem [2]. One possible solution to tolerate transient faults in interconnected systems is to make them *self-stabilizing*.

Regardless of its initial configuration, a *self-stabilizing* system eventually reaches a global configuration from where all its behaviors are legitimate [3]. Such a property is highly desirable in network protocols; self-stabilizing network protocols are immune to transient faults. We assume that there exists enough time for the protocol to stabilize before transient faults resume their occurrences. Figure 1.1 illustrates the legitimate and illegitimate configurations of a dynamic system together with deadlocks and non-progress cycles in illegitimate configurations. Note that although our main interest is in network protocols, component-based software systems can also benefit from self-stabilization as they have architectural constraints similar to those of a distributed systems.

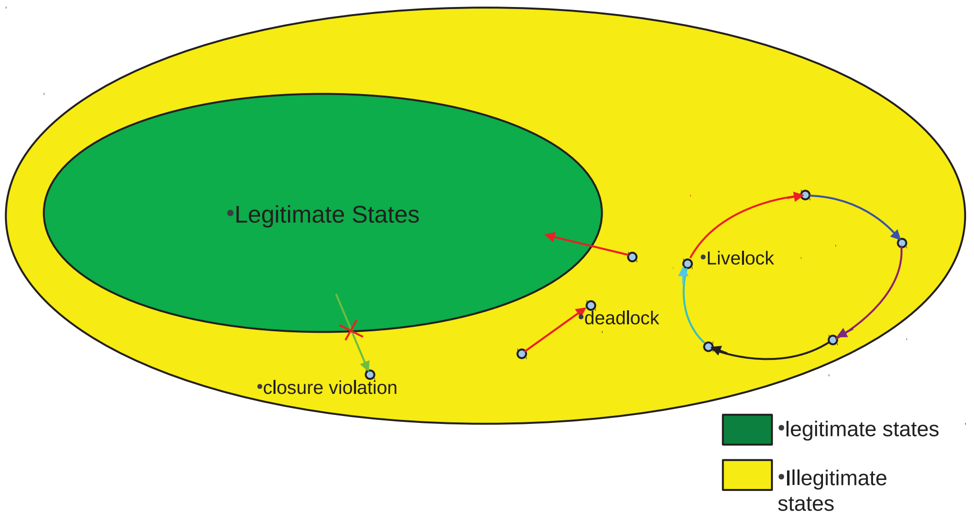


Figure 1.1: Partitioning of the Set of Configurations of a Dynamic System into Legitimate and Illegitimate Configurations

1.1 Motivation and Significance

Self-stabilization is naturally captured by living organisms. For instance, a wounded part of the human body takes its time to heal and eventually restores its initial healthy state. The human immunity system plays a major role in healing infections and as a result, recovering to the body's original normal status. On the other hand, system stability has been studied for a long time as a sub-discipline of control theory [4]. However, designing stabilization in computer systems is a non-trivial task [3], [5], [6]. First, self-stabilizing computer systems reach sets of configurations that are by themselves dynamic under legitimate behavior, unlike fixed stable points in control systems. Second, computer systems put no constraints, whatsoever, on the executions of their models. An arbitrary interleaving of the actions of processes is a valid behavior of a computer system, unlike the execution of control systems whose components are assumed to execute in a lockstep synchronization. Third, achieving stabilization in computer systems does not follow the constraints of the traditional feedback loop in control theory or the constraints of physical laws governing the controlled plant.

The difficulty of design and verification of self-stabilization has been reported by many researchers since the time Dijkstra introduced [7] self-stabilization in the seventies. For example, Gouda asserts that the time necessary to verify self-stabilization compared to the protocol design time is approximately ten to one [6]. Indeed, a stabilizing protocol should be able to recover to its legitimate behavior from *every possible* configuration. Moreover, each node in a distributed system is aware only of its neighborhood, thereby has limited observability of the global configuration. This limited scope usually prevents the node from evaluating the legitimacy of the global configuration and consequently, complicates potential corrective actions. Thus, the manual design and verification of self-stabilization are hard and error-prone. Therefore, it is of considerable importance to provide computer-aided means that facilitate the generation of stabilizing systems. The generated self-stabilizing systems are correct-by-construction, thereby eliminating the need for a proof of correctness. Furthermore, an understanding of the fundamental properties of stabilization enables the design of more efficient automated means for the verification and design of robust distributed protocols.

1.2 Problem Statement

We study the properties of transition systems [8] of self-stabilizing protocols. Given a non-stabilizing dynamic system/network protocol, we modify its transition relation such that the resulting protocol is self-stabilizing while maintaining the original behavior of the non-stabilizing protocol in the absence of transient faults. Figure 1.2 depicts a simplified view of our research problem.

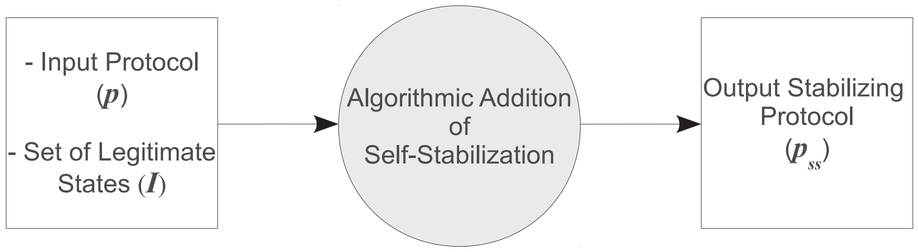


Figure 1.2: Algorithmic Addition of Self-Stabilization

Self-stabilization requires two properties, (1) closure and (2) convergence [9]. Closure states that if no faults occur, legitimate behaviors should always remain legitimate. Convergence requires that, from every possible configuration, a legitimate behavior is eventually reached in a finite number of steps when faults stop occurring. For finite-state converging protocols, it is necessary and sufficient that every illegitimate configuration is not a *deadlock* and that none of the illegitimate configurations participate in any *non-progress cycle*.

In this dissertation, we focused on designing convergence while maintaining closure through two approaches. In the first approach, we developed a set of heuristics for computer-aided design of self-stabilization. However, the difficulty of automatically synthesizing self-stabilization renders the use of complete algorithms infeasible for practical network sizes. To overcome this infeasibility, our second approach formulates theories for reasoning about deadlocks and livelocks in networks of arbitrary sizes. We demonstrated the existence of *local conditions*; i.e., verifiable for individual network nodes, that are necessary and sufficient for both deadlock-freedom and livelock-freedom of networks whose interconnection scheme is in the form of a unidirectional ring. The locality of these conditions establishes deadlock- and livelock-freedom as size-independent properties of the ring. Similar ideas could extend to arbitrary network topologies.

1.3 Organization

Chapter 2 presents basic definitions and a formal statement of the problem of adding convergence to non-stabilizing protocols/systems. In Chapter 3, we present our preliminary investigation on automated addition of convergence along with our experimental results. We have conducted our experiments using a software tool that we developed for automated addition of convergence[1]. We demonstrate an algorithmic solution to our problem statement that exploits the power of cluster computers in Chapter 4. We illustrate in Chapter 5 our initial results on properties of convergence that could be checked on individual network processes, thereby circumventing state explosion and the combinatorial nature of convergence design in the global state space. In Chapter 6, we develop a theory for local reasoning about livelocks in unidirectional rings; we prove necessary and sufficient conditions for livelock-freedom of an interesting subclass of protocols on unidirectional rings. To shed some light on the applications, we devise a method for rendering wireless sensor nodes fault-tolerant in Chapter 7. In Chapter 8, we summarize the architecture of two software tools that we have developed throughout our investigation of the automated design of stabilization. In Chapter 9, we provide an exposition and a taxonomy of related work to our research problem and in Chapter 10, we summarize our results and explore potential extensions of our work.

[1]A web interface for our software tool is accessible from the following link http://c28-0206-01.ad.mtu.edu:8888/SynStable/.

Chapter 2

Background[1]

In this chapter, we present our formal notations and definitions in Section 2.1 and formally state the problem of designing self-stabilizing finite-state systems from their non-stabilizing version in Section 2.2.

2.1 Preliminaries

In this section, we present the formal definitions of protocols, our distribution model (adapted from [13]), convergence and self-stabilization. Protocols are defined in terms of their set of variables, their transitions and their processes. The definitions of convergence and self-stabilization are adapted from [5]–[7], [9], [14].

2.1.1 Protocols as non-deterministic finite-state machines

A protocol $p(K)$ is a triplet $\langle \Phi_p,\ \Pi_p,\ \Delta_p \rangle(K)$ where K is a positive integer parameter and $\Phi_p(K) = \{v_0, \cdots, v_{M(K)-1}\}$ is a set of $M(K)$ variables where M depends on K. Each variable v_i in $\Phi_p(K)$ has a finite domain D_i ($0 \leq i \leq M(K)-1$). $\Pi_p(K) = \{P_0, \cdots, P_{K-1}\}$ is a set of K processes. $\Delta_p(K)$ is the protocol's *global transition relation* and will be defined later (p. 7). Every process $P_r = \langle R_r, W_r, \delta_r \rangle$ is a triplet such that $R_r \subseteq \Phi_p(K)$ is a subset of variables that process P_r can read ($0 \leq r \leq K-1$). The *locality* of P_r is the set of variables in R_r. $W_r \subseteq \Phi_p(K)$ is a subset of variables that

[1]This chapter is adapted from the preliminary sections of our own published references [10]–[12]. Reprinted with permission, please see the supplementary document to this dissertation for the copyright notice.

process P_r can write. We assume that $W_r \subseteq R_r$; i.e., P_r can only write variables that it can read.

A *global state* of $p(K)$ is a valuation of all variables in $\Phi_p(K)$. The *global state space* $S_p(K)$ is the set of all possible global states of $p(K)$. A *global state predicate* is any subset of $S_p(K)$ specified as a Boolean expression over variables of $\Phi_p(K)$. We say a global state predicate X *holds in a global state* s, denoted $s \in X$, *if and only if* X evaluates to true at s. The value of variable $v \in \Phi_p(K)$ at global state s is denoted $v(s)$. A *global transition* t of $p(K)$ is a pair of global states (s, s'): s is the source state of t and s' is the target state of t. Likewise, a *local state* s_r^l of P_r is a valuation of the variables in R_r $(0 \leq r \leq K-1)$. The *local state space* S_r^l is the set of all possible local states of P_r. A *local state predicate* is any subset of S_r^l specified as a Boolean expression over variables of R_r. We say that a local state predicate X_r *holds in a local state* s_r^l denoted, $s_r^l \in X_r$ if and only if X_r evaluates to true at s_r^l. The value of a variable $v \in R_r$ at local state s_r^l is denoted $v(s_r^l)$. A *local transition* t^l of P_r is a pair of local states $(s_r^l, {s_r^l}')$ of P_r such that, $\forall v \in (R_r - W_r) : v(s_r^l) = v({s_r^l}')$. δ_r denotes the set of local transitions of P_r $(0 \leq r \leq K-1)$.

Example: Three Coloring (TC). We consider a three coloring protocol over a bidirectional ring (adapted from [15]). We have $\Pi_{TC} = \{P_0, \cdots, P_{K-1}\}$, $\Phi_{TC} = \{c_0, \cdots, c_{K-1}\}$ with $D_r = \{0, 1, 2\}$ representing three distinct colors, where $0 \leq j \leq K-1$. Note that in this case $M(K) = K$. Every process P_r has one variable c_r defining its color, $R_r = \{c_{r-1}, c_r, c_{r+1}\}$ and $W_r = \{c_r\}$. $\lhd$

Two processes P_i and $P_j \in \Pi_p(K)$ $(0 \leq i, j \leq K-1)$ are *similar/symmetric* with respect to the bijection $\mathcal{F}_{ij} : R_i \rightarrow R_j$ if and only if $\delta_j = \mathcal{F}_{ij}(\delta_i)$[2]. Note that $\mathcal{F}_{ij}^{-1} = \mathcal{F}_{ji}$. We call $\mathcal{F}_{ij}$ a variable renaming/re-indexing function. We say that P_i and P_j are similar/symmetric if and only if there exists a bijection $\mathcal{F}_{ij} : R_i \rightarrow R_j$ such that P_i and P_j are similar/symmetric with respect to $\mathcal{F}_{ij}$. It is easy to prove thereof that symmetry/similarity is an equivalence relation on $\Pi_p(K)$.

Symmetry is one way that allows the recursive evaluation of the local transitions of δ_j from δ_i by finite means. As such, we can start reasoning about protocols with an unbounded number of processes; i.e., *parameterized protocols*. A parameterized protocol p is the unbounded set of finite-state protocols $p(K)$; where K is any positive integer. In Chapters 5 and 6, we develop a theory about the global properties of parameterized protocols on ring networks.

The *projection* $s \downarrow$ *Var* of a global state $s \in S_p(K)$ on a set of variables *Var* $\in \Phi_p(K)$

[2]The extension of $\mathcal{F}_{ij}$ to act on the local transition relation of P_r means the substitution of the variables in R_i of δ_i by the variables in R_j of δ_j.

is a valuation of every variable $v \in \mathit{Var}$ such that $v(s) = v(s \downarrow \mathit{Var})$. Likewise, we define the *projection* of a global transition $(s, s') \downarrow Var$ as the pair $((s \downarrow \mathit{Var}), (s' \downarrow \mathit{Var}))$. Inversely, every local state s_r^l of P_r is mapped to a set of global states $g^K(s_r^l) = \{s \in S_p(K) : \forall v \in R_r : v(s) = v(s_r^l)\}$. Likewise, every local transition t_r^l corresponds to a *group* of global transitions $g^K(s_r^l, {s_r^l}') = \{(s, s') \in S_p(K) \times S_p(K) : (\forall v \in R_r : v(s) = v(s_r^l) \wedge v(s') = v({s_r^l}')) \wedge (\forall v \notin W_r : v(s) = v(s'))\}$. Thus, $g^K(\delta_r)$ represents the set of global transitions of P_r in $p(K)$. We denote by $G(P_r)$ the set of potential transition groups of process P_r. Formally, $G(P_r) = \{g^K(t^r) | t^r$ is a *potential* local transition of P_r $\}$; i.e., t^r is any local transition that satisfies the read and write restrictions of P_r. The set of global transitions of $p(K)$ is the union of the set of global transitions of each process P_r, i.e.; $\Delta_p(K) = \cup_{r=0}^{K-1} g^K(\delta_r)$. As such, the *global transition relation* of $p(K)$ is representable as a directed graph $T_p(K)$ such that $S_p(K)$ and $\Delta_p(K)$ are its sets of vertices and arcs, respectively.

Notational convention. For abbreviation, we denote universally quantified statements over K; i.e., statements about a parameterized protocol p, by omitting K. For instance, we denote $\forall K : p(K)$ converges to $I(K)$ by p converges to I, and $\forall K : s \in g^K(s_r^l)$ by $s \in g(s_r^l)$.

TC Example. A group of global transitions for P_r is defined for the local transition where $c_{r-1} = 0$, $c_r = 0$, $c_{r+1} = 1$ and that assigns 2 to c_r. This group has 3^{K-3} transitions defined for every value of the variables unreadable by P_r. $\triangleleft$

Protocol Representation. We use a variant of Dijkstra's guarded commands language [16] to represent the set of local transitions of P_r (i.e., δ_r). A guarded command (i.e., *action*) is of the form $L : grd_r \rightarrow stmt_r$, where L is an optional label, grd_r is a local predicate of P_r, and $stmt_r$ is an assignment that updates variables of W_r *atomically*. Formally, an action $grd_r \rightarrow stmt_r$ includes a set of local transitions $(s^r, s^{r\prime})$ of P_r such that grd_r holds in every local state s^r and the atomic execution of $stmt_r$ results in a local state $s^{r\prime}$. An action $grd_r \rightarrow stmt_r$ is *enabled* in a local (global) state s^r (respectively, s_g) if and only if grd_r holds at s^r (respectively, s_g). The process P_r is *enabled*/has an *enablement* in s^r (respectively, s_g) if and only if there exists an action of P_r that is enabled at s^r (respectively, s_g). A local transition $(s^r, s^{r\prime})$ is enabled in s_g if $s_g \in g^K(s^r)$.

In Chapter 6, we consider unidirectional rings such that $R_r = \{x_{r-1}, x_r\}$ and $W_r = \{x_r\}$, for every process $P_r \in \Pi_p$ ($0 \leq r \leq K-1$). We denote a local transition $t^r = (s^r, s^{r\prime})$ of P_r on a unidirectional ring by $t^r_{(x_{r\ominus 1}(s^r))(x_r(s^r) \rightarrow x_r(s^{r\prime}))}$. We denote the set of local transitions of P_r by local transitions parameterized by values from D_r and $D_{r\ominus 1}$. For instance, the set $\tau_r = \{t^r_{(v)(v\ominus 1 \rightarrow v)} | v \in D_r\}$ is abbreviated as $t^r(v)$ since every local transition in τ_r is uniquely determined by the value of v.

TC Example. The process P_r $(0 \leq r \leq K-1)$ has the following action (addition and subtraction are performed in modulo K):

$$A_r: \ (c_r = c_{r-1}) \vee (c_r = c_{r+1}) \ \rightarrow \ c_r := \mathit{other}(c_{r-1}, c_{r+1})$$

If the color of P_r is equal to any of its neighbors, assign to P_r a color different from both of its neighbors. The function *other* non-deterministically returns a value different from both its arguments. For instance *other(c,c)* returns either $(c+1)$ *mod* 3 or $(c+2)$ *mod* 3; if $x \neq y$, then $\mathit{other}(x,y)$ returns the third remaining value. As such, $\delta_r = \{t_{v(v\rightarrow z)w}, t_{v(w\rightarrow z)w}$ $|v, w, z \in D_r$ and $(z \neq v) \wedge (z \neq w)\}$. $\lhd$

Faults. We consider *transient faults* that perturb the state of a protocol to illegitimate states without causing permanent damage (e.g., transient bit-flips). Formally, transient faults can be modeled as a set of transitions in $(S_p \times S_p)(K)$ that non-deterministically update protocol variables. We assume that transient faults occur a finite number of times. Such an assumption is necessary in order to ensure that a protocol can eventually achieve convergence.

2.1.2 Computations and execution semantics

A *computation* of a protocol $p(K)$ is a *maximal* walk σ in $T_p(K)$. σ is maximal in that either σ is infinite or if it is finite, then σ reaches a global state with no outgoing global transitions. We assume an interleaving semantics, where every global transition t_g of a computation σ belongs to the group of at least one local transition; i.e., no global transition corresponds to the joint action of more than one process. Thus, the sequence of local transitions of σ can be obtained by projecting every t_g in σ over the corresponding R_r of its local transition $(0 \leq r \leq K-1)$. A *computation prefix* σ_l is a finite walk in $T_p(K)$ that can be extended to a computation σ of $p(K)$.

2.1.3 Closure, Convergence and Self-Stabilization

A state predicate X is *closed in an action* $grd_r \rightarrow stmt_r$ if and only if executing $stmt_r$ from any state $s \in (X \wedge grd_r)$ results in a state in X. We say a state predicate X is *closed in a protocol* p if and only if X is closed in every action of p. In other words, *closure* [14] requires that every computation that starts in X remains in X.

Let I be a state predicate. We say that a protocol p *strongly converges to* I if and only if from any state, *every* computation of p reaches a state in I. A protocol p *weakly converges* to I if and only if from any state, *there exists* a computation of p that reaches a state in I. A protocol p is *strongly (respectively, weakly) self-stabilizing* to a state predicate I if and only if (1) I is closed in p and (2) p strongly (respectively, weakly) converges to I.

TC Example. The state predicate I_{color} captures the states in which any two neighboring processes have different colors. Formally, I_{color} is equal to $\forall r : 1 \leq r \leq K-1 \colon (c_{r-1} \neq c_r)$. The protocol TC is closed in I_{color} since no action is enabled in I_{color}. Thus, TC is *silent* in I_{color}.

In any state outside I_{color} there must be two neighboring processes that have the same colors. Thus, there is at least one enabled action in any state in $\neg I_{color}$. That is, there are no deadlock states in $\neg I_{color}$. Moreover, no cycles are formed in $\neg I_{color}$. Thus, TC is strongly stabilizing to I_{color}. $\lhd$

I is *locally conjunctive* if and only if for every K, $I(K)$ is a conjunction of K local state predicates LC_r, where LC_r specifies a local state predicate of P_r; i.e., $I(K) = \bigwedge_{r=0}^{K-1} LC_r$.

An *enablement* of P_r is a local state where P_r is enabled. A *corruption* (*non-corruption*) with respect to I is an enablement s_r^l of P_r such that $s_r^l \notin LC_r$ (respectively, $s_r^l \in LC_r$). Let I be a locally conjunctive closed predicate for p, a process P_r in a non-corrupt local state will never corrupt its own local state.

2.1.4 Deadlocks and Livelocks.

A *global deadlock* state s_d has no outgoing global transitions (i.e., no process is enabled), and no action of P_r is enabled in a *local deadlock* state s_d^l of P_r. A global deadlock state s_d (respectively, s_d^l) is legitimate if and only if $s_d \in I$ (respectively, $s_d^l \in LC_r$), otherwise s_d (respectively, s_d^l) is illegitimate. Notice that a parameterized protocol $p(K)$ is in a global deadlock state if and only if every process $P_r \in \Pi_p(K)$ $(0 \leq i \leq K-1)$ is in a local deadlock state. A global deadlock is illegitimate if and only if there exists a process P_r whose local deadlock is illegitimate.

In a finite-state parameterized protocol $p(K)$, *a livelock non-progress cycle* for a state predicate $I(K)$ is a computation $\ll sc_0, sc_1, \cdots, sc_{m-1}, \cdots \gg$ where $\exists m : m \geq 1 : (\forall i : i \in \mathbb{N} : sc_{i+m} = sc_i$ and $\forall i : 0 \leq i \leq m-1 : sc_i \notin I(K))$; i.e., an infinite repetition of a finite sequence of global states outside $I(K)$. In other words, a livelock is a cycle in $T_p(K)$.

Proposition 2.1.1. A protocol p *strongly converges to* I if and only if there are no global deadlock states in $\neg I$ and no livelocks in $\Delta_p \mid \neg I$.

When it is clear from the context, we shall omit the set of legitimate states I; e.g., instead of saying 'a livelock for $I(K)$', we say 'a livelock'.

Local Checkability. We follow Varghese [1] in defining locally checkable protocols. A protocol p is *locally checkable* with respect to a state predicate I if and only if when the global state of p is in $\neg I$, the local state of some process P_r is in $\neg I$. That is, any global state corresponding to that local state of P_r is in $\neg I$. For example, if the TC protocol has a global state in $\neg I_{color}$ then there exists r such that $(c_{r-1} = c_r)$; this can be checked by P_r. Thus, TC is locally checkable.

Local Correctability. A protocol p is *corrected to a state predicate* I if and only if the global state of p is in I. A protocol p is *locally correctable to a state predicate* I if and only if p can be corrected to I by correcting the local states of the processes of p. For example, if P_r in the TC protocol has a corrupted local state $(c_{r-1} = c_r)$, P_r ensures that $(c_{r-1} \neq c_r)$ and $(c_r \neq c_{r+1})$ by assigning to c_r a value different from both neighboring colors. Thus, TC is locally correctable.

2.1.5 Temporal Logic.

Temporal logic expresses facts/properties about different structures of time. It is a type of modal logic expressing properties of structures where truth of atomic *propositions* vary from one state/world to another. It is used extensively to specify concurrent and/or distributed protocols. We distinguish two major types of temporal logic according to the structure of their timeline: linear time and branching time. We follow Emerson *et al.* [17] in their definitions.

Linear Time Logic (LTL). In LTL, a structure/timeline is a sequence of states; i.e., from every state/world, there is *only one* possible future path. Syntactically, an LTL formula involves only quantifications over future states. In the sequel, we denote by A the set of atomic propositions or statements of interest: for example, $(x_2 > 2)$ is a proposition in the context of programming, where x_2 is an integer variable and 2 is an integer constant.

Linear Time Structure. A model in LTL is a linear time structure $M = \langle S, x, L \rangle$ such that:

- S is a set of *states/worlds*,

- $x : \mathbb{N} \to S$ is an infinite sequence of states and,
- $L : S \to 2^A$ is an assignment/labeling of atomic propositions to states.

Intuitively, we can think of M as a linear *truth* structure or as an infinite sequence of truth assignments to propositions of A.

Syntax of LTL. The set of propositional LTL formulae is the least set generated by the following rules:

- each $p \in A$ is a formula
- If p and q are formulae, then $p \wedge q$ and $\neg p$ are formulae.
- If p is a formula, then pUq "p until q" and Xp "next of p" are formulae.

Formulae like $\Diamond p$ ("eventually p") and $\Box p$ ("always p") are defined as abbreviations for $(\neg p \vee p)Up$ and $\neg\Diamond(\neg p)$, respectively.

Semantics of LTL. By semantics, we designate the relationship between LTL structures/models and LTL formulae. We say that $x \models \Phi$ (x satisfies Φ) if and only if formula Φ is "true" at timeline x. We denote x^i as the timeline starting at $x(i)$. x abbreviates x^0. We recursively define LTL semantics as follows. Let $P \in A$, p and q be LTL formulae.

- $x \models P$ if and only if $P \in L(x(0))$.
- $x \models p \wedge q$ if and only if $x \models p$ and $x \models q$.
- $x \models \neg p$ if and only if it is not the case that $x \models p$.
- $x \models pUq$ if and only if $\exists j(x^j \models q$ and $\forall k < j(x^k \models p))$.
- $x \models Xp$ if and only if $x^1 \models p$.

Example. Let L be defined by the linear sequence $(\{P\}, \{P, Q\}, \emptyset, \{Q\}, \{P\}, \{P\}, \{P\}, \cdots)$. It is not the case that $x^0 \models \Box P$ because $L(x(2)) = \emptyset$ or $P \notin L(x(3))$, while $x^4 \models \Box P$ because $\forall i \geq 4 : P \in L(x(i))$. $\lhd$

Branching Time Logic (BTL). In BTL, a structure/timeline is a tree of states; i.e., from every state/world, there are *multiple* possible future paths. Syntactically, BTL involves two types of quantifiers: state quantifiers as in LTL ($\Box$ and $\Diamond$) over possible future states in a path and path quantifiers over possible future paths. BTL has special cases like Computational Tree Logic (CTL) and (CTL*). In CTL every state quantifier should be preceded by a path quantifier; in CTL* such a restriction does not exist. These syntactical differences between LTL, CTL and CTL* reflect semantical differences with respect to expressive power and complexity of finite-model checking.

Tree-like Time Structure. A model in CTL is a tree-like time structure $M = \langle S, R, L\rangle$ such that:

- S is a set of *states/worlds*,
- $R \subseteq S \times S$ is a binary relation on states and,
- $L : S \to 2^A$ is an assignment/labeling of atomic propositions to states.

We denote by $\hat{R}$ the *unfolding* of R into an out-tree at a root state $r_0 \in S$. Intuitively, $\hat{R}$ augments S with a labeling from $\mathbb{N}$. We recursively define $\hat{S} = \{t \equiv (s,n) \in S \times \mathbb{N} : t = (r_0, 0) \vee (s \neq r_0 \wedge (\exists r \in S : \langle r, s\rangle \in R) \wedge ((r, n-1) \in \hat{S})))\}$. We define the unfolded binary relation $\hat{R} = \{\langle (s_1, x), (s_2, y)\rangle \in \hat{S} \times \hat{S} : y = x + 1 \wedge \langle s_1, s_2\rangle \in R\}$; i.e.; the labeling of $s \in S$ is the length of a path from r_0 to a state s. Hereafter, we refer to the tree-like time structure rooted at state $r_0 \in S$ as $\hat{R}(r_0)$.

Syntax of CTL.* The syntax of CTL* augments the syntax of LTL with two *path quantifiers* $\mathcal{A}$ "for all paths" and $\mathcal{E}$ "there exists a path". To this end, it is necessary to distinguish two types of formulae: *state formulae* and *path formulae*. Any formula is considered as a path formula. Path formulae are preserved under the LTL modal operators/state quantifiers: ($\Box$, $\Diamond$, X, U, other abbreviations) and logical connectives ($\wedge$, $\neg$, other abbreviations). State formulae are either propositions $P \in A$, quantified path formulae using $\mathcal{A}$ or $\mathcal{E}$. State formulae are preserved under logical connectives ($\wedge$, $\neg$, other abbreviations). The set of syntactically correct formulae of CTL* are state formulae.

Example. Let $P, Q \in A$. $(\Diamond\Box\mathcal{E}(P \vee \Box Q))$ is not a syntactically correct CTL* formula, while $(P \implies \mathcal{A}\mathcal{E}P)$ is a CTL* formula. $\lhd$

Semantics of CTL.* In the semantics of CTL*, meaning is assigned to both path and state formulae. A path formula has a model as a path structure M, x where x is a timeline (path). A state formula; i.e, a CTL* formula, has a model as a tree-like structure M, s_0 where $s_0 \in S$. A path x in M is an infinite sequence of states $s_0 s_1 \cdots s_i \cdots$ such that $\forall i \in \mathbb{N} : (s_i, s_{i+1}) \in R$.

Let $P \in A$, p and q be state formulae.

(S1) $M, s_0 \models P$ if and only if $P \in L(s_0)$.

(S2) $M, s_0 \models p \wedge q$ if and only if $M, s_0 \models p$ and $M, s_0 \models q$, $M, s_0 \models \neg p$ if and only if not ($M, s_0 \models p$).

(S3) $M, s_0 \models \mathcal{E}p$ if and only if $\exists$ fullpath $x = s_0 s_1 \cdots$ of $\hat{R}(s_0)$, $M, x \models p$. $M, s_0 \models \mathcal{A}p$ if and only if $\forall$ fullpath $x = s_0 s_1 \cdots$ of $\hat{R}(s_0)$, $M, x \models p$.

(P1) $M, x \models p$ if and only if $M, s_0 \models p$.

(P2) $M, x \models p \wedge q$ if and only if $M, x \models p$ and $M, x \models q$. $M, x \models \neg p$ if and only if not ($M, x \models p$).

(P3) $M, x \models pUq$ if and only if $\exists i : (M, x^i \models q$ and $(\forall k < j(M, x^k \models p)))$. $M, x \models Xp$ if and only if $M, x^1 \models p$.

Example. We illustrate the use of CTL* by a mutual exclusion example $p_{\texttt{mutual}}$. We define the quadruplet of $p_{\texttt{mutual}}$ as follows: $\Phi_p = \{x_1, x_2\}$, $\Pi_p = \{P_1, P_2\}$, $R_1 = R_2 = \Phi_p$, $W_1 = \{x_1\}$,$W_2 = \{x_2\}$, $D_1 = D_2 = \{N, T, C\}$ where N stands for *non-trying section*, T for *trying section* and C for *critical section*. To simplify our explanation, we illustrate Δ_p graphically in Figure 2.1.

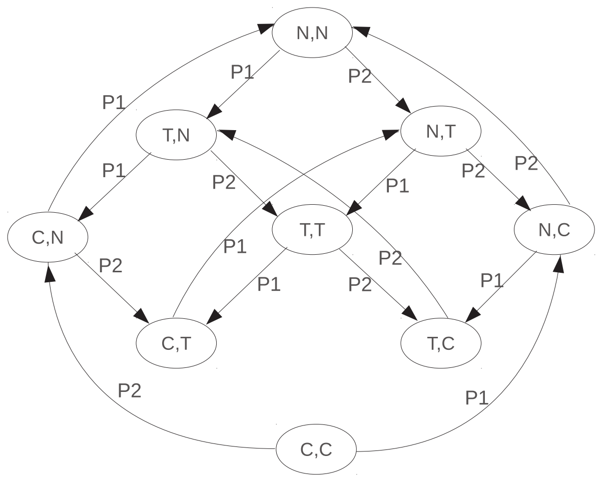


Figure 2.1: Transition Graph of Mutual Exclusion Example

The specification of $p_{\texttt{mutual}}$ is a conjunction of a *mutual exclusion* specification Ψ_m and a *starvation freedom* specification Ψ_s.

Mutual exclusion requires that it is never the case that $x_1 = C \wedge x_2 = C$; i.e, P_1 and P_2 should never be in their critical section simultaneously. Expressing this in CTL*, we write it as $\Psi_m = \mathcal{A}\Box(x_1 \neq C \vee x_2 \neq C)$. By unfolding $\Delta_p(N, N)$ to $\hat{\Delta_p}(N, N)$, we verify that, along every path from (N, N), it is never the case that $x_1 = C \wedge x_2 = C$: (C, C) is an unreachable state from state (N, N). Thus, $\hat{\Delta_p}(N, N) \models \Psi_m$.

Starvation freedom requires that both P_1 and P_2 eventually enter their critical section. In CTL*, we express this requirement as $\Psi_s = \mathcal{A}((x_1 = T \implies \Diamond x_1 = C) \wedge (x_2 = T \implies \Diamond x_2 = C))$. We demonstrate that there exists an infinite path in $\hat{\Delta_p}(N, N)$ along which the statement $x_1 = C$, ($x_2 = C$) never occurs following $x_1 = T$ ($x_2 = T$), respectively. One counterexample path is the non-progress cycle $(T, N), (T, T), (T, C), (T, N), \cdots$ in which P_1 starves ($x_1 = C$ never occurs following $x_1 = T$), violating the requirement of starvation freedom. Thus, it is not the case that $\hat{\Delta_p}(N, N) \models \Psi_s$. $\lhd$

2.2 Formal Problem Statement

Consider a non-stabilizing protocol $p = \langle \Phi_p, \Pi_p, \Delta_p \rangle$ and a state predicate I closed in p. Our objective is to generate a strongly stabilizing version of p, denoted p_{ss}, by *adding* convergence to I. We assume that p is correct as far as its original specification is concerned. Accordingly, we require that the behavior of p_{ss} in the absence of transient faults remains the same as p. With this motivation, during the synthesis of p_{ss} from p, no states (respectively, transitions) are added to or removed from I (respectively, $\Delta_p|I$). This way, p_{ss} behavior is exactly the same as p's behavior inside I. Moreover, if p_{ss} starts in a state outside I, p_{ss} will provide convergence to I.

Problem 2.2.1: Adding Convergence.

- **Input**: (1) a protocol $p = \langle \Phi_p, \Pi_p, \Delta_p \rangle$; (2) a state predicate I such that I is closed in p; and (3) a property of L_s converging, where $L_s \in \{$weakly, strongly$\}$.
- **Output**: A protocol $p_{ss} = \langle \Phi_p, \Pi_p, \Delta_{p_{ss}} \rangle$ such that the following constraints are met: (1) I is unchanged; (2) $\Delta_{p_{ss}}|I = \Delta_p|I$, and (3) p_{ss} is L_s converging to I.

Chapter 3

Lightweight Methods for Automated Design of Convergence[1]

In order to facilitate the design of self-stabilizing protocols, this chapter presents a lightweight method for algorithmic addition of convergence to finite-state non-stabilizing protocols, including non-locally correctable protocols. The proposed method enables the *reuse* of design efforts in the development of different self-stabilizing protocols. Moreover, for the first time (to the best of our knowledge), this chapter presents an algorithmic method for the addition of convergence to symmetric protocols that consist of structurally similar processes. The proposed approach is supported by a software tool that automatically adds convergence to non-stabilizing protocols. We have used the proposed method/tool to automatically generate several self-stabilizing protocols with up to 40 processes (and 3^{40} states) in a few minutes on a regular PC. Surprisingly, our tool has synthesized both protocols that are the same as their manually-designed versions as well as alternative solutions for well-known problems in the literature (e.g., Dijkstra's token ring, maximal matching, graph coloring, agreement and leader election in a ring). Moreover, the proposed method has helped us detect a design flaw in a manually designed self-stabilizing protocol.

3.1 Introduction

This chapter proposes a lightweight formal method for automated addition of convergence to network protocols (including non-locally correctable protocols). The approach is *lightweight* in that we start from an instance of a non-stabilizing protocol p with a fixed

[1]This chapter is an adaptation of our accepted publication in the ACM transactions on autonomous and adaptive systems [18].

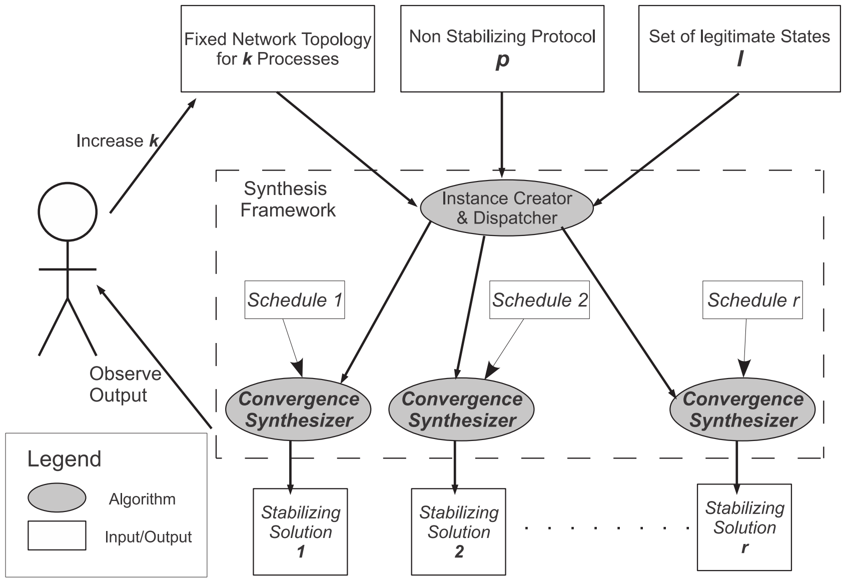

Fixed Network Topology for ***k*** Processes
Non Stabilizing Protocol ***p***
Set of legitimate States ***I***
Increase ***k***
Synthesis Framework
Instance Creator & Dispatcher
Schedule 1
Schedule 2
Schedule r
Observe Output
Convergence Synthesizer
Convergence Synthesizer
Convergence Synthesizer
Legend
Algorithm
Input/Output
Stabilizing Solution ***1***
Stabilizing Solution ***2***
Stabilizing Solution ***r***

Figure 3.1: The proposed lightweight method for automated design of convergence.

and small (i.e., handful) number of processes, denoted k, and add convergence to p for a set of legitimate states I (see Figure 3.1). (We assume that p has a static topology.) Then, we gradually increase the number of processes (or the variables domains) as long as the available computational resources permit us to benefit from automation. There are several advantages to this approach. First, we generate self-stabilizing versions of p that are correct-by-construction, thereby eliminating the need for a proof of correctness. Second, we facilitate the generation of an initial design of self-stabilizing protocols in a fully automatic way. Third, the issue of scalability is no longer a high-priority objective since a lightweight method benefits from available computational resources in order to provide useful insight for developers regarding the challenges of designing convergence when a protocol scales up (in terms of either the number of processes or the size of the domain of variables). Notice that our method does not require that the new processes be similar to existing processes. Fourth, while such a lightweight method can be applied only for the design of small protocols, it is an effective method for finding design flaws in manually-designed protocols (see Section 3.4.1 for an example).

In order to automate the addition of convergence (see the Convergence Synthesizer component in Figure 3.1), we present a method that includes three parts, namely automated design of weak convergence, automated design of a ranking function and automated design of strong convergence (see Figure 3.2). *Weak convergence* ensures that from every illegitimate state in $\neg I$, *there exists* an execution that eventually reaches a legitimate state in I, whereas *strong convergence* guarantees that from any state in $\neg I$, *every* execution eventually reaches a state in I. Observe that, a protocol that has strong convergence also guarantees weak convergence, but the reverse is not necessarily true [14]. We present a sound and complete algorithm that takes p and I and determines if weak convergence can be added to p, thereby generating a *weakly stabilizing* version of p. Our algorithm is complete in that if weak convergence to I can be added to p, then it will generate a weakly stabilizing version of p. Moreover, the generated weakly stabilizing protocol is correct by construction (i.e., soundness). If our algorithm cannot add weak convergence to p, then it means there is no weakly stabilizing version of p for the legitimate states I (see Figure 3.2). As a result, no strongly stabilizing version of p exists either. Thus, our algorithm also provides an impossibility test for the design of strong convergence.

After designing a weakly stabilizing version of the protocol p, denoted p_{ws}, we use p_{ws} to devise a sound heuristic for adding strong convergence to p towards generating a *strongly stabilizing* version of p, denoted p_{ss} (see Figure 3.2). Specifically, since from any state in $\neg I$ the protocol p_{ws} has at least one execution that reaches I, p_{ws} never deadlocks in $\neg I$, where from a *deadlock* state/configuration a protocol cannot execute any actions. Deadlock-freedom is a requirement for strong convergence too. However, strong convergence has another requirement, called *livelock*-freedom, that p_{ws} may fail to meet. Livelock-freedom requires that no execution of p_{ss} stays in $\neg I$ forever. One way to design p_{ss} using p_{ws} is to find a subset of the executions of p_{ws} that start in $\neg I$ and are deadlock-free and livelock-free in $\neg I$. While such a method is complete, it is computationally expensive due to the exponential number of subsets of executions of p_{ws}. To devise an efficient method, we proceed as follows (at the expense of completeness). First, we use the executions of p_{ws} for the algorithmic design of a function that ranks each state $s \in \neg I$ based on the length of the shortest execution of p_{ws} from s that reaches some state in I. Such a ranking method partitions the states in $\neg I$ to *Rank*$[1], \cdots,$ *Rank*$[M]$ ($M > 1$) (see Phase 2 in Figure 3.2) such that *Rank*$[j]$ is a subset of $\neg I$ from where the length of the shortest execution of p_{ws} to I is equal to j, where $1 \leq j \leq M$, and M denotes the total number of ranks, which is a finite value. Notice that the rank of all states in I is zero.

After computing the ranks/partitions of $\neg I$ using the executions of p_{ws}, we no longer need p_{ws}. Then, we eliminate any livelocks that may exists in the executions of the non-stabilizing protocol p in $\neg I$. Thus, at this point, p may have some deadlock states in $\neg I$. To design strong convergence, we systematically construct recovery paths from each deadlock state $s \in$ *Rank*$[j]$ to some state in *Rank*$[j-1]$, for $1 \leq j \leq M$, without creating livelocks (Phase 3 in Figure 3.2). If the inclusion of a recovery action from *Rank*$[j]$ to *Rank*$[j-1]$ results in creating a livelock with the previously included actions, we replace the added recovery action with another one to ensure livelock-freedom in each step. From a specific illegitimate state $s_i \in \neg I$, the success of convergence to some legitimate state $s_l \in I$ also depends on the *order/sequence* of processes that can execute from s_i to get

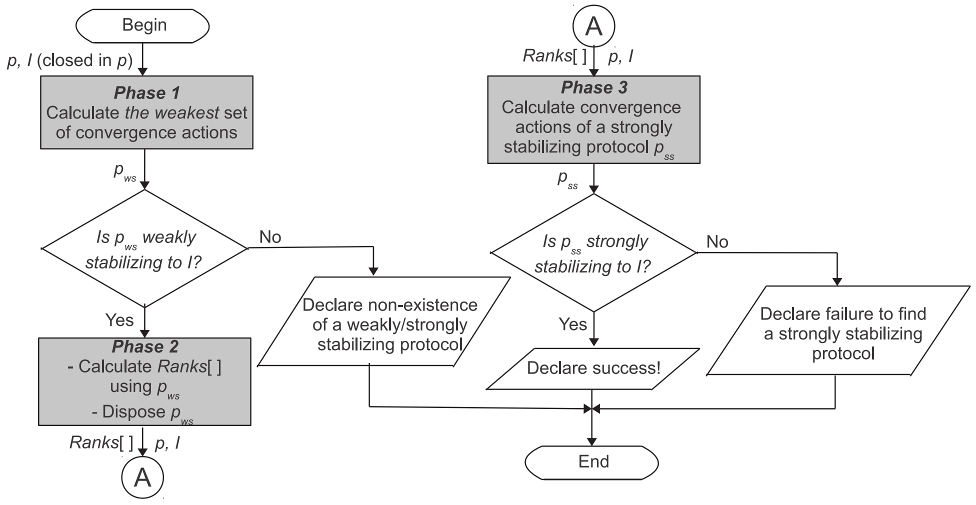

Begin
p, I (closed in p)
Phase 1
Calculate the weakest set of convergence actions
p_{ws}
Is p_{ws} weakly stabilizing to I?
No
Yes
Declare non-existence of a weakly/strongly stabilizing protocol
Phase 2
- Calculate Ranks[ ] using p_{ws}
- Dispose p_{ws}
Ranks[ ] p, I
A
Ranks[ ] p, I
Phase 3
Calculate convergence actions of a strongly stabilizing protocol p_{ss}
p_{ss}
Is p_{ss} strongly stabilizing to I?
No
Yes
Declare success!
Declare failure to find a strongly stabilizing protocol
End

Figure 3.2: Automated design of weak and strong convergence.

the global state of the protocol to s_l, called a *recovery schedule*. From s_i, there may be several recovery schedules that result in executions that reach some legitimate states without creating livelocks. Since during the execution of the heuristic the selected schedule remains unchanged, for each schedule, we can instantiate one instance of our heuristic on a separate machine (see Figure 3.1). If the proposed heuristic succeeds in finding a solution for a specific k and a specific schedule, then the resulting strongly stabilizing protocol p_{ss} is correct-by-construction for k processes; otherwise, we declare failure in designing strong convergence for that instance of the protocol. We also present a version of this heuristic that adds convergence to symmetric protocols that contain structurally similar processes, and maintain the symmetry during synthesis. That is, the synthesized self-stabilizing version of the non-stabilizing protocol is also symmetric. To the best of our knowledge, the proposed method is the first approach that automatically synthesizes symmetric self-stabilizing protocols from their non-locally correctable non-stabilizing versions.

Contributions. In summary, the contributions of this chapter are as follows. We present

- a lightweight formal method (see Figure 3.1) supported by a software tool that automates the generation of initial designs of self-stabilizing network protocols from their non-stabilizing versions;
- a sound and complete algorithm for the addition of weak convergence to non-stabilizing protocols;
- an impossibility test for the addition of weak/strong convergence;
- an algorithm for automated design of a ranking function that provides (i) a base set of recovery steps that should be included in any strongly stabilizing version of p

(i.e., a *necessary condition* for strong convergence), and (ii) a lower bound for all non-increasing ranking functions in terms of the speed of recovery to I (see Lemma 3.2.3 and Theorem 3.2.4);

- a sound heuristic for the addition of strong convergence to asynchronous, non-deterministic and non-locally correctable protocols that can have non-terminating computations, and
- a sound heuristic for adding convergence to *symmetric* protocols while preserving the symmetry of processes.

We have implemented the proposed approach in a software tool called STabilization Synthesizer (STSyn). (STSyn is available at http://c28-0206-01.ad.mtu.edu:8888/SynStable/.) The current implementation of STSyn is in C++ and uses Binary Decision Diagrams (BDDs) [19] to represent protocols in memory. STSyn has synthesized several self-stabilizing protocols (in a few minutes on a regular PC) similar to their manually-designed versions in addition to synthesizing alternative solutions. Thus far, STSyn has automatically generated instances of Dijkstra's token ring protocol [7] (2 different versions) with up to 5 processes, maximal matching on a ring [15] with up to 11 processes, three coloring of a ring with up to 40 processes, a three-ring self-stabilizing protocol with 9 processes, a leader election protocol with 5 processes on a ring and an agreement protocol with up to 6 processes. As far as we know, this is the first time that Dijkstra's self-stabilizing token ring and leader election in a ring are generated automatically. We have also used STSyn to find a livelock in a published self-stabilizing protocol for maximal matching [15] (see Section 3.4.1 for details). This contribution illustrates how our approach can be used for the debugging of manually-designed protocols.

We would like to note that while our focus is on network protocols, the proposed approach in this chapter can be applied to concurrent programs that run on either a single machine or a shared-memory multiprocessor. Moreover, the approach presented in Section 3.5 for automated design of symmetric self-stabilizing protocols can easily be tailored for automated design of self-stabilizing protocols that should converge under certain safety constraints (e.g., super-stabilizing systems [20]). The maximal matching and the coloring examples in this chapter illustrate that our approach can also be used for automated design of equilibrium in multi-agent systems.

Organization. Section 3.2 discusses a method for automated design of weak convergence as an approximation of strong convergence. Section 3.3 presents a sound and efficient heuristic for automated addition of strong convergence. Section 3.4 demonstrates some case studies. Section 3.5 presents a sound and efficient heuristic for the addition of convergence to *symmetric* protocols. Then, Section 3.6 discusses case studies for the heuristic in Section 3.5. Subsequently, Section 3.7 illustrates our experimental results, and Section 3.8 discusses some applications and limitations of the proposed approach. We make concluding remarks and discuss future work in Section 3.9.

3.2 Approximating Strong Convergence

This section presents a sound and complete algorithm for the addition of weak convergence to a network protocol p for a state predicate I that is closed in p (Phase 1 in Figure 3.2). We use the weakly stabilizing version of p (denoted p_{ws}) as an approximation of strong stabilization since p_{ws} provides the weakest set of possible computation prefixes that enable convergence from any state in $\neg I$ to I.

We present the algorithm AddWeak for the addition of weak convergence to p for the state predicate I. Specifically, AddWeak first includes any transition group g that meets the following constraints in the set of transitions of p_{ws} (see Step 1 in Figure 3.3): (1) g adheres to the read/write restrictions of some process $P_j \in \Pi_p$ (i.e., $g \in G(P_j)$), and (2) there is no transition in g that starts in I. This step includes in p_{ws} any transition group that could potentially be useful for strong convergence to I. Then, we check to see if there is a state $s_0 \in \neg I$ from where there is no computation prefix of p_{ws} that reaches I. If there exists such a state, then p_{ws} is not weakly stabilizing. Otherwise, we return p_{ws} as the weakly stabilizing version of the non-stabilizing protocol p.

```
AddWeak($p$: set of transition groups, $I$: state predicate ) {
  - $p_{ws} := p \cup \{g \mid \exists P_j \in \Pi_p : g \in G(P_j) : (\forall (s_0, s_1) : (s_0, s_1) \in g : s_0 \notin I)\}$      (1)
  - $noPrefix = \{s_0 | (s_0 \notin I) \wedge$ there is no computation prefix of $p_{ws}$ that starts in $s_0$ and
                                                                          reaches a state in $I\}$   (2)
  - if ($noPrefix \neq \emptyset$) then declare that no weakly stabilizing version of $p$ exists for $I$; exit;   (3)
  - return $p_{ws}$;                                                                                  (4)
}
```

Figure 3.3: Adding weak convergence.

Theorem 3.2.1 AddWeak is sound and complete, and its time complexity is polynomial in $|S_p|$.

Proof of soundness: We show that when AddWeak successfully returns a protocol p_{ws}, the protocol p_{ws} meets the three constraints of the output of Problem 2.2.1. First, no step of the algorithm AddWeak updates I in any way. Thus, I remains unchanged. Second, Step 1 of AddWeak ensures that no transition group that has a transition starting in I is included in p_{ws}. Thus, $\delta_{p_{ws}}|I = \delta_p|I$. Third, since AddWeak returns p_{ws} in Step 4 of Figure 3.3, we have *noPrefix* $= \emptyset$. As such, from every state $s_0 \in \neg I$, there exists a computation prefix that reaches a state in I. Thus, p_{ws} is weakly stabilizing to I.

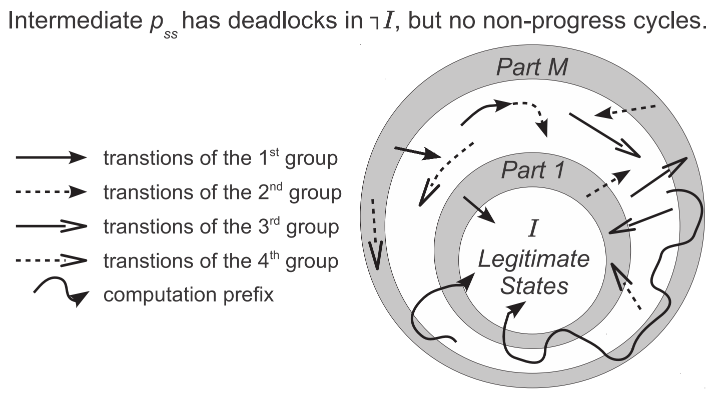
Proof of maximality: We discuss that p_{ws} is maximal. Line (1) of AddWeak includes any transition group g that satisfies the read/write restrictions of p and all transitions of g originate in $\neg I$. Any additional transition group added to p_{ws} would violate the closure of I, modify $\delta_p|I$ or violate the read/write restrictions of some process in Π_p. Therefore, p_{ws} includes a maximal set of transition groups; i.e., p_{ws} is maximal.

Proof of completeness: We demonstrate that if Problem 2.2.1 has a weakly stabilizing solution for a protocol p and a state predicate I, then AddWeak always returns a weakly stabilizing version of p. By contradiction, assume that AddWeak fails to generate a weakly stabilizing version of p. Moreover, assume that a weakly stabilizing version of p that meets the constraints of Problem 2.2.1 exists. Thus, there must be a set of transition groups that adhere to the read/write restrictions of the processes of p, include no transition starting in I, and form some computation prefix from any state in $\neg I$ that reaches a state in I, but AddWeak failed to find such a set of transition groups. This is a contradiction since the output of AddWeak is maximal. Therefore, AddWeak would have found a weakly stabilizing program.

Proof of polynomial-time complexity: It is straightforward to see that AddWeak has a constant number of steps, and every step of AddWeak can be performed in polynomial time in $|S_p|$. $\square$

Notice that if the AddWeak algorithm declares failure in adding weak convergence to p for a state predicate I, then p does not have a strongly stabilizing version either. If AddWeak returns a weakly stabilizing protocol p_{ws}, then there is some computation prefix from each state in $\neg I$; i.e., no state in $\neg I$ is deadlocked. However, p_{ws} may include non-progress cycles in $\neg I$; such non-progress cycles violate the requirements of strong convergence (see Proposition 2.1.1). Thus, one way to design a strongly stabilizing version of p is to find a subset of the transition groups included in p_{ws} (in Step 1 of Figure 3.3) such that no state in $\neg I$ is deadlocked and there is no non-progress cycle. Nonetheless, finding such a subset appears to be computationally expensive. In Section 3.3, we present a sound but incomplete heuristic for the construction of a set of transition groups that should be included in a strongly stabilizing version of p, denoted p_{ss}. The idea behind our heuristic is simple; partition $\neg I$ to a sequence of disjoint predicates around I, denoted $Part_1, \cdots, Part_M$ as depicted in Figure 3.4, and incrementally build the recovery paths from every state of $Part_i$ to $Part_{i-1}$ (for $1 \leq i \leq M$) while ensuring livelock-freedom and deadlock-freedom.

Notice that such a partitioning should be performed in a way that, using the transition groups of each process P_j in p_{ws}, convergence from $Part_i$ to $Part_{i-1}$ can be guaranteed.



Intermediate p_{ss} has deadlocks in $\neg I$, but no non-progress cycles.
Part M
Part 1
I
Legitimate States
transtions of the 1st group
transtions of the 2nd group
transtions of the 3rd group
transtions of the 4th group
computation prefix

Figure 3.4: Partitioning of $\neg I$.

Thus, the way $\neg I$ is partitioned directly affects the effectiveness of the heuristic. To elaborate on this, consider a state $s \in \neg I$ from where single-step recovery to I is impossible using the transition groups of p_{ws}. (The impossibility of such a single-step recovery from s can be due to the write restrictions of processes.) Now, if a partitioning method puts s in *Part*$_1$, then it will be impossible to ensure recovery from all states of *Part*$_1$ to I. Thus, such a partitioning method would not result in an effective method for adding strong convergence. Since from any state in $\neg I$ the weakly stabilizing protocol p_{ws} includes any potential computation prefix that reaches a state in I, it is important for a partitioning method to be consistent with how the computation prefixes of p_{ws} are formed. Towards this end, we present the ComputeRank algorithm that uses p_{ws} to partition $\neg I$ (Phase 2 in Figure 3.2).

```
ComputeRanks($p_{ws}$: set of transition groups, $I$: state predicate ) {
 /* $Rank$ is an array of state predicates. */
 - $explored := I$;    $i := 0$;    $Rank[i] := I$;                                   (1)
 - while ($Rank[i] \neq \emptyset$) {
     - $i := i + 1$;                                                                 (2)
     - $Rank[i] := \{s_0 \mid (s_0 \notin explored) \wedge$
                     $(\exists s_1, g : (s_1 \in explored) \wedge (g \in p_{ws}) : (s_0, s_1) \in g\}$;   (3)
     - $explored := explored \cup Rank[i]$ ;                                          (4)
     }
 - $M := i - 1$; // $M$ denotes the total number of ranks built around $I$            (5)
 - return $Rank[]$, $M$;                                                               (6)
}
```

Figure 3.5: Computing ranks and partitioning the set of illegitimate states $\neg I$.

ComputeRank takes the weakly stabilizing version of p (i.e., p_{ws}) and the state predicate I, and returns an array of state predicates *Rank*$[1], \cdots,$ *Rank*$[M]$, where M is the total number of partitions of $\neg I$. Each $Rank[i] \subseteq \neg I$ includes the set of states s from where the length of the shortest computation prefix of p_{ws} from s to I, called the *rank* of s, is equal to i, for $1 \leq i \leq M$. That is, $Rank[i]$ includes all states with rank i. Note that, for any state $s \in I$, the rank of s is zero. The loop in Figure 3.5 computes the set of backward reachable states from I, denoted *explored*, using the transitions of p_{ws}. In each iteration i, Line 3 calculates a set of states *Rank*$[i]$ outside *explored* from where some state in *explored* can be reached by a single transition of p_{ws}. The loop terminates when no more states can be added to *explored*.

Theorem 3.2.2 The ComputeRanks algorithm terminates in polynomial time in $|S_p|$, and correctly computes the length of the shortest computation prefix of p_{ws} from each state in $\neg I$ to I.

Proof. To illustrate the termination of ComputeRanks, we first make the following observation that before Step 4 of each iteration i of the while loop in ComputeRanks, the intersection of the state predicate *Rank*$[i]$ and the state predicate *explored* is empty. Thus, in each iteration, the size of the *explored* predicate increases in Step 4 of Figure 3.5. Notice that, this occurs because, by definition, from any state in $\neg I$, p_{ws} has at least one computation prefix that reaches I. Since our focus is on finite-state protocols, the *explored* predicate can at most become equal to S_p. It follows that, in some iteration, *Rank*$[i]$ becomes empty; hence termination. In the worst case, the number of iterations of the while loop is equal to the number of states in $\neg I$, where in each iteration the rank of only one state is computed. Further, each step of ComputeRanks can be performed in polynomial time in $|S_p|$. Thus, the time complexity of ComputeRanks is polynomial in $|S_p|$.

To illustrate the correctness of ComputeRanks, we show that the following loop invariant holds in the while loop in Figure 3.5:

Rank$[i]$ includes the set of states from where the shortest computation prefix of p_{ws} to I has length i

- *Initialization.* Before the loop in Figure 3.5 starts, we have *Rank*$[0] = I$, which preserves the loop invariant.

- *Maintenance.* We show that if the loop invariant holds before the i-th iteration of the loop, where $i \geq 0$, then it also holds before the $(i+1)$-th iteration. Thus, we assume that before i-th iteration, *Rank*$[i]$ includes the set of states from where the length of the shortest computation prefix of p_{ws} to I is i. Step 2 increments the loop counter. Thus, the iteration number becomes $i+1$. Before Step 4, the predicate *explored* includes all states from where the length of the shortest computation prefix of p_{ws} to I is at most i. Since Step 3 computes the set of states from where some state in *explored* can be reached by a single transition of p_{ws}, the rank of the states in *Rank*$[i+1]$ is $i+1$. It follows that before iteration $i+1$ the loop invariant holds.

- *Termination.* The condition that causes the termination of the loop is that in some iteration j, *Rank*$[j]$ becomes empty. That is, there are no more states from where *explored* can be reached by the computations of p_{ws}. By the Maintenance property, the loop invariant holds before j. That is, *Rank*$[j-1]$ holds the set of states from where the shortest computation prefix of p_{ws} to I has length $j-1$. Thus, upon termination, the invariant holds for each *Rank*$[i]$, for $1 \leq i \leq j-1$. Notice that, when the loop terminates, the total number of partitions/ranks is $j-1$, where j is the last value assigned to the loop counter i.

The only way ComputeRanks misses to place a state s in a rank is that there is no computation prefix of p_{ws} from s to some state in I. This is impossible since by construction p_{ws} is a weakly stabilizing protocol. □

We would like to note that the ranks *Rank*$[1], \cdots,$ *Rank*$[M]$ have the interesting property that, starting from a state s_0 in some rank *Rank*$[j]$, where $1 \leq j \leq M$, any strongly stabilizing version of p (irrespective of how it has been designed) cannot converge to I in less than j steps. More precisely, starting in $s_0 \in$ *Rank*$[j]$, any strongly stabilizing version of p should go through the ranks *Rank*$[j-1]$, *Rank*$[j-2], \cdots,$ *Rank*$[1]$, *Rank*$[0]$. This property provides (i) a *necessary condition* for strong convergence; i.e., a base set of recovery steps that should be included in any strongly stabilizing version of p, and (ii) a lower bound for all non-increasing ranking functions in terms of the speed of recovery to I. Before we prove this claim, we provide the following definition:

Definition. We call a transition $t = (s_0, s_1)$ *rank decreasing if and only if* $s_0 \in$ *Rank*$[i]$ and $s_1 \in$ *Rank*$[i-1]$ $(0 < i \leq M)$.

Lemma 3.2.3 If p_{ws} is a weakly stabilizing version of a non-stabilizing protocol p for a state predicate I, then p_{ws} excludes any transition (s_0, s_1) that decreases the ranks calculated by ComputeRanks() more than one unit.

Proof. By contradiction, let p_{ws} include a transition (s_0, s_1) such that $s_0 \in$ *Rank*$[i]$ and $s_1 \in$ *Rank*$[j]$, and $i - j > 1$. Then, ComputeRanks() has missed s_0 as a state that is backward reachable from s_1 in a single step by the transition groups of p_{ws}. This contradicts with the correctness of ComputeRanks() (demonstrated in Theorem 3.2.2). □

Theorem 3.2.4 If p_{ss} is a strongly stabilizing version of p that meets the requirements of Problem 2.2.1 for a predicate I that is closed in p, then every computation prefix of p_{ss} that starts in a state $s_0 \in$ *Rank*$[i]$ $(i > 0)$ includes a rank-decreasing transition starting in *Rank*$[j]$ for every j where $0 < j \leq i$.

Proof. First, we recall that the transition groups of p_{ss} form a subset of the transition groups of p_{ws} computed by AddWeak. By assumption, p_{ss} strongly converges to I. Hence, every computation of p_{ss} that starts in *Rank*$[i]$ has a prefix $\sigma = \ll s_0, s_1, \cdots, s_f \gg$ where $s_f \in I$. Based on Lemma 3.2.3, a transition of p_{ws} can at most decrease the rank of a *state* by 1. Hence, to change the rank from i to 0, σ should at least include a transition from *Rank*$[i]$ to *Rank*$[i-1]$, a transition from *Rank*$[i-1]$ to *Rank*$[i-2]$, $\cdots$, and a transition from *Rank*$[1]$ to I. □

The significance of the results of this section is multi-fold. First, Lemma 3.2.3 and Theorem 3.2.4 respectively provide necessary conditions for weak and strong convergence. Second, given a non-stabilizing protocol p and a state predicate I closed in p, Theorem 3.2.4 presents a lower bound on the number of steps required for strong convergence to $\neg I$. Third, while existing methods in the literature [21], [22] use manually-designed (strictly decreasing) ranking functions for the verification of strong convergence, ComputeRanks() provides an *algorithmic* method for assigning a unique rank to each state, thereby creating a ranking function. Fourth, Theorem 3.2.4 implies that, for the design of strong convergence, a ranking function need not necessarily be strictly decreasing. Instead, we need guarantees for (1) having at least j rank-decreasing transitions from each rank j, and (2) ensuring livelock and deadlock-freedom. That is, a strongly stabilizing protocol may fluctuate between ranks before it eventually converges to I. We note that, the ranks computed in this section are different from the convergence stairs [22] (used for *verifying* strong convergence) in that each *Rank*$[i]$ need not be closed in p_{ss}.

Hereafter, we no longer need p_{ws}; rather we use the ranks as a guide for systematic inclusion of transition groups in p_{ss} while guaranteeing livelock-freedom. Next section presents a sound heuristic for the design of strong convergence based on the results of this section.

3.3 Algorithmic Design of Strong Convergence

In this section, we present a sound heuristic for adding strong convergence (Phase 3 in Figure 3.2). The proposed heuristic (in Figure 3.6) incrementally includes recovery transitions (and their associated group), where a *recovery transition* (s_0, s_1) is such that s_0 is a deadlock state; i.e., (s_0, s_1) *resolves* the deadlock state s_0. A *recovery transition group* is a transition group that includes a recovery transition. The recovery transition groups are included under the following constraints:

- (C1) all transitions of a recovery transition group must start in $\neg I$. (Recall that, due to read restrictions, all transitions in a group must be either included or excluded.);
- (C2) a recovery transition group must include a rank-decreasing transition from *Rank*$[i]$ to *Rank*$[i-1]$, for some $1 \leq i \leq M$;

- (C3) no transition of a recovery transition group participates in a non-progress cycle outside I, and

- (C4) a recovery transition group includes no transition that reaches a deadlock state.

The AddStrong heuristic in Figure 3.6 takes a non-stabilizing protocol p, a state predicate I (closed in p), the ranks calculated by the ComputeRanks routine in Section 3.2 and an integer array *schedule* that represents a preferred order based on which processes in Π_p are used for the inclusion of recovery transition groups during the design of strong convergence. An example recovery schedule for the TR protocol is $\{P_1, P_2, P_3, P_0\}$; i.e., *schedule*$[1] = 1$, *schedule*$[2] = 2$, *schedule*$[3] = 3$ and *schedule*$[4] = 0$. That is, when adding recovery from a deadlock state s_d, we first check the ability of P_1 in including a recovery transition group from s_d, then the ability of P_2 and so on. Note that, the ranks are computed if the AddWeak algorithm (presented in Section 3.2) generates a weakly stabilizing version of p. In other words, the AddStrong heuristic is used if there exists a weakly stabilizing version of p. In Step 1 of AddStrong, we remove from p any transition group whose transitions start in $\neg I$. Such groups have no transitions that can take part in any computation of p that starts in I. Later on, during the synthesis of strong convergence, these groups will be considered for designing the convergence of p_{ss} to I. In practice, p rarely includes such transition groups, nonetheless, we must consider removing them to ensure the soundness of our heuristic. If the remaining transition groups of p include transitions that form any non-progress cycles in $\neg I$, then we declare failure in synthesizing p_{ss} and *exit* (Step 3 in Figure 3.6). The reason behind this step is that resolving those cycles requires the elimination of groups having transitions in $\delta_p|I$, which violates the second constraint in the output of Problem 2.2.1. We implement an existing algorithm due to Gentilini *et al.* [23] (see Detect_SCC in Line 2 of AddStrong) for the detection of Strongly Connected Components (SCCs) that are created by transitions of $\delta_p|\neg I$. A SCC is a state transition graph in which every state is reachable from any other state. Thus, a SCC may include multiple cycles. Detect_SCC returns an array of state predicates, denoted *SCCs*, where each array cell contains the states of a SCC. Detect_SCC also returns the number of SCCs. If no cycles exist in $\neg I$, then we initialize p_{ss} by p and move on to the subsequent steps where we incrementally include recovery transition groups in p_{ss} until all deadlocks in $\neg I$ are resolved and the computations of p_{ss} are livelock-free in $\neg I$.

Before including recovery transition groups in p_{ss}, we compute the deadlock states in $\neg I$, denoted *deadlockStates* (see Step 5 in Figure 3.6). To systematically include recovery transition groups in p_{ss}, we sweep the ranks from bottom up and explore the possibility of resolving deadlock states in each *Rank*$[i]$, for i from 1 to M. Each round of sweeping is called a *pass*. We go through three passes by invoking the Pass_Template function (see Figure 3.7) in Lines 6, 7 and 8 of Figure 3.6.

```
AddStrong(p: set of transition groups; I: state predicate;
                    Rank[1], ···, Rank[M]: state predicate; schedule[1..K]: array of integers) {
/* Rank[1], ···, Rank[M] are computed by ComputeRanks. */
/* schedule is an array representing a preferred order based on which processes are used
/* in the inclusion of transition groups for recovery from Rank[i] to Rank[i − 1]. */

  - $p := p - \{g | (g \in p) \wedge (\forall(s_0, s_1) : (s_0, s_1) \in g : s_0 \notin I)\}$; // g denotes a transition group   (1)
  - SCCs, numOfSCCs := Detect_SCC(p, $\neg I$);                                                      (2)
        // SCCs is an array of state predicates in which each array cell includes the states in a
        // Strongly Connected Component (SCC) formed by transitions of $\delta_p | \neg I$.
        // numOfSCCs dentoes the size of the array SCCs.
  - if (numOfSCCs $\neq 0$) then declare failure in adding strong convergence to p; exit;          (3)
  - else $p_{ss} := p$;                                                                              (4)
  - deadlockStates := { $s_0 \mid s_0 \notin I \wedge (\forall s_1, g: (s_0, s_1) \in g: g \notin p_{ss})$};   (5)
  - deadlockStates, $p_{ss}$ := Pass_Template(C1234, deadlockStates, I, Rank[1], ···, Rank[M],
                                                                   $p_{ss}$, schedule[1..K]); (6)
  - deadlockStates, $p_{ss}$ := Pass_Template(C123, deadlockStates, I, Rank[1], ···, Rank[M],
                                                                   $p_{ss}$, schedule[1..K]); (7)
  - deadlockStates, $p_{ss}$ := Pass_Template(C13, deadlockStates, I, Rank[1], ···, Rank[M],
                                                                   $p_{ss}$, schedule[1..K]); (8)
  - if (deadlockStates $\neq \emptyset$) then declare failure in adding strong convergence to p; exit;   (9)
  - else return $p_{ss}$;                                                                           (10)
}
```

Figure 3.6: Proposed heuristic for adding strong convergence.

- *Pass C1234: Adding recovery from Rank*$[i]$ *to Rank*$[i-1]$ *excluding transitions that reach deadlocks.* To build upon any already existing computation prefix that starts in $\neg I$ and reaches a state in I, in this pass, we include only those recovery transition groups whose transitions terminate in a non-deadlock state. Specifically, we iterate through each *Rank*$[i]$ ($0 < i \leq M$) and explore the possibility of including any recovery transition group g in p_{ss} such that (1) g resolves some deadlock state in *Rank*$[i]$ (see the predicate *From* in Line 3 of Figure 3.7), (2) g includes a rank-decreasing transition (i.e., constraint C2), and (3) g meets the constraints (C1), (C3) and (C4). To enforce the constraints (C1) and (C4), we construct the set of transitions *ruledOutTrans* (see Line 5 of Figure 3.7) that includes the transitions that either start in I or reach a deadlock state. Then, to enforce the constraint (C3) in each iteration, we invoke the Add_Convergence algorithm of Figure 3.8 (see Line 7 of Figure 3.7) with *ruledOutTrans* passed to it as an actual parameter. Add_Convergence adds recovery from states in predicate *From* to states of *To*. If all deadlock states are resolved in some iteration, then p_{ss} is a strongly stabilizing protocol that converges to I.

- *Pass C123: Adding recovery from Rank*$[i]$ *to Rank*$[i-1]$ *including transitions that reach deadlocks.* This pass relaxes the constraint (C4) while including recovery transition groups. Specifically, in Lines 2-8 of Figure 3.7, the predicates *From* and *To* are computed in the same way as in Pass C1234, nonetheless, we have *ruledOutTrans*= $\{(s_0, s_1) \mid (s_0 \in I)\}$ (Line 6 in Figure 3.7). That is, we permit the inclusion of recovery groups that include transitions reaching deadlock states (i.e., we relax the constraint (C4)).

```
Pass_Template($passNo$: integer; $deadlockStates$, $I$: state predicate; $p_{ss}$: set of transition groups;
                    $Rank[1], \cdots, Rank[M]$: state predicate; $schedule[1..K]$: array of integers) {
- If ($passNo = $ C1234 $\vee$ $passNo = $ C123), then {                                                    (1)
   - for $i := 1$ to $M$ {          // Go through each rank                                                  (2)
      - $From := \{s \mid s \in Rank[i] \ \wedge \ s \in deadlockStates\}$;                                   (3)
      - $To := \{s \mid s \in Rank[i-1]\}$;                                                                   (4)
      - If ($passNo = $ C1234) then
            - $ruledOutTrans := \{(s_0, s_1) \mid (s_0 \in I) \vee (s_1 \in deadlockStates)\}$;               (5)
      - Else
            - $ruledOutTrans := \{(s_0, s_1) \mid (s_0 \in I)\}$;                                             (6)
      - $deadlockStates$, $p_{ss}$ := Add_Convergence($From$, $To$,
                              $I, p_{ss}$, $ruledOutTrans$, $schedule[1..K]$, $passNo$);                      (7)
      - If ($deadlockStates = \emptyset$) then return $deadlockStates$ , $p_{ss}$;                            (8)
         } // for
   - return $deadlockStates$ , $p_{ss}$;                                                                      (9)
  } // if
- Else if ($passNo = $ C13) then {                                                                           (10)
   - $From := \{s \mid s \in deadlockStates\}$;                                                              (11)
   - $To := true$ ;                                                                                          (12)
   - $ruledOutTrans := \{(s_0, s_1) \mid (s_0 \in I)\}$;                                                     (13)
   - $deadlockStates$, $p_{ss}$ := Add_Convergence($From$, $To$,
                              $I, p_{ss}$, $ruledOutTrans$, $schedule[1..K]$, $passNo$);                     (14)
   - return $deadlockStates$ , $p_{ss}$;                                                                     (15)
        }
   - Else        declare invalid pass number!; exit;                                                         (16)
}
```

Figure 3.7: Pass_Template is invoked three times by the proposed heuristic with different inputs.

- *Pass C13: Adding recovery from any remaining deadlock states to wherever possible.* In Lines 10-15 of Figure 3.7, we explore the feasibility of adding recovery transitions from remaining deadlock states to any state without adhering to the ranking constraint (i.e., relaxing the constraint C2). As such, we invoke Add_Convergence only once with *From* $=$ *deadlockStates*, *To* $= true$ and *ruledOutTrans*$= \{(s_0, s_1) \mid (s_0 \in I)\}$.

While the passes become less restrict in the order we have presented them, developers have the liberty to perform these passes in any order. Specifically, designers have two options in determining the order of executing these passes (out of the six possible permutations/orders): either a specific order is determined using some background knowledge about p, or in a brute-force fashion, one can invoke our heuristic with all possible 6 permutations of steps 6, 7 and 8 in Figure 3.6 to see which order generates a strongly stabilizing version of p. For example, if the input non-stabilizing protocol is empty (i.e., has no transition groups), then starting with Pass C1234 is not a good idea because all states in $\neg I$ are deadlock and the constraint (C4) prohibits the inclusion of any recovery transition group. After the execution of the passes, if there are still some unresolved deadlock states, then AddStrong declares failure and terminates (see Step 9 in Figure 3.6). Otherwise, p_{ss} is returned as the strongly stabilizing version of p.

Adding convergence from a state predicate to another. Depending on the selected pass (the first parameter of Pass_Template), Pass_Template simply prepares the appropriate input parameters of the Add_Convergence routine and then returns the results computed by the Add_Convergence routine in Figure 3.8. Add_Convergence adds recovery transition groups from a state predicate *From* to another state predicate *To*. Such an inclusion of recovery transition groups in the protocol p_{ss} is performed (i) under the read/write restrictions of processes, (ii) without creating non-progress cycles in $\neg I$, (iii) without including any transition group that is ruled out by the constraints of that pass, denoted *ruledOutTrans*, and (iv) based on the recovery schedule given in the array *schedule*[]. We shall invoke Add_Convergence in Passes C1234, C123 and C13 with different values for its input parameters (see Lines 7 and 14 of Pass_Template)).

In each iteration of the for loop in Add_Convergence, we use the routine Add_Recovery to check whether the transition groups that adhere to the read/write restrictions of process $P_{sch[j]}$ can add recovery from $From$ to To. This addition of recovery is performed while excluding any transition in the set of transition groups *ruledOutTrans* (see Line 1 of Add_Recovery in Figure 3.8). Once a recovery transition is added, we need to make sure that its groupmate transitions do not create non-progress cycles with the groupmates of the transitions of p_{ss}. For this reason, we use the Identify_Resolve_Cycles (see Figure 3.8) routine in Line 2 of Add_Recovery.

The Identify_ Resolve_Cycles (see Figure 3.8) routine identifies any SCCs that are created in $\neg I$ due to the inclusion of new recovery transitions in p_{ss}. The for-loop in Line 3 of Identify_ Resolve_Cycles determines a set of groups of transitions *badTrans* that include at least a transition (s_0, s_1) that starts and ends in a SCC; i.e., (s_0, s_1) participates in at least one non-progress cycle. Step 3 in Add_Recovery excludes such groups of transitions from the set of groups of transitions added for recovery. As such, the remaining groups add recovery without creating any cycles.

TR Example. For the TR example introduced in Section 3.2, the state predicate I is equal to S_1 (defined in Section 3.2). ComputeRanks calculates two ranks ($M = 2$) that cover the entire predicate $\neg I$. The non-stabilizing TR protocol does not have any non-progress cycles in $\neg S_1$. The recovery schedule is P_1, P_2, P_3, P_0. We could not add any recovery transitions in Pass C1234 as the groups that do not terminate in deadlock states cause cycles. In Pass C123, we add the recovery action $x_j = x_{j\ominus 1} \oplus 1 \rightarrow x_j := x_{j\ominus 1}$, for $1 \leq j \leq 3$, without introducing any cycles. No new transitions are included in P_0. The union of the added recovery action and the action A_j in the non-stabilizing TR protocol results in the action $x_j \neq x_{j\ominus 1} \rightarrow x_j := x_{j\ominus 1}$ for the domain {0, 1, 2}. Notice that, the synthesized TR protocol

```
Add_Convergence($From$, $To$, $I$: state predicate; $p_{ss}$, $ruledOutTrans$: set of transition groups,
                                    $schedule[1..K]$: integer array; $passNo$: integer)
 /* $schedule$ is an array representing a preferred schedule based on which
 /* processes are used in the design of convergence. */
 /* $G(P_i)$ denotes the set of transition groups that adhere to the read/write restrictions of */
 /* a process $P_i$. Note that not all groups in $G(P_i)$ may be included in the process $P_i$ */
 /* of the stabilizing protocol $p_{ss}$. */

{ - for $j := 1$ to $K$ {
  // use the schedule in array $sch$ for adding recovery
        - $p_{ss}$ := Add_Recovery($From$, $To$, $I, G(P_{sch[j]})$, $p_{ss}$, $ruledOutTrans$);                 (1)
        - $deadlockStates$ := { $s_0 \mid s_0 \notin I \wedge (\forall s_1, g\colon (s_0, s_1) \in g\colon g \notin p_{ss})$};      (2)
        - if ($deadlockStates = \emptyset$) then return $deadlockStates$, $p_{ss}$;                            (3)
        - if ($passNo$ = C1234) then
              $ruledOutTrans := \{(s_0, s_1) \mid (s_0 \in I) \vee (s_1 \in deadlockStates)\}$;                 (4)
          } // for loop
  - return $deadlockStates$, $p_{ss}$;                                                                   (5)
}

Add_Recovery($From$, $To$, $I$: state predicate;$G(P)$, $p_{ss}$, $ruledOutTrans$: set of transition groups)
{ - $addedRecovery$ := { g | (g ∈ $G(P)$) ∧
        (∃ (s0,s1) : (s0,s1) ∈ g ∧ s0 ∈ $From$ ∧ s1 ∈ $To$ ∧ g ∉ $ruledOutTrans$) }                        (1)
  - $badTrans$ := Identify_Resolve_Cycles($p_{ss}$,$addedRecovery$, $\neg I$);                             (2)
  - return ($p_{ss} \cup (addedRecovery - badTrans)$);                                                   (3)
}

Identify_Resolve_Cycles($p_{ss}$, $addedTrans$: set of transition groups; $X$: state predicate)
{ - $badTrans := \emptyset$; // transitions to be removed from cycles.                                  (1)
  - $SCCs$, $numOfSCCs$ := Detect_SCC($p_{ss} \cup addedTrans$, $X$);                                     (2)
   // $SCCs$ is an array of state predicates in which each array cell includes the states in an SCC.
  - for $i := 1$ to $numOfSCCs$ {
      - $groupsInSCC$ := { g | g ∈ $addedTrans$ ∧ (∃ (s0,s1) ∈ g :: s0 ∈ $SCCs[i]$ ∧ s1 ∈ $SCCs[i]$)};(3)
      - $badTrans := badTrans \cup groupsInSCC$;      }                                                  (4)
  - return $badTrans$;                                                                                   (5)
}
```

Figure 3.8: Add convergence from a state predicate *From* to another state predicate *To*.

is the same as Dijkstra's token ring protocol in [7]. ◁

Theorem 3.3.1 AddStrong is sound, and has a polynomial time complexity in $|S_p|$.

Proof. The heuristic AddStrong ensures that no transition originating in I will be included. Moreover, the heuristic only adds new recovery transition groups in p_{ss}. Thus, throughout the execution of the heuristic, I remains unchanged and $\delta_{p_{ss}} \mid I = \delta_p \mid I$. Hence, the first two constraints of Problem 2.2.1 are met.

The only step where the heuristic exits successfully is where it returns p_{ss} when no more deadlock states exist; hence deadlock-freedom. Now, we illustrate the livelock-freedom of p_{ss} in $\neg I$. By contradiction, consider a computation $\sigma = \ll s_0, s_1, \cdots \gg$ that includes a non-progress cycle in $\neg I$. Since the state space of p is finite, there must be some state s_i that is revisited in σ. Nonetheless, *Identify_Resolve_Cycles* routine ensures that no cycles are formed in $\neg I$ every time a recovery action is added by *Add_Convergence*. Thus, the computation σ must include a state in I. Therefore, the returned protocol p_{ss} is strongly converging to I.

It is straightforward to see that Steps 1-5 of AddStrong can be performed in polynomial time in $|S_p|$. In Pass C1234, we iterate through M ranks. In each iteration, we invoke Add_Convergence, which includes a for-loop that iterates K times, where K is the number of processes. The Add_Convergence routine takes at most linear time in $|S_p|$. In the worst case, each rank would include a single state, and hence M would be in the order of $|S_p|$. Pass C123 goes through the same number of iterations. Thus, the time complexity of the first and second passes is at most polynomial in $|S_p|$. In pass C13, we only invoke Add_Convergence once. Therefore, the time complexity of AddStrong is polynomial in $|S_p|$. $\square$

Comment on completeness. AddStrong is incomplete in that for some protocols it may fail to add strong convergence while there exist a strongly stabilizing version of the input non-stabilizing protocol p that meet the constraints of Problem 2.2.1. One reason behind such incompleteness is our cycle resolution method where we eliminate any newly added transition group that has a transition which participates in some cycle. This is not the best way to resolve cycles. Consider a scenario where we add two transition groups each have transitions that participate in the same cycle. Thus, eliminating one of them would resolve the cycle. Nonetheless, our method removes both. Such a conservative cycle resolution method is a cause of incompleteness. Another cause of incompleteness is the greedy approach in which we discard only recently added groups that participate in cycles. A more efficient cycle resolution method considers the possibility of removing the previously included transitions as well; i.e., backtracking, which is beyond the scope of this chapter. We would like to note that while performing the passes in different orders may increase the likelihood of finding a solution, it does not provide a complete method because completeness also depends on the aforementioned factors.

3.4 Case Studies

In this section, we present more case studies for the addition of strong convergence to illustrate the applicability of the proposed heuristic in different settings and to emphasize that the manual design of convergence is error prone. Section 3.4.1 discusses the synthesis of a strongly stabilizing maximal matching protocol, Section 3.4.2 presents a stabilizing three coloring protocol, and Section 3.4.3 presents our synthesis of a three-ring token ring protocol.

3.4.1 Maximal Matching on a Bidirectional Ring

The Maximal Matching (MM) protocol (presented in [15]) has K processes $\{P_0, \cdots, P_{K-1}\}$ located on a ring, where $P_{(i\ominus 1)}$ and $P_{(i\oplus 1)}$ are respectively the left and right neighbors of P_i. The left neighbor of P_0 is P_{K-1} and the right neighbor of P_{K-1} is P_0. Each process P_i has a variable m_i with a domain of three values {left, right, self} representing whether or not P_i points to its left neighbor, right neighbor or itself. Intuitively, two neighbor processes are *matched* if and only if they point to each other. More precisely, process P_i is *matched* with its left neighbor $P_{(i\ominus 1)}$ (respectively, right neighbor $P_{(i\oplus 1)}$) if and only if $m_i =$ left and $m_{(i\ominus 1)} =$ right (respectively, $m_i =$ right and $m_{(i\oplus 1)} =$ left). When P_i is matched with its left (respectively, right) neighbor, we also say that P_i *has a left match* (respectively, *has a right match*). Process P_i points to itself if and only if $m_i =$ self. Each process P_i can read the variables of its left and right neighbors. P_i is also allowed to read and write its own variable m_i. The non-stabilizing protocol is empty; i.e., does not include any transitions. Our objective is to automatically generate a strongly stabilizing protocol that converges to a state in $I_{MM} = \forall i : 0 \leq i \leq K-1 : LC_i$, where LC_i is a local state predicate of process P_i as follows

$$LC_i \equiv (m_i = \text{left} \Rightarrow m_{(i\ominus 1)} = \text{right}) \wedge (m_i = \text{right} \Rightarrow m_{(i\oplus 1)} = \text{left}) \wedge \\ (m_i = \text{self} \Rightarrow (m_{(i\ominus 1)} = \text{left} \ \wedge m_{(i\oplus 1)} = \text{right}\ ))$$

In a state in I_{MM}, each process is in one of these states: (i) matched with its right neighbor, (ii) matched with left neighbor or (iii) points to itself, and its right neighbor points to right and its left neighbor points to left. The protocol MM is silent in I_{MM}. We have automatically synthesized stabilizing MM protocols for $K = 5$ to 11 in a few minutes. Due to space constraints, we present only the actions of P_0 in a synthesized protocol for $K = 5$ (see [24] for the actions of all processes).

$$\begin{array}{ll}
m_4 = \text{left} \wedge m_0 \neq \text{self} \wedge m_1 = \text{right} & \longrightarrow m_0 := \text{self} \\
(m_0 = \text{self} \wedge m_4 = \text{right}) \vee (m_0 \neq \text{left} \wedge m_1 \neq \text{self} \wedge m_4 = \text{right}) & \longrightarrow m_0 := \text{left} \\
(m_0 = \text{self} \wedge m_1 = \text{left}) \vee (m_0 \neq \text{right} \wedge m_1 = \text{left} \wedge m_4 = \text{left}) & \longrightarrow m_0 := \text{right}
\end{array}$$

If the left neighbor of P_0 (i.e., P_4) points to its left and its right neighbor (i.e., P_1) points to its right and P_0 does not point to itself, then it should point to itself. P_0 should point to its left neighbor in two cases: (1) P_0 points to itself and its left neighbor points to right, or (2) P_0 does not point to its left, its right neighbor does not point to itself, and its left neighbor points to right. Likewise, P_0 should point to its right neighbor in two cases: (1) P_0 points to itself and its right neighbor points to left, or (2) P_0 does not point to its right, its right neighbor points to left and its left neighbor points to left. These actions are different from the actions in the manually design MM protocol presented by Gouda and Acharya [15] as follows ($1 \leq i \leq K$):

$$\begin{array}{lcl} m_i = \text{left} \wedge m_{(i\ominus 1)} = \text{left} & \longrightarrow & m_i := \text{self} \\ m_i = \text{right} \wedge m_{(i\oplus 1)} = \text{right} & \longrightarrow & m_i := \text{self} \\ m_i = \text{self} \wedge m_{(i\ominus 1)} \neq \text{left} & \longrightarrow & m_i := \text{left} \\ m_i = \text{self} \wedge m_{(i\oplus 1)} \neq \text{right} & \longrightarrow & m_i := \text{right} \end{array}$$

Observe that the actions of processes in Gouda and Acharya's protocol are *symmetric*, whereas in our synthesized protocol they are not. In a symmetric protocol, the actions of each process can be obtained from the actions of another process by a re-indexing (renaming) of variables. This difference motivated us to investigate the causes of such differences. While analyzing Gouda and Acharya's protocol, we found out that their protocol includes a non-progress cycle starting from the state $\langle left, self, left, self, left\rangle$ with a schedule P_0, P_1, P_2, P_3, P_4 repeated twice, where the tuple $\langle m_0, m_1, m_2, m_3, m_4\rangle$ denotes a state of the MM protocol. This experiment illustrates how difficult the design of strongly convergent protocols is and how automated design can facilitate the design of convergence.

3.4.2 Three Coloring

In addition to the TC protocol presented in Section 2.1, STSyn synthesized an alternative strongly stabilizing `TC` protocol with 40 processes with the following actions labeled by process numbers ($1 < i \leq 40$).

$$\begin{array}{lcl} P_1\text{:}\ \ (c_1 = c_0) \vee (c_1 = c_2) & \longrightarrow & c_1 := \text{other}(c_0, c_2) \\ P_i\text{:}\ \ (c_{(i\ominus 1)} \neq c_i) \wedge (c_i = c_{(i\oplus 1)}) & \longrightarrow & c_i := \text{other}(c_{(i\ominus 1)}, c_{(i\oplus 1)}) \end{array}$$

Notice that P_0 has no actions. Moreover, this protocol is different from the TC protocol presented in [15].

3.4.3 Three-Ring Token Ring

In order to illustrate that our approach is applicable for more complicated topologies, in this section, we demonstrate how we added convergence to an extended version of Dijkstra's token ring.

The non-stabilizing Three-Ring Token Ring (TR^2) protocol. The TR^2 protocol includes 9 processes located in three rings A, B and C (see Figure 3.9). In Figure 3.9, the arrows show the direction of token passing. Process *PA*$_i$ (respectively, *PB*$_i$ and *PC*$_i$), $0 \leq i \leq 1$, is the predecessor of *PA*$_{i\oplus 1}$ (respectively, *PB*$_{i\oplus 1}$ and *PC*$_{i\oplus 1}$). Process *PA*$_2$ (respectively, *PB*$_2$ and *PC*$_2$) is the predecessor of *PA*$_0$ (respectively, *PB*$_0$ and *PC*$_0$). Each process *PA*$_i$ (respectively, *PB*$_i$ and *PC*$_i$), $0 \leq i \leq 2$, has an integer variable a_i (respectively, b_i and c_i) with the domain {0, 1, 2}.

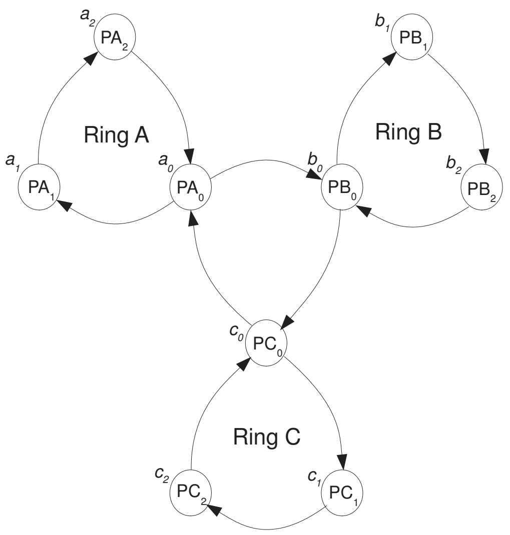


Figure 3.9: The Three-Ring Token Ring (TR^2) protocol.

Process *PA*$_i$, for $1 \leq i \leq 2$, *has the token* if and only if $(a_{i\ominus 1} = a_i \oplus 1)$. Intuitively, *PA*$_i$ has the token if and only if a_i is one unit less $a_{i\ominus 1}$. Process *PA*$_0$ has the token if and only if $(a_0 = a_2) \wedge (c_0 = c_2) \wedge (a_0 = c_0)$; i.e., *PA*$_0$ has the same value as its predecessor and that value is equal to the values held by *PC*$_0$ and *PC*$_2$. Process *PB*$_0$ has the token if and only if $(b_0 = b_2) \wedge (a_0 = a_2) \wedge ((b_0 \oplus 1) = a_0)$. That is, *PB*$_0$ has the same value as its predecessor and that value is one unit less than the values held by *PA*$_0$ and *PA*$_2$. Process *PC*$_0$ has the token if and only if $(c_0 = c_2) \wedge (c_0 \oplus 1 = b_0) \wedge (b_0 = b_2)$. That is, *PC*$_0$ has the same value as its predecessor and that value is one unit less than the values held by *PB*$_0$ and *PB*$_2$. Process *PB*$_i$ (respectively, *PC*$_i$) $(1 \leq i \leq 2)$ *has the token* if and only if $(b_{i\ominus 1} = b_i \oplus 1)$ (respectively, $c_{i\ominus 1} = c_i \oplus 1$). The TR^2 protocol also has a variable $turn \in \{0, 1, 2\}$; ring A executes only if *turn* $= 0$, ring B executes if *turn*$= 1$ and ring C executes if *turn*$= 2$.

Using the following actions, the non-stabilizing TR^2 circulates the token in rings A, B and C ($i = 1, 2$):

$$
\begin{array}{lll}
AC_0: \ (a_0 = a_2) \ \wedge \ turn = 0 & \longrightarrow & \text{if } (a_0 = c_0) \ \ a_0 := a_2 \oplus 1; \\
 & & \text{else} \qquad\quad turn := 1; \\
AC_i: \ (a_{i\ominus 1} = a_i \oplus 1) & \longrightarrow & a_i := a_{i\ominus 1};
\end{array}
$$

Notice that the action AC_i is a parameterized action for processes *PA*$_1$ and *PA*$_2$. The actions of the processes in ring B are as follows ($i = 1, 2$):

$$
\begin{array}{lll}
BC_0: \ (b_0 = b_2) \ \wedge \ turn = 1 & \longrightarrow & \text{if } (a_0 \neq b_0) \ \ b_0 := b_2 \oplus 1; \\
 & & \text{else} \qquad\quad turn := 2; \\
BC_i: \ (b_{i\ominus 1} = b_i \oplus 1) & \longrightarrow & b_i := b_{i\ominus 1};
\end{array}
$$

The actions of the processes in ring C are as follows ($i = 1, 2$):

$$
\begin{array}{lll}
CC_0: \ (c_0 = c_2) \ \wedge \ turn = 2 & \longrightarrow & \text{if } (c_0 \neq b_0) \ \ c_0 := c_2 \oplus 1; \\
 & & \text{else} \qquad\quad turn := 0; \\
CC_i: \ (c_{i\ominus 1} = c_i \oplus 1) & \longrightarrow & c_i := c_{i\ominus 1};
\end{array}
$$

Set of legitimate states I. Consider a state s_0 where $(\forall i : 0 \leq i \leq 2 : (a_i = 0) \wedge (b_i = 0) \wedge (c_i = 0))$ and $turn = 0$ in s_0. The predicate I contains all the states that are reached from s_0 by the execution of actions AC_i, BC_i and CC_i, for $0 \leq i \leq 2$. Starting from the state s_0, process *PA*$_0$ has the token and starts circulating the token until the protocol reaches the state s_1, where $(turn(s_1) = 1) \wedge (\forall i : 0 \leq i \leq 2 : (a_i(s_1) = 1) \wedge (b_i(s_1) = 0) \wedge (c_i(s_1) = 0))$; i.e., *PB*$_0$ has the token. Process *PB*$_0$ circulates the token until the protocol reaches a state s_2, where $(turn(s_2) = 2) \wedge (\forall i : 0 \leq i \leq 2 : (a_i(s_2) = 1) \wedge (b_i(s_2) = 1) \wedge (c_i(s_2) = 0))$; i.e., process *PC*$_0$ has the token. This way the token circulation continues in the three rings. In other words, the predicate I includes all states where there is *exactly one* token in the rings. Thus, $I = I_A \ \wedge \ I_B \ \wedge \ I_C$ where:

$$
\begin{aligned}
I_A = \{s \mid\ & (\forall i : 0 \leq i \leq 2 : a_i(s) = a_{i\oplus 1}(s)) \vee \\
& ((turn(s) = 0)\ \wedge\ (\exists j : 1 \leq j \leq 2 : (a_{j\ominus 1}(s) = a_j(s) \oplus 1) \wedge \\
& \quad (\forall k : 0 \leq k < j-1 : a_k(s) = a_{k\oplus 1}(s)) \wedge \\
& \quad (\forall k : j \leq k < 2 : a_k(s) = a_{k\oplus 1}(s)))\ \}
\end{aligned}
$$

$$
\begin{aligned}
I_B = \{s \mid\ & (\forall i : 0 \leq i \leq 2 : b_i(s) = b_{i\oplus 1}(s)) \vee \\
& ((turn(s) = 1)\ \wedge\ (\exists j : 1 \leq j \leq 2 : (b_{j\ominus 1}(s) = b_j(s) \oplus 1) \wedge \\
& \quad (\forall k : 0 \leq k < j-1 : b_k(s) = b_{k\oplus 1}(s)) \wedge \\
& \quad (\forall k : j \leq k < 2 : b_k(s) = b_{k\oplus 1}(s)))\ \}
\end{aligned}
$$

$$
\begin{aligned}
I_C = \{s \mid\ & (\forall i : 0 \leq i \leq 2 : c_i(s) = c_{i\oplus 1}(s)) \vee \\
& ((turn(s) = 2)\ \wedge\ (\exists j : 1 \leq j \leq 2 : (c_{j\ominus 1}(s) = c_j(s) \oplus 1) \wedge \\
& \quad (\forall k : 0 \leq k < j-1 : c_k(s) = c_{k\oplus 1}(s)) \wedge \\
& \quad (\forall k : j \leq k < 2 : c_k(s) = c_{k\oplus 1}(s)))\ \}
\end{aligned}
$$

The state predicate I_A (respectively, I_B and I_C) includes the states in which either all a (respectively, b and c) values are equal or it is the turn of ring A (respectively, B and C) and there is only one token in ring A (respectively, B and C).

Adding Convergence to TR^2 Protocol. In the absence of transient faults, there is exactly one token in the set of legitimate states I of TR^2. However, transient faults may set the variables to arbitrary values from their domains and either create multiple tokens in the rings, or perturb the state of TR^2 to a deadlock state outside I. Let $\langle turn, a_0, a_1, a_2, b_0, b_1, b_2, c_0, c_1, c_2\rangle$ denote the global state of the protocol TR^2. Then, no process has the token and no action is enabled in the state $s_d = \langle turn, 1, 1, 2, 1, 1, 2, 1, 1, 2\rangle$, where $turn$ could take any value from its domain. Thus, s_d is a global deadlock that can be reached by transient faults. STSyn generates a strongly self-stabilizing version of this protocol that ensures recovery to states where only one token exists in the rings. The actions AC$_i$, BC$_i$ and CC$_i$ ($1 \leq i \leq 2$) are modified in the revised protocol.

```
$AC_{00}$ :  $(a_0 = a_2)\ \wedge\ turn = 0$    $\longrightarrow$   if $(a_0 = c_0)$   $a_0 := a_2 \oplus 1$;
                                                       else             $turn := 1$;
$AC_{10}$ :  $(a_0 = a_1 \oplus 1)$             $\longrightarrow$   $a_1 := a_0$;
$AC_{11}$ :  $(a_1 = a_0 \oplus 1)$             $\longrightarrow$   $a_1 := a_0 \ominus 1$;
$AC_{20}$ :  $(a_1 \neq a_2)$                   $\longrightarrow$   $a_2 := a_1$;
```

$$\begin{array}{llcll}
BC_{00}: & (b_0 = b_2)\ \wedge\ turn = 1 & \longrightarrow & \text{if } (b_0 = a_0) & b_0 := b_2 \oplus 1; \\
 & & & \text{else} & turn := 2; \\
BC_{10}: & (b_0 = b_1 \oplus 1) & \longrightarrow & b_1 := b_0; & \\
BC_{11}: & (b_1 = b_0 \oplus 1) & \longrightarrow & b_1 := b_0 \ominus 1; & \\
BC_{20}: & (b_1 \neq b_2) & \longrightarrow & b_2 := b_1; &
\end{array}$$

$$\begin{array}{llcll}
CC_{00}: & (c_0 = c_2)\ \wedge\ turn = 2 & \longrightarrow & \text{if } (c_0 = b_0) & c_0 := c_2 \oplus 1; \\
 & & & \text{else} & turn := 0; \\
CC_{10}: & (c_0 = c_1 \oplus 1) & \longrightarrow & c_1 := c_0; & \\
CC_{11}: & (c_1 = c_0 \oplus 1) & \longrightarrow & c_1 := c_0 \ominus 1; & \\
CC_{20}: & (c_1 \neq c_2) & \longrightarrow & c_2 := c_1; &
\end{array}$$

Notice that the action AC_{20} for *PA*$_2$ is the union of the existing action $AC_2 : (a_1 = a_2 \oplus 1) \longrightarrow a_2 := a_1$ and the new convergence action $(a_2 = a_1 \oplus 1) \longrightarrow a_2 := a_1$ added to *PA*$_2$. (the new action is the union of these actions because the variable domain is $\{0, 1, 2\}$.) The same is true for actions BC_{20} and CC_{20} for *PB*$_2$ and *PC*$_2$. STSyn did not include an action $AC_{21} : (a_2 = a_1 \oplus 1) \longrightarrow a_2 := a_1 \ominus 1$ in process *PA*$_2$ (that is symmetric to AC_{11} in process *PA*$_1$) because it would have created a livelock as follows. Starting from the state $\langle 0, 0, 2, 0, 0, 2, 0, 0, 2, 0\rangle$, the following actions are executed in ring A: the if part of action AC_{00}, AC_{21}, AC_{11} and the else part of AC_{00}. By these actions, ring A goes through the states $\langle 0, 2, 0\rangle$, $\langle 1, 2, 0\rangle$, $\langle 1, 2, 1\rangle$ and $\langle 1, 0, 1\rangle$, where the triplet $\langle a_0, a_1, a_2\rangle$ denotes the state of ring A. Subsequently, rings B and C go through similar state transitions until the entire protocol reaches the global state $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0, 1\rangle$. In the next round of token passing, each ring transitions through the following states $\langle 1, 0, 1\rangle$, $\langle 2, 0, 1\rangle$, $\langle 2, 0, 2\rangle$, $\langle 2, 1, 2\rangle$, thus reaching the global state $\langle 0, 2, 1, 2, 2, 1, 2, 2, 1, 2\rangle$. In the third round, each ring creates the sequence of states $\langle 2, 1, 2\rangle$, $\langle 0, 1, 2\rangle$, $\langle 0, 1, 0\rangle$, $\langle 0, 2, 0\rangle$, thereby getting back to the global state $\langle 0, 0, 2, 0, 0, 2, 0, 0, 2, 0\rangle$. Since none of the states of the above cycle is a legitimate state, it forms a livelock in the set of illegitimate states. That is why the action AC_{20} (respectively, BC_{20} and CC_{20}) in process *PA*$_2$ (respectively, *PB*$_2$ and *PC*$_2$) is structurally different from the actions of process *PA*$_1$ (respectively, *PB*$_1$ and *PC*$_1$).

3.5 Algorithmic Design of Strong Convergence in Symmetric Protocols

In this section, we investigate the addition of strong convergence to symmetric non-stabilizing protocols towards synthesizing a symmetric self-stabilizing version thereof.

A protocol p is *symmetric* if and only if for every pair of the processes P_i and P_j of p, the code of P_j can be obtained from P_i by a simple renaming (re-indexing), and vice versa [25]. This suggests that, for every pair of processes P_i and P_j, δ_j and δ_i are identical (*similar*) up to a re-indexing of processes. In this section, we present a modification of our heuristic in Section 3.3 to account for symmetric protocols by assuming that the input protocol p and the output protocol p_{ss} both consist of similar processes. We also demonstrate the soundness and polynomial-time complexity of our heuristic.

Add_Convegence_Sym is the core of each pass where similarity between processes is enforced (see Figure 3.10). It adds recovery groups from a state predicate *From* to another state predicate *To*. These recovery groups are included in p_{ss} such that no cycles are created in $\neg I$ and any groups ruled out by the constraints of the current pass, denoted *ruledOutTrans* (Section 3.3), are excluded. We invoke a revised version of Pass_Template (Figure 3.7) for passes C1234, C123 and C13, where Add_Convergence is replaced with Add_Convergence_Sym in Lines 7 and 14 of Figure 3.7.

Add_Convergence_Sym starts by initializing p_{ss} to the current protocol transitions; it invokes Substitute that takes the transition groups of a representative process P_r and substitutes its variables with P_j variables, for every j (Line 1). In this way, Substitute enforces symmetry through variable substitution in a representative set of transition groups. A candidate set of recovery transition groups for the representative process P_r is computed in Line 2 (*RecoveryGroups*$_r$). We use a subscript r for a set of groups and/or states to denote a representative set. Line 3 updates p_{ss} based on the new value of *RecoveryGroups*$_r$ by invoking UpdateGroups(). Update_Groups computes the corresponding recovery groups for each process P_j ($1 \leq j \leq K$). Line 4 computes the SCCs and their number in $\delta_{p_{ss}}|\neg I$. In case there are cycles, Lines 5-7 update p_{ss} to remove potential newly added groups causing these cycles. Line 8 updates the set of deadlock states.

Note. The predicate *ruledOutTrans* allows us to extend our approach by defining some *Safety* requirements. For instance, it is possible to define a set of *Bad* groups for every process P_i, say *Safety*$_i$, and rule them out by augmenting (disjuncting) *ruledOutTrans* with *Safety*$_i$ for every process P_i. This approach allows our heuristic to synthesize *superstabilizing* protocols. A superstabilizing protocol is a self-stabilizing protocol that maintains a safety property during its convergence to its legitimate set of states [20]. The safety predicate is usually a property of the network topology that should be maintained during convergence. We use such safety constraints to guide our heuristic in synthesizing two of the case studies (Section 3.6). **Theorem 3.5.1** The heuristic Add_Convergence_Sym

is sound, and has a polynomial time complexity in $|S_p|$.

Proof We demonstrate that (1) p_{ss} is cycle free outside I, (2) p_{ss} is a symmetric protocol and, (3) if the heuristic returns successfully, p_{ss} is deadlock free. We show that Add_Convergence_Sym maintains (1) and (2) if its input set of processes is symmetric

```
Add_Convergence_Sym($From$, $To$, $I$: state predicate; $P_r$, $ruledOutTrans$: set of transition groups)
{
  - $p := \emptyset$;     for j := 1 to K     { $p := p \cup Substitute(P_r, Variables_j)$; }     $p_{ss} := p$ ;     (1)
   // $P_r$ is a representative process of the symmetric protocol.
   // $Substitute$ replaces the variables of its first parameter with the variables of process $j$,
   // denoted $Variables_j$, thereby generating a set of transition groups of $P_j$ that
   // are isomporphic to a set of groups of $P_r$.
  - $RecoveryGroups_r$ :=Add_Recovery_Sym($From$, $To$, $I$, $G(P_r)$, $ruledOutTrans$);     (2)
  - $p_{ss}$:= Update_Groups($P_r \cup RecoveryGroups_r$);     (3)
  - $SCCs$, $numOfSCCs$ := Detect_SCC($p_{ss}$, $\neg I$);     (4)
  -if ($numOfSCCs \neq 0$) {
      - $BadGroups_r$ :=Identify_Resolve_SCC($p_{ss}$, $p_{ss}$, $\neg I$);     (5)
      - $RecoveryGroups_r$ :=$RecoveryGroups_r - BadGroups_r$;     (6)
      - $p_{ss}$ := Update_Groups($P_r \cup RecoveryGroups_r$);     (7)
      }
  - $deadlockStates$ := { $s_0 \mid s_0 \notin I \wedge (\forall s_1, g: (s_0, s_1) \in g: g \notin p_{ss})$};     (8)
  - return $deadlockStates$, $p_{ss}$;     (9)
}

Add_Recovery_Sym($From$, $To$, $I$: state predicate; $G(P)$, $ruledOutTrans$: set of transition groups)
{
  - $addedRecovery$:= { g | (g $\in G(P)$) $\wedge$
                  ($\exists$ ($s_0$,$s_1$) : ($s_0$,$s_1$) $\in$ g $\wedge$ $s_0 \in From \wedge$ $s_1 \in To \wedge$ g $\notin ruledOutTrans$) }     (1)
  - return $addedRecovery$;     (2)
}

Update_Groups($P_r$: set of transition groups)
{
  - $p_{ss} := \emptyset$;     for j := 1 to K     $p_{ss} := p_{ss} \cup Substitute(P_r, Variables_j)$;     (1)
  - return $p_{ss}$;     (2)
}
```

Figure 3.10: Convergence synthesis for symmetric protocols

and cycle free outside I. In fact, Add_Convergence_Sym adds a set of recovery groups to an intermediate symmetric protocol (represented by P_r) that are similar and cycle free. Lines 5-7 compute and remove the groups causing cycles in p_{ss}. If no groups are added to p_{ss}, it remains unchanged and its cycle freedom is established by previous calls to Add_Convergence_Sym in previous passes and/or ranks. Thus, cycle-freedom is maintained. The procedure Update_Groups maintains symmetry by adding the same recovery group (up to variables substitution) to each process P_j where P_j's are all similar (Line 1). Initially, the input protocol p is symmetric and cycle free, and hence iterative invocations of Add_Convergence_Sym maintains cycle-freedom and symmetry during each iteration of the heuristic. If the invocation of Pass_Template for the passes C123, C1234 and C13 returns a valid protocol where *deadlockStates* $= \emptyset$, then deadlock-freedom in $\neg I$ is guaranteed (see Lines 8, 9 and 15 in Figure 3.7). This proves that p_{ss} is (1) cycle free outside I, (2) symmetric and (3) deadlock free outside I. Therefore, p_{ss} is a symmetric protocol strongly converging to I.

The proof of polynomial-time complexity is similar to the proof of polynomial-time complexity of the heuristic presented in Section 3.3, hence omitted. □

MM Example. In this section, we illustrate how a symmetric version of the MM protocol can be synthesized by our heuristic for $K = 9$. Since the input protocol for MM is empty, Pass C1234 is likely to terminate without including any recovery groups (Recall that Pass C1234 adds only groups having no transitions terminating in deadlock states). Accordingly, we let our heuristic follow a modified order for the passes: Pass C123, Pass C1234 and then Pass C13.

For Pass C123, our heuristic adds a set of recovery groups represented as the following actions. These actions have transitions starting in Rank[1] and reaching I_{MM} for each P_i ($0 \leq i \leq K-1$):

A_{11}: $(m_i \neq \text{self}) \wedge (m_{i\ominus 1} = \text{left}) \wedge (m_{i\oplus 1} = \text{right}) \longrightarrow m_i := \text{self}$
A_{12}: $(m_i \neq \text{left}) \wedge (m_{i\ominus 1} = \text{right}) \wedge (m_{i\oplus 1} \neq \text{left}) \longrightarrow m_i := \text{left}$
A_{13}: $(m_i \neq \text{right}) \wedge (m_{i\ominus 1} \neq \text{right}) \wedge (m_{i\oplus 1} = \text{left}) \longrightarrow m_i := \text{right}$

These added actions already resolve half of the deadlock states in $\neg I_{MM}$. None of them interfere and cause non-progress cycles in $\neg I$. An example of a remaining deadlock state is one in which m_i =self, for each P_i ($0 \leq i \leq K-1$). Continuing in Pass C123, our heuristic adds a set of recovery groups represented as the following actions (A_{21} and A_{22}). These actions have transitions starting in Rank[2] and reaching a state in Rank[1] for each P_i ($0 \leq i \leq K-1$):

A_{21}: $(m_i = \text{self}) \wedge (m_{i\ominus 1} = \text{self}) \wedge (m_{i\oplus 1} = \text{self}) \longrightarrow m_i := \text{left} \mid \text{right}$
A_{22}: $(m_i = \text{self}) \wedge (m_{i\ominus 1} = \text{right}) \wedge (m_{i\oplus 1} = \text{left}) \longrightarrow m_i := \text{left} \mid \text{right}$

In Pass C123, we examine the addition of the actions A_{31} and A_{32} (see below). However, since they create a non-progress cycle for each process P_i between the local states $\langle$ self, self, right $\rangle$ and $\langle$ self, left, right $\rangle$, we exclude them to be in the synthesized protocol. (The local states represent valuations of $\langle m_{i\ominus 1}, m_i, m_{i\oplus 1}\rangle$.) The notation $m_i :=$ left | right signifies a nondeterministic assignment to m_i of one of the values on the right hand side separated by |.

A_{31}: $(m_i = \text{left}) \wedge (m_{i\ominus 1} = \text{self}) \wedge (m_{i\oplus 1} = \text{right}) \vee$
$(m_i = \text{right}) \wedge (m_{i\ominus 1} = \text{left}) \wedge (m_{i\oplus 1} = \text{self}) \longrightarrow m_i := \text{self}$
A_{32}: $(m_i = \text{self}) \wedge (m_{i\ominus 1} = \text{self}) \wedge (m_{i\oplus 1} = \text{right}) \vee$
$(m_i = \text{self}) \wedge (m_{i\ominus 1} = \text{left}) \wedge (m_{i\oplus 1} = \text{self}) \longrightarrow m_i := \text{left} \mid \text{right}$

The actions of A_{41} and A_{42} have at least one transition starting in Rank[3] and going to Rank[2]. They are included in the intermediate program.

A_{41}: $(m_i = \text{right}) \wedge (m_{i\ominus 1} = \text{self}) \wedge (m_{i\oplus 1} = \text{right}) \longrightarrow m_i := \text{self} \mid \text{left}$
A_{42}: $(m_i = \text{left}) \wedge (m_{i\ominus 1} = \text{left}) \wedge (m_{i\oplus 1} = \text{self}) \longrightarrow m_i := \text{self} \mid \text{right}$

For higher ranks, there are no more transition groups to add in Pass C123. Nonetheless, there are still deadlock states in $\neg I_{MM}$. For example, the global state ⟨self, self, right, left, self, right, left, right, left ⟩ is a deadlock state in $\neg I_{MM}$ because it has two neighboring nodes having the value self. Note that by symmetry, all cyclic permutations of this deadlock state are deadlocks as well. In Pass C1234, our heuristic adds the actions A_{51} and A_{52} to p_{ss}. These actions include transitions from Rank[2] to Rank[1] that resolve the remaining deadlock states.

A_{51}: $(m_i = \text{self}) \wedge (m_{i\ominus 1} = \text{self}) \wedge (m_{i\oplus 1} = \text{right}) \longrightarrow m_i := \text{left}$
A_{52}: $(m_i = \text{self}) \wedge (m_{i\ominus 1} = \text{left}) \wedge (m_{i\oplus 1} = \text{self}) \longrightarrow m_i := \text{right}$

No more deadlock states remain in $\neg I_{MM}$ and the obtained solution is the union of all the included actions ($A_{11}, A_{12}, A_{13}, A_{21}, A_{22}, A_{41}, A_{42}, A_{51}, A_{52}$). To gain more confidence in the implementation of STSyn, we model-checked the above solution for $5 \leq K \leq 10$. (The corresponding Promela [26] models are available at http://cs.mtu.edu/~anfaraha/CaseStudy/PromelaCode.) STSyn synthesized a different stabilizing MM version for $K = 6$ and $K = 8$, which we model-checked for $5 \leq K \leq 10$.

$$m_{i\ominus 1} = \text{left} \wedge m_i \neq \text{self} \wedge m_{i\oplus 1} = \text{right} \longrightarrow m_i := \text{self}$$

$$(m_i = \text{self}) \wedge (m_{i\ominus 1} = \text{right} \vee m_{i\oplus 1} = \text{self}) \longrightarrow m_i := \text{left}$$
$$(m_i = \text{self}) \wedge (m_{i\ominus 1} = \text{self} \vee m_{i\oplus 1} = \text{left}) \longrightarrow m_i := \text{right}$$

$$m_i \neq \text{left} \wedge m_{i\ominus 1} = \text{self} \wedge m_{i\oplus 1} = \text{right} \longrightarrow m_i := \text{left}$$
$$m_i \neq \text{right} \wedge m_{i\oplus 1} = \text{self} \wedge m_{i\ominus 1} = \text{left} \longrightarrow m_i := \text{right}$$

$$m_i = \text{right} \wedge m_{i\ominus 1} = \text{right} \wedge m_{i\oplus 1} \neq \text{left} \longrightarrow \quad m_i := \text{left}$$
$$m_i = \text{left} \wedge m_{i\oplus 1} = \text{left} \wedge m_{i\ominus 1} \neq \text{right} \quad \longrightarrow \quad m_i := \text{right}$$

The first action of P_i points to itself only when both its neighbors point to other processes ($P_{i\oplus 1}$ points to $P_{i\oplus 2}$ and $P_{i\ominus 1}$ points to $P_{i\ominus 2}$). The second action is enabled when P_i points to itself and either its left (right) neighbor points to P_i so it matches with its left (right) neighbor or a left (right) neighbor points to itself so P_i matches with the right (left). The third action considers the case when P_i points to its left (right), this neighbor points to its left (right) neighbor and P_i's right (left) neighbor points to itself, then P_i points to its right (left) neighbor. The fourth action is enabled when P_i points to its left (right) neighbor that does not match with P_i while its right (left) neighbor points to P_i, then P_i matches with its right (left) neighbor. $\triangleleft$

3.6 Case Studies for Symmetric Protocols

In this section, we present some of our case studies for the addition of strong convergence to symmetric processes. Section 3.6.1 discusses a stabilizing symmetric three coloring protocol. Section 3.6.2 presents the synthesis of a leader election protocol over a ring. Finally, Section 3.6.3 demonstrates the solution STSyn generates for an agreement protocol.

3.6.1 Three Coloring

Our heuristic synthesized a symmetric TC protocol for $5 \leq K \leq 11$ as follows ($0 \leq i < K$):

$$P_i\text{:}\ \ (c_i = c_{i\ominus 1}) \vee (c_i = c_{i\oplus 1}) \qquad \longrightarrow \qquad c_i := \text{other}(c_{i\ominus 1}, c_{i\oplus 1})$$

Due to the fact that I_{color} is locally checkable and correctable, the synthesized protocol is straightforward in the sense that, each process action has the form $\neg LC_i \rightarrow$ *establish* LC_i. The assignment of the new color does not violate the local predicates $LC_{i\ominus 1}$, LC_i and $LC_{i\oplus 1}$ where $LC_i = (c_i \neq c_{i\oplus 1})$; that is why TC is locally correctable.

3.6.2 Leader Election

We synthesize a Leader Election (LE) protocol adopted from Huang *et al.* [27]. LE is defined over a bidirectional ring with K processes. Each process P_i has a variable $x_i \in \{0, 1, ..., K-1\}$ $(0 \leq i \leq K-1)$ where x_i is an identifier for P_i. A stable state is such that x_i uniquely identifies P_i. The set of legitimate states for LE is defined as $I_{leader} = \bigwedge_{i=1}^{i=K-1}(x_i \ominus x_{i\ominus 1} = x_{i\oplus 1} \ominus x_i)$, where K is a prime value. For a composite K, Huang *et al.* demonstrate the impossibility of having an SS protocol.

Given I_{leader} and an empty input protocol p, STSyn synthesizes the solution in [27] up to 5 processes. To reach this solution, we consider the requirement that x_i can be increased by 1 modulo K; any transition group containing a transition that increases x_i by more than one unit is excluded.

$$\begin{array}{lcl} (x_i \ominus x_{i\ominus 1}) < (x_{i\oplus 1} \ominus x_i) & \longrightarrow & x_i := x_i \oplus 1 \\ x_{i\ominus 1} = x_i = x_{i\oplus 1} & \longrightarrow & x_i := x_i \oplus 1 \end{array}$$

STSyn generated a solution for $K = N = 5$ in 2 seconds but could not synthesize a solution for $K = N = 7$ in a reasonable amount of time (less than 12 hours). This can be explained by the increase in the domain size and its impact on the exponential growth of the size of BDDs.

3.6.3 Agreement

We present a symmetric protocol on a bidirectional ring where the processes need to agree on a specific value: from an initial arbitrary state, all the variables should eventually be equal to one another. The ring has K processes P_i $(0 \leq i \leq K-1)$. Each process P_i can write its local variable x_i where $x_i \in \{0, \cdots, L-1\}$. Each process P_i can read its left $x_{i\ominus 1}$, right $x_{i\oplus 1}$ and its own variable x_i. The set of legitimate states is $I_{agreement} = \bigwedge_{i=1}^{i=K-1}(x_{i\ominus 1} = x_i)$. The protocol is not locally correctable: the establishment of $x_{i\ominus 1} = x_i$ by an action P_i can invalidate $x_i = x_{i\oplus 1}$. This fact complicates the search for a solution with similar processes. The input protocol p is empty. We restrict the set of groups to be included in our solution only to groups increasing the value of x_i. Thus, STSyn generates the following solution for up to 6 processes.

$$\begin{array}{lll} x_i < x_{i\ominus 1} & \longrightarrow & x_i := x_{i\ominus 1} \\ x_i < x_{i\oplus 1} & \longrightarrow & x_i := x_{i\oplus 1} \end{array}$$

Figure 3.11 illustrates a summary of the case studies we have conducted in terms of being locally-checkable/correctable, whether STSyn generated alternative solutions for them and the maximum size of their state space.

Case Study	**Locally Checkable**	**Locally Correctable**	**Synthesized Alternative Solutions**	**# of States**
Three-Coloring	Yes	Yes	Yes	3^{40}
Matching	Yes	No	Yes	3^{11}
Token Ring	No	No	Yes	5^5
Three-Ring	No	No	No	3^{10}
Leader Election	Yes	No	No	5^5
Agreement	Yes	No	No	6^6

Figure 3.11: Summary of case studies.

3.7 Experimental Results

While the significance of our work is in enabling the automated design of convergence, we would like to report the potential bottlenecks of our work in terms of tool development. Thus, in this section, we discuss our experimental results. We conducted our experiments on a Linux Fedora 10 distribution personal computer, with a 3GHz dual core Intel processor and 1GB of RAM. We have used C++ and the CUDD/GLU [28] library version 2.1 for BDD [19] manipulation in the implementation of STSyn.

In order to guarantee accuracy in the measurements of execution times, we repeat our experiments for every sample point in the graph until the statistical average of the execution times is bounded within an error margin of $\pm 0.5\%$. We observe that for smaller values of independent variables; i.e., the number of processes or the domain size, the required number of experiments is around 4. However, for increasing values of independent variables, we require around 20 repetitions of the experiment to guarantee the required error margin. For instance, in the 3-coloring protocol, 3 to 5 repetitions of our experiment for each value of the number of processes smaller than 25 suffice to guarantee our error margin. However, for larger number of processes, we had to repeat our experiment from 18 to 21 times.

For execution times, we illustrate the time for computing ranks, the SCC detection time

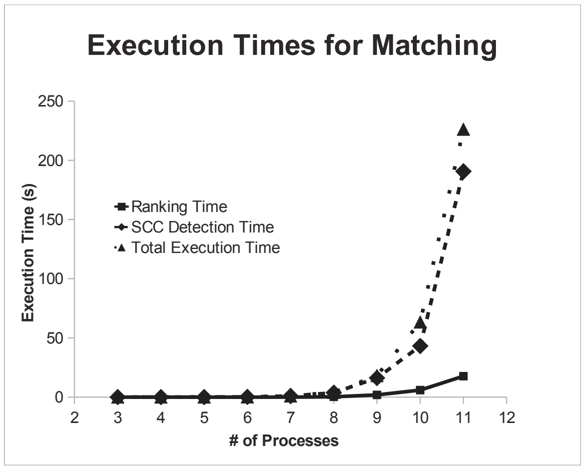


Figure 3.12: Time spent for adding convergence to matching versus the number of processes.

and the total execution time versus either the number of processes or the domain size of the variables. For memory usage, we illustrate the average number of BDD nodes for an SCC and total number of nodes in the synthesized BDD versus the number of processes or the domain size of the variables. The average number of BDD nodes is computed as the total number of nodes in all SCC's detected during the execution divided by their corresponding number of SCC's. We illustrate our results for asymmetric case studies except for the agreement protocol which we synthesized using our heuristic for symmetric protocols.

Figures 3.12 and 3.13 respectively represent how execution time and memory usage of synthesis grow as we increase the number of processes in the matching protocol. We measure the memory usage in terms of the number of BDD nodes rather than in kBytes for the following reason: in a platform-independent fashion, the number of BDD nodes reflects how space requirements of our heuristic grow during synthesis. Observe that, for maximal matching, increasing the number of processes significantly increases the execution time and memory usage of synthesis. Nonetheless, since the domain size is constant, we were able to scale up the synthesis and generate a strongly stabilizing protocol with 11 processes in a few minutes.

Figures 3.14 and 3.15 respectively demonstrate execution time and memory usage of adding convergence to the TC protocol. We have added convergence to the coloring protocol for 8 versions from 5 to 40 processes with a step of 5. Since the added recovery transitions for the coloring protocol do not create any SCCs outside $I_{coloring}$, we have been able to scale up the synthesis and generate a stabilizing protocol with 40 processes.

While both $I_{coloring}$ in the coloring protocol and I_{MM} in the matching protocol are locally checkable for each process P_i, we note that the cost of synthesizing a maximal matching

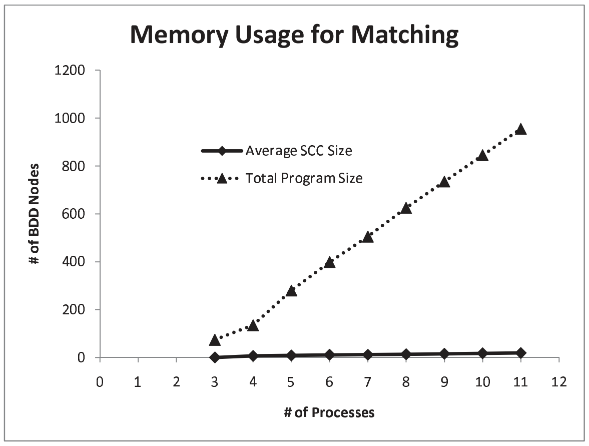


Figure 3.13: Space usage for adding convergence to matching versus the number of processes.

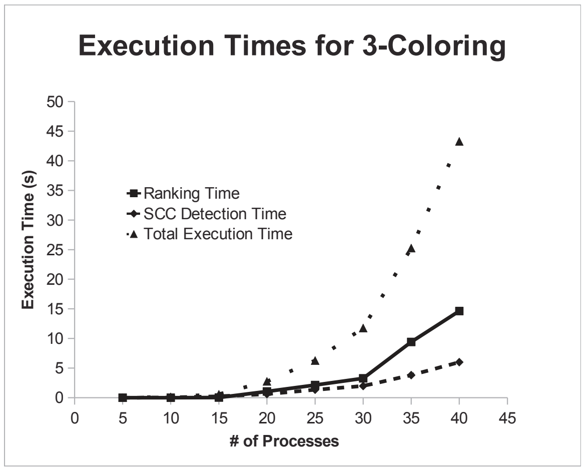


Figure 3.14: Time spent for adding convergence to 3-Coloring versus the number of processes.

protocol is higher in part because the MM protocol is not locally correctable, whereas the TC protocol is. In the MM protocol, consider a case where the first conjunct of the local predicate LC_i is false for P_i. That is, $m_i =$ left and $m_{i\ominus 1} \neq$ right. If P_i makes an attempt to satisfy its local predicate LC_i by setting m_i to self, then LC_i may become invalid if $m_{i\ominus 1} \neq$ left. The last option for P_i would be to set m_i to right, which may not make the second conjunct true if $m_{i\oplus 1} \neq$ left. Thus, the success of P_i in correcting its local predicate depends on the actions of its neighbors as well. Such dependencies cause cycles outside I_{MM}, which complicate the design of convergence. By contrast, in the coloring protocol, each process can easily establish its local predicate $c_{(i\ominus 1)} \neq c_i$ by selecting a color that is different from its left and right neighbors.

Figures 3.16 and 3.17 respectively illustrate how execution time and memory usage of

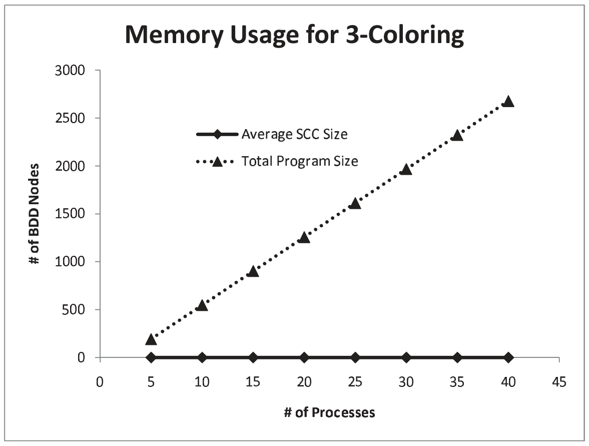


Figure 3.15: Space usage for adding convergence to 3-Coloring versus the number of processes.

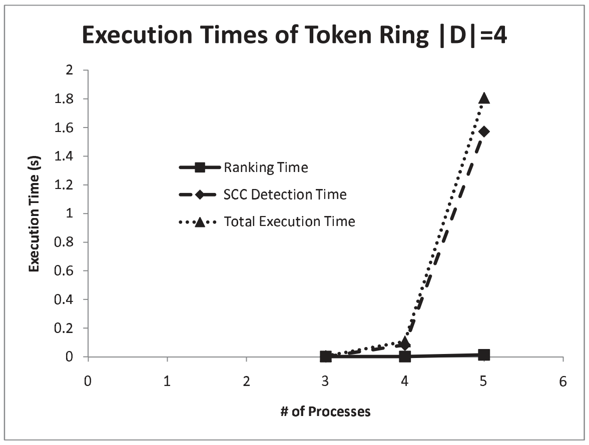


Figure 3.16: Time spent for adding convergence to Token Ring versus the number of processes.

synthesis increase for the token ring protocol as we keep size of the domain of x variables constant (i.e., $|D| = 4$) and increase the number of processes.

Figures 3.18 and 3.19 respectively illustrate how execution time and memory usage of synthesis increase for the agreement symmetric protocol as we keep the size of the domain of x variables constant (i.e., $|D| = 3$) and increase the number of processes. Figures 3.20 and 3.21 respectively illustrate how execution time and memory usage of synthesis increase for the symmetric agreement protocol as we keep the number of processes constant ($|P| = 5$) and increase the domain size of x variables. Observe that the SCC detection time is our major bottleneck.

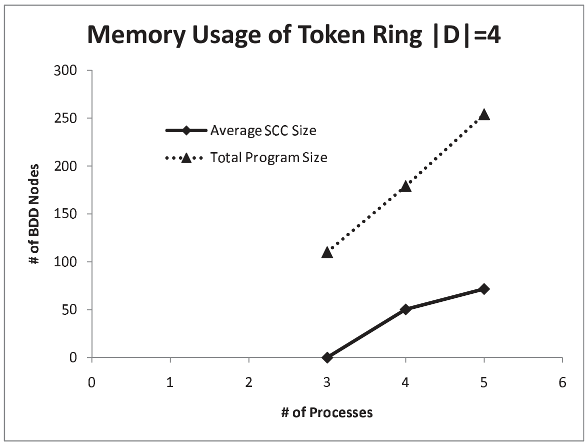


Figure 3.17: Space usage for adding convergence to Token Ring versus the number of processes.

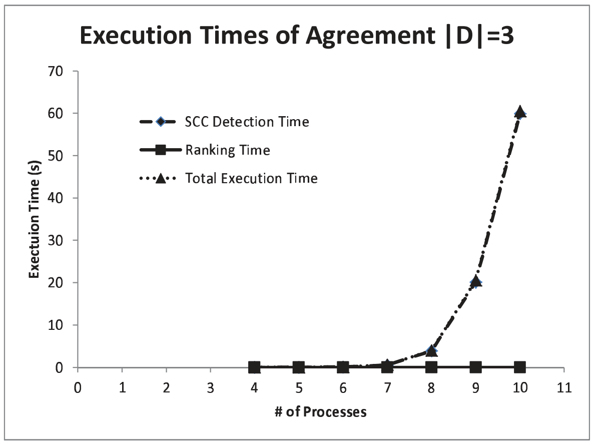


Figure 3.18: Time spent for adding convergence to Agreement versus the number of processes.

3.8 Discussion

In this section, we discuss issues related to the applications, strengths and some limitations of our lightweight method.

Applications. There are several applications for the proposed lightweight method. First, STSyn can be integrated in model-driven development environments (such as Unified Modeling Language [29] and Motorola WEAVER [30]) for protocol design and visualization. For the implementation of STSyn, we have benefited from commonly-used data structures (e.g., BDDs [19]) that are applied in the implementation of model checkers. However, our heuristics automatically synthesize the convergence actions necessary to

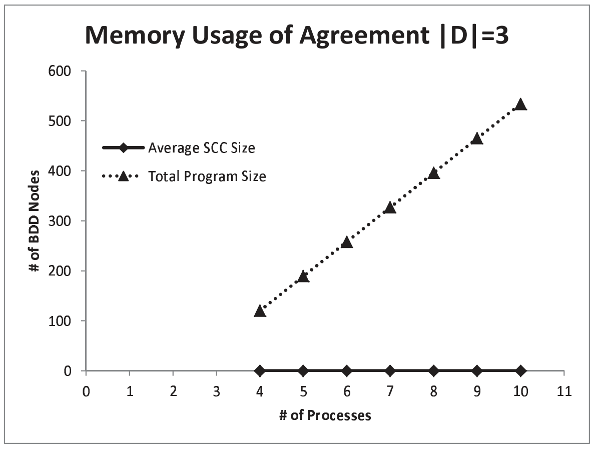


Figure 3.19: Space usage for adding convergence to Agreement versus the number of processes.

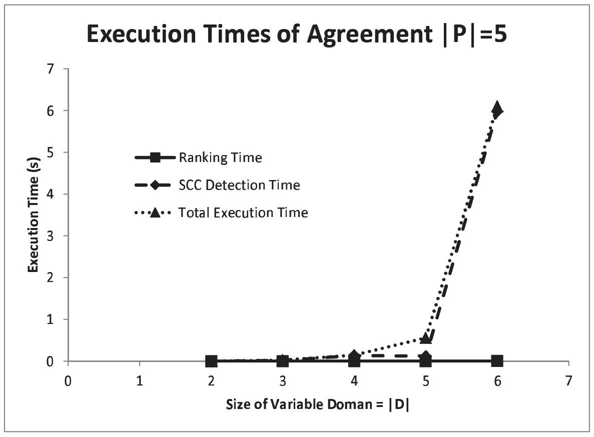


Figure 3.20: Time spent for adding convergence to Agreement versus the domain size.

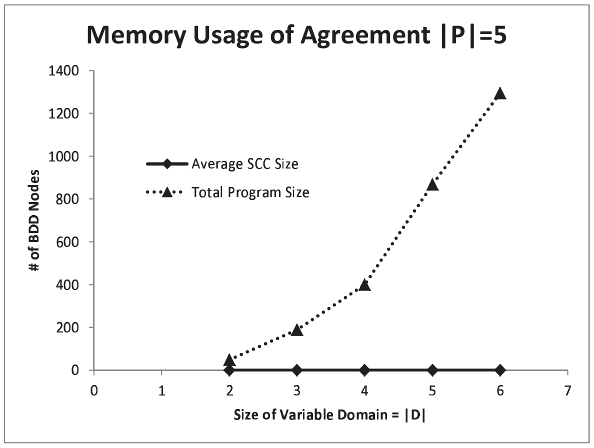


Figure 3.21: Space usage for adding convergence to Agreement versus the domain size.

make a protocol self-stabilizing instead of just verifying them. While model checkers generate scenarios as to how a protocol fails to self-stabilize, the burden of revising the protocol so it becomes self-stabilizing remains on the shoulders of designers. Our heuristics revise a protocol towards generating a self-stabilizing version thereof. As such, an integration of our heuristics with model checkers can greatly benefit the designers of self-stabilizing protocols.

Limitations. The objectives of this research place scalability at a low degree of priority as the philosophy behind our lightweight method is to benefit from automation as long as available computational resources permit. Nonetheless, we have analyzed the behavior of STSyn regarding time/space cost (see Section 3.7). The extent to which we can currently scale up a protocol depends on many factors including the number of processes, variable domains and topology. For example, while our tool is able to synthesize a self-stabilizing protocol with up to 40 processes for the 3-coloring problem, it is only able to find solutions for Dijkstra's token ring with up to 5 processes, each with a variable domain size of 5. One of the major factors affecting the scalability of our heuristic is the cycle resolution problem. The number of cycles mainly depends on the size of the variable domains and the size of the transition groups (which is also determined by the number of unreadable variables and their domains). Our experience shows that the larger the size of the groups and the variable domains, the more cycles we get. We believe that scaling-up our heuristic is strongly dependent on our ability to scale-up cycle resolution, which is the focus of one of our current investigations. Although the proposed heuristic does not scale-up systematically for all input protocols, our lightweight approach allows designers to have some concrete examples of a possibly general self-stabilizing version of a non-stabilizing protocol.

3.9 Summary and Extensions

We presented a lightweight method for automated addition of convergence to non-stabilizing network protocols to make them self-stabilizing, where a self-stabilizing protocol recovers/converges to a set of legitimate states from *any* state in its state space. The addition of convergence is a problem for which no polynomial-time algorithm is known yet, nor is there a proof of NP-completeness for it (though it is in NP). As a building block of our lightweight method, we presented a heuristic that automatically adds strong convergence to non-stabilizing protocols in polynomial time (in the state space of the non-stabilizing protocol). We also presented a sound and complete method for automated design of weak convergence (Theorem 3.2.1). While most existing manual/automatic methods for the addition of convergence mainly focus on locally-checkable/correctable protocols, our method automates the addition of convergence to non-locally-checkable/correctable protocols. We also presented a heuristic for synthesizing symmetric self-stabilizing protocols. We have implemented our heuristics in a software tool, called STabilization Synthesizer (STSyn), using which we have automatically generated many stabilizing protocols including different versions of Dijkstra's token ring protocol [7], maximal matching, three coloring in a ring, a three-ring token passing protocol, a leader election protocol and an agreement protocol. STSyn has generated alternative solutions and has facilitated the detection of a design flaw in a manually-designed maximal matching protocol [15].

In Chapter 4, we investigate the parallelization of our heuristics towards exploiting the computational resources of computer clusters for automated design of self-stabilization. In this chapter, we considered the addition of convergence under the interleaving semantics. We will study the automated design of convergence under the concurrent semantics. In addition to what we mentioned in Section 3.8, another important open problem focuses on the generalization of synthesized solution. In other words, we would like to address this question: *If a synthesized symmetric protocol is self-stabilizing for K processes, then can we generalize its structure for $K + 1$ processes?* In this regard, we plan to study how increasing the number of symmetric processes in a self-stabilizing protocol affects its transition system. In particular, we will consider how non-progress cycles and deadlocks will be formed when we extend the protocol's configuration space of K processes to that of $K + 1$ processes. To this end, we demonstrate theories that characterize deadlocks and non-progress cycles in arbitrary-sized rings in Chapters 5 and 6.

Chapter 4

Swarm Synthesis of Convergence for Symmetric Protocols[1]

This chapter presents a novel non-deterministic method for algorithmic addition of convergence to non-stabilizing symmetric protocols. The proposed method exploits randomization and parallelization in order to expand the scope of the search for self-stabilizing versions of non-stabilizing protocols. Such a non-deterministic method enables an embarrassingly parallel framework that exploits the computational resources of computer clusters for automated design of self-stabilizing protocols. We have implemented our approach in a software tool and have synthesized several new self-stabilizing solutions for well-known protocols in the literature (e.g., maximal matching, graph coloring and leader election on a ring). Our case studies demonstrate that the proposed method is able to automatically generate self-stabilizing versions of non-stabilizing protocols in cases where existing automated methods fail. As a result, the proposed approach increases the likelihood of success in synthesizing the self-stabilizing versions of non-stabilizing protocols.

4.1 Introduction

This chapter presents a novel non-deterministic method that adds convergence to *finite-state symmetric* protocols, thereby exploiting the computational resources of computer clusters towards automated design of self-stabilizing protocols. A *symmetric* protocol is defined by

[1]This chapter is an adaptation of our own publication in the proceedings of the ninth European Dependable Computing Conference (EDCC), Sibiu Romanina, May 2012 [11]. Reprinted with permission, please see the supplementary document to this dissertation for the copyright notice.

a representative/template process from which the code of each process can be generated by a simple variable renaming. We would like to emphasize that improving the scalability of automated design is not an objective of this chapter. Instead, by proposing a non-deterministic heuristic, we would like to increase the likelihood of success in adding convergence to non-stabilizing protocols.

The proposed method includes three phases, namely *ranking and approximation, ordering recovery actions* and *spawning synthesizer threads*. The first phase,*Ranking and approximation*, is described in details in Chapter 3.

In the second phase, the proposed method *orders* each possible recovery action based on the smallest rank – called the *rank of the recovery action* – from which that action can execute and can take the protocol's state to a state with a smaller rank. Notice that the ranking of recovery actions depends on the ranking of states in $\neg I$. For example, the bold transitions in Figure 4.1 represent a recovery action whose rank is 2. The output of Phase 2 is an array *Groups* such that each element *Groups*$[i]$ is an *ordered* list of candidate recovery actions whose rank is i.

In the third phase, for each i from 1 to the total number of ranks/partitions of $\neg I$, the proposed method traverses the ordered list *Groups*$[i]$ and includes an action A in the stabilizing protocol *if and only if* A resolves some deadlock states in $\neg I$ without forming cycles with previously included recovery actions. A deadlock state has no outgoing transitions (see Figure 4.1). If cycles are formed, then A is excluded from the stabilizing protocol and the subsequent actions in the list *Groups*$[i]$ are similarly considered. If at the end of the third phase there are still some deadlock states, then we permute the candidate recovery actions in each *Groups*$[i]$ and re-do the third phase. Such a reordering allows us to consider subsets of recovery actions that we have not considered so far. This will increase the likelihood of finding a self-stabilizing version of p by exploiting randomization. Towards this end, we create several parallel instances of the proposed method. Each instance permutes the candidate actions in each rank until either a solution is found or an upper bound (specified for the time (respectively, space) of synthesis) is reached.

We have designed and implemented the proposed approach in a software tool, called the parallel STabilization Synthesizer (pSTSyn). Using pSTSyn, we have generated the self-stabilizing versions of several symmetric self-stabilizing protocols including maximal matching, graph coloring, agreement and leader election on a ring. pSTSyn has generated new self-stabilizing protocols (for coloring, matching and leader election) that we could not synthesize with existing heuristics (see Section 4.4).

Organization. Section 4.2 illustrates the complexity of adding convergence through

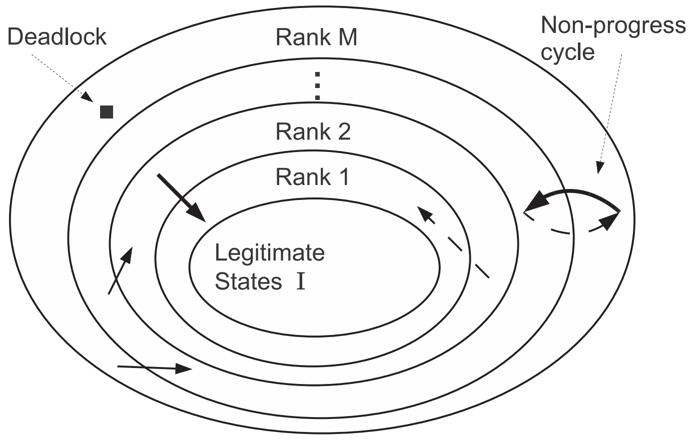


Figure 4.1: Ranking and approximating self-stabilization.

a running example. Section 4.3 presents a new method that enables the addition of convergence in an embarrassingly parallel fashion, called the *swarm synthesis* of convergence. Section 4.4 demonstrates some case studies synthesized by pSTSyn and provides the experimental results on time (respectively, space) efficiency of swarm synthesis. Section 4.5 makes concluding remarks and discusses potential extensions of our work.

4.2 Complexity of Synthesizing Convergence

We illustrate the complexity of designing strong convergence throughout the following example.

Example: Token Ring (TR). To demonstrate the complexity of the problem, we present the problem of adding convergence in the context of an instance of the Token Ring (TR) protocol (adapted from [7]) with three processes P_0, P_1 and P_2. Each process P_j has an integer variable a_j, where $0 \leq j < 3$, with a domain $\{0, 1, 2\}$. The processes are located in a ring where each process P_j $(1 \leq j \leq 2)$ has a predecessor P_{j-1} and a successor P_{j+1}, where addition and subtraction are in modulo 3. The predecessor of P_0 is P_2 and its successor is P_1. Each process is allowed to read/write its own variable and can read the variable of its predecessor. Action Γ_0 in Figure 4.2 belongs to P_0, and the parameterized action Γ_j denotes the action of each process P_j, where $1 \leq j \leq 2$.

Γ_0: $a_2 = a_0 \quad\quad\quad \rightarrow a_0 := a_2 + 1$
Γ_j: $a_{j-1} = a_j + 1 \rightarrow a_j := a_{j-1}$

Figure 4.2: Actions of the Token Ring protocol where $j = 1, 2$.

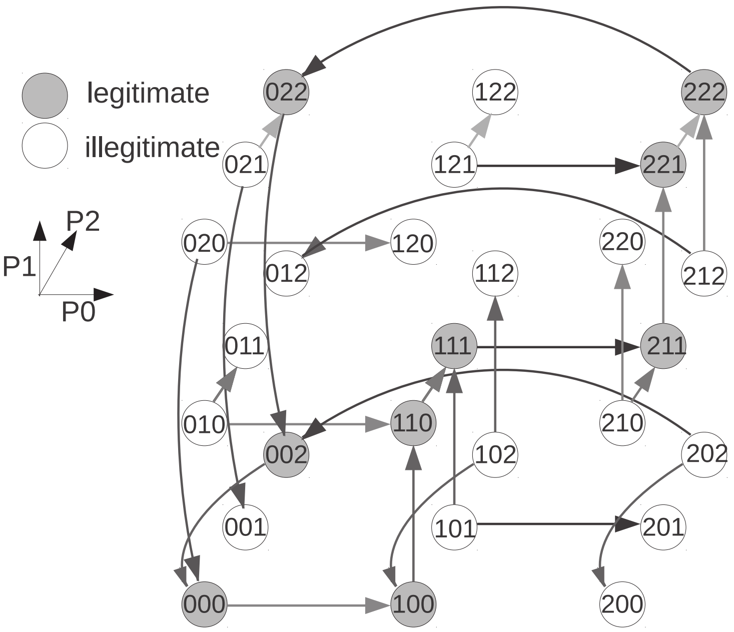


Figure 4.3: State transition graph of the non-stabilizing Token Ring protocol

When the values of a_0 and a_2 are equal, P_0 increments a_0 by one (see action Γ_0). Each process P_j increments a_j only if a_j is one unit less than a_{j-1}, for $1 \leq j \leq 2$. By definition, process P_j, for $1 \leq j \leq 2$, *has a token* if and only if $a_j + 1 = a_{j-1}$. Process P_0 *has a token* if and only if $a_0 = a_2$. The state predicate I_{TR} below captures the set of *legitimate* states where exactly one token exists in the ring.

$$((a_0 = a_1) \wedge (a_1 = a_2)) \vee ((a_1 + 1 = a_0) \wedge (a_1 = a_2)) \vee ((a_0 = a_1) \wedge (a_2 + 1 = a_1))$$

Figure 4.3 illustrates the state transition graph of the non-stabilizing TR protocol in a three dimensional representation. The nodes of this graph denote global states and arrows represent global transitions. The gray nodes represent legitimate states in I_{TR} and white nodes depict illegitimate states in $\neg I_{TR}$. Each dimension denotes the execution of transitions/actions of a specific process (e.g., the x-Axis depicts the transition groups belonging to P_0). Let $\langle a_0, a_1, a_2 \rangle$ denote a state of TR. The three digits in each node respectively represent the values of a_0, a_1 and a_2. Notice that, starting from the state $\langle 0, 0, 0 \rangle$ the execution of the TR protocol remains in the set of legitimate states (i.e., gray states). However, the occurrence of transient faults may perturb the state of TR to one of the illegitimate states. For example, consider the deadlock state $s_0 = \langle 1, 2, 0 \rangle$ that has no outgoing transition. From s_0, no process can execute. Thus, we need to include recovery transitions (along with their groupmates) to resolve s_0. Likewise, any other deadlock state should be resolved. Such inclusion of recovery transitions should be performed under the constraint of not creating cycles whose all states are illegitimate; i.e., non-progress cycles.

Figure 4.5 illustrates a set of recovery transition groups included in TR that resolves all deadlocks. This set includes the labeled actions in Figure 4.4, where A actions belong to process P_1 and B actions belong to P_2. While all deadlocks are resolved by the actions A_1, A_2, A_3, B_1, B_2 and B_3, the transition groups included form a cycle in $\neg I_{TR}$ that could prevent recovery to I_{TR} (see bold arrows in Figure 4.5). However, if we replace the actions A_1 and B_1 with actions $A_4 : (a_0 = 1) \wedge (a_1 = 2) \rightarrow a_1 := 1$ and $B_4 : (a_1 = 1) \wedge (a_2 = 2) \rightarrow a_2 := 1$, then a self-stabilizing TR protocol is generated (see Figure 4.6).

A_1: $(a_0 = 1) \wedge (a_1 = 2) \quad \rightarrow a_1 := 0$
$A_2 : (a_0 = 2) \wedge (a_1 = 0) \quad \rightarrow a_1 := 1$
$A_3 : (a_0 = 0) \wedge (a_1 = 1) \quad \rightarrow a_1 := 2$
B_1: $(a_1 = 1) \wedge (a_2 = 2) \quad \rightarrow a_2 := 0$
$B_2 : (a_1 = 2) \wedge (a_2 = 0) \quad \rightarrow a_2 := 1$
$B_3 : (a_1 = 0) \wedge (a_2 = 1) \quad \rightarrow a_2 := 2$

Figure 4.4: Transition groups included in the TR protocol for deadlock resolution.

A deterministic heuristic that selects the first set of actions would fail to add convergence, whereas one that selects the second set would succeed in adding convergence to TR. Thus, to increase the likelihood of success, it is desirable to design non-deterministic methods that simultaneously explore the possibility of adding convergence for different sets of recovery actions by exploiting parallel machines. The next section presents such a non-deterministic method for algorithmic design of convergence.

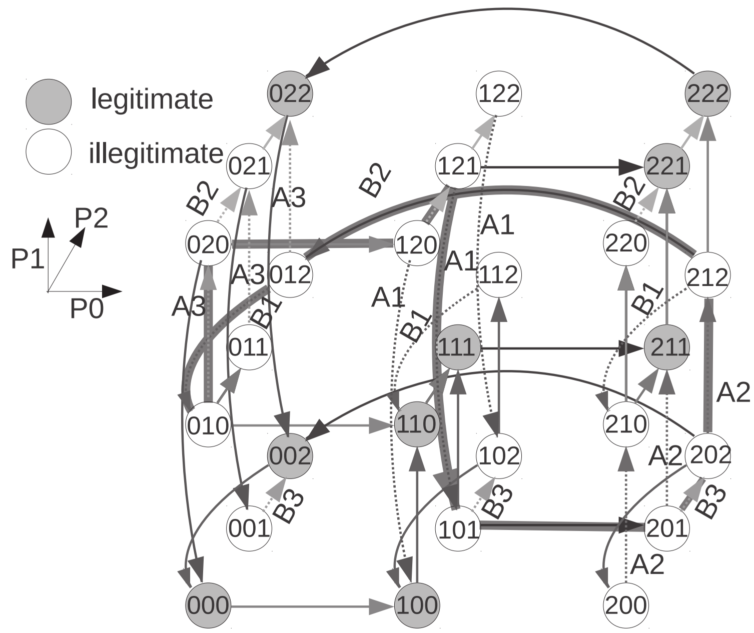


Figure 4.5: Inclusion of actions A_1, A_2, A_3, B_1, B_2 and B_3 creates a non-progress cycle in $\neg I_{TR}$.

4.3 A Method for Swarm Synthesis

In this section, we present a method for adding strong convergence to symmetric protocols by exploiting randomization and parallelism. The proposed approach enables the addition of convergence in an embarrassingly parallel fashion, called *swarm synthesis*. To solve Problem 2.2.1, the proposed method (see Figure 4.7) includes three phases, namely *Rank and Approximate, Order Recovery Groups* and *Spawn Synthesizers*. These three phases are initiated by the Main component. In the first phase, Main computes the ranks and a candidate set of recovery transition groups from which the synthesizer chooses a subset for strong convergence. In the second phase, Main orders the set of candidate recovery groups based on the smallest rank from which a recovery group can provide convergence to I. In the third phase, Main spawns parallel threads of the Looper routine. Each Looper independently considers different permutations of candidate recovery groups. This way, our heuristic explores a larger subset of the solution space.

Phase 1: Rank and Approximate. In the rank-and-approximation phase (see Figures 4.7 and 4.8), the Rank-Approximate routine computes a set of Recovery Transition Groups (RTGs) $\delta_{r_{ws}} = \{g : g \in G(P_r) \wedge \forall (s, s') \in g : s \notin I\}$. Since $G(P_r)$ comprises all transition groups that can be included in P_r, the set of transitions $\delta_{r_{ws}}$ captures all possible candidate RTGs that satisfy the read/write restrictions of P_r and exclude any transition starting in I. Thus, the transitions of $\delta_{r_{ws}}$ do not violate the closure of I, and maintain constraint (2)

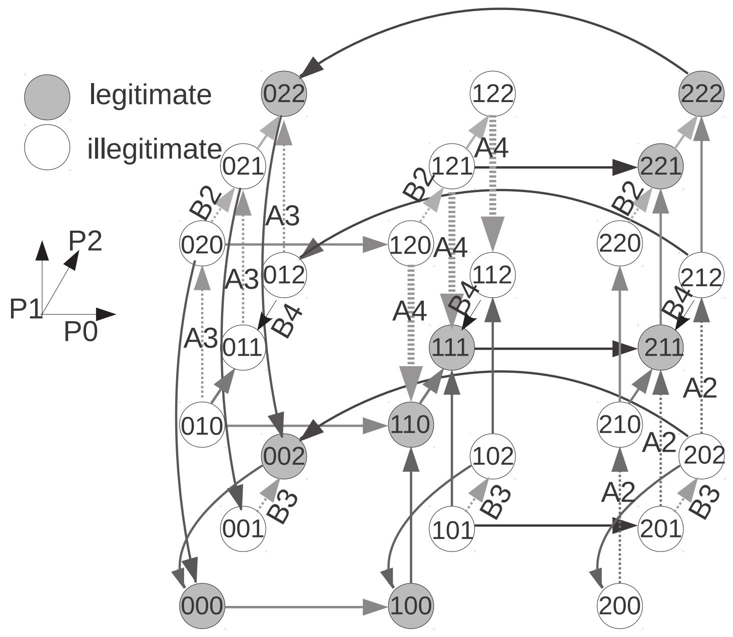


Figure 4.6: Inclusion of actions A_4, A_2, A_3, B_4, B_2 and B_3 results in a self-stabilizing version of TR.

of the output part of Problem 2.2.1. We denote by p_{ws} the protocol whose representative process $P_{r_{ws}}$ comprises the transition groups $\delta_{r_{ws}} \cup \delta_r$. Hence, the computations of p_{ws} consist of transitions in the RTGs of $\delta_{p_{ws}} = (\cup_{r=0}^{K-1} \delta_{r_{ws}}) \cup \delta_p$. Rank-Approximate computes the rank of every state s in $\neg I$, where *Rank*(s) is the length of the shortest computation prefix of p_{ws} from s to some state in I (see Figure 4.1). Note that *Rank*$(s) = 0$ if and only if $s \in I$. Moreover, *Rank*$(s) = \infty$, if and only if there is no computation prefix of p_{ws} from s that reaches a state in I. *NumRanks* denotes the total number of ranks. If each state $s \in \neg I$ has a finite rank, then including the computation prefixes originating at s would result in a weakly stabilizing version of p; i.e., p_{ws} is a weakly stabilizing protocol. (Please see [10] for a formal proof of correctness.) Otherwise, a self-stabilizing version of p does not exist.

TC Example. For the TC example, Rank-Approximate computes $P_{r_{ws}}$ as follows.

$$B_r : \ (c_r = c_{r-1}) \vee (c_r = c_{r+1}) \ \rightarrow \ c_r := \textit{other}(c_r, c_r)$$

That is for every r, $P_{r_{ws}}$ updates c_r to whichever color is different from its current value. $P_{r_{ws}}$ contains every possible RTG that adheres to read/write restrictions of TC and has no transitions starting in I_{color}. Note that the protocol represented by $P_{r_{ws}}$ has no deadlocks in I_{color}, but may have non-progress cycles. Moreover, for any strongly stabilizing solution $P_{r_{ss}}$, we have $P_{r_{ss}} \subset P_{r_{ws}}$.

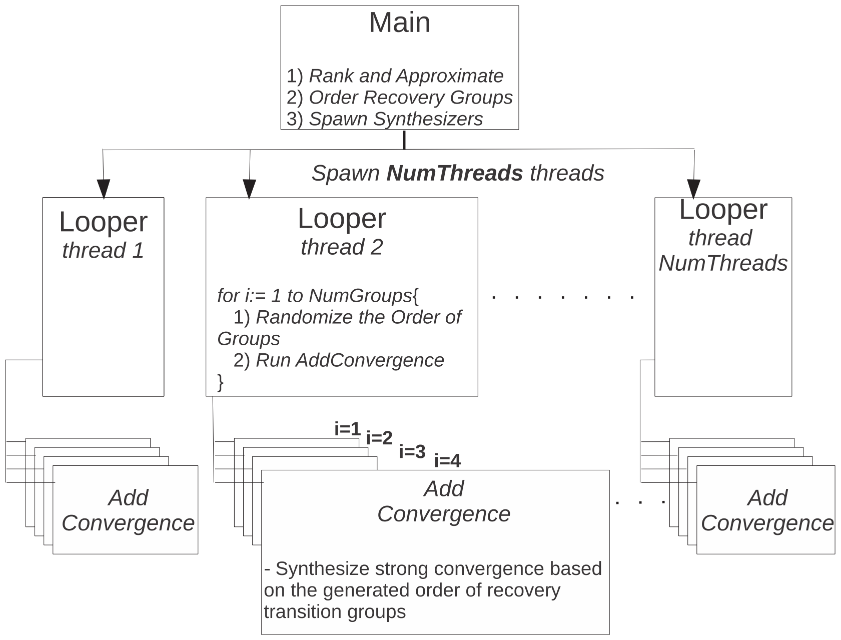


Figure 4.7: Overview of swarm synthesis of convergence.

The number of ranks in TC depends on K. We denote a global state of TC by $\langle c_0, \cdots, c_{K-1}\rangle$ where $c_i \in \{0,1,2\}$ $(0 \le i \le K-1)$. For $K = 7$, the global state $\langle 0,1,0,1,0,0,2\rangle$ has rank 1; P_5 can execute a single assignment to $c_5 := 1$ to render the global state $\langle 0,1,0,1,0,1,2\rangle \in I$. The global state $\langle 0,1,0,1,0,0,0\rangle$ is of rank 2, at least two processes should write their corresponding variables to render TC's state in I. P_5 executes $c_5 := 1$ and P_0 executes $c_0 := 2$ to set TC's global state to $\langle 2,1,0,1,0,1,0\rangle \in I$. The TC protocol has a number of ranks proportional to its number of variables. Every protocol transition that decreases the rank of a state in $\neg I_{color}$ establishes two adjacent conjuncts of I_{color}. Thus, TC has $\lfloor \frac{K+1}{2} \rfloor$ ranks. $\triangleleft$

Phase 2: Order Recovery Groups. This phase of the proposed method takes the ranks generated by Phase 1 and computes a partial order of all *candidate* RTGs based on ranks. Phase 2 is executed by the Order routine in Line 2 of Figure 4.8. A transition (s_0, s_1) is *rank decreasing* if and only if *Rank*(s_1) < *Rank*(s_0). We say that the *rank of an RTG* g is $i > 0$ if and only if i is the smallest rank from where g includes a rank decreasing transition. Notice that some RTGs may have no rank-decreasing transitions, called the *rankless* RTGs. Thus, considering all RTGs with a specific rank, we generate a partial order of RTGs as an array of lists, denoted *Groups*$[]$, where *Groups*$[i]$ is an ordered list of all RTGs whose rank is i (see Line 2 of Figure 4.8). The ranked RTGs may fill the array *Groups*$[]$ up to a certain rank, denoted *maxRTGRank*, which may not necessarily be equal to *NumRanks*. We then insert all rankless RTGs in *Groups*$[$*maxRTGRank*$+1]$. The array *RankSize* has *maxRTGRank*$+1$ elements, where *RankSize*$[i]$ contains the number of RTGs whose rank is i; i.e., size of *Groups*$[i]$.

TC Example. We partition the RTGs of $P_{r_{ws}}$ according to their ranks. RTGs of Rank 1; i.e., having a transition that changes the global state of p_{ws} from rank 1 to rank 0, are of the form:

Rank 1: $(c_{r-1} = c_r) \vee (c_{r+1} = c_r) \rightarrow c_r := \mathit{other}(c_{r-1}, c_{r+1})$

In fact, by updating c_r's value to one that is different from both its neighbors, all the transitions in every $g \in$ Rank 1 are rank decreasing. The transitions of Rank 1 are assigned to *Groups*[1]. The remaining RTGs in $(P_{r_{ws}} -$ Rank 1$)$ are rankless and are assigned to *Groups*[2]. That is, *maxRTGRank*$=\ 2$. As a shorthand for $c_{r-1} = e_{r-1} \wedge c_r = e_r \wedge c_{r+1} = e_{r+1} \rightarrow c_r := e'_r$, we omit the variable names and represent an RTG by $e_{r-1}e_re_{r+1} \rightarrow e_{r-1}e'_re_{r+1}$. We enumerate the list of candidate RTGs in $P_{r_{ws}}$ and their corresponding ranks as computed by Rank-Approximate and Order.

Groups[1]=$\{001 \rightarrow 021, 112 \rightarrow 102, 220 \rightarrow 210, 000 \rightarrow 010, 000 \rightarrow 020, 111 \rightarrow 101, 111 \rightarrow 121, 222 \rightarrow 212, 222 \rightarrow 202, 022 \rightarrow 012, 100 \rightarrow 120, 211 \rightarrow 201, 002 \rightarrow 012, 110 \rightarrow 120, 221 \rightarrow 201, 011 \rightarrow 021, 122 \rightarrow 102, 200 \rightarrow 210\}$

Groups[2]=$\{002 \rightarrow 022, 110 \rightarrow 100, 221 \rightarrow 211, 001 \rightarrow 011, 112 \rightarrow 122, 220 \rightarrow 200, 011 \rightarrow 001, 122 \rightarrow 112, 200 \rightarrow 220, 022 \rightarrow 002, 100 \rightarrow 110, 211 \rightarrow 221\}$ $\triangleleft$

```
Algorithm 1: Main
  Input  : NumThreads, K: integer, $P_r$: set of representative transition
           groups, I: set of states
  Output: $P_{r_{ss}}$: set of representative transition groups, success:
           boolean, Groups[..]: array of lists of transition groups,
           RankSize[..]: array of integers, NumRanks, maxRTGRank:
           integer
1 $P_{r_{ws}}$, Ranks, NumRanks ← Rank-Approximate($P_r$, I, K);
2 Groups, NumGroups, RankSize, maxRTGRank ← Order($P_{r_{ws}}$, Ranks,
  NumRanks);
3 foreach $0 \leq Thd < NumThreads$ do
4     $P_{r_{ss}}$, success ← Looper(Thd, $P_r$, I, K, Groups, NumGroups,
      RankSize, maxRTGRank);
5 return;
```

Figure 4.8: The Main routine.

Phase 3: Spawn Synthesizers. After creating a partial order of RTGs based on the ranks, the Main routine spawns a fixed number of Looper threads (see Lines 3-4 of Figure 4.8) each with a unique identifier *ThreadIndex*. We set the number of threads based on the available computational resources of the computer cluster. Each Looper thread randomly reorders the RTGs of each rank (using the ShuffleGroups routine in Figure 4.9) and invokes the AddConvergence routine in an iterative fashion (see the for-loop in Figure 4.9). The total number of RTGs in all ranks is denoted by *NumGroups*. The ShuffleGroups routine generates permutations that depend upon the *ThreadIndex* of that Looper and the iteration i of the for-loop in Figure 4.9. ShuffleGroups computes a unique integer from the pair $(i, \textit{ThreadIndex})$ which is *PIndex*= $i * \textit{NumThreads} + \textit{ThreadIndex}$. *PIndex* is an input to a standard routine for generating a permutation of a given size. We reuse a variant of Algorithm *L* mentioned by Knuth in Section 7.2.1.2 of [31] to establish a one-to-one correspondence between *PIndex* and the generated permutation. Since the values of *PIndex* generated in each thread are unique to that thread, different Loopers explore different permutations. We place an upper bound *NumGroups* on the number of iterations of the for-loop in Lines 2-8 of Looper to avoid an exponential number of iterations (in the number of candidate RTGs) during synthesis. Once a Looper thread succeeds in synthesizing a self-stabilizing protocol, a termination signal is sent to all the Loopers.

```
Algorithm 2: Looper
  Input  : ThreadIndex, K, maxRTGRank, NumGroups: integer, $P_r$: set of
           representative groups, I: set of states, Groups[..]: array of
           vectors, RankSize[..]: array of integers
  Output: $P_{r_{ss}}$: set of representative groups, success: boolean
1 success ← false; $P_{r_{ss}} \leftarrow P_r$;
2 for i ← 0 to NumGroups-1 do
3     $Groups_{interm}$ ← ShuffleGroups(Groups, RankSize[..],
      maxRTGRank+1, ThreadIndex, i);
4     $success_{interm}$, $P_{interm}$ ←AddConvergence($P_r$, I, K, $Groups_{interm}$);
5     if ($success_{interm}$ = true) then
6         $P_{r_{ss}} \leftarrow P_{interm}$;
7         success ← true;
8         return;
9 return;
```

Figure 4.9: The Looper routine.

The AddConvergence routine (see Line 4 of Figure 4.9 and Figure 4.10) takes a representative process P_r, a state predicate I that is closed in the symmetric protocol represented by P_r, the number of processes K, and the permuted RTGs in the array *Groups*$[]$. The objective of AddConvergence is to check whether convergence can be designed by incremental inclusion of RTGs of *Groups*$[i]$ in P_r, for $1 \leq i \leq \textit{maxRTGRank}$ $+1$ (see the for-loops in Lines 4-5 of Figure 4.10). Initially, we assign P_r to a representative process $P_{r_{ss}}$ that is updated during the inclusion of RTGs (Line 1 in Figure 4.10). Notice that initially P_r represents the non-stabilizing protocol that does not guarantee convergence to I. Our goal is to include a subset of RTGs in the set of transitions of $P_{r_{ss}}$ such that the resulting protocol ensures convergence to I. The Unfold routine instantiates $P_{r_{ss}}$ for all $0 \leq r \leq K-1$ to generate the transition system of an intermediate synthesized protocol p_{interm}. Lines 4-14 consider RTG *Groups*[i][j] where i is the rank of the RTG and j is the index of the RTG in rank i. Starting from *Groups*$[1]$, an RTG g is included if and only if g resolves some deadlock states in $\neg I$ and the inclusion of g preserves the cycle-freedom of transitions starting in $\neg I$ (Lines 6-9 in Figure 4.10). Line 6 computes the set of source states of the current RTG and assigns it to *Pre*. Line 7 checks if *Pre* resolves any deadlocks, otherwise the current RTG (*Groups*[i][j]) is skipped. We reuse a symbolic cycle detection algorithm due to Somenzi *et al.* [32] that we have implemented in the DetectCycles routine (see Line 8 in Figure 4.10). If an RTG creates a cycle, then we skip its inclusion and check the feasibility of including the next RTG in the list *Groups*$[1]$. The motivation behind the inclusion of individual RTGs is to simplify cycle-resolution. If we include more than one RTG per iteration and cycles form in p_{interm}, deciding on which RTG to remove for cycle resolution would involve a computational overhead. Upon the inclusion of an RTG in $P_{r_{ss}}$ (Line 9), we unfold the updated structure of $P_{r_{ss}}$ to update the intermediate protocol p_{interm}, which will be used to recalculate the deadlock states (Lines 10-11 in Figure 4.10). After all RTGs in *Groups*$[1]$ are checked for inclusion, then we respectively perform the same analysis on the RTGs in *Groups*$[2]$, *Groups*$[3]$, $\cdots$, *Groups*$[$*maxRTGRank*$+1]$.

```
Algorithm 3: AddConvergence
  Input  : $P_r$: set of transition groups, $I$: set of states, $K$: integer,
           $Groups[maxRTGRank{+}1]$: array of vectors of groups
  Output: $P_{r_{ss}}$: set of transition groups, success: boolean
1  $P_{r_{ss}} \leftarrow P_r$;
2  $p_{interm} \leftarrow$ Unfold($P_{r_{ss}}$, $K$) ;    /* generates the whole transition
   system by instantiating $P_{r_{ss}}$ for all processes */
3  $Deadlocks \leftarrow \{s : (s \in \neg I) \wedge (\forall g, s_1 : (s, s_1) \in g : g \notin p_{interm})\}$;
4  for $i \leftarrow 1$ to $maxRTGRank\ {+}1$ do
5      for $j \leftarrow 1$ to $RankSize[i]$ do
6          $Pre \leftarrow \{s_0 \colon \exists (s_0, s_1) : (s_0, s_1) \in Groups[i][j]\}$;
7          if $Pre \cap\ Deadlocks \neq \emptyset$ then
8              if DetectCycles($p_{interm}$,$Groups[i][j]$) =false then
                   /* Detects cycles created due to including the
                      transition group $Groups$[i][j]                 */
9                  $P_{r_{ss}} \leftarrow P_{r_{ss}} \cup Groups[i][j]$;
10                 $p_{interm} \leftarrow$ Unfold($P_{r_{ss}}$, $K$);
11                 $Deadlocks \leftarrow$
                   $\{s : (s \in \neg I) \wedge (\forall g, s_1 : (s, s_1) \in g : g \notin p_{interm})\}$;
12                 if $Deadlocks = \emptyset$ then
13                     success $\leftarrow$ true;
14                     return;
15 success $\leftarrow$ false;
16 return;
```

Figure 4.10: The AddConvergence routine.

Theorem 4.3.1 (Soundness)**.** AddConvergence *is sound and has a polynomial-time complexity in* $|S_p|$.

Proof. If AddConvergence declares success by returning a solution, then its exit point is Line 14 and this is only possible if *Deadlocks*$=\emptyset$. In Line 11, *Deadlocks* is assigned the set of deadlocks of p_{interm}. Lines 9-11 are executed *if and only if* the inclusion of the last candidate transition group does not cause non-progress cycles. Consequently, p_{interm} has no cycles and no deadlocks. Thus, $P_{r_{ss}}$ represents a symmetric protocol that has no deadlocks and is cycle-free in $\neg I$; i.e., $P_{r_{ss}}$ represents a self-stabilizing symmetric protocol (see Proposition 2.1.1).

The nested for-loop in AddConvergence runs for at most the number of all possible groups in a process. Kulkarni and Arora demonstrate that the total number of groups is polynomial in $|S_p|$ [13]. Moreover, the time complexity of the cycle detection algorithm in [32] is polynomial in $|S_p|$ too. $\square$

TC Example. $P_{r_{ws}}$ has 30 candidate RTGs partitioned into 2 ranks. We consider a possible order that ShuffleGroups generates.

Groups[1]=$\{000 \rightarrow 010, 000 \rightarrow 020, 111 \rightarrow 101, 111 \rightarrow 121, 222 \rightarrow 212, 222 \rightarrow 202, 002 \rightarrow 012, 110 \rightarrow 120, 221 \rightarrow 201, 001 \rightarrow 021, 112 \rightarrow 102, 220 \rightarrow 210, 011 \rightarrow 021, 122 \rightarrow 102, 200 \rightarrow 210, 022 \rightarrow 012, 100 \rightarrow 120, 211 \rightarrow 201\}$

Groups[2]=$\{011 \rightarrow 001, 122 \rightarrow 112, 200 \rightarrow 220, 002 \rightarrow 022, 110 \rightarrow 100, 221 \rightarrow 211, 001 \rightarrow 011, 112 \rightarrow 122, 220 \rightarrow 200, 022 \rightarrow 002, 100 \rightarrow 110, 211 \rightarrow 221\}$

Groups[1] and *Groups*[2] are the input to AddConvergence in the same order presented. We trace how AddConvergence includes the candidate RTGs to obtain $P_{r_{ss}}$ for $K = 7$. For the TC input protocol, *Deadlocks*= $\neg I_{color}$. For $i = 1$, AddConvergence includes $000 \rightarrow 010$ for $j = 1$. AddConvergence leaves out $000 \rightarrow 020$ for $j = 2$ because this RTG resolves no additional deadlocks. The included RTGs within the for-loop (Lines 4-14) are $P_{r_{ss}} = \{000 \rightarrow 010, 111 \rightarrow 101, 222 \rightarrow 212, 002 \rightarrow 012, 110 \rightarrow 120, 221 \rightarrow 201, 001 \rightarrow 021, 112 \rightarrow 102, 220 \rightarrow 210\}$. None of the RTGs in $P_{r_{ss}}$ forms non-progress cycles; all of their transitions are rank decreasing. This is a property of TC that is not necessarily valid for other input protocols. Moreover, 9 RTGs of *Groups*[1] are not included in $P_{r_{ss}}$ because they do not resolve additional deadlocks. The omitted transitions depend on the order in *Groups*[1]: the last 6 RTGs are skipped because all deadlocks are resolved when $j = 12$. RTGs at $j = 2$, $j = 4$ and $j = 6$ are also not included in p_{interm} because they resolve the same sets of deadlock states resolved by RTGs at $j = 1$, $j = 3$ and $j = 5$, respectively. Hence, for the given *Groups*[], AddConvergence obtains a solution at $i = 1$ and $j = 12$. $P_{r_{ss}}$ is represented by the action $(c_{r-1} = c_r) \rightarrow c_r :=$ *other*(c_{r-1}, c_{r+1}). $\lhd$

4.4 Case Studies

In this section, we present some of the case studies that we have conducted with pSTSyn. The implementation of pSTSyn is in C++ and we use Binary Decision Diagrams (BDDs) [19] to represent protocols in memory. We have deployed pSTSyn on a computer cluster with 24 nodes. Each node is an Intel(R) Xeon(R) CPU 5120 @ 1.86GHz (4 cores) with 4GB RAM and the Linux operating system (kernel 2.6.9-42.ELsmp). The Looper threads are created using the MPICH2-1.3.2p1 run-time system. Section 4.4.1 discusses how swarm synthesis simultaneously generates multiple solutions of a Maximal Matching protocol that would have been impossible to generate with existing automated approaches. Section 4.4.2 presents a self-stabilizing agreement protocol. We present a stabilizing solution to a Leader Election protocol for 5 processes and three alternative solutions to Maximal Matching in [33].

4.4.1 Maximal Matching

The Maximal Matching (MM) protocol (adapted from [15]) has $K > 3$ processes $\{P_0, \cdots, P_{K-1}\}$ located on a ring, where $P_{(i-1)}$ and $P_{(i+1)}$ are respectively the left and right neighbors of P_i, and addition and subtraction are in modulo K ($0 \leq i < K$). Each process P_i has a variable m_i with a domain of three values {left, right, self} representing whether or not P_i points to its left neighbor, right neighbor or itself. Process P_i is *matched* with its left neighbor $P_{(i-1)}$ (respectively, right neighbor $P_{(i+1)}$) if and only if $m_i =$ left and $m_{(i-1)} =$ right (respectively, $m_i =$ right and $m_{(i+1)} =$ left). Each process P_i can read the variables of its left and right neighbors. P_i is also allowed to read and write its own variable m_i. The non-stabilizing protocol is empty; i.e., does not include any transitions. Our objective is to automatically generate a strongly stabilizing protocol that converges to a state in $I_{MM} = \forall i : 0 \leq i \leq K-1 : LC_i$, where LC_i is a local state predicate of process P_i as follows

$$\begin{aligned}&(m_i = \text{left} \Rightarrow m_{(i-1)} = \text{right}) \wedge (m_i = \text{right} \Rightarrow m_{(i+1)} = \text{left})\\ &\qquad \wedge (m_i = \text{self} \Rightarrow (m_{(i-1)} = \text{left} \ \wedge m_{(i+1)} = \text{right}))\end{aligned}$$

In a state in I_{MM}, each process is in one of these states: (i) matched with its right neighbor, (ii) matched with left neighbor or (iii) points to itself, and its right neighbor points to right and its left neighbor points to left. The MM protocol is silent in I_{MM} in that after stabilizing to I_{MM}, the actions of the synthesized MM protocol should no longer be enabled. Below actions illustrate a new solution for the MM problem synthesized by pSTSyn. For example, the first action means that a process sets m_i to self if it is not pointing to itself, its left neighbor points to left and its right neighbor points to right. Other actions can be interpreted similarly.

$$\begin{array}{lll}(\mathrm{m}_{i-1}=\text{left}) \wedge (\mathrm{m}_i \neq \text{self}) \wedge (\mathrm{m}_{i+1}=\text{right}) & \longrightarrow & \mathrm{m}_i:=\text{self}\\ (\mathrm{m}_{i-1} \neq \text{left}) \wedge (\mathrm{m}_i=\text{self}) \wedge (\mathrm{m}_{i+1} = \text{self}) & \longrightarrow & \mathrm{m}_i:=\text{left}\\ (\mathrm{m}_{i-1}=\text{right}) \wedge (\mathrm{m}_i \neq \text{left}) \wedge (\mathrm{m}_{i+1}=\text{right}) & \longrightarrow & \mathrm{m}_i:=\text{left}\\ (\mathrm{m}_{i-1}=\text{right}) \wedge (\mathrm{m}_i=\text{right}) \wedge (\mathrm{m}_{i+1} \neq \text{right}) & \longrightarrow & \mathrm{m}_i:=\text{left}\\ (\mathrm{m}_i=\text{self}) \wedge (\mathrm{m}_{i+1}=\text{left}) & \longrightarrow & \mathrm{m}_i:=\text{right}\end{array}$$

$(\mathrm{m}_{i-1} \neq \text{right}) \wedge (\mathrm{m}_i = \text{left}) \wedge (\mathrm{m}_{i+1} \neq \text{right}) \quad \longrightarrow \quad \mathrm{m}_i := \text{right}$
$(\mathrm{m}_{i-1} = \text{left}) \wedge (\mathrm{m}_i \neq \text{right}) \wedge (\mathrm{m}_{i+1} \neq \text{right}) \quad \longrightarrow \quad \mathrm{m}_i := \text{right}$

The diversity of solutions we have generated demonstrates the effectiveness of exploiting randomization and parallelism in automating the design of self-stabilization. We have synthesized the matching protocol for $5 \leq K \leq 14$. Figures 4.11 and 4.12 respectively represent the time and space costs of synthesis, where memory costs are represented in terms of BDD nodes. *Synthesis time* captures the time spent in Phases 2 and 3 excluding the cycle detection time: we illustrate cycle detection time by a separate curve.

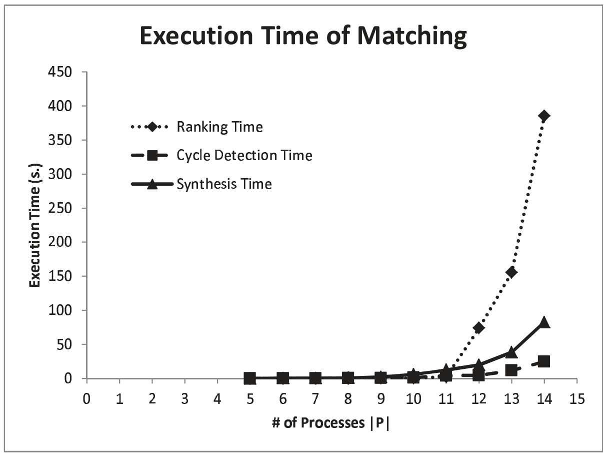


Figure 4.11: Time spent for adding convergence to matching versus the number of processes

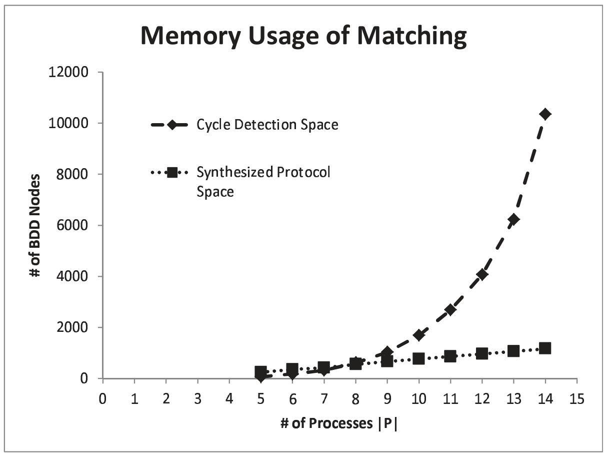


Figure 4.12: Space usage for adding convergence to matching versus the number of processes

Notice that pSTSyn synthesizes a self-stabilizing version of MM in less than 10 minutes. While the ranking time is significant, it is performed only once as a preprocessing phase.

Figure 4.12 demonstrates that as we scale up the number of processes the space cost of cycle detection becomes a bottleneck due to the large size of BDDs. We are currently working on more efficient cycle detection methods. The number of ranks in MM protocol increases linearly with the number of processes K. Extrapolating our experimental results, we obtain the relationship *NumRanks*$= 2(\lfloor \frac{K-1}{3} \rfloor + 1)$ where $3 \leq K \leq 14$. The representative process of the synthesized solution presented here has 17 RTGs. Due to symmetry, the number of RTGs is independent of K.

4.4.2 Agreement

We present a symmetric protocol on a bidirectional ring where the processes need to agree on a specific value: from an initial arbitrary state, all the variables should eventually be equal to one another. The ring has K processes P_i $(0 \leq i \leq K-1)$. Each process P_i can write its local variable a_i where $a_i \in \{0, \cdots, |D|-1\}$. Each process P_i can read its left a_{i-1}, right a_{i+1} and its own variable a_i (operations on process and variable indices are modulo K). The set of legitimate states is $I_{agreement} = \bigwedge_{i=1}^{i=K-1}(a_{i-1} = a_i)$. The protocol is not locally correctable: the establishment of $a_{i-1} = a_i$ by an action of P_i can invalidate $a_i = a_{i+1}$. This fact complicates the search for a solution with similar processes. Nonetheless, pSTSyn generates the following protocol with 6 processes from an empty protocol. In this case, our solution is generalizable for any K.

$$(a_i > a_{i-1}) \vee (a_i > a_{i+1}) \longrightarrow a_i := \mathsf{min}(a_{i-1}, a_{i+1})$$

Figures 4.13 and 4.14 respectively illustrate the impact of the domain size of a_i values on time (respectively, space) efficiency of synthesis. ($|P|$ denotes the number of processes.) Notice that synthesis time grows slowly (note the scale of the y axis in Figure 4.13), whereas memory costs increase exponentially as we increase the domain of a_i. The reason behind this is that the size of transition groups (respectively, the size of BDDs representing them) increases, thereby raising the number of cycles and the time needed for cycle detection. The number of ranks in the agreement protocol depends on the number of processes K and the domain size $|D|$. For example, when $|D| = 2$, *NumRanks*$= \lfloor \frac{K}{2} + 1 \rfloor$. The number of RTGs of the representative process depends only on $|D|$ and is independent of K due to symmetry. For $|D| = 3$, the number of RTGs is 13. In general, the number of RTGs is in $O(|D|^3)$ because $|R_r| = 3$. The intuition behind this reasoning is that the number of local states of a process P_r is in the order of the number of all possible valuations of the variables in R_r.

Keeping the domain size constant (equal to 3), we can scale up the synthesis up to 22 processes (see Figures 4.15 and 4.16). We observe that the super linear jump in the synthesis time is due to the thrashing phenomenon when the BDD sizes go beyond a threshold and secondary memory has to be used. ($|D|$ denotes the domain size of a_i.)

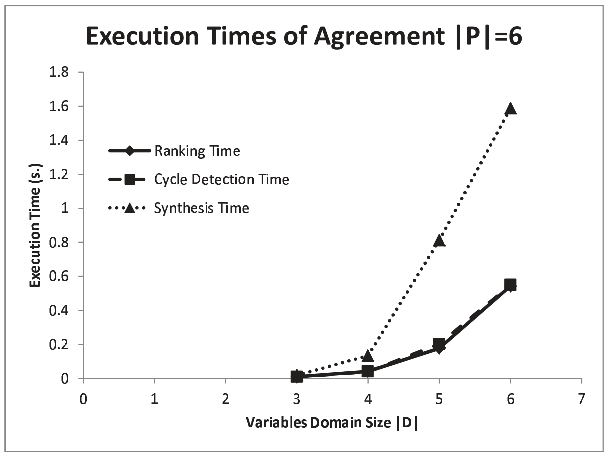


Figure 4.13: Time spent for adding convergence to agreement versus the size of the variable domain

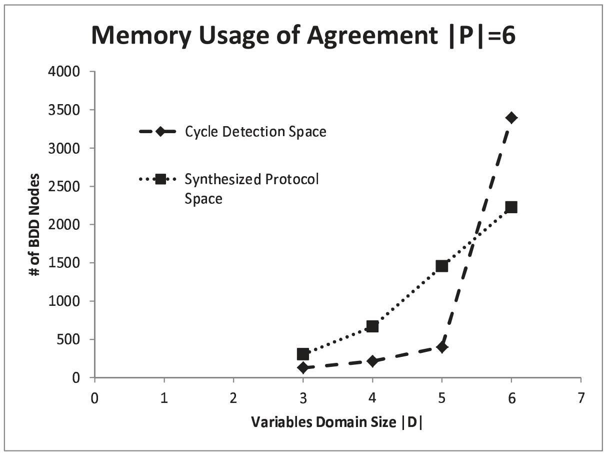


Figure 4.14: Space usage for adding convergence to agreement versus the size of the variable domain

Scalability vs. generalization. In this section, we presented our experimental results just to provide a measure as to how much time (respectively, space) is required for the synthesis of self-stabilizing protocols with a few processes. While scaling up the synthesis is useful, scalability is not a high priority objective for us; rather our approach is based on a paradigm of *synthesize in small scale and generalize*. We are currently working on a family of synthesis methods that generate p_{ss} for small values of K (e.g., $K \leq 20$) such that p_{ss} is *generalizable*; i.e., p_{ss} preserves its convergence property for *any* K.

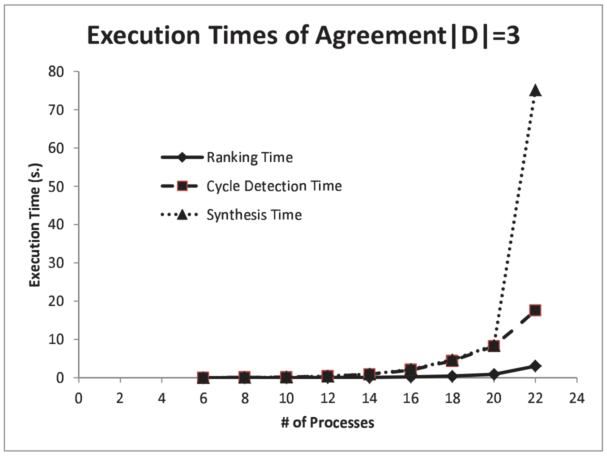


Figure 4.15: Time spent for adding convergence to agreement versus the number of processes

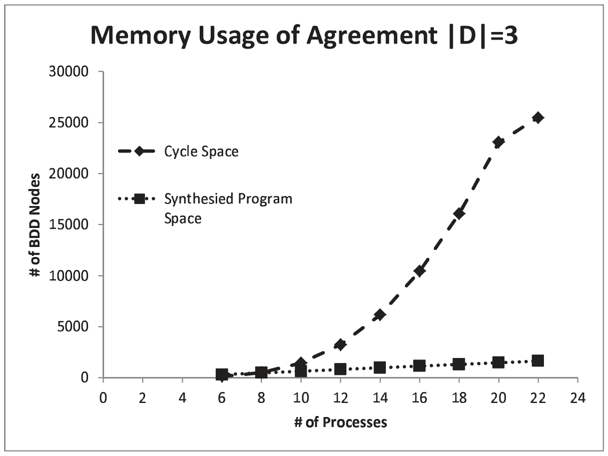


Figure 4.16: Space usage for adding convergence to agreement versus the number of processes

4.5 Summary and Extensions

We presented a swarm synthesis method that exploits randomization and parallelism to add convergence to non-stabilizing symmetric protocols. While the problem of adding convergence to non-stabilizing protocols is known to be in NP, we are not aware of any algorithm that adds convergence in polynomial-time (in protocol state space) [13]. We conjecture that adding convergence is most likely an NP-hard problem due to the exponential number of the combinations of recovery actions that could resolve deadlocks without creating non-progress cycles in the set of illegitimate states of the protocol.

Existing methods [10], [34] for adding convergence perform a deterministic search in the state space of non-stabilizing protocols to synthesize necessary convergence actions. However, such techniques often search only a part of the problem space due to their deterministic nature, thereby resulting in premature failures in finding a self-stabilizing protocol. Moreover, swarm verification methods [35] only verify the correctness of an existing protocol rather than synthesizing a correct protocol. The main contribution of the proposed approach is to increase the likelihood of success in automated design of self-stabilization by exploiting randomization and parallelism. We have implemented the proposed approach in a software tool, called pSTSyn, that has automatically generated new solutions for several well-known protocols such as maximal matching, graph coloring, agreement and leader election on a ring. To the best of our knowledge, pSTSyn is the first tool that enables swarm synthesis of convergence for symmetric protocols.

There are several extensions to this work that we would like to investigate. First, we plan to devise a method for swarm synthesis of convergence for asymmetric protocols. Second, we will investigate how pSTSyn can be used for adding convergence to protocols with dynamic topologies (e.g., overlay networks). The design of protocols with dynamic topologies is especially challenging as the locality of each process may change, thereby changing the transition groups that are created due to different scope of readability for processes. As a result, in a dynamic network, for each configuration of the network topology we have a distinct set of transition groups that form the transition system of the protocol. Parallelism can be especially beneficial in tackling this problem.

Chapter 5

Local Reasoning for Global Convergence of Parameterized Rings[1]

This chapter presents a method that can generate self-stabilizing *parameterized* protocols; i.e., correct for arbitrary number of finite-state processes. Specifically, we present *necessary and sufficient* conditions specified in the local state space of the representative process of parameterized rings for deadlock-freedom in their global state space. Moreover, we introduce sufficient conditions that guarantee livelock-freedom in arbitrary-sized unidirectional rings. We illustrate the proposed approach in the context of several classic examples including a maximal matching protocol and an agreement protocol. More importantly, the proposed method lays the foundation of an approach for automated design of global convergence in the local state space of the representative process.

5.1 Introduction

In this chapter, we present a local reasoning method for the design of global convergence in parameterized protocols with the ring topology. Recall that in a parameterized protocol, the code of each process is instantiated from the code of a *representative process* by variable substitution. The entire reasoning in the proposed method is performed in the local state space of the representative process. To ensure convergence to a set of legitimate states I (specified as the conjunction of a set of local constraints), starting from any state $s \in \neg I$, every execution of the protocol from s should eventually reach a state in I. Thus, a protocol

[1]This chapter is adapted from our own publication [12] in the proceedings of the International Conference on Distributed Computing Systems, June 2012. Reprinted with permission, please see the supplementary document to this dissertation for the copyright notice.

must ensure that it is deadlock-free in $\neg I$. Moreover, there must be no cycles formed by processes' actions such that all states of the cycle belong to $\neg I$; i.e., *livelock*-freedom. For a parameterized protocol, deadlock/livelock-freedom properties must hold for any number of processes in the ring. To address this problem, we present *necessary and sufficient* conditions specified in the local state space of the representative process for deadlock-freedom in the global state space of the ring (with an arbitrary number of processes). Moreover, we introduce sufficient conditions that guarantee livelock-freedom in arbitrary-sized unidirectional rings. Our sufficient conditions are weaker than what is proposed in existing methods. For instance, as demonstrated in Section 5.3, it is unclear how existing methods [36], [37] can be used to design convergence for an agreement protocol. To validate our necessary and sufficient conditions, we apply them for the design of several parameterized self-stabilizing protocols on a ring including maximal matching and agreement protocols.

Organization. We present a necessary and sufficient condition for deadlock-freedom in parameterized rings in Section 5.2. In Section 5.3, we introduce the notion of a local transition graph and illustrate how we use it to reason about non-terminating computations in unidirectional rings. We make concluding remarks and outline future work in Section 5.5.

5.2 Deadlock-Freedom

In this section, we present a necessary and sufficient condition for the deadlock-freedom of bidirectional parameterized rings. Specifically, we define a relation between the local states of the representative process P_r. This relation captures the way the local states of a process are related with the local states of its neighboring processes. Using this relation, we present a necessary and sufficient condition defined in the local state space of P_r for global deadlock-freedom of p.

The Right Continuation Relation for Rings. Due to the locality of each process, a local state s_i^l of process P_i restricts the allowable set of local states for each successor P_j of P_i. P_j *is a successor* of P_i (respectively, P_i is a predecessor of P_j) if and only if $W_i \cap R_j \neq \emptyset$. In a bidirectional ring of size K, $P_{(i+1)\mathsf{mod}K}$ and $P_{(i-1)\mathsf{mod}K}$ are right and left successors of P_i, respectively. As such, the *right* (respectively, *left*) *continuation* of a local state s_i^l of P_i is a local state s_{i+1}^l of P_{i+1} (respectively, s_{i-1}^l of P_{i-1}) such that for every $x \in R_i \cap R_{i+1}, x(s_i^l) = x(s_{i+1}^l)$ (respectively, $x \in R_i \cap R_{i-1}, x(s_i^l) = x(s_{i-1}^l)$)[2]; i.e., a continuation of a local state in P_r is a possible local state of P_r's successor. Notice that for a unidirectional ring, we only define a right continuation relation.

[2]Addition and subtractions of indices are in modulo K.

Definition 5.2.1. *A directed Right Continuation Graph (RCG$_p$) of a ring is a pair (V_r, S_R) such that:*

1. V_r is a set of vertices representing local states of the representative process P_r.

2. S_R denotes the set of arcs of RCG$_p$, called *s-arcs*, where $S_R = \{(s^l_1, s^l_2) \in V_r \times V_r : \forall x \in R_r \cap R_{r+1} : x(s^l_1) = x(s^l_2)$ and P_{r+1} is a right successor of $P_r\}$. The left continuation relation, denoted S_L, can be defined symmetrically. Thus, S_R is sufficient to define the continuation relation for bidirectional rings.

Our definition of continuation relation naturally extends to network topologies other than rings. For instance, we construct the continuation graph of a tree from the locality of a non-root process that includes the writable variables of its parent, itself and its children.

Example 5.2.1. In maximal matching over a bidirectional ring, all processes are similar. P_r has P_{r+1} as the right successor and P_{r-1} as the right predecessor. $R_r = \{m_{r-1}, m_r, m_{r+1}\}$, $W_r = \{m_r\}$ and $M(K) = K$. Moreover, $W_r \cap R_{r+1} = W_r \cap R_{r-1} = \{m_r\}$. $R_r \cap R_{r+1} = \{m_r, m_{r+1}\}$. $D_r = \{$left, right, self$\}$ contains the values of m_r meaning that P_r points to its predecessor, successor or itself, respectively. We illustrate the right continuation graph over the local state space of P_r in Figure 5.1. The three values inside each vertex represent the values of m_{r-1}, m_r and m_{r+1} in the corresponding local state of P_r. For example, when P_r is in the local state rrs, its right successor can be in either one of the local states rss, rsr or rsl. That is why there are three outgoing s-arcs from rrs to rss, rsr and rsl, respectively. The set of local legitimate states LC_r is defined by the Boolean expression ($m_r =$ right $\wedge m_{r+1} =$ left) $\vee$ ($m_{r-1} =$ right $\wedge m_r =$ left) $\vee$ ($m_{r-1} =$ left $\wedge m_r =$ self $\wedge m_{r+1} =$ right). The gray vertices in Figure 5.1 denote the local legitimate states.

Due to symmetry, local state spaces of all similar processes are captured by P_r's local state space S^l_r. We observe that any directed cycle of length L in Figure 5.1 represents a possible valuation of local states to a ring of processes of size $k \times L$ (k a positive integer). For instance, the cycle between the local state rsr and srs represents the global state of any ring of size $2k$ where if m_i =right then m_{i+1} =self and vice versa ($0 \leq i \leq 2k - 1$). $\lhd$

Theorem 5.2.2 (Deadlock-Freedom in Parametrized Rings). *A parameterized protocol $p(K)$ over a ring topology is deadlock-free outside $I(K)$ for every K if and only if the induced subgraph*[3] *of RCG$_p$ over local deadlocks has no directed cycles containing a local state/vertex in $\neg LC_r$.*

[3]An induced subgraph $G' = (V', E')$ of a directed graph $G = (V, E)$ is such that $V' \subset V$, E' is the maximum subset of E such that the source and target vertices of every arc in E' are in V'.

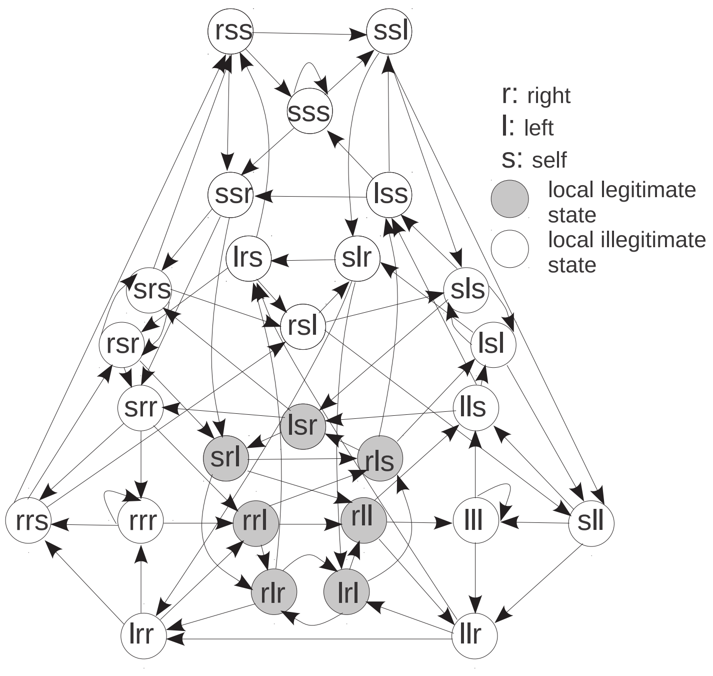


Figure 5.1: Continuation relation over all local states of Maximal Matching

Proof. $\Rightarrow$: Let $p(K)$ be a parameterized protocol that is deadlock-free outside $I(K)$ for every K. By contradiction, assume that RCG$_p$ has a directed cycle over its local deadlocks $C = \{s^l_0, s^l_1, \cdots, s^l_{n-1}\}$ and for some $0 \leq j \leq n-1$, $s^l_j \notin LC_r$. By definition of RCG$_p$, s^l_{i+1} is a right continuation of s^l_i for every $0 \leq i < n-1$ and s^l_0 is a right continuation of s^l_{n-1}. By assigning to P_i the local state s^l_i for $0 \leq i \leq n-1$, we construct a ring R of size $k \times n$ (k is a positive integer) in which every P_i is locally deadlocked. Moreover, for some j, P_j is in a local state $s^l_j \notin LC_r$. Because $I(K)$ is locally conjunctive, the corresponding global state of R is a global deadlock outside $I(K)$. This contradicts our premise.

$\Leftarrow$: Let RCG$_p$'s induced subgraph over local deadlocks have no directed cycles with a local state $s^l_j \notin LC_r$. By contradiction, assume that for some K, $p(K)$ is globally deadlocked outside $I(K)$. It follows that every process P_i of $p(K)$ is in a local deadlock s^l_{di} ($0 \leq i \leq K-1$) among which there exists a local deadlock $s^l_{dj} \notin LC_r$. By definition of the continuation relation, RCG$_p$ captures *every possible* right continuation of every local state of P_r. Hence, for every $0 \leq i \leq K-1$, $(s^l_{di}, s^l_{di+1}) \in$ RCG$_p$. Since $p(K)$ is a ring of local deadlocks, the induced subgraph of RCG$_p$ over local deadlocks should have a directed cycle containing s^l_{dj}. $\square$

We illustrate the application of Theorem 5.2.2 by the following examples.

Example 5.2.2 (Deadlock-Free Generalizable Maximal Matching)**.** We consider the following parameterized protocol for maximal-matching on a bidirectional ring. We have automatically synthesized this protocol for $K = 6$ using the STabilization Synthesizer tool (STSyn) [10].

$$\begin{array}{ll}
m_{r-1} = \text{left} \wedge m_r \neq \text{self} \wedge m_{r+1} = \text{right} & \rightarrow m_r := \text{self} \\
m_{r-1} = \text{self} \wedge m_r = \text{self} \wedge m_{r+1} = \text{self} & \rightarrow m_r := \text{right} \mid \text{left} \\
m_{r-1} = \text{right} \wedge m_r = \text{self} & \rightarrow m_r := \text{left} \\
m_r = \text{self} \wedge m_{r+1} = \text{left} & \rightarrow m_r := \text{right} \\
m_{r-1} = \text{right} \wedge m_r = \text{right} \wedge m_{r+1} \neq \text{left} & \rightarrow m_r := \text{left} \\
m_{r-1} \neq \text{right} \wedge m_r = \text{left} \wedge m_{r+1} = \text{left} & \rightarrow m_r := \text{right} \\
m_{r-1} = \text{self} \wedge m_r \neq \text{left} \wedge m_{r+1} = \text{right} & \rightarrow m_r := \text{left} \\
m_{r-1} = \text{left} \wedge m_r \neq \text{right} \wedge m_{r+1} = \text{self} & \rightarrow m_r := \text{right}
\end{array}$$

Figure 5.2 illustrates the RCG$_p$ of Example 5.2.2 induced over its local deadlocks. As we can see, there are no directed cycles that include local illegitimate states. This proves the deadlock freedom of the parametrized maximal matching protocol in Example 5.2.2. To gain more confidence, we have model-checked this protocol for $6 \leq K \leq 10$ and found no deadlocks. $\triangleleft$

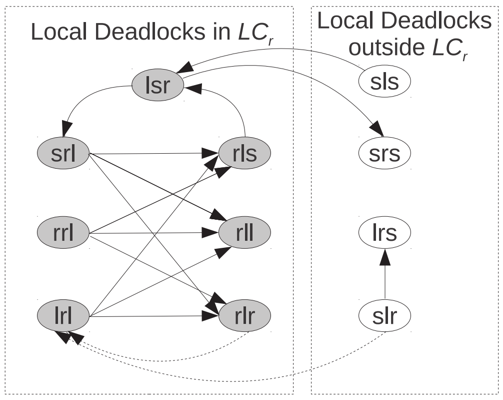


Figure 5.2: Continuation Relation over local deadlocks of Example 5.2.2

Example 5.2.3 (Non-generalizable Maximal Matching)**.** We automatically synthesized the following protocol that stabilizes for 5 processes and has deadlocks for a ring of size 6. We illustrate how the right continuation relation helps us reason about global deadlocks.

$$\begin{array}{ll}
m_{r-1} = \text{left} \wedge m_r \neq \text{self} \wedge m_{r+1} = \text{right} & \rightarrow m_r := \text{self} \\
m_{r-1} = \text{right} \wedge m_r = \text{self} \wedge m_{r+1} = \text{left} & \rightarrow m_r := \text{right} \\
m_{r-1} = \text{self} \wedge m_r = \text{self} \wedge m_{r+1} = \text{self} & \rightarrow m_r := \text{right}
\end{array}$$

m_{r-1} =right $\wedge m_r$ =right $\wedge m_{r+1} = left \rightarrow$ m_r :=left
m_{r-1} =self $\wedge m_r$ =self $\wedge m_{r+1}$ =right $\rightarrow$ m_r :=left
m_{r-1} =right $\wedge m_r$ $\neq$left $\wedge m_{r+1} \neq$ left $\rightarrow$ m_r :=left
m_{r-1} $\neq$right $\wedge m_r$ $\neq$right $\wedge m_{r+1}$ =left $\rightarrow$ m_r :=right

Figure 5.3 illustrates a subgraph of the RCG in Figure 5.1 that has been induced over the local deadlocks of the maximal matching protocol presented in Example 5.2.3. There are only two directed cycles having local illegitimate deadlocks in Figure 5.3. Both cycles include the local state ⟨*left,left,self*⟩. The first directed cycle has length 4:⟨lls, lsr, srl, rll⟩ and represents global deadlocks $\langle$left,self,right,left $\rangle^k$ in rings whose size is a multiple of 4. The second directed cycle has length 6. $\lhd$

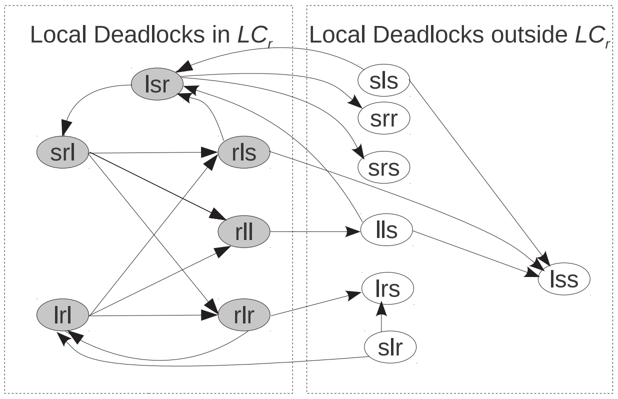


Figure 5.3: Continuation Relation over local deadlocks of Example 5.2.3

5.3 Livelock-Freedom

In this section, we focus on the following problem: *For a parameterized protocol $p(K)$ with a unidirectional ring topology and a conjunctive predicate $I(K)$, determine whether $p(K)$ is livelock-free outside $I(K)$ for all K without exploring the global state space of $p(K)$*. For simplicity, we establish the following assumptions:

1. Every process P_i is *self-terminating*. As such, every sequence of local transitions of P_i terminates in a local deadlock.
2. No process P_i has *self-enabling actions*. An action A: $\mathsf{guard}_A \rightarrow \mathsf{statement}_A$ is self-enabling if there exists a global transition $(s_g, s'_g) \in A$ such that $s_g \in \mathsf{guard}_A$

and $s'_g \in \mathsf{guard}_A$. Likewise, an action B is *self-disabling* if and only if B disables guard_B after executing $\mathsf{statement}_B$.

Assumption 2 is at no loss of protocol's generality because self-enabling actions can be transformed into self-disabling without adding either deadlocks or livelocks in $\neg I$. If P_i is self enabling, then for some local state s^l_{i1} of P_i, there exists a sequence of local states $\langle s^l_{i1}, s^l_{i2}, \cdots, s^l_{ik} \rangle$ of P_i such that $(s^l_{ij}, s^l_{i(j+1)})$ is a local transition of P_i $(1 \leq j \leq k-1)$ and s^l_{ik} is a local deadlock (Item 1 prohibits local non-terminating computations). We substitute every local transition $(s^l_{ij}, s^l_{i(j+1)})$, where $1 \leq j \leq k-1$, with (s^l_{ij}, s^l_{ik}). This substitution renders P_i self-disabling and preserves reachability to s^l_{ik} from every local state s^l_{ij}. Moreover, it does not introduce new local deadlock states.

Lemma 5.3.1 (Enablement Propagation)**.** *Let C=$\ll c_1, \cdots, c_k, \cdots \gg$ be a computation of a parameterized protocol $p(K)$ on a unidirectional ring of size K. $\forall k > 1$:* **If** $(\exists j$: *P_j is* enabled *in c_k* **and** *P_j is* disabled *in c_{k-1})* **then** *$\exists i$: P_j is the successor of P_i* **and** $(c_{k-1}, c_k) \in g^K(\delta_i)$.

Proof. The fact that P_j is not enabled in c_{k-1} and enabled in c_k means that (c_{k-1}, c_k) writes a variable $x \in R_j$. Then $x \in W_i$ of some process such that $(c_{k-1}, c_k) \in g(\delta_i)$. It follows that $\{x\} \subset W_i \cap R_j$, hence P_j is a successor of P_i. ☐

The significance of Lemma 5.3.1 is to illustrate that in the course of a program computation, a disabled process is enabled only by the action of its predecessor. In other words, a process can only pass *enablement* to its successor. To represent the propagation of enablement in the local state space of the representative process P_r, we augment the RCG with the local transitions of P_r, called *t-arcs*. Thus, the augmented RCG has two types of arcs: *s-arcs* that represent the continuation relation and the transfer of control to possible local states of successor processes, and t-arcs representing local transitions of P_r. We call the new RCG, the Local Transition Graph (LTG).

Definition 5.3.2. The Local Transition Graph (LTG) of p is a triplet LTG$_p = (V, T, S)$. V is a set of vertices representing the local state space of P_r, T is the set of t-arcs and S captures the set of s-arcs. We construct LTG$_p$ as follows:

1. For the representative process P_r, assign a vertex in V corresponding to each local state of P_r.

2. In V, add a t-arc (v_r, v'_r) to T to represent a local transition of P_r.

3. For every local state/vertex in V, add an s-arc (v_r, v_r') to S if v_r represents a local state of P_r and v_r' represents a possible local state of the successor of P_r.

Example 5.3.1 (Binary Agreement). Consider a binary agreement protocol on a unidirectional ring such that $M(K) = K$, $R_r = \{x_{r-1}, x_r\}$, $W_r = \{x_r\}$, $D_r = \{0, 1\}$. The representative process P_r has the following actions:

$$t_{10}{}^{r} : x_{r-1} = 0 \wedge x_r = 1 \rightarrow x_r := 0$$
$$t_{01}{}^{r} : x_{r-1} = 1 \wedge x_r = 0 \rightarrow x_r := 1$$

Intuitively, P_r sets x_r to x_{r-1} whenever $x_r \neq x_{r-1}$. Thus, a local legitimate state is such that $x_r = x_{r-1}$; i.e., the protocol is in a global legitimate state when all variables have equal values. In Figure 5.4, the left hand-side graph represents the continuation graph of the agreement (denoted RCG$_p$) and the right hand-side graph is the LTG of agreement (denoted LTG$_p$).

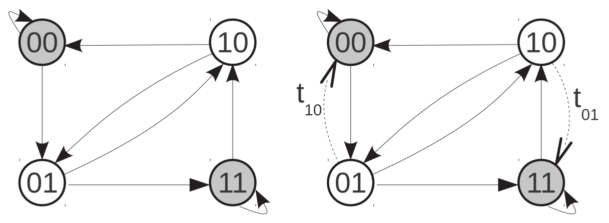


Figure 5.4: RCG$_p$ and LTG$_p$ of the Agreement protocol

Definition 5.3.3 (Collision). *Let $p(K)$ be a parameterized protocol with a unidirectional ring topology and P_j be the successor of P_i. Let s_i^l and s_j^l be local states where P_i and P_j are both enabled, respectively. A* collision *is an execution of any local transition of P_i enabled at s_i^l.*

Lemma 5.3.4 (Enablement Conservation in a Unidirectional Ring). *Let $p(K)$ be a parameterized protocol on a unidirectional ring of size K. L is a livelock of $p(K)$* if and only if *in every global state of L, the number of enabled processes is the same and greater than zero.*

Proof. Only if: Let s be some global state of L. Assume the number of enabled processes at s is $|E|$. From Assumption 2, every local transition of any process P_i disables P_i. Since every process in a unidirectional ring has one successor, a local transition of any enabled process will not increase $|E|$. It follows that $|E|$ can either stay constant or decrease. However, if an execution of a transition at s decreased $|E|$ to $|E| - 1$, then since $|E| - 1$ cannot increase in subsequent transitions, s cannot be re-encountered in L following s. Therefore, s cannot be in a livelock L. Consequently, $|E|$ is constant in any livelock on a unidirectional ring.

If: In every global state of L, there exists and enabled process that executes. Since $p(K)$ has a finite number of states, L is a livelock. $\square$

Corollary 5.3.5 (Absence of Collisions in Livelocks in Unidirectional Rings)**.** *If L is a livelock on a unidirectional ring then for every global transition t in L, there is no collision t^l such that $t \in g(t^l)$.*

Proof. In a unidirectional ring, a collision decreases the number of enabled processes by 1. This is in contradiction with Lemma 5.3.4. $\square$

Corollary 5.3.6 (Insensitivity to Weak Fairness)**.** *Let $p(K)$ be a parameterized protocol on a unidirectional ring of size K. If L is a livelock of $p(K)$ then there is no continuously enabled process in L.*

Proof. Let s_g be a global state of L where every process of $p(K)$ is enabled. Hence, any execution of any enabled process will cause a collision. From Corollary 5.3.5, s_g cannot be in L. It follows that in every global state of L, there exists a disabled process. According to Lemmas 5.3.1 and 5.3.4, a constant number of enablements propagate along the arcs of the unidirectional ring. Hence, disabled local states propagate in the opposite direction. Thus, every process in the ring will eventually be disabled in the reverse direction. $\square$

Corollary 5.3.6 implies that the assumption of the existence of a weakly fair scheduler[4] does not simplify the design of livelock-freedom in unidirectional rings because no process stays continuously enabled in a livelock.

Effect of Fairness and Execution Semantics on Livelocks. In a livelock L of a ring $p(K)$, every process participates in L; i.e., every process executes infinitely often. Therefore, every livelock is an *impartially fair* computation.[5] Moreover, Lemma 5.3.10 asserts that, in a livelock L in a unidirectional ring, no matter in what order independent local transitions execute, the resultant global state is the same. It follows that the simultaneous execution of two local transitions $t^{r\oplus 1}$ and t^r is equivalent to the sequential execution $\ll t^{r\oplus 1}, t^r \gg$, in $P_{r\oplus 1}$ then P_r, respectively. Thus, $T_p(K)$ also captures the behaviors of livelocks in which multiple processes execute simultaneously, provided that $|E| < K$. The case of a fully synchronous livelock; i.e., when $|E| = K$, is not captured by $T_p(K)$ since every sequential propagation of enablements has a collision. However, Lemma 5.3.4 still holds for a fully synchronous livelock. Thus, our results hold in a parameterized ring for the most general execution semantics and fairness assumptions.

[4]A weakly fair scheduler infinitely often executes any action that is continuously enabled.
[5]A computation C is impartially fair if every process infinitely often participates in C.

Lemma 5.3.7 (Local Illegitimacy)**.** *Let L be a livelock of a parameterized protocol $p(K)$ on a unidirectional ring. Then, for every global state of L there exists a process P_i in an illegitimate local state.*

Proof. Every global state of L is in $\neg I$. Since I is locally conjunctive, for every global state of L, there exists LC_i that evaluated to false by the local state of P_i. In other words, there exists a process P_i whose local state is in $\neg LC_i$. $\square$

In every global state of a livelock L, there exists an enabled process P_i and some process P_j in an illegitimate local state: notice that we do not rule out the possibility of $i = j$, in this case P_i's local state is a corruption.

Lemma 5.3.8 (Local Corruptions)**.** *Let L be a livelock of a parameterized protocol $p(K)$ on a unidirectional ring. Then, for some global state of L there exists a process P_i having a corruption.*

Proof. From Lemma 5.3.7, every global state of L has a process P_i in an illegitimate local state. Lemmas 5.3.1 and 5.3.4 establish that enabled local states propagate along a unidirectional ring without collisions. By contradiction, assume that at every global state of L, all enabled processes are in non-corruptions. Due to closure of $I(K)$ in $p(K)$, a propagation of a non-corruption in any process P_i should leave P_i in a local legitimate deadlock. As such, eventually every process P_i will be in a legitimate state. This contradicts Lemma 5.3.7. Therefore, there exists a global state of L where some P_i is in a corruption. $\square$

To understand how livelocks represent themselves in LTG, we observe that each sequence *Sch* of local transitions representing a livelock belongs to an equivalence class of permutations of *Sch* whose local transitions preserve some irreflexive and transitive precedence relation.

Definition 5.3.9 (Livelock Induced Precedence Relation $\prec$)**.** *Let a livelock L be represented by a sequence of local transitions Sch=$\ll t^l_0, t^l_1, \cdots, t^l_{n-1} \gg$. We say t^l_i precedes t^l_j, denoted $t^l_i \prec t^l_j$ if and only if*

1. *the execution of t^l_i enables t^l_j or,*
2. *if t^l_j executes, then it collides with t^l_i enabled in P_i or,*
3. *if 1 and 2 are false, then there exists t^l_k in Sch such that $t^l_i \prec t^l_k$ and $t^l_k \prec t^l_j$.*

Example 5.3.2 (Binary Agreement)**.** Consider an instance of the agreement protocol where $K = 4$. As illustrated in the top part of Figure 5.6, we examine a livelock L represented by the global computation prefix $\ll$ 1000, 1100, 0100, 0110, 0111, 0011, 1011, 1001 $\gg$ repeated k times (k is an arbitrary integer). We represent L by the sequence of local transitions *Sch*=$\ll$ ${t_{01}}^1$, ${t_{10}}^0$, ${t_{01}}^2$, ${t_{01}}^3$, ${t_{10}}^1$, ${t_{01}}^0$, ${t_{10}}^2$, ${t_{10}}^3$ $\gg$. The superscript denotes the process index and the subscript ij represents a change of value in x_r from i to j.

Figure 5.5 illustrates the dependencies between the local transitions of L. Two local transitions t_i^l and t_j^l are *independent* if and only if $t_i^l \not\prec t_j^l$ and $t_j^l \not\prec t_i^l$. Since we have only three pairs of independent local transitions ((t_{01}^2, t_{10}^0), (t_{01}^3, t_{10}^1) and (t_{01}^0, t_{10}^2)), the precedence relation allows $8 = 2^3$ possible precedence-preserving permutations of *Sch*. Figure 5.6 depicts L and another livelock generated by a permutation of *Sch* preserving the precedence relation in Figure 5.5.

Lemma 5.3.10 (Precedence Relation Reduction)**.** *Let $p(K)$ be a parameterized protocol on a unidirectional ring of size K. If $p(K)$ has a livelock L, for some K, whose local transitions are represented by a sequence Sch=$\ll t_0^l, t_1^l, \cdots, t_{n-1}^l \gg$ then every precedence-preserving permutation of Sch represents a livelock of $p(K)$.*

Proof. Let *Sch'* be a precedence-preserving permutation of *Sch* obtained by swapping two arbitrary independent local transitions t_i^l and t_j^l where $i < j$. Now consider the subsequence *Middle*=$\ll t_{i+1}^l, \cdots, t_{j-1}^l \gg$ of *Sch*, since swapping of t_i^l and t_j^l in *Sch'* is precedence-preserving, each of t_i^l and t_j^l form independent pairs with every local transition in *Middle*. If it is not the case, a swap of t_i^l and t_j^l would have violated the precedence relation. Since *Sch* is precedence-preserving and every transition in *Middle* is independent of t_i^l and t_j^l, for every $t_k^l : t_k^l \prec t_j^l$, t_k^l occurs in *Sch* before t_i^l and for every $t_k^l : t_i^l \prec t_k^l$, t_k^l occurs in *Sch* after t_j^l. Thus, the execution of *Sch'* proceeds as follows. Every transition t_k^l for $k < i$ executes exactly as in *Sch*. Now t_j^l is enabled since all local transitions preceding it already executed, hence, t_j^l executes as in *Sch*. None of the transitions in *Middle* depends on t_i^l or on t_j^l and they execute as in *Sch*. t_k^l ($k \geq j$) execute as in *Sch* since all their preceding transitions already executed. Since no local transition has been disabled due to the precedence-preserving swap, *Sch'* represents a new livelock L'. □

Lemma 5.3.10 establishes our observation for a reduction based on an irreflexive partial order. Godefroid [38] originally introduced partial order reduction to simplify automatic verification. We accordingly reduce our search for livelocks in unidirectional rings to a search for a representative livelock that we call a *contiguous livelock*.

Let L be a livelock on a unidirectional ring having $|E|$ enablements. As illustrated in the bottom part of Figure 5.6, a *contiguous livelock* C_L has a global state where $|E|$ adjacent

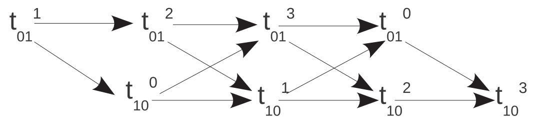


Figure 5.5: Precedence relation for local transitions in Example 5.3.2

processes are enabled. The subsequent global states of C_L are such that only the rightmost enablement in the segment of adjacent processes propagates while the remaining $|E| - 1$ enablements do not propagate. After $K - |E|$ propagations of the rightmost enablement, a new global state with $|E|$ adjacent enablements is reached. Figure 5.6 illustrates this scenario for $K = 4$ and $|E| = 2$. Notice that a K times repetition of the scenario in Figure 5.6 results in a full rotation of the segment of adjacent enablements. Corollary 5.3.11 directly follows from Lemma 5.3.10.

Corollary 5.3.11. $p(K)$ *has a livelock* if and only if $p(K)$ *has a contiguous livelock.*

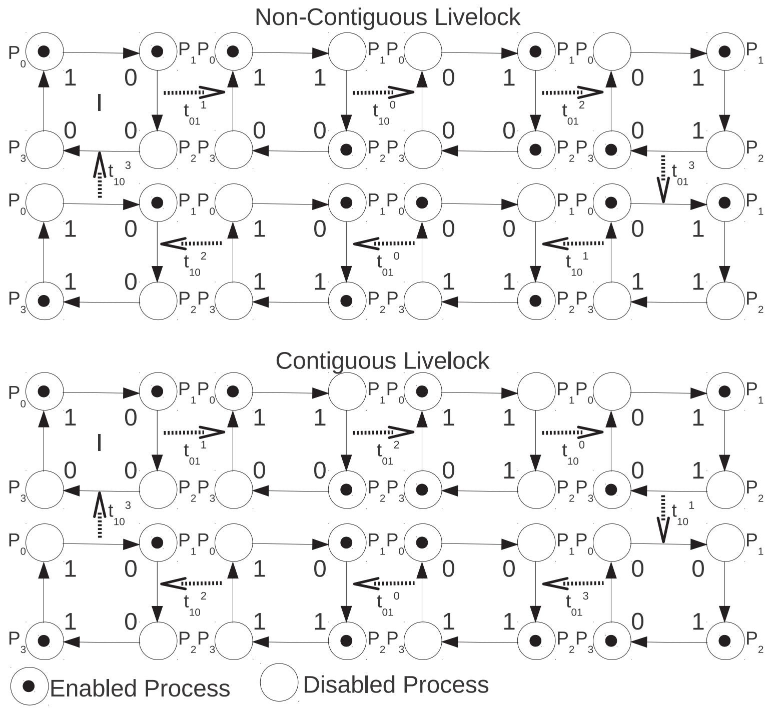


Figure 5.6: Two precedence-preserving livelocks for Example 5.3.2. The starting global state is marked by "I"

Lemma 5.3.12 demonstrates the kind of structure LTG$_p$ has when $p(K)$ has a contiguous livelock. We call this structure a *contiguous trail* of LTG$_p$.

Lemma 5.3.12 (Representation of a Contiguous Livelock in LTG$_p$)**.** *Let the parameterized protocol* $p(K)$ *on a unidirectional ring have a contiguous livelock* C_L *with* $|E|$ *enablements. Then, LTG*$_p$ *has an alternating trail* T_R *of the following format.*

1. if $|E| = 1$, then T_R is an alternating trail of a t-arc followed by an s-arc and vice versa.

2. if $|E| > 1$, then T_R is an alternation of two types of walks: w_1 and w_2. w_1 consists of $|E|$ consecutive s-arcs such that every vertex/local state in w_1 has an outgoing t-arc in w_2. w_2 has $2(K - |E|)$ arcs of an alternating walk of t-arcs and s-arcs.

We call T_R a contiguous trail of LTG$_p$.

Proof. If $|E| = 1$, then there exists only one enablement in the ring. An enablement propagation at a process P_i corresponds to a t-arc $(s_i^l, s_i^{l'})$. Now, P_{i+1}, the successor of P_i, is in an enabled local state s_{i+1}^l that is a right continuation of $s_i^{l'}$. Therefore, there exists an s-arc from $s_i^{l'}$ to s_{i+1}^l. Following a similar reasoning for every process P_i that propagates a single enablement along C_L, we conclude that T_R is a trail of alternating s-arcs and t-arcs when $|E| = 1$.

If $|E| > 1$, C_L consists of two types of computations. The first type of computation is such that $p(K)$ is in a global state s^c where $|E|$ enabled processes are adjacent, which implies a walk of type w_1 of $|E|$ consecutive s-arcs in T_R. Moreover, every local state in w_1 is an enablement that will eventually propagate. Thus, every local state in w_1 should have an outgoing t-arc participating in T_R but not in w_1. The second type of computation is the rightmost enablement propagation through the execution of $K - |E|$ local transitions. Using a similar reasoning as in the case where $|E| = 1$, the second type of computation is represented by a walk of type w_2 in T_R consisting of an alternating t-arc followed by an s-arc and vice versa. As such, the length of the alternating walk w_2 is $2(K - |E|)$. Since C_L is an alternation of both types of computations, T_R is an alternation of both types of walks: w_1 and w_2. Moreover, every s-arc in a walk of type w_1 should reach a target local state that is a source of a t-arc in a walk of type w_2 in T_R. □

In a global livelock, the partial observation of each process P_i on W_i results in a repetitive sequence of values that we call a *pseudo-livelock*. For example, a local transition t_{02} represented by the action $y = 0 \wedge x = 0 \rightarrow x := 2$ and a local transition t_{20} whose action is $y = 1 \wedge x = 2 \rightarrow x := 0$ form a psuedo-livelock; if we project each local transition on x, we obtain the local transitions t'_{02} and t'_{20} represented by $x = 0 \rightarrow x := 2$ and $x = 2 \rightarrow x := 0$, respectively. t'_{02} and t'_{20} form the repeating sequence of values $\ll 0, 2 \gg^k$ for x. However, neither of $\{t_{02}, t_{20}\}$ enables the other because of different values of the unwritable variable y. Note that a pseudo-livelock in P_r does not imply the existence of a global livelock.

Theorem 5.3.13 establishes a sufficient condition for livelock-freedom in unidirectional rings.

Theorem 5.3.13 (Sufficient Conditions for Livelock Freedom)**.** *For some K, if L is a livelock in a parameterized protocol $p(K)$ on a unidirectional ring, then LTG$_p$ has a contiguous directed trail T_R in LTG$_p$ such that:*

1. *There exists an illegitimate local state in T_R, and,*

2. *All t-arcs of T_R form pseudo-livelocks.*

Proof. From Lemma 5.3.10, $p(K)$ has a livelock L if and only if $p(K)$ has a contiguous livelock C_L. Lemma 5.3.12 implies that LTG$_p$ has a contiguous trail T_R representing C_L.

According to Lemma 5.3.8, there exists a global state in L such that some process is corrupted. Since T_R is a representation of C_L on a ring, we conclude that some vertex in T_R represents a local illegitimate state. This proves Item 1.

Since L is a livelock, for every P_i, the projection of every global transition t_i in L on the writable variables of P_i; a.k.a., $t_i \downarrow W_i$, induces a repetitive sequence of values for variables in W_i . Therefore, t-arcs in T_R form a pseudo-livelock. This proves Item 2. $\square$

Note that we use the contrapositive of Theorem 5.3.13 to prove livelock-freedom.

Example 5.3.3 (Binary Agreement)**.** In the right hand-side of Figure 5.4, the local illegitimate states are $\{10, 01\}$, however, it is sufficient to resolve either of them to obtain a continuation graph that has no directed cycles passing by illegitimate deadlocks. Since including just one of the candidate local transitions does not form pseudo-livelocks, both solutions are livelock free; hence convergence. If we unnecessarily include both t_{01} and t_{10} that form a pseudo-livelock, we observe $T_R = \ll$ 01, t_{10}, 00, s, 01, s, 10, t_{01}, 11, s, 10, s, 01 $\gg$ as an alternating trail satisfying the implications of Lemma 5.3.12 (see Figure 5.4). Moreover, t_{01} and t_{10} form a pseudo-livelock. Hence, including both t_{01} and t_{10} does not satisfy the sufficient conditions of the contrapositive of Theorem 5.3.13. Notice that if we apply constraint satisfaction for cyclic constraint graphs as described in [36], there is no way to differentiate between the case where only one of the convergence actions $\{t_{01}, t_{10}\}$ is included in p_{ss}, and the case where we include both convergence actions in p_{ss}.

5.4 Application in Automated Design of Convergence

This section presents an outline for a method that synthesizes global convergence for parameterized protocols in the local state space of the representative process (without exploring the global state). Previous work on automated design of convergence [10], [34], [39] mainly explores the global state space of a protocol to synthesize recovery from any illegitimate state. Moreover, existing work addresses the synthesis of convergence for protocols with a fixed number of processes; i.e., synthesized solutions are not generalizable. Thus, the proposed method in this section enables a significant improvement in the time/space complexity of automated design of convergence.

5.4.1 Synthesis Methodology

Given a parameterized protocol p over a ring whose representative process is P_r and whose set of legitimate states is defined by LC_r, we construct LTG$_p$ as in Section 5.3.

1. Identify the subset ${D_L}^l \subset {S_r}^l$ of local deadlocks of P_r. Form the induced subgraph of RCG$_p$ over ${D_L}^l$.

   3-coloring example. Since the input protocol p for 3-coloring is empty, we have ${D_L}^l = {S_r}^l$ (Figure 5.7)$\lhd$.

2. Identify a subset *Resolve* $\subset \neg LC_r \cap {D_L}^l$ of local deadlocks that should be resolved by local t-arcs in the revised protocol p_{ss}. As such, RCG$_{p_{ss}}$ is the induced subgraph of RCG$_p$ over ${D_L}^l -$ *Resolve* should represent a deadlock free protocol for every K. By Theorem 5.2.2, RCG$_{p_{ss}}$ has no directed cycles through any local deadlock in LC_r if and only if $p_{ss}(K)$ has no deadlocks for every K. As such, *Resolve* captures a minimal subset of local deadlocks of p that should be resolved in p_{ss}. One way to compute *Resolve* is as a *minimal feedback subset*[6] of RCG$_p$ restricted to be a subset of $\neg LC_i$. Therefore, all minimal feedback subsets that are subsets of $\neg LC_r$ are possible candidates for *Resolve*.

   3-coloring example. A parameterized 3-coloring protocol over a unidirectional ring is defined by a process P_r, a set of variables $\Phi_p(K) = \{c_0, \cdots, c_{K-1}\}$ such that c_r takes values from a domain $D_r = \{0, 1, 2\}$. A local legitimate state of P_r is such that P_r's color is different from its predecessor's; i.e., $LC_r = (c_r \neq c_{r-1})$. In Figure 5.7, the set of illegitimate local states identified by uncolored vertices is $\{00, 11, 22\}$.

[6]A feedback subset *FS* of a directed graph G is a subset of vertices of G such that, when omitted from the G, induces a subgraph of G with no directed cycles. *FS* is minimal when it has no subset that is a feedback set.

Since every illegitimate local state has a self-loop, *Resolve*$= \{00, 11, 22\}$. We denote a possible local transition of P_r by t_{ij} where $i, j \in D_r$, such that $t_{ij} : c_{r-1} = c_r = i \rightarrow c_r := j$. $\lhd$

3. Identify *Candidates*$_r$ as the set of all possible candidate local transitions ${t_r}^l$ of P_r that resolve every local deadlock in *Resolve*. ${t_r}^l = ({s_0}^l, {s'_0}^l) \in$ *Candidates*$_r$ is a local transition of P_r such that ${s_0}^l \in$ *Resolve* and ${s'_0}^l \notin$ *Resolve*. As such, we guarantee that all actions are self-disabling as in Assumption 2 of Section 5.3.

   3-coloring example. The set of candidate local transitions in Figure 5.7 that resolve all local deadlocks in *Resolve* is $\{t_{01}, t_{02}, t_{10}, t_{12}, t_{20}, t_{21}\}$. $\lhd$

4. Identify a subset of *Non-Pseudo-Livelocks* (*NPL*) of *Candidates*$_r$ such that:

   (a) Local transitions in *NPL* do not form pseudo-livelocks.

   (b) Local transitions in *NPL* resolve every local deadlock in *Resolve*.

   If such *NPL* exists, declare success (Theorem 5.3.13).

   3-coloring example It is sufficient to include only one local transition originating at every local deadlock to resolve it. For example, it is sufficient to include either t_{01} or t_{02}, but not both, to resolve the local deadlock 00. Every local deadlock in *Resolve* is the source state of two possible local transitions in *Candidates*$_r$. As such, 2^3 possible subsets of *Candidates*$_r$ render 3-coloring deadlock free for any K. These subsets are $\{\{t_{01}, t_{12}, t_{20}\}, \{t_{01}, t_{12}, t_{21}\}, \{t_{01}, t_{10}, t_{20}\}, \{t_{01}, t_{10}, t_{21}\}, \{t_{02}, t_{12}, t_{20}\}, \{t_{02}, t_{12}, t_{21}\}, \{t_{02}, t_{10}, t_{20}\}, \{t_{02}, t_{10}, t_{21}\}\}$. However, every subset has a pseudo-livelock. For example, local transitions $\{t_{01}, t_{12}, t_{20}\}$, when projected on W_r, form the pseudo-livelock $\ll 0, 1, 2 \gg^k$. Likewise, any two local transitions t_{ij}, t_{ji} form a pseudo-livelock. $\lhd$

5. Identify a subset of *Pseudo-Livelocks* (*PL*) of *Candidates*$_r$ such that:

   (a) Local transitions in *PL* resolve every local deadlock in *Resolve*.

   (b) Local transitions in *PL* have subsets forming pseudo-livelocks. Otherwise, local transitions in *PL* would have been in *NPL* and we should not have reached the current step.

   (c) Each pseudo-livelock in *PL* is not forming a contiguous trail T_R in LTG$_p$ as in Lemma 5.3.12.

   If such *PL* exists, there are no pseudo-livelocks in *PL* whose t-arcs form contiguous trails. Consequently, we can conclude from Theorem 5.3.13 that p_{ss} is livelock free for every size of the ring. Otherwise, declare failure.

   3-coloring example. Every subset of t-arcs forming a pseudo-livelock corresponds to a contiguous livelock. For example, in Figure 5.7, $\{t_{01}, t_{12}, t_{20}\}$ forms a pseudo-livelock and creates the contiguous trail $T_R = \{00, 01, 11, 12, 22, 20\}$ that includes illegitimate local states. The sufficient conditions for livelock freedom in the contrapositive of Theorem 5.3.13 are not satisfied. Therefore, we declare failure. $\triangleleft$

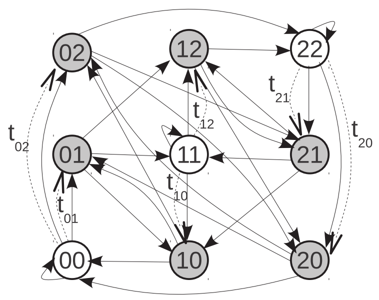


Figure 5.7: LTG$_p$ of 3-coloring example

5.4.2 Further Examples

In this subsection, we apply our proposed methodology to design three protocols: binary agreement, two-coloring and sum-not-two protocols. In the latter example, we illustrate how the conditions of Theorem 5.3.13 are strictly sufficient for livelock-freedom, however, they are weak enough to provide a converging solution on a symmetric unidirectional ring.

Agreement example. We investigate a parameterized binary agreement protocol as in Example 5.3.2. A local legitimate state is such that $x_r = x_{r-1}$; i.e., the protocol stabilizes when all variable values are equal.

Figure 5.8 represents LTG$_p$ of the parametrized agreement protocol. t_{01} and t_{10} are local transitions resolving illegitimate local states. t_{01} : $(x_r < x_{r-1}) \rightarrow x_r := x_{r-1}$ or t_{10} : $(x_{r-1} < x_r) \rightarrow x_r := x_{r-1})$.

In Figure 5.8, the local illegitimate states are ${D_L}^l = \{10, 01\}$, however, it is sufficient to resolve either of them to obtain a continuation relation that has no directed cycles passing by illegitimate deadlocks. Therefore, *Resolve*$= \{01\}$ or *Resolve*$= \{10\}$. As such, including either t_{01} or t_{10} (but not both!) renders the protocol deadlock free. Since including just one of the candidate local transitions does not form pseudo-livelocks, both solutions are livelock free. Hence follows convergence.

If we unnecessarily include both t_{01} and t_{10} that form a pseudo-livelock, we observe $T_R =\ll$ 01, t_{10}, 00, s, 01, s, 10, t_{01}, 11, s, 10, s, 01 $\gg$ as an alternating trail satisfying the implications of Lemma 5.3.12. Moreover, t_{01} and t_{10} form a pseudo-livelock. Hence, including both t_{01} and t_{10} does not satisfy the sufficient conditions of the contrapositive of Theorem 5.3.13.

Notice that if we apply constraint satisfaction for cyclic constraint graphs as described in reference [36], there is no way to differentiate between the case where only one of the convergence actions $\{t_{01}, t_{10}\}$ is included in p_{ss}, and the case where we include both convergence actions in p_{ss}. In fact, both constraint graphs are the same since the set of legitimate states does not change. Moreover, our methodology computes a possibly strict subset of local deadlocks outside LC_r and still guarantees deadlock freedom for every K. $\lhd$

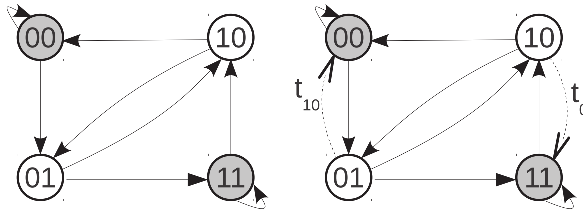


Figure 5.8: RCG$_p$ and LTG$_p$ of Agreement Example

Two-coloring example. For a 2-coloring protocol whose RCG and LTG are represented in Figure 5.9, $R_r = \{c_{r-1}, c_r\}$ and $W_r = \{c_r\}$. $D_r = \{0, 1\}$ and $LC_r = c_r \neq c_{r-1}$. A legitimate local state is such that a process and its predecessor should have different colors.

Unlike deadlock states in agreement, 2-coloring requires the resolution of both illegitimate local deadlocks ${D_L}^l$ =*Resolve*= $\{00, 11\}$ because they have self-loops of s-arcs[7]. However, the resolution of both local deadlocks results in a directed trail T_R as in Lemma 5.3.12 $\ll$ 00, t_{01}, 01, s, 11, t_{10}, 10, s, 00 $\gg$ and not satisfying the sufficient conditions in the contrapositive of Theorem 5.3.13. As such, we cannot conclude livelock freedom of 2-coloring for arbitrary K. In fact, 2-coloring self-stabilizing protocols are impossible in unidirectional rings [40], however our lack of necessary conditions for livelock freedom prevents us from deducing any impossibility results. $\lhd$

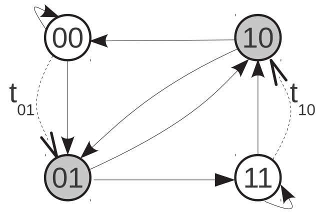


Figure 5.9: LTG$_p$ of the Two Coloring Example

[7]Recall that for deadlocks-freedom, we make sure that there are no directed cycles over local deadlocks in RCG that include illegitimate local states.

Sum-not-two example. We present a hypothetical example to illustrate the interplay between having a trail, having pseudo-livelocks and having both. The Sum-Not-Two protocol on a unidirectional ring is such that P_r reads x_{r-1} and x_r and writes x_r. For simplicity of presentation, we restrict our example such that x_r takes values in $\{0, 1, 2\}$. A local legitimate state is such that $x_r + x_{r-1} \neq 2$. The input protocol p is empty.

Since p is empty, the set of local deadlocks outside LC_r is $\neg LC_r = \{20, 11, 02\}$. For a deadlock free protocol, no proper subset of $\neg LC_r$ can be resolved to render p_{ss} deadlock free for every K. Thus, *Resolve*$= \{20, 11, 02\}$.

Figure 5.10 illustrates LTG$_p$ of Sum-Not-Two protocol with all candidate t-arcs included. Every local deadlock has two possible t-arcs that resolve it and hence, we have 2^3 possibilities for *Candidates*$_r$. The following two possibilities form pseudo-livelocks and each of them participate in a trail: $\{\{t_{21}, t_{10}, t_{02}\}, \{t_{01}, t_{12}, t_{20}\}\}$. For example, the first possibility participates in the trail: $T_R = \ll 02, t_{21}, 01, s, 11, s, 11, t_{10}, 10, s, 02, s, 20, t_{20}, 22, s, 20, s \gg$. This possibility forms a pseudo-livelock and participates in a trail T_R as implied by Lemma 5.3.12. Hence, sufficient conditions of the contrapositive of Theorem 5.3.13 are not satisfied by the first possible set of candidates and we cannot include this set.

In fact, if we examine T_R, it should represent a contiguous livelock L having $|E| = 2$ and only one propagation of enablement; i.e., $K - |E| = 1$. Hence, T_R is possibly representing a livelock in a ring where $K = 3$. However, if we try to reconstruct the global livelock of a ring of three processes using T_R, we fail! In other words, T_R does not represent a real livelock and due to the lack of necessity, we could not include $\{t_{21}, t_{10}, t_{02}\}$ in p_{ss}.

None of the remaining candidate subsets of t-arcs forms a trail whose t-arcs are pseudo-livelocks. For example, let *Candidates*$_r = \{t_{21}, t_{12}, t_{01}\}$. Here, t_{21} and t_{12} form a pseudo-livelock, however, there is no trail where they solely participate and that has the properties implied in Lemma 5.3.12. Moreover, there is a trail that includes all the three t-arcs together, but since, together, they do not form a pseudo-livelock, conditions of the contrapositive of Theorem 5.3.13 remain satisfied. As such, including $\{t_{21}, t_{12}, t_{01}\}$ in p_{ss} renders Sum-Not-Two converging. The following action captures *Candidates*$_r$: $(x_r + x_{r-1} = 2) \wedge (x_r \neq 2) \rightarrow x_r := (x_r + 1) \bmod 3$, $(x_r + x_{r-1} = 2) \wedge (x_r = 2) \rightarrow x_r := (x_r - 1) \bmod 3$.

To prove convergence of our proposed solution using constraint satisfaction, we must ingenuously identify a partitioning of the protocols actions. We argue that our methodology bypasses constraint satisfaction in this respect as we directly design/verify convergence through local state space exploration. $\lhd$

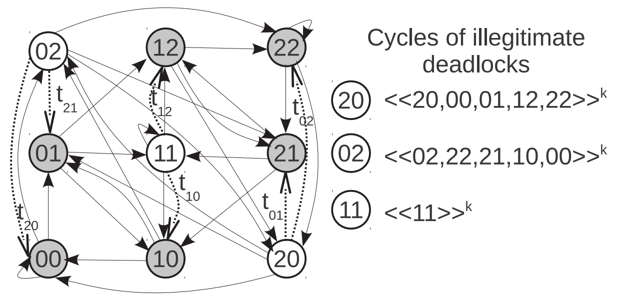


Figure 5.10: LTG$_p$ of the Sum-Not-Two example including every candidate t-arc. To the right, we demonstrate a s-cycle for each individual local deadlock that we resolved.

5.5 Summary and Extensions

This chapter proposed a method for local reasoning about global convergence of parameterized network protocols with the ring topology. In such protocols, the code of each process is instantiated from the parameterized code of a representative/template process by variable substitution. Parameterized ring protocols have important applications as they can be used to construct more complicated topologies where multiple rings are intertwined (e.g., multi-ring token passing in Chapter 3). Global convergence to a set of legitimate states I requires both deadlock-freedom and livelock-freedom in $\neg I$. While most existing design methods enable the design of convergence by reasoning in the global state space of a protocol, this chapter takes a different approach of reasoning in the local state space of the representative process to ensure global convergence. Specifically, we presented necessary and sufficient conditions for deadlock-freedom, and sufficient conditions for livelock-freedom in parameterized unidirectional rings.

We would like to extend this work in several directions. First, we plan to investigate local reasoning for global convergence of parameterized protocols with topologies other than rings (e.g., tree, mesh, etc.). Second, we are currently investigating sufficient conditions for bidirectional rings. Third, another interesting problem is automation. We will design synthesis algorithms that can automate the generation of the LTG graphs and can revise the graphs so they meet our conditions for deadlock/livelock-freedom. Such a synthesis in local state space is a significant paradigm shift with respect to previous work on automated design of convergence in global state [10], [13], [34], which could result in producing software tools that are substantially more efficient in automated design of parameterized self-stabilizing protocols.

Chapter 6

An Exact Algebraic Characterization of Livelocks in Unidirectional Rings

This chapter presents necessary and sufficient conditions for the livelock-freedom of a subclass of unidirectional rings in which processes are self-disabling; i.e., in each execution turn, the execution of an action of some process disables all actions of that process. We present our necessary and sufficient conditions in an algebraic setting that significantly simplifies reasoning about global livelocks in the local state space of the processes. The proposed approach enables a design and verification method that solely relies on local reasoning, thereby eradicating the need for reasoning about livelocks in the global state space. The proposed approach has several applications including the design of global convergence for self-stabilizing systems and the verification of livelock-freedom in unidirectional rings. We evaluate the proposed necessary and sufficient conditions in the context of several examples including Dijkstra's self-stabilizing token ring protocol. In fact, using the proposed necessary and sufficient conditions, we illustrate a simplified proof of the livelock-freedom of Dijkstra's self-stabilizing token ring protocol.

6.1 Introduction

Livelocks are among the least understood types of concurrency flaws due to their dynamic, global and arbitrary nature [41]. We understand by a livelock/non-progress cycle, a sequence of undesirable global configurations that indefinitely repeat, thereby preventing further progress towards a global desirable configuration. Examples include process starvation for a shared resource [42], multi-token circulation in a mutual exclusion protocol [7], and the design of convergence in distributed protocols [9], [43]; i.e., protocols that recover to a desired behavior regardless of their initial configuration. To design livelock-free protocols irrespective of their number of processes, we investigate properties of global livelocks as perceived by a single process; i.e., local properties of livelocks. In this chapter, we establish necessary and sufficient local conditions for global livelock-freedom in arbitrary-sized/parameterized unidirectional rings.

The problem of livelock-freedom in a parameterized network is generally undecidable. Apt and Kozen [44] reduce the non-halting problem to livelock-freedom verification of a parameterized network. Suzuki [45] establishes a similar result for parameterized unidirectional rings of symmetric processes. Consequently, most of livelock verification techniques are either (1) incomplete [46]–[50]; i.e., provide strictly sufficient conditions for livelock-freedom, (2) are applicable to a strict subclass of models of computation [51]–[54], (3) or are not amenable to automation by using manually designed ranking functions [22], [46], [55]. Due to their inherent incompleteness, the approaches in the first category do not exactly characterize livelocks, thereby preventing impossibility proofs similar to the results exhibited by Shukla et al. [40] for symmetric coloring protocols. The second category includes techniques that impose strong restrictions on unidirectional rings, thereby could not explore the convergence of simple protocols. For instance, all our case studies do not fall in the subclass captured by approaches in (2). Unlike the previous two techniques, manual methods in category (3) require human ingenuity to create strictly decreasing ranking functions in order to demonstrate convergence/livelock-freedom; these methods are not generally suitable for automation.

We extend the second category of techniques by exactly characterizing livelocks in a subclass of parameterized protocols on unidirectional rings. Due to Suzuki's undecidability result [45], we inevitably consider unidirectional rings whose processes are *self-disabling*. A process is self-disabling if any of its execution steps disable all actions of that process until a step of its predecessor process re-enables it. A process is enabled/has an enablement if and only if its condition for taking a step is satisfied for the current value of its local variables, otherwise it is disabled. In Chapter 5, we established that a unidirectional ring propagates a fixed number of enablements along its links if and only if the ring has a livelock (Property 5.3.4). We formalize the notions of propagation and process local loops by using local binary relations ϵ_r and η_r on the steps of every process P_r. Theorem 6.2.4 establishes our main result: a unidirectional ring has a livelock if and only if propagations are cyclic structure-preserving for local loops, for every process P_r, and there exists a global configuration from which enablement propagation starts.

Contributions. We establish an equivalence between the set of livelocks in a unidirectional ring and algebraic properties of binary relations representing local loops and local propagations of each process in the ring, respectively. The local nature of our algebraic characterization significantly simplifies reasoning about livelocks in parameterized unidirectional rings that are not necessarily fully symmetric while avoiding exploration of the global state space. Herlihy and Shavit [56] reduce the proof of existence of wait-free protocols to reasoning about topological properties of combinatorial representations of the protocols. In a similar spirit, our algebraic representation reduces design and verification of livelock-freedom to the analysis of a set of algebraic equations on compositions of binary relations. Our algebraic approach exactly characterizes all the livelocks Dijkstra's token ring [7] has, including a case where processes execute in full synchrony and the number of processes is greater than the size of the variables' domains by one.

Organization. We incrementally develop our formal framework for reasoning about livelocks and establish our main theorem in Section 6.2. Section 6.3 illustrates how we utilize our formal framework to characterize livelocks/livelock-freedom through three classical protocols on unidirectional rings. Section 6.4 summarizes our contributions and discusses potential extensions.

6.2 Algebraic Properties of Livelocks

In this section, we establish necessary and sufficient conditions for livelock-freedom in parameterized rings in terms of binary relations on local transitions of each process of the ring. In Subsection 6.2.1, we define a set of binary relations (on the local transitions of processes) that we shall use in Subsection 6.2.2 to specify necessary and sufficient conditions for the existence of livelocks in unidirectional rings.

6.2.1 Binary Relations on Local Transitions

We define our binary relations on the set of local transitions of processes. We say a transition $t^i = (s^i_0, s^i_1)$ in a process P_i *partially enables* a transition $t^j = (s^j_0, s^j_1)$ in a process P_j if and only if for every variable $x \in W_i \cap R_j$, we have $x(s^i_1) = x(s^j_0)$. The idea behind partial enablement is that the complete enablement of a local transition t^r of P_r depends on x_r and $x_{r \ominus 1}$. Thus, when a process P_r executes, it may update x_r that could in turn enable its successor depending on the value of $x_{r \oplus 1}$. Using the notion of partial enablement, we relate each local transition of a process with the rest of its local transitions and with the local transitions of its successor. Formally, we define a binary relation $\mathcal{H}_r \subseteq \delta_r \times \delta_r$ that is equal to the following set of pairs of local transitions of P_r: $\mathcal{H}_r = \{((s^r_i, {s^r}'_i), ({s^r}_j, {s^r}'_j)) | x_r({s^r}'_i) = x_r({s^r}_j)\}$.

A cycle of local transitions in $\mathcal{H}_r$ is a *pseudolivelock* whose set of pairs of local transitions defines a relation $\eta_r \subseteq \mathcal{H}_r$. We denote by $L^r_p \subseteq \delta_r$ the subset of transitions of P_r that participate in the pseudolivelock η_r.

Likewise, we define a *local propagation* of P_r to $P_{r\oplus 1}$ as a binary relation $\mathcal{E}_r \subseteq \delta_r \times \delta_{r\oplus 1}$ where $(t^r, t^{r\oplus 1}) \in \mathcal{E}_r$ if and only if t^r partially enables $t^{r\oplus 1}$. $\mathcal{E}_r = \{((s^r, s^{r\prime}), (s^{r\oplus 1}, s^{r\oplus 1\prime})) | x_r(s^{r\prime}) = x_r(s^{r\oplus 1})\}$. A *feasible propagation* of the transitions of a pseudolivelock η_r of P_r to the transitions of a pseudolivelock $\eta_{r\oplus 1}$ is a local propagation such that $\epsilon_r \subseteq L^r_p \times L^{r\oplus 1}_p$ is left-total in L^r_p and onto on $L^{r\oplus 1}_p$; i.e., ϵ_r is a *total* binary relation.

Notation. Let R be a binary relation on a subset of local transitions of $p(K)$, $R(t_i)$ denotes some arbitrary transition t_j such that $(t_i, t_j) \in R$. In other words, $t_j = R(t_i)$ is a shorthand for $(t_i, t_j) \in R$. We denote relation composition from left to right by '$\circ$' and a sequence of composition of binary relations $a_x \circ \cdots \circ a_y$ by $a_{x \rightsquigarrow y}$, where composition is along a clockwise direction of the ring if and only if $x < y$. a^m denotes the composition of m copies of a; i.e., exponentiation. a^{-1} denotes the inverse relation of a. e denotes the *identity* binary relation; i.e., $e = \{(x, x) \in A \times A\}$, for some set A.

Example 6.2.1 (3-valued Agreement)**.**

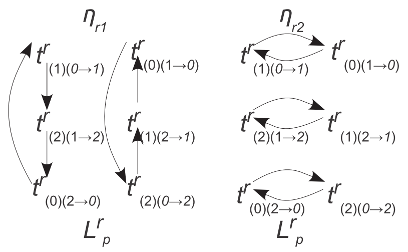
Consider the local transitions δ_r of a 3-valued agreement protocol defined as follows for $0 \le r \le K - 1$.

$$\begin{aligned}
&t^r_{(2)(0\to 2)} : x_{r\ominus 1} = 2 \wedge x_r = 0 \longrightarrow x_r := 2\\
&t^r_{(2)(1\to 2)} : x_{r\ominus 1} = 2 \wedge x_r = 1 \longrightarrow x_r := 2\\
&t^r_{(1)(0\to 1)} : x_{r\ominus 1} = 1 \wedge x_r = 0 \longrightarrow x_r := 1\\
&t^r_{(1)(2\to 1)} : x_{r\ominus 1} = 1 \wedge x_r = 2 \longrightarrow x_r := 1\\
&t^r_{(0)(2\to 0)} : x_{r\ominus 1} = 0 \wedge x_r = 2 \longrightarrow x_r := 0\\
&t^r_{(0)(1\to 0)} : x_{r\ominus 1} = 0 \wedge x_r = 1 \longrightarrow x_r := 0
\end{aligned}$$

We summarize the local transitions of the protocol by the set of parameterized transitions: $t^r_{(v)(v\oplus 1\to v)}$ and $t^r_{(v\oplus 1)(v\to v\oplus 1)}$.

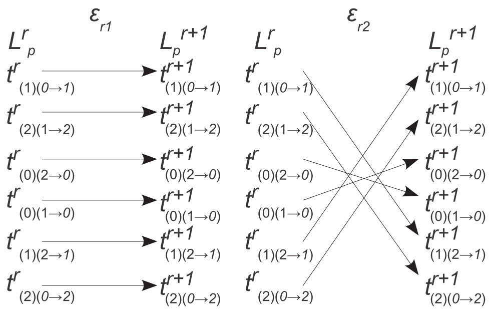
Figure 6.1 illustrates $\mathcal{H}_r$ of the local transitions in Example 6.2.1. Every related pair of local transitions satisfies our definition. For instance, $(t^r_{(1)(2\to 1)}, t^r_{(2)(1\to 2)}) \in \mathcal{H}_r$ since $t^r_{(1)(2\to 1)}$ sets x_r to 1 while the local source state of $t^r_{(2)(1\to 2)}$ is partially enabled at $x_r = 1$. We apply the same reasoning to every pair of local transitions in δ_r in order to build $\mathcal{H}_r$. Using a fixpoint computation on $\mathcal{H}_r$, we determine that in this case $\eta_r = \mathcal{H}_r$ and $L^r_p = \delta_r$. For the 3-valued agreement, η_r is a strongly connected component that can be decomposed into the union of the following two bijections: η_{r1} and η_{r2}. $\eta_{r1}(t^r_{(v)(v\oplus1\rightarrow v)})= t^r_{(v\ominus1)(v\rightarrow v\ominus1)}$, $\eta_{r1}(t^r_{(v)(v\ominus1\rightarrow v)}) = t^r_{(v\oplus1)(v\rightarrow v\oplus1)}$, $\eta_{r2}(t^r_{(v)(v\oplus1\rightarrow v)}) = t^r_{(v\oplus1)(v\rightarrow v\oplus1)}$ and $\eta_{r2}(t^r_{(v\oplus1)(v\rightarrow v\oplus1)}) = t^r_{(v)(v\oplus1\rightarrow v)}$.

Figure 6.2 depicts $\mathcal{E}_r$ of the local transitions in Example 6.2.1. Every connected pair of local transitions satisfies our definition of $\mathcal{E}_r$. For instance, $(t^r_{(1)(2\rightarrow1)}, t^{r\oplus1}_{(1)(0\rightarrow1)}) \in \mathcal{E}_r$ since the local source state of $t^{r\oplus1}_{(1)(0\rightarrow1)}$ and the local target state of $t^r_{(1)(2\rightarrow1)}$ share the value of $x_r = 1$. We apply the same reasoning to every pair of local transitions in $\delta_r \times \delta_{r\oplus1}$ to build $\mathcal{E}_r$. A potential feasible propagation for the transitions of a given pseudolivelock $\eta_r \in \mathcal{H}_r$ is a total binary relation on $L^r_p \times L^{r\oplus1}_p$.



η_{r1}
η_{r2}
$t^r_{(1)(0\rightarrow1)}$
$t^r_{(2)(1\rightarrow2)}$
$t^r_{(0)(2\rightarrow0)}$
$t^r_{(0)(1\rightarrow0)}$
$t^r_{(1)(2\rightarrow1)}$
$t^r_{(2)(0\rightarrow2)}$
L^r_p
L^r_p

Figure 6.1: Pseudolivelocks Over Local Transitions of a 3-Agreement Protocol



ε_{r1}
ε_{r2}
L^r_p
L^{r+1}_p
$t^r_{(1)(0\rightarrow1)}$
$t^r_{(2)(1\rightarrow2)}$
$t^r_{(0)(2\rightarrow0)}$
$t^r_{(0)(1\rightarrow0)}$
$t^r_{(1)(2\rightarrow1)}$
$t^r_{(2)(0\rightarrow2)}$
$t^{r+1}_{(1)(0\rightarrow1)}$
$t^{r+1}_{(2)(1\rightarrow2)}$
$t^{r+1}_{(0)(2\rightarrow0)}$
$t^{r+1}_{(0)(1\rightarrow0)}$
$t^{r+1}_{(1)(2\rightarrow1)}$
$t^{r+1}_{(2)(0\rightarrow2)}$

Figure 6.2: Feasible Propagations Over Local Transitions of a 3-Agreement Protocol

We decompose $\mathcal{E}_r$ into two disjoint bijections $\epsilon_{r1} : L^r_p \rightarrow L^{r\oplus1}_p$ and $\epsilon_{r2} : L^r_p \rightarrow L^{r\oplus1}_p$.

- $\epsilon_{r1}(t^r_{(v)(v\oplus1\rightarrow v)}) = t^{r\oplus1}_{(v)(v\oplus1\rightarrow v)}$,
- $\epsilon_{r1}(t^r_{(v)(v\ominus1\rightarrow v)}) = t^{r\oplus1}_{(v)(v\ominus1\rightarrow v)}$,
- $\epsilon_{r2}(t^r_{(v)(v\oplus1\rightarrow v)}) = t^{r\oplus1}_{(v)(v\ominus1\rightarrow v)}$, and

- $\epsilon_{r2}(t^r_{(v)(v\ominus 1\rightarrow v)}) = t^{r\oplus 1}_{(v)(v\oplus 1\rightarrow v)}$.

Figure 6.2 captures the decomposition of ϵ_r into ϵ_{r1} and ϵ_{r2}. $\lhd$

6.2.2 Conditions for Livelock-Freedom in Unidirectional Rings

We incrementally establish a set of necessary and sufficient conditions for livelock-freedom in unidirectional rings.

Lemma 6.2.1 (Existence of a pseudolivelock)**.** *If $p(K)$ has a livelock L, then for every process P_r there exists a pseudolivelock $\eta_r \subseteq \mathcal{H}_r$ strongly connecting L^r_p.*

Proof. Let $p(K)$ have a livelock L. Lemma 5.3.1 implies that every P_r participates in L. Project L on x_r to obtain the recurrent sequence of local transition $\sigma_r = \ll t^r_0, t^r_1, \cdots, t^r_i, t^r_{i\oplus 1}, \cdots, t^r_{m-1} \gg^*$ of P_r. As such, every t^r_i in σ_r is infinitely often enabled, thus, for all $0 \leq i \leq m-1$: t^r_i partially enables $t^r_{i\oplus 1}$ through the update of x_r. Therefore, there exists a cycle $\eta_r = \{(t^r_i, t^r_{i\oplus 1})\} \subseteq \mathcal{H}_r$ and η_r strongly connects $L^r_p = \{t^r_i\}$, where L^r_p is the set of local transitions in σ_r. Hence, $\eta_r \subseteq \mathcal{H}_r$ is a pseudolivelock that strongly connects L^r_p, for every P_r. $\square$

Lemma 6.2.2 asserts the existence of feasible propagations $\epsilon_r \subseteq (L^r_p \times L^{r\oplus 1}_p) \cap \mathcal{E}_r$ for every P_r in a livelock of $p(K)$.

Lemma 6.2.2 (Existence of a feasible propagation)**.** *If $p(K)$ has a livelock L, then for every P_r there exists a feasible propagation designated by a total binary relation $\epsilon_r \subseteq (L^r_p \times L^{r\oplus 1}_p)$ where L^r_p is the set of local transitions of some pseudolivelock generated by L in P_r.*

Proof. Since $p(K)$ has a livelock L, every P_r has a pseudolivelock η_r whose set of local transitions is designated by L^r_p (Lemma 6.2.1). In L, for every P_r, the execution of every local transition in L^r_p passes an enablement to $P_{r\oplus 1}$ as implied by Properties 5.3.1 and 5.3.4. As such, the execution of every $t^r \in L^r_p$ partially enables a transition $t^{r\oplus 1} \in L^{r\oplus 1}_p$, for every $0 \leq r \leq K-1$. Otherwise, we have two cases: (1) there exists a local transition in L^r_p that does not enable any transition in $L^{r\oplus 1}_p$, thereby not propagating an enablement to $P_{r\oplus 1}$ or, (2) there exists a local transition in $L^{r\oplus 1}_p$ that is not enabled by any transition in L^r_p, thereby blocking the pseudolivelock $\eta_{r\oplus 1}$. Both cases contradict the existence of L. Therefore, ϵ_r is a total binary relation consisting of the set of pairs $(t^r, t^{r\oplus 1}) \in L^r_p \times L^{r\oplus 1}_p$ such that the execution of t^r partially enables $t^{r\oplus 1}$; i.e., $\epsilon_r \subseteq ((L^r_p \times L^{r\oplus 1}_p) \cap \mathcal{E}_r)$. $\square$

For simplicity of presentation, we define the binary relation $\kappa_r \subseteq L_p^r \times L_p^{r\ominus 1}$ linking two local transitions $(t^r, t^{r\ominus 1}) \in L_p^r \times L^{r\ominus 1}$ if and only if there exists $t_p^{r\ominus 1} \in L_p^{r\ominus 1}$ such that $t_p^{r\ominus 1}$ partially enables $t^{r\ominus 1}$ in the pseudolivelock $\eta_{r\ominus 1}$ and $t_p^{r\ominus 1}$ partially enables t^r in the feasible propagation $\epsilon_{r\ominus 1}$. Definition 6.2.3 formalizes this notion.

Definition 6.2.3 (Contiguousness). $\kappa_r = \epsilon_{r\ominus 1}^{-1} \circ \eta_{r\ominus 1}$.

We establish our necessary and sufficient conditions for the existence of livelocks in the following Theorem.

Theorem 6.2.4. $p(K)$ *has a livelock* L if and only if *for every* P_r*, there exists a pseudolivelock* $\eta_r \subseteq \mathcal{H}_r$*, strongly connecting* L_p^r*, and a feasible propagation* $\epsilon_r : L_p^r \rightarrow L_p^{r\oplus 1}$ *that satisfy the following two equations:*

1. Ring Reconstruction Equation: $\exists |E| > 0$: $\kappa_{r\rightsquigarrow r-|E|+1} = \left\{ \begin{array}{ll} \epsilon_{r\rightsquigarrow r+K-|E|-1} & |E| < K \\ e & |E| = K \end{array} \right\}$, *and*

2. Enablement Flow Equation: $\eta_r \circ \epsilon_r = \epsilon_r \circ \eta_{r\oplus 1}$.

Proof. Necessity of the Conditions for the Existence of Livelocks: Since $p(K)$ has a livelock L, it follows from Lemmas 6.2.1 and 6.2.2 that for every P_r, there exist a pseudolivelock $\eta_r : L_p^r \rightarrow L_p^r$ and a feasible propagation $\epsilon_r : L_p^r \rightarrow L_p^{r\oplus 1}$.

1. *Ring Reconstruction Equation*. Consider a livelock L with $|E|$ ($|E| > 0$) enablements in the ring of size K. There exists a contiguous livelock C_L that preserves the partial order imposed on the local transitions of L (Lemma 5.3.11). Consider a global state of C_L where $|E|$ enablements are adjacent in processes $P_{r-|E|+1}$ to P_r, and t_j^r is enabled in P_r. A propagation of the rightmost enablement in C_L is a sequence of $K - |E|$ propagations from P_r to $P_{r+K-|E|-1}$. By definition of ϵ_r and κ_r, the enabled sequence of local transitions along the direction of propagation of the rightmost enablement of C_L is $\ll t_j^r, \epsilon_r(t_j^r), (\epsilon_r \circ \epsilon_{r\oplus 1})(t_j^r), \cdots, \epsilon_{r\rightsquigarrow(r+K-|E|-1)}(t_j^r) \gg$; where $t_j^{r\oplus K-|E|-1} = \epsilon_{r\rightsquigarrow(r+K-|E|-1)}(t_j^r)$. On the other hand, going anticlockwise along the contiguous sequence of enabled local transitions reached after $K - |E|$ propagations yields the same $t_j^{r\oplus K-|E|-1} = \kappa_{r\rightsquigarrow(r-|E|+1)}(t_j^r)$. The Ring Reconstruction Equation follows immediately.

2. *Flow Equation*. For some arbitrary P_r, let $t_j^r \in L_p^r$ be an arbitrary enabled local transition. Since L is a livelock, every local transition in L_p^r eventually executes in the sequence projected from L on x_r and captured by η_r; i.e., $\sigma_r = \ll t_j^r, \eta_r(t_j^r), (\eta_r \circ$

$\eta_r)(t_j^r), \cdots, \eta_r^{m_r-1}(t_j^r) \gg^*$. Since every local transition of L_p^r is self-disabling, σ_r can execute if and only if every local transition in σ_r gets enabled by some local transition $t_j^{r\ominus 1} \in \sigma_{r\ominus 1}$. In fact, the enablement of every t_j^r in a unidirectional ring depends only on the values of $x_{r\ominus 1}$ and x_r, i.e.; can only be enabled by the execution of $\eta_r^{-1}(t_j^r) = t_{j\ominus 1}^r$ and $\epsilon_{r\ominus 1}^{-1}(t_j^r) = t_j^{r\ominus 1}$. Note that local transitions as illustrated in Figure 6.4 are not all necessarily distinct, in fact m is the least common multiple of m_r and $m_{r\oplus 1}$. In L, $t_{j\ominus 1}^r \prec t_j^{r\ominus 1}$ since the execution of $t_j^{r\ominus 1}$ either (1) collides with $t_{j\ominus 1}^r$ if $|E| > 1$ or (2) transitively depends on t_j^r due to a consecutive propagation on the ring if $|E| = 1$; i.e., $t_j^{r\ominus 1} = \epsilon_{r\rightsquigarrow r+K-1}(t_{j\ominus 1}^r)$. In case (1), clearly $t^{r\ominus 1} = \kappa_r(t_{j\ominus 1}^r)$, while in case (2), using the Reconstruction Equation, we reach the same result. Thus, $\kappa_r(t_{j\ominus 1}^r) = t_j^{r\ominus 1}$ for $|E| > 0$ (Figure 6.3). Combining identities to obtain relations only on t_j^r, we obtain $(\kappa_r \circ \epsilon_{r\ominus 1})(t_j^r) = \eta_r(t_j^r)$. By substitution for κ_r from Definition 6.2.3, we obtain the Enablement Flow Equation.

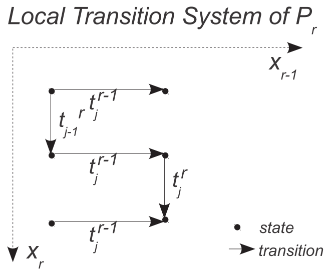


Figure 6.3: The source state of t_i^r and $t_j^{r\ominus 1}$ share the value of $x_{r\ominus 1}$, that is why $\kappa_r(t_i^r) = t_j^{r\ominus 1}$ for any $|E| > 0$

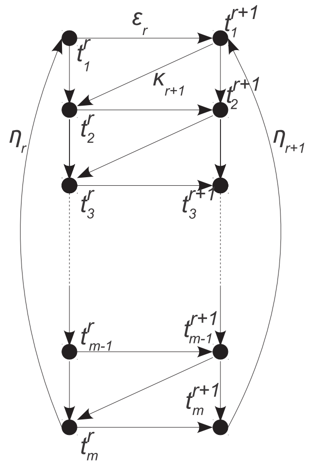


Figure 6.4: ϵ_r preserves the cyclic structure of η_r and $\eta_{r\oplus 1}$.

Sufficiency of the Conditions for the Existence of Livelocks: We demonstrate that, if for every P_r, there exists η_r and ϵ_r satisfying the Flow and the Ring Reconstruction Equations

(κ_r is redundant according to Definition 6.2.3), then $p(K)$ has a contiguous livelock with $|E|$ enablements. We construct a livelock with $|E|$ enablements as follows:

Initial State. Let $t^r \in L^r_p$ be a local transition in the pseudolivelock η_r whose cyclic structure is preserved by the feasible propagation ϵ_r, where $0 \leq r \leq K-1$. Since the Reconstruction Equation holds for some $|E|$ and K, we construct a global state s_I of $p(K)$ with $|E|$ enabled local transitions $t^r = (s^r, s^{r\prime})$ in L^r_p where $0 \leq r \leq |E|-1$, $t^{|E|-1}$ being the rightmost enabled local transition. For each enabled P_r ($0 \leq r \leq |E|-1$) in s_I, let $t^r = \kappa_{r\oplus 1}(t^{r\oplus 1})$. Each disabled process P_r, $|E| \leq r \leq K-1$ is in the target local state $s^{r\prime}$ of its corresponding disabled local transition t^r. For each disabled P_r ($|E| \leq r \leq K-1$) in s_I, let $t^{r\oplus 1} = \epsilon_r(t^r)$. It is easy to verify that the Reconstruction Equation holds for $t^{|E|}$.

Execution. Substituting the result of Definition 6.2.3 into the Flow Equation, we obtain an equivalent form of the Flow Equation where $\kappa_r \circ \epsilon_{r\ominus 1} = \eta_r$. We use structural induction to demonstrate that every disabled P_r eventually gets enabled provided that the Flow Equation holds at every process.

- *Base Case.* We demonstrate that the execution of $t^{|E|-1}$ enables $P_{|E|}$. We apply the latter form of the Flow Equation at $P_{|E|}$ in the initial global state s_I to obtain that $(\kappa_{|E|} \circ \epsilon_{|E|-1})(t^{|E|}) = \eta_{|E|}(t^{|E|})$. Substituting the value of $\kappa_{|E|}(t^{|E|})$ from s_I, we obtain that $\epsilon_{|E|-1}(t^{|E|-1}) = \eta_{|E|}(t^{|E|})$. However, since $t^{|E|}$ already executed in $P_{|E|}$; i.e, is disabled in target local state $s^{|E|\prime}$, $\eta_{|E|}(t^{|E|})$ is partially enabled by $t^{|E|}$ and the execution of $t^{|E|-1}_j$ fully enables $\eta_{|E|}(t^{|E|})$ in $P_{|E|}$.
- *Induction Hypothesis.* The execution of every disabled process P_q for $|E| \leq q < r$ enables $P_{q\oplus 1}$.
- *Induction Step.* P_r is enabled in $\eta_r(t^r)$, $P_{r\oplus 1}$ is disabled in $s^{r\oplus 1\prime}$ and $t^{r\oplus 1}$ partially enables $\eta_{r\oplus 1}(t^{r\oplus 1})$ in $P_{r\oplus 1}$. In s_I, $t^{r\oplus 1} = \epsilon_r(t^r)$, therefore, (1) $t^{r\oplus 1}$ partially enables $(\epsilon_r \circ \eta_{r\oplus 1})(t^r)$. In P_r, (2) $\eta_r(t^r)$ partially enables $(\eta_r \circ \epsilon_r)(t^r)$. Since the Flow Equation holds at P_r, we conclude from (1) and (2) that $\eta_r(t^r)$ and $t^{r\oplus 1}$ partially enable the same local transition $\eta_{r\oplus 1}(t^{r\oplus 1})$. By the induction hypothesis, $\eta_r(t^r)$ is enabled in P_r, and $\eta_{r\oplus 1}(t^{r\oplus 1})$ is partially enabled by the executed $t^{r\oplus 1}$. Thus, the execution of $\eta_r(t^r)$ in P_r fully enables $\eta_{r\oplus 1}(t^{r\oplus 1})$.
- *Terminal Case.* At P_{K-1}, applying the Flow Equation yields $\epsilon_{K-1}(\eta_{K-1}(t^{K-1})) = \eta_K(t^0)$. However, the execution of t^{K-1} does not enable $\eta_K(t^0)$ and disables t^0 that has not executed yet, thereby contradicting the Flow Equation. Thus, no collision occurs at P_{K-1} with P_0.

This proves that as long as the Flow Equation holds, collisions do not occur and propagation always occurs for every P_r.

By repeating the similar structural induction for every propagation of the rightmost enablement, and since η_r is a cycle, we construct a contiguous livelock C_L of $p(K)$ having $|E|$ enablements. Thus follows from Lemma 5.3.11 that $p(K)$ has a livelock L. □

The local nature of pseudolivelocks and feasible propagations allows us to exactly characterize livelocks in a unidirectional ring for arbitrary K. Intuitively, the Flow Equation asserts that $\epsilon_r \subseteq L_p^r \times L_p^{r\oplus1}$ is a *cyclic structure-preserving* total binary relation. Given an initial global state from which pseudolivelocks could start their execution by satisfying the Reconstruction Equation, the Flow Equation guarantees a seamless propagation of enablements for every process P_r. Therefore, a livelock exists. Conversely, the existence of a livelock L under the assumption of self-disablement imposes an order on the execution of enabled local transitions in adjacent processes to avoid collisions (Lemma 5.3.10). As such, the structure of pseudolivelocks, whose connectedness is preserved by feasible propagations, is the only representation for L.

Example 6.2.2 (3-valued Agreement)**.**

Consider the pseudolivelock represented by $\mathcal{H}_r = \eta_{r1} \cup \eta_{r2}$ as in Figure 6.1. Consider $\mathcal{E}_r = \epsilon_{r1} \cup \epsilon_{r2}$ as depicted in Figure 6.2. For simplicity of presentation, we check for potential livelocks formed by η_{r1} and ϵ_{r1} restricted to $L_p^r = \{t^r_{(v\oplus1)(v\to v\oplus1)} | 0 \leq v \leq 2\}$.

In Figure 6.1, $\eta_{r1}(t^r(v)) = t^r(v \oplus 1)$. Figure 6.2 illustrates that $\epsilon_{r1}(t^r(v)) = t^{r\oplus1}(v)$. Checking the Flow Equation, we obtain $(\eta_{r1} \circ \epsilon_{r1})(t^r(v)) = \epsilon_{r1}(t^r(v\oplus1)) = t^{r\oplus1}(v\oplus1)$. The right hand side of the Flow Equation evaluates to $(\epsilon_{r1} \circ \eta_{(r\oplus1)_1})(t^r(v)) = \eta_{(r\oplus1)_1}(t^{r\oplus1}(v))$ $=t^{r\oplus1}(v \oplus 1)$. Therefore, the Enablement Flow Equation holds for η_{r1} as a pseudolivelock and ϵ_r^1 as a feasible propagation. Since our derivation is independent of r, the Flow Equation holds for every P_r. Note that $\eta_{r1} \circ \epsilon_{r2} \neq \epsilon_{r2} \circ \eta_{(r\oplus1)_2}$; i.e., ϵ_{r2} does not preserve the cylic structure of η_{r1} in $\eta_{(r\oplus1)_2}$.

To apply the Reconstruction Equation, we evaluate $\kappa_r(t^r(v)) = (\epsilon_{r\ominus1}^{-1} \circ \eta_{r\ominus1})(t^r(v))$=$t^{r\ominus1}(v \oplus 1)$. We substitute $t^r(v)$ into the Reconstruction Equation to obtain $t^{r-|E|}(v \oplus |E|) = t^{r-|E|}(v)$. In other words, $|E| = 0$ in modulo 3; i.e., $|E|$ is a multiple of 3 regardless of the value of K. Thus, 3-valued agreement has a livelock formed by local transitions in L_p^r. This completes our proof. ◁

Notice that, using the Reconstruction Equation at some local transition $t_j^r \in L_p^r$, we can form an initial global state of a contiguous livelock in $p(K)$ with $|E|$ enabled processes. The universally quantified version of the Reconstruction Equation follows directly from the iterative application of the Flow Equation to the initial global state. In this way, the Reconstruction Equation needs to hold only for some process P_r and at some local transition in L_p^r.

Example 6.2.3 (6-valued Agreement)**.**

To appreciate the independence of the Flow and Reconstruction Equations, consider a 6-valued agreement protocol where $K = 4$ and consisting of transitions in $\delta_r = L_p^r = \{t^r_{(v\oplus1)(v\rightarrow v\oplus1)} | 0 \leq v \leq 5\}$. Despite the fact that the Flow Equation holds at every P_r, using a similar reasoning to the one in Example 6.2.2, the Ring Reconstruction Equation holds only for $|E| = 0$ when $K = 4$. In fact, substitution in the reconstruction equation yields that $|E|$ is a multiple of 6. This demonstrates that 6-valued agreement is livelock free for $2 \leq K \leq 5$. $\triangleleft$

Corollary 6.2.5 establishes another form of the Ring Reconstruction Equation.

Corollary 6.2.5 (Calculation of $|E|$)**.** *If $p(K)$ has a livelock L with $|E|$ enablements, then $\eta_r^{|E|} = \epsilon_{r\rightsquigarrow r+K-1}$, for every process P_r.*

Proof. We present a sketch of the proof. We advise the reader to workout the details of the algebra as an exercise. Theorem 6.2.4 implies the Flow and Reconstruction Equations. Substitute for κ_r from Definition 6.2.3 in the Ring Reconstruction Equation to obtain an expression in terms of η_r and ϵ_r. Right and left-compose the Flow Equation by ${\epsilon_r}^{-1}$. The result allows to drag left all of η_r in the Reconstruction Equation and the lemma follows. $\square$

Corollary 6.2.5 is not sufficient for proving the existence of livelocks. To illustrate that, we consider a Sum-Not-Two Example.

Example 6.2.4 (Sum-Not-Two)**.**

The Sum-Not-Two protocol has variables' domains $D_r = \{0, 1, 2\}$. The set of legitimate states is such that, for every P_r, $x_r + x_{r\ominus1} \neq 2$.

The set of local transitions is $\delta_r = \{t^r_{(v)((2\ominus v)\rightarrow(3\ominus v))} | v \in D_r\}$. Thus, δ_r has only three local transitions in a pseudolivelock; i.e., $L_r^p = \delta_r$. Since the transitions of δ_r are fully determined by v, we use the shorthand $t^r(v)$ for $t^r_{(v)((2\ominus v)\rightarrow(3\ominus v))}$. $\eta_r(t^r(v)) = t^r_{(v\ominus1)((3\ominus v)\rightarrow(1\ominus v))} = t^r(v \ominus 1)$. $\epsilon_r(t^r(v)) = t^{r\oplus1}(3 \ominus v)$. The left hand side of the Flow Equation evaluates to $t^{r\oplus1}(1 \ominus v)$ while the right hand side evaluates to $t^{r\oplus1}(2 \ominus v)$. Thus, the Flow Equation does not hold and $p(K)$ is livelock-free.

On the other hand, applying Corollary 6.2.5 generates a pair of values for K and $|E|$ where K is even and $|E|$ is a multiple of 3. Consequently, Corollary 6.2.5 provides a necessary but insufficient condition for the existence of livelocks. $\triangleleft$

Example 6.2.5 (l-valued Agreement)**.**

Consider a general agreement protocol where $D_r| = \{0, 1, \cdots, l-1\}$ for all r. The set of local legitimate states of P_r is captured by $x_{r\ominus 1} = x_r$. Consider the following candidate protocol.

$$t^r(v, w) : (x_{r\ominus 1} = w) \wedge (x_r = v) \to x_r := w$$

where $v, w \in D_r$ are parameters. We demonstrate how to determine what type of constraints are put on $t^r_{(v,w)}$.

By definition of η_r, we get $\eta_r(t^r(v, w))$ = $t^r(w, y)$, where $y \in D_r$. Applying ϵ_r to $t^r(w, y)$ generates $t^{r\oplus 1}(z, y)$. Computing the right hand side of the Flow Equation in a similar way, we obtain $\epsilon_r(t^r(v, w)) = t^{r\oplus 1}(y_1, w)$ and $\eta_{r\oplus 1}(t^{r\oplus 1}(y_1, w)) = t^{r\oplus 1}(w, z_1)$. The Flow Equation holds if and only if $w = z$ and $y = z_1$. Thus, $\epsilon_r(t^r(v, w)) = t^{r\oplus 1}(v, w)$. In other words, the Flow Equation holds for an $l-1$ Agreement protocol if and only if a local transition in an arbitrary pseudolivelock of P_r enable exactly the same local transition in $P_{r\oplus 1}$.

Let m denote the length of a sequence of local transitions inside repeat inside a pseudolivelock; i.e., the multiplicity/periodicity of η_r. The calculation of $|E|$ according to Corollary 6.2.5 yields $\eta_r^{|E|}(t^r(v, w)) = t^r(v, w)$, this is possible if and only if $|E|$ is a multiple of m.

As such, we exactly characterize the set of possible livelocks in an $l-1$ agreement protocol of arbitrary size. We conclude that breaking symmetry, guarantees livelock-freedom for a general agreement protocol. $\triangleleft$

6.3 Additional Examples

In this section, we examine three classical protocols on a unidirectional ring to exactly determine their livelocks.

Example 6.3.1 (l-coloring)**.**

A general parameterized set of local transitions for an l-coloring protocol has the form: $t^r(v, w) : (v)(v \to w)$, where $v \neq w$. We start by applying the Flow Equation to $t^r(v, w)$.

$\eta_r(t^r(v,w)) = t^r(w,y)$ and $\epsilon_r(t^r(w,y)) = t^{r\oplus1}(y,z)$. $\epsilon_r(t^r(v,w)) = t^{r\oplus1}(w,y_1)$ and $\eta_{r\oplus1}(t^{r\oplus1}(w,y_1)) = t^{r\oplus1}(y_1,z_1)$. The Flow Equation holds if and only if $y_1 = y$ and $z_1 = z$. Substituting into the definitions of η_r and ϵ_r, we deduce that $\epsilon_r = \eta_r\circ$ *shift*$_1$ = *shift*$_1 \circ \eta_{r\oplus1}$, where *shift*$_k(t^r(v)) = t^{r\oplus k}(v)$. Clearly, *shift*$_K$ = *shift*$_1^K$ = e. We identify this commutativity property as a *shift invariance* property.

By substitution into the result of Corollary 6.2.5 and reuse of the shift invariance property, we obtain that $\eta_r^{|E|} = \eta_r^K$. Thus follows that $K - |E|$ is a multiple of m, where m is the length of the cycle of pseudolivelock η_r. Note that shift invariance holds if and only if L_p^r are all symmetric; i.e., have the same set of transitions upto shifts.

Since every local state of P_r where $x_r = x_{r\ominus1}$ should have an outgoing local transition in a self-stabilizing coloring protocol [57], our result implies an impossibility of self-stabilizing symmetric l-coloring over unidirectional rings.

Example 6.3.2 (Sum-Not-($l-1$)).

In a Sum-Not-($l-1$) protocol, a legitimate local state is such that $x_{r\ominus1} + x_r \neq l-1$, where $D_r = \{0, 1, \cdots, l-1\}$. A general parameterized action detects whether the sum of $x_{r\ominus1}$ and x_r is $l-1$ and updates the value of x_r accordingly. We study the conditions under which livelocks form in Sum-Not-($l-1$).

A parameterized local transition of P_r has the form $t^r(v) : (l-1\ominus v)(v \to f^r(v))$, where $f^r : D_r \to D_r$ is an arbitrary bijection. Applying the Flow Equation on $t^r(v)$, $(\eta_r\circ\epsilon_r)(t^r(v)) = \epsilon_r(t^r(f^r(v))) = t^{r\oplus1}(l-1\ominus f^r(v))$. $(\epsilon_r\circ\eta_{r\oplus1})(t^r(v)) = \eta_{r\oplus1}(t^{r\oplus1}(l-1\ominus v))$ = $t^{r\oplus1}(f^{r\oplus1}(l-1\ominus v))$. Hence, the Flow Equation holds if and only if $f^{r\oplus1}(l-1\ominus v) \oplus f^r(v) = l-1$. Choosing $f^r(v) = v \oplus i_r$ for every r; the Flow Equation reduces to $l-1\ominus(v\oplus i_r)$ = $(l-1\ominus v)\oplus i_{r\oplus1}$. This holds only when $i_r \oplus i_{r\oplus1} = 0$ in modulo l. In fact, choosing $i_r = i_{r\oplus1} \neq l/2$ guarantees livelock freedom.

Example 6.3.3 (Dijkstra's Token Ring).

Dijkstra's token ring [7] is a mutual exclusion protocol over a unidirectional ring where $D_r = \{0, \cdots, l-1\}$. A legitimate global state of the ring is such that $|E| = 1$. Dijkstra provides a protocol where δ_0 has the parameterized local transition $t^0(v) : (v)(v \to v\oplus 1)$ and δ_i $(1 \leq i \leq K-1)$ consists of the local transitions $t^i(v,w) : (w)(v \to w)$, $(v \neq w)$. We illustrate a proof of livelock-freedom for $|E| > 1$ when $l \geq K-1$ by using the Flow Equation in Theorem 6.2.4 and Corollary 6.2.5.

At P_0, we apply the Flow Equation to obtain, $\eta_0(t^0(v))$ = $t^0(v\oplus1)$ and $\epsilon_0(t^0(v\oplus1))$ = $t^1(w, v\oplus2)$. $\epsilon_0(t^0(v))$ = $t^1(w_1, v\oplus1)$ and $\eta_1(t^1(w_1, v\oplus1))$ = $t^r(v\oplus1, z_1)$. The Flow

Equation holds if and only if $w = v \oplus 1$ and $z_1 = v \oplus 2$. Thus, $\epsilon_0(t^0(v)) = t^1(v, v \oplus 1)$, $w_1 = v$, $\eta_1(t^1(v, v \oplus 1)) = t^1(v \oplus 1, v \oplus 2)$ and $L_p^1 = \{t^1(v, v \oplus 1) | v \in D_1\}$.

At P_i, $1 \leq i \leq K - 2$, $(\eta_i \circ \epsilon_i)(t^1(v, v \oplus 1)) = \epsilon_i(t^i(v \oplus 1, v \oplus 2)) = t^{i\oplus 1}(w, v \oplus 2)$. $(\epsilon_i \circ \eta_{i\oplus 1})(t^1(v, v \oplus 1)) = \eta_{i\oplus 1}(t^{i\oplus 1}(w_1, v \oplus 1)) = t^{i\oplus 1}(v \oplus 1, z_1)$. The Flow Equation holds if and only if $w = v \oplus 1$ and $z_1 = v \oplus 2$. It follows that $\epsilon_i(t^i(v, v \oplus 1)) = t^{i\oplus 1}(v, v \oplus 1)$ and $w_1 = v$. Thus, $\eta_{i\oplus 1}(t^{i\oplus 1}(v, v \oplus 1)) = t^{i\oplus 1}(v, v \oplus 1)$ and $L_p^{i\oplus 1} = \{t^{i\oplus 1}(v, v \oplus 1) | v \in D_{i\oplus 1}\}$.

So far, the feasible propagation at P_{K-1} is the only undetermined relation. We apply the Flow Equation at P_{K-1}. $(\eta_{K-1} \circ \epsilon_{K-1})(t^{K-1}(v, v \oplus 1)) = \epsilon_{K-1}(t^{K-1}(v \oplus 1, v \oplus 2)) = t^0(v \oplus 2)$. $(\epsilon_{K-1} \circ \eta_0)(t^{K-1}(v, v \oplus 1)) = \eta_0(t^0(v \oplus 1)) = t^0(v \oplus 2)$. Hence, the Flow Equation holds at every P_r.

To summarize our results, we have $\eta_r(t^r(v, v \oplus 1)) = t^r(v \oplus 1, v \oplus 2)$ for $r \neq 0$, $\eta_0(t^0(v)) = t^0(v \oplus 1)$, $\epsilon_r(t^r(v, v \oplus 1)) = t^{r\oplus 1}(v, v \oplus 1)$ for $1 \leq r \leq K - 2$, $\epsilon_0(t^0(v)) = t^1(v, v \oplus 1)$ and $\epsilon_{K-1}(t^{K-1}(v, v \oplus 1)) = t^0(v \oplus 1)$.

Applying Corollary 6.2.5 at P_0, we obtain that $t^0(v \oplus |E|) = t^0(v \oplus 1)$; this holds if and only if $|E| - 1$ is a multiple of l: $t^0(v)$ increases v by 1 in modulo l. That is to say, $|E| = (k \times l) + 1$, where $k \in \mathbb{N}$. It follows that $|E| \in \{1, l + 1, 2l + 1, \cdots, kl + 1, \cdots\}$; these are the values of $|E|$ for which Dijkstra's token ring has livelocks. To guarantee livelock freedom for $|E| \neq 1$, $l \geq K$ should always hold. For the case where $l = K - 1$, Dijkstra's token ring has a livelock with K enablements, however, this livelock exactly occurs when all processes synchronously execute; i.e., under fully synchronous execution semantics. In other words, Dijkstra's token ring has no livelocks with $|E| > 1$ for $l \geq K-1$ assuming that all K processes can never execute together. For $l \geq K$, the protocol has no livelocks with $|E| > 1$ assuming no constraints, whatsoever, on the execution semantics. Our method detects that the only class of livelocks in Dijkstra's token ring are propagations of only one enablement: this is a legitimate behavior of Dijkstra's token ring. Since our equations capture the set of all livelocks in $p(K)$, this proves that Dijkstra's token ring has no livelocks outside $I(K)$.

6.4 Summary and Extensions

Livelocks are among the most intricate concurrency flaws in distributed algorithms. Verification of livelocks in arbitrary-sized rings is generally undecidable [45]. We devised an exact characterization of livelocks in a subclass of protocols on unidirectional rings whose processes are self-disabling; i.e., the execution of an action of some process disables all the actions of that process. Our characterization reduced verification and design of livelock-freedom to the solution of equations on compositions of binary relations representing processes local looping: pseudolivelocks, and dependency among local loops: feasible propagations. Due to the local nature of our binary relations, our characterization is independent of the number of processes in the network, thereby circumventing global state space exploration. We illustrated the feasibility of our mathematical model through four examples: agreement, l-coloring, sum-not-$(l-1)$ and the classical Dijktra's token-ring [7].

One major application of our framework is in the automatic design and verification of convergence [9]. By combining methods for local reasoning about deadlocks, described in Chapter 5, with our algebraic characterization of livelocks. We shall investigate sound and complete algorithms – that hinder altogether state explosion – for the automatic synthesis of convergence in unidirectional rings. We plan to generalize our current results to a wider class of protocols in two directions: (1) we shall consider arbitrary network topologies with self-disabling processes, (2) we shall investigate unidirectional rings whose processes are not necessarily self-disabling. In the first case, our current characterization is sufficient for proving the existence of livelocks in a sub-ring of the arbitrary network. In the second case, the undecidability of the problem enforces us to study non-identical necessary and sufficient conditions for livelock-freedom.

Chapter 7

Application: Fault Tolerance of Wireless Sensor Nodes[1]

Most existing techniques for the design and implementation of fault tolerance use resource redundancy. As such, due to scarcity of resources, it is difficult to directly apply them for adding fault tolerance to sensor nodes in Wireless Sensor Networks (WSNs). Thus, it is desirable to develop techniques that implement fault tolerance under the constraints of memory and processing power of sensor nodes. We present a novel method for designing recovery from transient faults that cause non-deterministic bit-flips in the task queue of the scheduler of TinyOS, which is the operating system of choice for sensor nodes. Specifically, our approach exploits computational redundancy for the design of recovery instead of using resource redundancy. The presented fault-tolerant task queue recovers from bit-flips with significantly lower space/time overhead compared with the Error Correction Codes.

7.1 Introduction

Wireless Sensor Networks (WSNs) are increasingly used in mission-critical applications (e.g., body sensor networks, habitat monitoring, flood forecasting, etc.), where they have to be deployed in harsh environments (e.g., volcano, forest, battle field, etc.). On one hand, WSNs must exhibit a high degree of service dependability due to application requirements, and on the other hand, unexpected environmental events, i.e., *faults*, may negatively

[1]This chapter is adapted with permission from our publication [58] in the proceedings of the Conference on Software Engineering and Knowledge Engineering (SEKE 2011), Miami, Florida, July 2011. Please refer to the corresponding permission letter in the supplementary document to this dissertation.

affect their quality of service. For example, *transient faults* may cause non-deterministic bit-flips in the main memory of sensor nodes (a.k.a. *motes*), thereby perturbing the state of the running program to an arbitrary state in its state space. Since the quality of the service provided by the entire sensor network heavily relies on the dependability of the controlling software of motes, fault tolerance techniques should be applied to improve the dependability of motes. Nonetheless, due to their limited computational (e.g., memory and processing power) and energy resources, it is impractical to apply the traditional fault tolerance methods (e.g., Error Correction Code (ECC) [59]) to motes. This chapter proposes a novel method that exploits computational redundancy for the addition of recovery to transient bit-flips in the task queue of TinyOS [60].[2]

Most existing techniques [61]–[65] present solutions for the design of fault-tolerant protocols for WSNs rather than focusing on the fault tolerance of individual sensor nodes. For instance, techniques for reliable transmission are mostly based on redundant and/or multi-path retransmission [63]. Several methods exist for (i) designing self-stabilizing WSN communication protocols [64] that ensure a correct synchronization among sensor nodes starting from an arbitrary non-synchronized state, and (ii) providing recovery for data dissemination in WSNs [65]. ECC methods (e.g., Hamming code [59]) often require extensive memory redundancy for storing extra parity bits in code words. Moreover, decoding/coding algorithms in these methods are computationally expensive.

We propose a novel approach that enables space/time-efficient recovery to transient Bit-Flips (BFs) in motes. The proposed approach is based on the detection of the violations of invariance conditions that must always be true and dynamic corrections of such violations. Specifically, we focus on the task queue of the TinyOS as it is one of the most critical components of the kernel of TinyOS and its structure is heavily sensitive to BFs. We first define conditions under which the task queue has a valid structure, called the *structural invariant*. Then, before and after the addition/removal of a task to/from the task queue, we check whether the structural invariant holds. In case of the violation of the invariant, we identify different failure scenarios created due to the occurrence of BFs and systematically correct them, thereby recovering to the structural invariant. The proposed approach enables the detection and correction of multiple BFs in a single-byte variable in a space/time-efficient fashion. Compared with the Hamming Code (HC) [59], our approach needs at least 20% less memory and performs at least twice as fast as HC. The time complexity of our approach is linear in the size of the task queue. We also note that HC cannot correct multiple BFs whereas our approach enables the correction of multiple BFs as long as they occur in the same variable. Furthermore, for some special cases, the proposed approach corrects BFs in multiple variables as well.

Organization. Section 7.2 illustrates the structure of the TinyOS task queue. Then, Section

[2]TinyOS is the operating system of choice for sensor nodes in WSNs.

7.3 presents our approach for the detection and correction of transient bit flips in the TinyOS task queue. We also demonstrate the superiority of the space/time efficiency of the proposed approach compared with the ECC methods. Section 7.4 makes concluding remarks and outlines future research directions.

7.2 Structural Invariance of TinyOS Task Queue

In this section, we define what constitutes a valid structure of TinyOS's task queue. In Tiny OS version 2.x, the task queue is a linked list of task identifiers implemented as a statically allocated array of 256 entries (see Figure 7.1). Figure 7.2 illustrates the implementation of the task queue in nesC [66], which is a component-based variant of the C programming language used for application development on TinyOS. Each identifier (ID) is an integer between 0 and 255 inclusive. The set of variables of interest are `m_head`, `m_tail` and `m_next[256]`. `m_head` holds the index of the oldest ID in the queue, and `m_tail` holds the ID of the most recent task inserted in the queue. Every value in `m_next` is an ID for the next task to be executed and an index (i.e., pointer) to the successor entry in `m_next`. A distinguished task has the identifier `NO_TASK` $= 255$. `NO_TASK` is the value of `m_next[m_tail]` and it is the successor of all non requesting identifiers. For example, as depicted in Figure 7.1, a queue state s_1 consists of `m_head`=12, `m_next[12]`=3, `m_next[3]`=255, `m_tail`=3, and $\forall j : (j \neq 12) :$ `m_next[j]`=255.

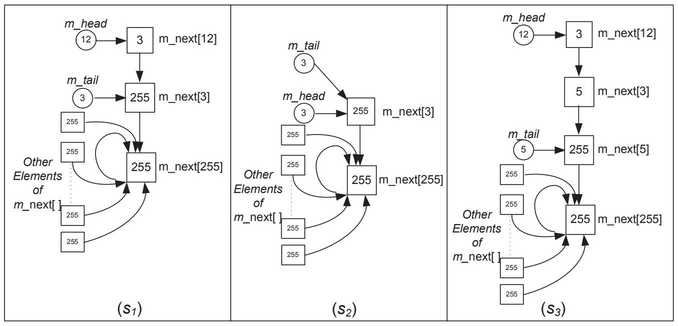


Figure 7.1: Example states of the task queue.

The state s_1 represents a task queue having only two pending tasks of identifiers 12 and 3 respectively. The effect of `popTask()` on s_1 is a transition to state s_2 (see Figures 7.2 and 7.1). In state s_2, `m_head`=3, `m_next[3]`=255, `m_tail`=3, and $\forall j : (0 \leq j \leq 255) :$ `m_next[j]`=255. The effect of `pushTask(5)` on s_1 is a transition to state s_3 (see Figures 7.2 and 7.1), where `m_head`=12, `m_next[12]`=3, `m_next[3]`=5, `m_next[5]`=255, `m_tail`=5, and $\forall j : (j \neq 12) \wedge (j \neq 3) \wedge (0 \leq j \leq 255)$: `m_next[j]`=255.

```
inline uint8_t popTask()
{
 if( m_head != NO_TASK ) {
   uint8_t id = m_head;
   m_head = m_next[m_head];
   if( m_head == NO_TASK )   m_tail = NO_TASK;
   m_next[id] = NO_TASK;
   return id;   }
  else      return NO_TASK;
  }

bool isWaiting( uint8_t id )
 { return (m_next[id] != NO_TASK) || (m_tail == id); }

bool pushTask( uint8_t id )  {
 if( !isWaiting(id) )  {
   if( m_head == NO_TASK ) { m_head = id;  m_tail = id; }
   else  {  m_next[m_tail] = id;  m_tail = id;   }
   return TRUE;
 } else   return FALSE;      }
```

Figure 7.2: Excerpt of the Tiny OS Scheduler.

Structural Invariant. A *valid* state of the task queue is a state where the queue has a linear structure with its head (m_head) pointing to its beginning and its tail (m_tail) pointing to the most recently added identifier to the task queue. Each element of m_next with a non-255 ID is reachable from the head. Each entry of m_next that is not in the queue holds the value of NO_TASK, and m_next[m_tail] is equal to NO_TASK. Moreover, the task IDs belong to the interval $0 \leq ID \leq 255$. Figure 7.3-(a) illustrates a sample valid state of the task queue. Furthermore, any operation performed on the queue should remove an element from the head (i.e., popTask()), add an element to the tail pushTask() or leave the structure of the queue and the task IDs unchanged.

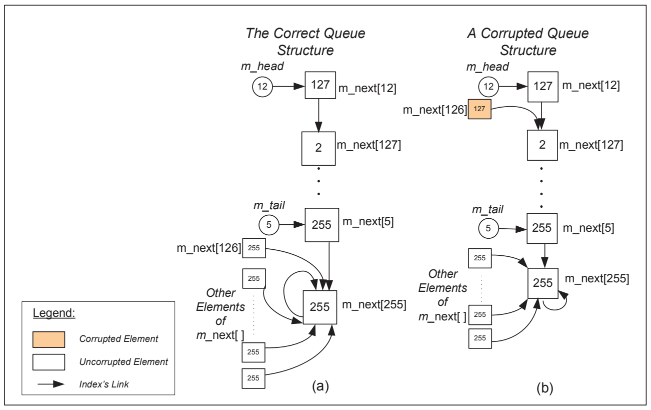


Figure 7.3: Valid and invalid task queue structures.

Transient faults. Transient faults may toggle multiple bits in a single variable; i.e., m_head, m_tail or a memory cell of m_next[]. The case of multi-variable corruption is the subject of our current investigation. Bit-flips may perturb a task ID and the structure of the task queue to an invalid state. For example, Figure 7.3 demonstrates how resetting the most significant bit of m_next[126] could change its content from 255 to 127, thereby pointing to m_next[127] instead of pointing to NO_TASK.

7.3 Addition of Recovery

Section 7.3.1 analyzes the memory and time requirements of correcting BFs with the Hamming code. Section 7.3.2 illustrates how our approach enables recovery from BFs by detecting invalid queue structures and correcting them.

7.3.1 Correcting Bit-Flips with ECC

One approach for recovery from transient faults that cause bit-flips is to use error detection and correction codes such as the Hamming Code (HC) [59]. However, due to high memory/CPU cost of the encoding/decoding algorithms these approaches seem impractical in the context of WSNs. For example, there are two ways to deal with bit-flips in the task queue using HC; consider either individual memory cells of the `m_next[]` array as separate data words, or the entire 256 bytes of the task queue as one data word.

In the first case, each cell of the `m_next[]` array should be encoded before storing a value and it should be decoded before reading its contents. To encode 8 bits of data with HC, we need 4 extra parity bits, which results in a code word with 12 bits in the following format: $p_1p_2d_1p_3d_2d_3d_4p_4d_5d_6d_7d_8$, where d_j denotes data bits for $1 \leq j \leq 8$, and p_i represents the parity bits for $1 \leq i \leq 4$. The encoding algorithm of HC determines the 12-bit code word by multiplying a 12×8 matrix by a vector made of the data bits. Such a matrix multiplication takes 96 multiplications and 84 additions; i.e., totally 180 *basic operations* in addition to one read and write operation on each memory cell, where a basic operation includes arithmetic and logical operations as well as comparisons and load/store. The decoding algorithm also multiplies a 4×12 matrix by a vector containing the 12-bit code word, which results in a 4-bit syndrome vector representing the position of the corrupted bit. (Thus, each decoding takes 48 multiplications and 44 additions, totally 92 basic operations.) Notice that for each byte allocated in `m_next[]` 4 extra bits should be considered for parity. That is, $256/2 = 128$ extra bytes should be allotted along with the 256 bytes allocated for `m_next[]`. Besides, every time a task ID is stored/retrieved to/from a memory cell in `m_next[]`, the encoding/decoding algorithm must be executed. That is, for one round of detection and correction, $256 \times (180 + 92) = 69632$ basic operations should be performed.

In the second case, the queue comprises a bit pattern with $256 \times 8 = 2024$ bits, for which $1 + log\ 2024 = 12$ parity bits are needed in HC. Thus, the size of the code word is equal to $2024 + 12 = 2036$ bits. The encoding takes $2024 \times 2036 = 4120864$ multiplications and $2023 \times 2036 = 4118828$ additions; i.e., totally 8239692 basic operations. For decoding, we will need 12×2036 multiplications and 12×2035 additions resulting in 48852 basic operations. This analysis clearly illustrates the impracticality of using HC on sensor nodes with a small memory and a limited processing power. While other ECC methods (e.g., Forward Error Correction (FEC) [67] and Reed-Solomon (RS) [68]) can correct cases where multiple variables are corrupted, they are even more expensive than HC in terms of either space or time. For example, the RS method needs t bits for the correction of $\lfloor t/2 \rfloor$ bits and $\mathcal{O}(t^2)$ is its time complexity.

7.3.2 Adding Recovery to Task Queue

This section illustrates how we enable recovery to a valid queue structure from transient BFs. Figure 7.4 depicts a state machine that demonstrates the impact of transient faults and how recovery should be achieved. Since the task queue is a centralized program running on a single CPU, we can benefit from a high atomicity model in which a set of instructions can be performed atomically. In fact, the nesC language provides atomic blocks that capture a sequence of statements that are supposed to be executed without interruption. The essence of the addition of recovery in high atomicity [10] is based on detecting the violation of the invariant due to the occurrence of faults, and providing recovery from every invalid state to the invariant. Thus, we present the function `DetectCorrect()` (see Figure 7.5) that we add to the Tiny OS scheduler to enable the detection and correction of BFs before and after any push/pop operations on the task queue. The function `DetectCorrect()` should be invoked in an atomic block (i.e., `atomic{DetectCorrect()}`) to ensure that detection and correction are not interrupted during execution. Depending on the harshness of the environment where the motes are deployed, the period of invoking `DetectCorrect()` could be changed by the developers; i.e., `DetectCorrect()` can be invoked in an adaptive fashion by the scheduler of TinyOS in order to enable a tradeoff between the degree of dependability and the energy cost of providing recovery. Next, we explain different parts of `DetectCorrect()` to illustrate how detection and correction are achieved.

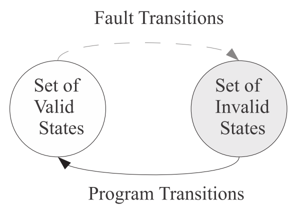


Figure 7.4: Adding recovery to the task queue.

Data structures. To detect and correct corruptions of the task queue, `DetectCorrect()` gathers some information about the structure of the queue and stores them in these data structures (see Figure 7.5):

```
DetectCorrect(int q_size) {
 uint8_t previous;
 uint8_t current=m_head;
 uint8_t Index=0;
 uint8_t dangleElem=0;
 uint8_t non255 =0;      // Number of non-255 elements
 uint8_t qLength =0;     // Number of elements in the queue
 uint8_t cyclePoint =0;  // The corrupted element that
                         // points back and creates a cycle

 component visited = new BitVectorC(256);
 visited.clearAll();
 component pointedTo = new BitVectorC(256);
 pointedTo.clearAll();
 component pointsToNoTask = new BitVectorC(256);
 pointsToNoTask.clearAll();

 bool cyclic = FALSE;
 bool pointedByHead = FALSE;
```

Figure 7.5: Data structures.

(1) `previous`, `current`, `dangleElem` and `cyclePoint` are pointers that are used during the traversal of the queue; (2) `non255` keeps the number of memory cells in `m_next` that contain non-255 values; (3) `qLength` stores the number of non-255 elements reachable from the head of the queue (`m_head`); (4) the `visited` bit vector allocates one bit corresponding to each element of `m_next` illustrating whether or not it has been visited previously in a queue traversal for cycle detection; (5) the `pointedTo` bit vector keeps a bit for each element of `m_next` demonstrating whether or not that element is being pointed to by some other element; (6) the `pointsToNoTask` bit vector allocates a bit corresponding to each memory cell of `m_next` that contains `NO_TASK`; (7) the `cyclic` flag is set if a cycle is detected in the structure of the task queue, and (8) the `pointedByHead` flag is true if and only if there is an element in the queue that is pointed by both `m_head` and another element in the queue. We use the `pointedByHead` flag in detecting/correcting the corruption of `m_head`. Notice that, we allocate 96 bytes for the bit vectors and 7 bytes for other variables (i.e., 103 bytes totally) capturing local variables; i.e., when `DetectCorrect()` returns this memory is released.

Initialization. In this step, we first count the total number of elements in `m_next` that contain non-255 values. Then, we initialize the `pointedTo` and `pointsToNoTask` bit vectors. This step incurs $256 \times 15 = 3840$ basic operations on our solution.

```
// Count the number of non-255 elements in array m_next
 for(Index=0; Index<NO_TASK; ++Index)
    if (m_next[Index] != NO_TASK) non255++;
// Determine the elements that are being pointed to
 for(Index=0; Index<NO_TASK; ++Index)
    pointedTo.set(m_next[Index]);
// Determine the elements that point to NO_TASK
 for(Index=0; Index<NO_TASK; ++Index)
    if(m_next[Index] == NO_TASK) pointsToNoTask.set(Index);
```

Detection and correction of `m_head`. Since the traversal of the queue for subsequent processing is performed using the `m_head` pointer, we first ensure that `m_head` is corrected. If `m_head` points to `NO_TASK` (see Figure 7.6), then we set `m_head` to the index of the element to which no other element points, and exit (because, by assumption, our focus is on single-variable corruption). Otherwise, we detect whether `m_head` points to another non-255 element in the queue. (Please see Figure 7.7 and the first for-loop in the else part of Figure 7.6.) If so, then we set `m_head` to the index of the non-255 element to which no other element points. (See the second for-loop in the else part of Figure 7.6.) Figure 7.7 illustrates a case where the value of `m_head` has been corrupted from 12 to 4.

```
if (m_head == NO_TASK)  {
  for(Index=0; Index<NO_TASK; ++Index)
    if(!pointedTo.get(Index) && m_next[Index] != NO_TASK) {
            m_head = Index;  return; }      }
else {
  for(Index=0; Index<NO_TASK; ++Index)
    if(pointedTo.get(Index) && Index == m_head) {
                    pointedByHead = TRUE; break; }
  if (pointedByHead)
   for(Index=0; Index<NO_TASK; ++Index)
    if(!pointedTo.get(Index) && m_next[Index] != NO_TASK) {
                  m_head = Index;  return; }
       }
```

Figure 7.6: Detect and correct `m_head`.

The time complexity (and energy consumption) of this step is proportional to the maximum number of basic operations. If `m_head` $=$ `NO_TASK`, the `for-loop` in the `if` part of Figure 7.6 will be executed, which has one comparison and one increment for the loop counter in each iteration. Moreover, the `if-statement` inside the `for-loop` performs two load operations, one comparison and two logical operations per iteration. Thus, in the worst case, we have 7 basic operations in each iteration of this `for-loop`, which results in $256 \times 7 = 1792$ basic operations if `m_head` $=$ `NO_TASK`. A similar reasoning illustrates that, in the worst case, we perform $256 \times 15 = 3840$ basic operations if `m_head` $\neq$ `NO_TASK`. Therefore, since either the `if` part or the `else` part is executed in Figure 7.6, the correction of `m_head` takes at most 3840 basic operations.

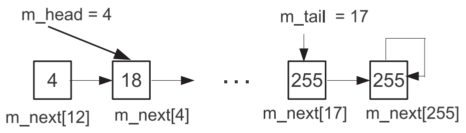


Figure 7.7: Corruption of `m_head`.

Detection and correction of cyclic structures. The `do-while` loop in the below code uses the `visited` bit pattern to determine whether there is a cycle in the queue. This loop also stores the number of elements in the queue that are reachable from `m_head` in the `qLength` variable.

A cycle could be formed in two ways: either the tail points back to some element including

```
// Detect cycles
do {
   if(!visited.get(current))       visited.set(current);
   else { cyclic = TRUE;
          cyclePoint = previous;     break;  }
   previous=current;
   current=m_next[previous];
   qLength++;
     } while(current != NO_TASK && m_tail!=previous);

// Correct cycles
if (cyclic && (cyclePoint == m_tail)) {
           m_next[cyclePoint] =  NO_TASK; return; }
if (cyclic && cyclePoint != m_tail)
  for(Index=0; Index<NO_TASK; ++Index)
    if(!pointedTo.get(Index) &&
          (m_next[Index] != NO_TASK) && (Index != m_head)) {
          m_next[cyclePoint] = Index;  return; }
```

itself (see Figure 7.8-(a)), or another element points back to some element including itself (see Figure 7.8-(b)). If a cycle is detected, then the `cyclic` flag is set and the index of the element pointing back is stored in `cyclePoint`. In case `cyclePoint` is equal to `m_tail`, then that means the tail of the queue is pointing back to some element instead of pointing to `NO_TASK`. Otherwise, to fix the cycle, we set the contents of `m_next[cyclePoint]` to the index of the element that has become *dangled* due to the cycle creation; i.e., the element to which no element points, does not point to `NO_TASK`, and is not equal to `m_head`. This correction will take at most $256 \times 26 = 6656$ basic operations.

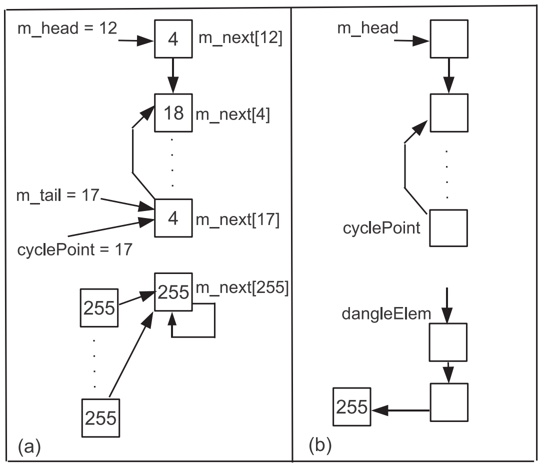


Figure 7.8: Cyclic corruption of the task queue.

Detection and correction of queue size and non-255 elements. To detect discrepancies in the size of the task queue, we add a new variable `q_size` to the TinyOS scheduler to store the size of the queue outside the `DetectCorrect` function. Nonetheless, `q_size` could be perturbed by transient faults. The first `if statement` in Figure 7.9 corrects `q_size`. Notice that `DetectCorrect` can simultaneously correct `q_size` and `m_head`, which is a special case of correcting MBFs in multiple variables. Moreover, faults may change the value of an array element from 255 to some other value. This means that that element points to some queue element. Such a link is not part of the task queue and should be eliminated. To this end, we assign 255 to an element to which no other element points, points to a non-255 element and is not equal to `m_head`. The `for-loop` in the second `if statement` could take at most $256 \times 11 = 2816$ basic operations.

```
if (qLength == non255) {
    if (q_size != qLength)  { q_size = qLength; return; }
        else return;  // Task queue is NOT corrupted.
         }

if (non255 > q_size) // some 255 element has become non255
   for(Index=0; Index<NO_TASK; ++Index)
     if (!pointedTo.get(Index) &&
        (m_next[Index] != NO_TASK) && (Index != m_head)) {
                  m_next[Index] = NO_TASK;  return;  }
```

Figure 7.9: Detect and correct queue size.

Detection and correction of `m_tail`. To detect and correct the corruptions of `m_tail`, we set `m_tail` to the index of the first element whose contents point to `NO_TASK` (see below). For example, in Figure 7.10, `m_tail` is set to 17. The `for-loop` in the below code performs at most 2560 basic operations.

```
for(Index=0; Index<NO_TASK; ++Index) {
    if ((!pointsToNoTask.get(Index)) &&
        (m_next[m_next[Index]] == NO_TASK) &&
                (m_tail != m_next[Index])) {
              m_tail = m_next[Index];  return;  } }
```

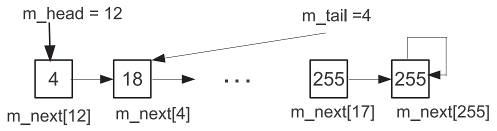


Figure 7.10: Corruption of `m_tail`.

Detection and correction of corrupted acyclic structures. If faults corrupt a non-255 element so it points to one of its successors, then a structure similar to Figure 7.11 could be created. In this example, the contents of `m_next[4]` is changed from 18 to 25 and `m_next[18]` becomes unreachable from head; i.e., a *dangling* element. One way to detect this case is to simply compare `qLength` with the number of non-255 elements; if `qLength` $\neq$ `non255`, then either this case has occurred or the corruption of `m_head`. Nonetheless, if the code of the `DetectCorrect()` routine reaches this point, then it means that `m_head` has the correct value.

The identification of the `corrupted element` in Figure 7.11 is not straightforward. Our strategy is to determine the index of the element in the queue that is pointed by two internal elements of the queue (see `m_next[25]` in Figure 7.11). Such an element must be in the fragment of the queue that starts with the dangling element. Thus, we first find the index of the dangling element by the first `for-loop` in Figure 7.12. If there is such a dangling element, then we reset the `pointedTo` bit vector. Then, in the first `do-while`

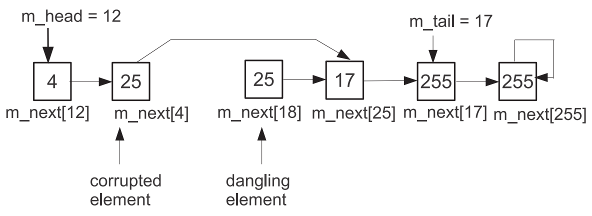


Figure 7.11: Corrupted acyclic structure.

in Figure 7.12, we start setting the bits of `pointedTo` corresponding to the fragment of the queue that starts with the dangling element. In the second `do-while`, we search the first fragment of the queue (starting from `m_head`) for the element that points to an element whose corresponding bit is already set in the `pointedTo` vector. Once we find such an element, we set its content to the index of the dangling element, and the queue is corrected. This step includes $27 \times 256 = 6912$ basic operations in the worst case.

```
dangleElem = NO_TASK;
for(Index=0; Index<NO_TASK; ++Index)
   if(!pointedTo.get(Index) &&
       m_next[Index] != NO_TASK &&
       Index != m_head) {
                   dangleElem = Index; break; }

if (dangleElem == NO_TASK)  return;

pointedTo.clearAll();
current = dangleElem;

do {
     pointedTo.set(m_next[current]);
     previous=current;
     current=m_next[previous];
     } while(current != NO_TASK && m_tail!=previous);

current = m_head;

do {
    if (pointedTo.get(m_next[current]) {
             m_next[current] = dangleElem; return; }
    previous=current;
    current=m_next[previous];
     } while(current != NO_TASK && m_tail!=previous);
   }
```

Figure 7.12: Detect and correct acyclic structures.

Time complexity of `DetectCorrect()`. Since the code of `DetectCorrect()` does not include nested for-loops, its time complexity is linear in the size of the task queue. Figure 7.13 presents a comparison of the time/space cost of the proposed method of this chapter with two scenarios of using the Hamming code for correction of BFs: HC1 represents the case where each element of `m_next` is encoded with HC, and HC2 denotes the case where the entire `m_next` is encoded as a single word. Notice that, our approach outperforms HC1 in terms of both time and space efficiency, respectively by a factor of 20% and 60%. More importantly, the required memory (i.e., 103 bytes) is temporary; i.e., when `DetectCorrect()` returns this memory is released. The HC2 method seems impractical due to expensive computing requirements.

Approach	Memory Cost	# of Operations
Hamming Code for each element of `m_next` (HC1)	128 Bytes	$\simeq$ 70000
Hamming Code for the entire `m_next` (HC2)	12 bits	$\simeq$ 4.17 million
Proposed Method	103 Bytes	28000

Figure 7.13: Space/Time cost of correction of BFs.

Scope of correction. The scope of correction in these three methods is different. Figure 7.14 demonstrates that our approach can correct multiple bit-flips in a single variable, which cannot be achieved by HC1 and HC2. However, HC1 can correct single bit-flips in multiple variables, which we do not currently have a solution for it.

Approach	Corrects SBFs	Corrects MBFs in a Variable	Corrects SBFs in Multi Vars	Corrects MBFs in Multi Vars
HC1	Yes	No	Yes	No
HC2	Yes	No	No	No
Proposed Method	Yes	Yes	No	No

Figure 7.14: Scope of correction for Single Bit-Flips (SBFs) and Multiple Bit-Flips (MBFs).

Fault tolerance of `DetectCorrect()`. In case transient faults perturb the local variables and/or the control flow of `DetectCorrect()`, the current round of execution of `DetectCorrect()` may not recover the structure of the task queue. However, since `DetectCorrect()` is executed repeatedly and transient faults eventually stop occurring, `DetectCorrect()` will eventually provide recovery.

7.4 Summary and Extensions

We presented a novel method for the detection and correction of transient Bit-Flip (BF) in the task queue of the TinyOS, which is the operating system of choice for sensor nodes. Since motes have limited computational and energy resources, instead of using resource redundancy, the proposed approach exploits computational redundancy to efficiently recover from transient BFs that corrupt the contents and the structure of the task queue. The essence of our approach is based on the detection of invalid structures of the queue that might be created due to transient faults. Upon reaching an invalid structure, we analyze the structure of the task queue to determine which failure scenario has occurred and recover to a valid state. Using this method, we can correct Multiple BFs (MBFs) in single-byte variables. We illustrate that the proposed approach can provide a better time/space efficiency with respect to Error Correction Codes such as the Hamming code [59] (see Figures 7.13 and 7.14).

Several techniques exist for designing fault-tolerant data structures. Aumann *et al.* [69] present alternative implementations for pointer-based data structures by adding redundant links. Finocchi *et al.* [70] provide resilient search trees through periodic checkpoints. Jørgensen *et al.* [71] devise a method that ensures the resilience of priority queues by storing pointers in resilient memory locations. By contrast, our approach continuously monitors a structural invariance and provides recovery if the invariant is violated.

Future/ongoing work focuses on techniques for the correction of MBFs in multiple variables. Moreover, we would like to leverage our work in Chapter 3 on automated addition of convergence for the addition of recovery to data structures. Specifically, we modeled a state as a unique valuation of variables of primitive types (e.g., Boolean and integer). Nonetheless, we need to create richer models that capture the state of complex data structures. We will also work on models where ECC methods and our approach are used in a hybrid fashion.

Chapter 8

Tools for Automated Synthesis of Convergence

In this chapter, we demonstrate the architecture and design of two software tools that we have developed to synthesize convergence. In Section 8.1, we illustrate the constituents of STSyn, a software tool that synthesizes convergence on single processor based on the algorithms in Chapter 3. In Section, 8.2, we explain pSTSyn; the tool that exploits the power of computer clusters as in Chapter 4 to synthesize convergence in symmetric protocols.

8.1 STabilization Synthesizer (STSyn)

STSyn is a software tool that modifies a non-stabilizing protocol p, represented in guarded command-like language, and generates a protocol p_{ss} that has exactly the same state space. Moreover, p_{ss} behaves like p starting from any legitimate state; i.e., a state in I. Starting from any global state outside I, p_{ss} strongly converges to I. p_{ss} is represented in the same language as p. A web interface for STSyn is available at http://c28-0206-01.ad.mtu.edu:8888/SynStable/.

STSyn implementation consists of the following modules:

- **Input Protocol Parser.** The *input protocol parser* transforms the input language of the protocol into our native data structures representation. We imported the input language parser of an automated synthesis tool, *sycraft*, by Bonakdarpour *et al.* [72]. For details about the syntax and semantics of the input protocol language, please refer to sycraft's user manual http://www.cse.msu.edu/~borzoo/sycraft/sycraft-user-manual.pdf.

In this version of STSyn, we use Multivalued Decision Diagrams (MDD's) [73] to represent protocols as logic expressions in the working memory of STSyn. An MDD is a representation of a logic expression over variables that generally assume multiple values. This representation is in the form of a directed acyclic graph whose nodes represent variables and arcs represent variable valuations. The sink vertices of the acyclic graph are evaluations to either *true* or *false*. A valuation of the logic expression is therefore a path in the graph following a specific order for variable valuation.

- **Protocol Encoder.** The *protocol encoder* provides the essential functionalities to manipulate the protocol representation including adding/removing groups of transitions to a process, forming groups for sets of transitions of a specific process, enforcing write restrictions on a set of protocol transitions and manipulating sets of states.

- **Convergence Synthesizer.** The *convergence synthesizer* implements the algorithms in Chapter 3 to add convergence to an input protocol p. If the synthesizer succeeds in adding convergence to p, it generates a self-stabilizing protocol p_{ss}, otherwise, the synthesizer declares failure.

- **Cycle Detection and Resolution.** An essential module to implement our synthesis algorithms is the cycle detection and resolution module. We implemented a set of cycle detection algorithms [23], [32], [74] that run on implicit/symbolic representations of directed graphs. A representation of a graph is symbolic if it is in the form of a logic expression over possible valuations of its vertices and arcs. An MDD is one example of an implicit representation of the transition relation of a protocol.

- **State Space Partitioning and Ranking.** Prior to synthesis, STSyn exhibits a partitioning of the global state space into ranks based on the shortest distance from a global state to the set of legitimate states. Partitioning and ranking is a pre-synthesis step which generates an approximate weakly stabilizing version of the input protocol. During synthesis, the set of partitioned global states guides our heuristics to increase their chance of finding a strongly stabilizing solution.

- **Output Generator.** STSyn generates two output files. The first file is a *.log*, a log file that includes the details of ranking, cycle resolution and synthesis operation. The second file is a *.fout*; an output file that includes the guarded commands of the output protocol p_{ss}, if any. We reuse a multi-valued logic minimizer (MVSIS) from the University of California, Berkley [75] to minimize our output representation[1].

[1]Information about MVSIS can be found in this link http://embedded.eecs.berkeley.edu/mvsis/.

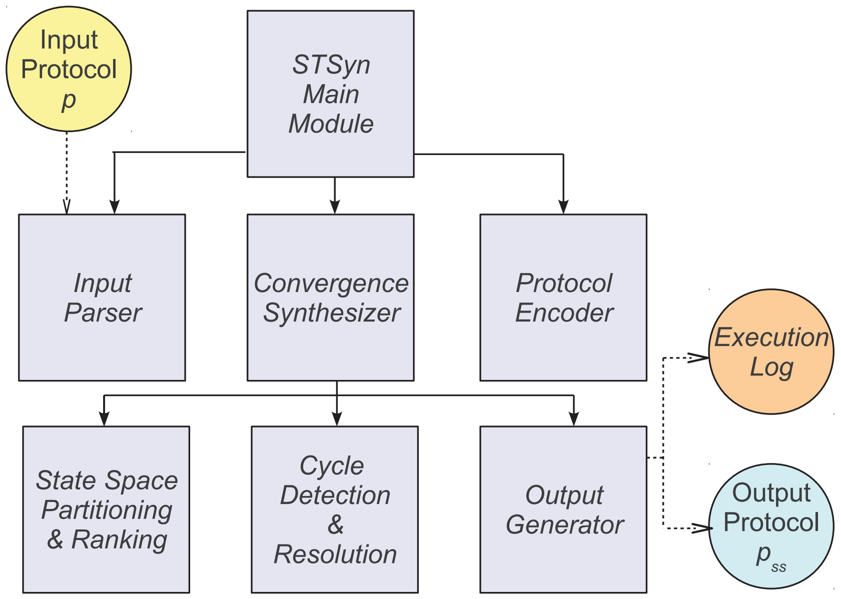


Figure 8.1: Modules Included in the Implementation of STSyn

Figure 8.1 summarizes the architecture of STSyn.

8.2 Parallel STabilization Synthesizer (pSTSyn)

Similar to STSyn, pSTSyn is a software tool that generates a stabilizing protocol p_{ss} provided with an input non-stabilizing protocol p. pSTSyn exploits the computational redundancy available in computer clusters. The tool runs parallel independent versions of the convergence synthesizer on an input protocol and leverages randomization in order to explore disjoint regions of the solution space.

pSTSyn consists of the following modules per each parallel thread:

- **Protocol Encoder.** In addition to the functionalities of the protocol encoder in STSyn, pSTSyn manipulates the global transition relation of p by adding/removing single transitions groups. Such fine grained manipulation allows the implementation of backtracking algorithms for a group by group addition to the working set of transition groups. Read and write restrictions are imposed on the transitions of a process by applying *existential quantification* over unreadable variables and imposing equality constraints between the *pre* and *post-transition* values of unwritable variables. Pre-transition and post-transition values of unwritable variables are the values of the unwritable variables in the source and target states of a global transition, respectively.

- **Convergence Synthesizer.** The *convergence synthesizer* implements the algorithms in Chapter 4 to add convergence to an input protocol p. The main difference is that transition groups are included one-by-one to the working set of transition groups. In addition, we implemented a backtracking version of the convergence synthesizer that is complete. However, backtracking demonstrates impracticality due to its exponentially growing time complexity in the size of the global state space.
- **Cycle Detection and Resolution.** This module is almost similar to its counterpart in STSyn. The only difference is that it makes use of the fact that groups are added one-by-one, so the transition relation depth-first search algorithm does not need to rerun every time a transition group is added. In other words, cycle resolution memorizes its previous state, thus optimizes its search for cycles by adding new transition groups.
- **State Space Partitioning and Ranking.** This module is exactly the same as its counterpart in STSyn
- **Groups Shuffler.** A *group shuffler* randomly shuffles the transition groups in every partition such that each of them constitutes a seed to a different thread of pSTSyn. We chose our shuffler in such a way that the synthesizer explores disjoint areas of the solution space. As such, we increase the number of discovered stabilizing solutions, if any.

Chapter 9

Related Work: A Taxonomy

In this chapter, we classify the existing work related to design and verification of self-stabilization in particular and of properties expressible in temporal logic in general.

We adopt the following two criteria for our literature classification: the discipline and the problem type. Three disciplines are of interest: *control-theoretic* where stabilization is studied in the context of Discrete-Event Systems (DES) [4], *game-theoretic* (*open systems*) where the behavior of the system under study is affected by the environment as an opponent [76], and *distributed algorithms* where the dynamic system/program and the environment's behavior are modeled as a whole closed entity; i.e., *closed system* [77]. We consider two major problems: *analysis (verification)* vs. *design (synthesis)*. Researchers tackle these problems using different approaches. For *analysis*, there is a model-theoretic vs. a proof-theoretic approaches. Whereas in *design*, we encounter *specification-based synthesis* vs. *program revision*. Such taxonomy partitions the related work into twelve different categories. Figure 9.1 illustrates these categories with key works/authors in each category. Note that we further refine the revision-based design of distributed algorithms to *manual* vs. *automated* methods. This is the category where our contributions reside. In addition, in distributed algorithms verification, we only list work related to *automated verification* of self-stabilization due to the extensive scope of automated verification, especially *model checking* [78]. We quote Baier and Katoen's definition [79]:

> Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state) in that model.

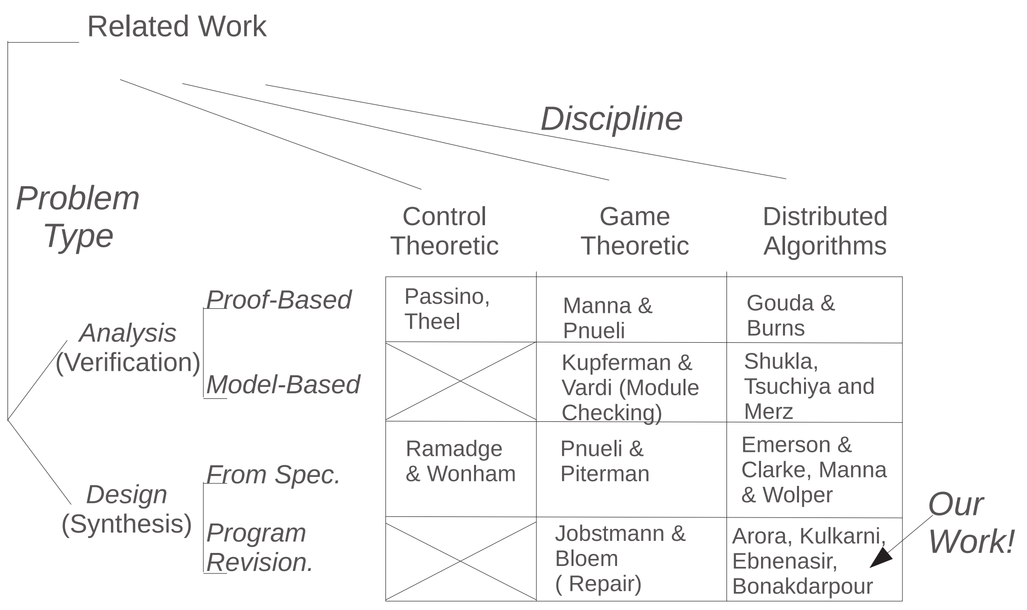


Figure 9.1: Taxonomy and key authors

9.1 Control-Theoretic Approach

The concept of stabilization in control systems is well-established and mature. However, controller synthesis in Discrete-Event Systems (DES) is relatively a recent field [4]. Control-theoretic methods take a model of an uncontrolled plant and the specification of a desired controlled plant, often expressed in terms of a formal language. Then they synthesize a DES controller whose composition with the uncontrolled plant results in a system that meets the formal specification of the controlled plant. In particular, the DES controller restricts events applicable to the plant based on the current state of the plant. The problem is set up as a feedback control loop as encountered in the context of control theory.

9.1.1 Design

In the eighties, Ramadge and Wonham introduced the synthesis of supervisory control in DES [80]. They established a formal language framework for the study of DES controller synthesis. A set of event sequences; a.k.a., strings, defines all the *admissible* behaviors of the system in terms of a formal language $\mathfrak{L}$ over a finite alphabet of possible events. In addition, a subset of this language $\mathfrak{L}_m$ consists of the *marked* behaviors of the system; i.e., the behaviors reaching a required set of states called *marked* states. For example, in the context of a car assembly line, a painted car defines a desirable state and any behavior of the assembly line that reaches this state is a marked behavior/string. Supervisory control defines a function that restricts allowable events based on the current system's state. The generated language is $\mathfrak{L}_s \subset \mathfrak{L}_m$. Lin and Wonham introduce decentralized supervisory control [81]. Decentralized control has the effect of grouping event sequences into equivalence classes due to the partial observation of the current system state by a system's submodule.

In contrast to the formal language approach, Ozveren and Willsky adopt a state-based approach to controller synthesis [82]. They define the controller synthesis problem for regulatory purposes, i.e., the controller guarantees that the language of the plant has finite prefixes in the set of *illegitimate* states. Their notion of closure is weaker than the one we introduce in Chapter 2. They define a stabilizing behavior as a string of events that visits *infinitely often* a legitimate state. They provide a polynomial time algorithm in the size of the state space for synthesizing a stabilizing controller. However, their approach does not handle decentralization and partial information.

9.1.2 Analysis

Passino *et al.* pioneered the study of DES stability analysis [83]–[85]. The authors reused several notions of stability in classical control theory like *Lyapunov Stability* and *Lagrange Stability* [86]. A Lyapunov function is a scalar real-valued function of the state of a control system. The system analyzer chooses a strictly decreasing function along the trajectories of the dynamic system in such a way that the function value vanishes at the (presumably) stable points of the dynamic system. The choice of a suitable Lyapunov function is not systematic and depends on the analyzer's ingenuity. Lagrange stability focuses on *boundedness* of motion. It defines an invariant set of states within which system trajectories are always confined. For example,

the number of customers in a cashier line should not exceed five

is a boundedness condition. In either Lyapunov or Lagrange stability, a *metric* is defined on the set of system states. A metric is a scalar positive-valued measure on pairs of states: for example, the number of '1's in the bitwise XOR of two bit vectors is a metric on a bit-vector space. This is how neighborhoods are defined on the state space and accordingly, monotonic decrease of Lyapunov functions and boundedness of invariant sets are defined. Notice how Lyapunov stability captures the concept of convergence while Lagrange stability captures closure. However, our definitions in Chapter 2 of closure and convergence do not require a metric space.

Oehlerking and Theel [87] use *Lyapunov* functions to verify convergence in distributed systems. To this end, a transformation from the distributed system model to a DES representation is necessary. They then take the matrix form of a discrete linear-time system and apply Lyapunov method to prove the stability of the distributed protocol. Their work differs from Passino *et al.*'s in that, in the latter, they directly verify a nonlinear DES model using a Lyapunov function; i.e., there is no need for a transformation from a distributed system model to a difference-equation representation.

Kumar and Garg characterize properties of stabilization in terms of *formal languages* [88]. They demonstrate different types of stability as *language* properties. They argue that the traditional notion of stabilization; i.e., the definition based on partitioning the state space into legal and illegal states, is not general enough for the purposes of the author. They introduce a language-based definition of stabilization where legal behaviors may include a bounded number of iterations in a non-progress cycle if they eventually reach a legal state. They prove that such type of stabilization cannot be implemented using *static* feedback controller; i.e., a controller that takes its decision based only on state information. They suggest a *dynamic* feedback controller that takes its decision based on the current computation prefix (possibly including cycles) rather than the current state. In addition, they provide algorithms for the verification of language stability. They introduce the notion of ω- language stability which is stability for infinite-length behaviors.

Young and Garg [89] define stabilization in such a way that the whole transition system consists of only *legal* behaviors: from any state of the systems, all computations are legal. Their main contribution includes methods for determining how system specification can be strengthened/weakened to become satisfied by an existing stabilizing system according to their own definition of stabilization.

Our proposed approach considers a state-based method unlike Ramadge *et al.*'s original setting. Supervisory control addresses the synthesis of *safety* properties[1] and a subset of liveness[2] called (*nonblocking*) properties. In particular, we consider the set of *marked* states of a DES. A nonblocking supervisory control enforces that every string in $\mathfrak{L}_s$ can be extended to a string with a *marked* state in $\mathfrak{L}_s$. However, to the best of our knowledge, we are not aware of contributions in synthesizing nonblocking-decentralized controllers in a state-based approach. Moreover, controller synthesis covers only regular languages whereas stabilization in distributed systems is considered in the context of languages with

[1]Safety properties require that nothing "bad" should happen and are verifiable by examining finite prefixes of computations.

[2]Liveness properties mandate that something "good" should happen and are generally verifiable by examining infinite length computations. Note that any program property can be specified as the intersection of *safety* and *liveness* properties [90].

infinite-length strings; i.e., protocol executions that possibly do not terminate (exception is work by Kumar and Garg [88]. In this vein, it is worth mentioning that the approach by Young and Garg [89] helped to generate a specification for a stabilizing system defined in terms of *regular languages*; i.e., a specification that is reusable as an input to the controller synthesis problem.

9.2 Game-Theoretic Approach

Game-theoretic approaches model protocols/programs/dynamic systems as transition systems with two players in action: the concurrent program, and the environment. We can think of the environment as an opponent trying to perturb the program's behavior. As players, the program and the environment take turns when acting upon the program's state. Such programs correspond to *open systems* as they interact with their environment.

The interface between the environment and the program is through input/output (output/input) of the program (environment), respectively. In this setting, the environment generates possibly infinite length strings as input to the concurrent program and the program instantly responds interactively with infinite length output strings to the environment. The environment behavior is uncontrollable, and most generally, malicious. The requirements on the program's behavior are expressed as constraints on its input/output. We call these constraints the system's specification. A realization of the program is a description of its semantics such that the program's behavior satisfies the input/output specification.

9.2.1 Synthesis

The problem of synthesis from specification takes as input a representation of constraints on the input/output infinite length strings of the program and generates as output a realization (an automaton, a circuit) of these constraints. Such constraints constitute a specification to which the concurrent program should adhere.

Historically, automated synthesis of a program from its specification has been suggested by Church [91], [92] and has been thereafter addressed as Church's problem in the framework of mathematical logic. Under the same framework, two attempts have provided a solution to Church's problem either as an emptiness problem to a tree automaton [93], or as a solution to a two-party game [94].

9.2.1.1 Synthesis from Specification

In the context of open systems synthesis, Pnueli and Rosner [95], [96] introduced a method for synthesizing a module interacting with its environment from a temporal logic specification on its input and output. Their approach is an extension of the idea of finding a proof of validity of a temporal logic formula. The validity proof constructs a model for the temporal logic specification. In the case of a propositional temporal logic, validity is decidable. They demonstrate that, for an open system, the synthesized protocol/circuit satisfies the temporal logic formula, for all possible input valuations, unlike the closed system case where it is only sufficient to find one satisfying valuation for the input. This result stems from the fact that the designer has no control over the environment, and hence the synthesized model satisfies a universally quantified BTL formula. The main result of this paper is that for a specification $\Psi(x, y)$ (x is the input to the concurrent program and y is the output from the concurrent program) to be *implementable*, it is necessary and sufficient for a BTL formula to be satisfiable. $\Psi(x, y)$ is said to be implementable[3] iff for $x, y \in D$, there exists $f_P : D^+ \rightarrow D$; i.e., a function mapping non-zero length strings over D to elements of D such that for $x_i = a_1a_2a_3...a_i$, every possible behavior $\langle a_1, f_P(x_1)\rangle, \langle a_2, f_P(x_2)\rangle, \langle a_3, f_P(x_3)\rangle, \cdots, \langle a_i, f_P(x_i)\rangle, \cdots$ satisfies $\Psi(x, y)$. This reduces the synthesis problem to a satisfiability proof when x and y belong to finite domains. In the finite state case, the authors extend the work of Rabin *et al.* [93]. They build a labeled tree from the formula $\Psi(x, y)$ with values of pairs $\langle x, y\rangle$ at each node of the tree. Then, they transform it to a finite state tree automaton using techniques from [93]. However, the presented approach has an improved time complexity. The synthesis algorithm is polynomial in the number of states of the automaton and takes a double-exponential time as a function of the temporal logic formula length.

More recently, Piterman and Pnueli [97] consider a subclass of useful temporal properties for which Church's problem is solvable in at most a cubic-time in the size of the state space. Such an approach, together with the considerable increase in the computational power of current computers, revived research for instances of practical size. In particular, hardware synthesis from specification attracts Jobstmann, Bloem *et al.* to develop a tool [98] implementing the method in [97] and synthesize - for the first time - real circuits from specifications [99].

A hardness result is established by Pnueli and Rosner for the implementation of a synchronous reactive system on a given distributed architecture. They proved that the

[3]Realizability is the ability of an open system to satisfy its specification for all possible inputs from the environment. In other words, it is having a winning strategy for the open system in its two-party game with the environment: the environment plays a value of x and the game plays a value of y. However, realizability does not restrict a specific model for an open system as the definition of implementability herein. Thus, implementability in that sense is a special case of realizability.

existence of an open system implementation of a propositional temporal logic specification on a given distributed architecture is semi-decidable[4] [100]. Kupferman and Vardi [101] identify a decidable subproblem by considering a subset of architectures where information is restricted to flow between processes in a specific order. These results provide insight as to how distribution complicates the synthesis problem.

9.2.1.2 Program Revision

Program revision is the modification of an existing program to render it satisfying a modified specification. Jobstmann *et al.* consider program repair/revision as a strategy design in a game [102]. They design a heuristic to find a memoryless strategy for the product automaton of the original program with an LTL specification automaton[5]. A memoryless strategy means that repair modifies existing program text without adding new variables. The authors assumed that the required repair is localized in the program text and accordingly, represent this portion of the code as an unknown action. A winning strategy of the product automaton is an assignment of a specific action to the unknown part of code. As such, for every possible input sequence from the environment, the program satisfies its LTL specification. The authors demonstrate that the decision version of this problem is NP-Complete and they present a heuristic for its solution. This heuristic, for the subclass of LTL safety properties expressed as invariants, becomes a complete algorithm. An invariant is a property satisfied by every reachable program state during execution. Jobstmann *et al.* use symbolic methods to extract the repair from the winning strategy. Griesmayer *et al.* [104] apply Jobstmann's work by considering *repair-as-a-game* of Boolean C programs using memoryless stack-less strategies. A Boolean C program has only Boolean variables. Stack-less strategy means that any repair should not make use of the stack. They provide a model for Boolean C programs and apply an algorithm similar to the one in [102] to identify a repair for a suspected statement in the program text. Their contribution is the generation of such repairs for real code.

We are not aware of studies of self-stabilization within the open systems framework. In fact, convergence assumes that perturbations to the system's state stop occurring during recovery; i.e., convergence does not require recovery *in the presence of* perturbations of the environment. That is why it is adequate to study self-stabilization in a closed-system context.

[4]A decision problem (yes/no problem) is semi-decidable if there is no algorithm that answers "No" on all its negative instances. Moreover, a semi-decidable problem has only algorithms that answer "yes" on its positive instances and possibly do not halt on some of its "No" instances.

[5]A specification automaton is a finite state machine whose set of computations exactly matches the computations satisfying a given LTL property. For further reading, consult Vardi *et al.* [103].

9.2.2 Verification

In verification of reactive modules/open systems, all the possible environment behaviors are considered by the verification algorithm. Pnueli's seminal work on program verification presented a proof-theoretic approach to program correctness [105][6]. Kupferman and Vardi introduce *module checking* [108] for the algorithmic verification of finite-state reactive (open) systems. A module is an open system interacting with its environment. Its input x takes arbitrary values from the environment and the module should interactively compute its output y to satisfy its specification. The authors partition the module's state space into system and environment states. A system state can only be altered by program actions while an environment state can only be altered by the action of the environment. It turns out that model-based automated verification of both closed systems and open systems, with respect to LTL specifications, belongs to the class of PSPACE problems (in the length of the temporal specification). However, the authors' [108], [109] main result demonstrates that module checking against CTL specifications is harder than its closed system counterpart; the former is EXPTIME-Complete while the latter has a polynomial-time algorithm in the length of the specification. Kupferman and Vardi argued that although such results are discouraging, fragments of CTL (for instance, universally quantified CTL) are as efficient as their closed systems counterparts; i.e., modules in this fragment of CTL are checkable in polynomial-time. These results raise questions about how feasible is CTL versus LTL. Despite that CTL and LTL have incomparable expressive power, arguments about the superiority of CTL over LTL – which has simpler syntax and semantics – in model-based verification are not settled yet.

Vardi *et al.* [110] consider cases where all the environment behaviors should either be taken into account, or not, depending on the kind of temporal logic used for the specification. Kupferman *et al.* [109] distinguish cases where the environment has incomplete knowledge about the system, and prove their respective complexity bounds; this is analogous to decentralized/distributed systems.

9.3 Distributed Algorithms Approach

Self-stabilization was originally defined in the context of distributed algorithms [7]. Except for a handful of papers, boosting research in self-stabilization did not take place until Lamport's ACM address [111] stating that self-stabilization is one of Dijkstra's most brilliant ideas.

[6]For a comprehensive treatment of temporal logic in specification and verification of sequential and concurrent programs, please consult [106], [107].

Design of self-stabilization is a hard problem. To overcome some of its difficulties, authors propose weaker types of stabilization. Gouda *et al.* [14] introduce *weak stabilization* as a *possibility* for a system to reach its legitimate behavior: *the existence* of a computation that reaches a legitimate behavior from every state is necessary and sufficient for weak stabilization. Whereas *strong stabilization* requires that *every* computation from every state reaches a legitimate behavior [6]. As such, every strongly stabilizing protocol is weakly stabilizing; the converse is not necessarily true. Burns *et al.* introduce *pseudostabilization* as another relaxation of strong stabilization: stabilizing computations can bounce between legitimate and illegitimate states as long as they will eventually settle to a legitimate behavior [112].

Burns *et al.* [113] treat the effect of execution semantics on self-stabilization. In *interleaving semantics*, only one enabled process is allowed to execute at a time, while in *concurrent semantics*, subsets of enabled processes are allowed to execute simultaneously. The authors define conditions on protocols such that stabilization under *interleaving* execution semantics are sufficient for stabilization under *concurrent (subset)* semantics. Follow-up work provide transformations of stabilizing protocols under interleaving semantics to stabilize under concurrent semantics [114], [115]. Gradinariu and Tixeuil [115] propose a conflict manager at each network node to prevent two neighboring nodes from reading a shared variable concurrently; i.e., provide local mutual exclusion between neighboring processes. This way, a concurrent execution of processes is equivalent to an interleaved execution; in fact, the execution of non-neighboring processes do not interfere.

9.3.1 Manual Design

Most of the work done for designing convergence/stabilization advocates manual methods. In this subsection, we study the main approaches to manual design of self-stabilization.

9.3.1.1 Convergence Stairs/Ranking Functions

Manual methods define a decreasing ranking function to generate the recovery actions [55], [116]. Equivalently, manual methods use a layering/ranking approach to construct an order on sets of states from where the protocol eventually reaches its legitimate behavior [21], [22].

The fact that a strongly stabilizing system has no cycles in its set of illegitimate states

(Chapter 2) means that its corresponding transition graph is acyclic (in the set of illegitimate states). The transitive closure of the digraph/transition relation can be used to define a partial order on its set of vertices/states such that (s, s') is an arc/transition iff $(s' < s)$. Equivalently, a ranking function f_r is a monotonic function on the set of states; i.e., it preserves the partial order. A candidate ranking function is such that $f_{rl}(s)$ is the longest path from s to a legitimate state. f_r partitions the set of illegitimate states into *ranks* or *stairs*. Two states s_1 and s_2 have the same rank iff $f_r(s_1) = f_r(s_2)$. In the special case of f_{rl}, two states have the same rank if their longest paths to some legitimate state are of the same length. Consequently, partial orders on states, monotonic ranking functions and convergence stairs represent equivalent techniques for proving convergence.

Some manual methods use layering and modularization [21], [117] to enable the design of self-stabilization by incremental construction of convergence using either strictly decreasing [55] or non-increasing ranking functions [116]. Arora *et al.* [36], [117] provide a method based on constraint satisfaction, where they create a dependency graph of local constraints whose satisfaction guarantees recovery of the entire system. The dependency graph has arcs labeled by protocol actions (equivalently by the constraints they should establish). Each node is labeled by a variable name. An arc (n_1, n_2) corresponds to an action that reads variables of nodes n_1 and n_2 and modifies variables in node n_2 to establish its labeling constraint. As such, this graph models the interdependency between constraints and how establishing some of the constraints can violate others. The authors define three classes of dependency graphs: (1) Acyclic graphs, (2) acyclic graphs with self-loops and (3) cyclic graphs reducible to acyclic graphs with self-loops. In case (1), no cyclic corruption of constraints can occur then, eventually, all constraints will be established. In case (2), the designer should guarantee that establishing a constraint does not violate constraints of the loops. In case (3), the authors partition convergence actions into layers, each with its own acyclic constraint graph. Actions of a higher layer K should eventually establish all its constraints and consequently, pass control to actions of layer $K-1$. This process continues until all constraints are established.

9.3.1.2 Local Checking and Correction, Snapshots and Global Reset

Katz and Perry [118] present a general (but expensive) method for transforming a non-stabilizing system to a stabilizing one by taking global snapshots[7] and resetting the global state of the system if necessary. Their method assumes that there is one master process that initiates global resets based on the snapshot result; i.e, whether the global snapshot is a legitimate state or not.

[7]A global snapshot is a distributed computation where a process initiates a request to read the values of all the variables in the network and eventually receives these values.

To avoid the expense of taking global snapshots, some of the design methods focus on a class of protocols whose set of legitimate states can be checked and corrected by a single component (i.e., locally checked/corrected). Varghese [1] define local checking and local correcting of protocols. Consider L to be the set of legitimate states for the protocol p. p is locally checkable if there exists $L' \subset L$ such that L' is a conjunction of local predicates (a.k.a, a predicate defined over the local variables in r_j for a process P_j, for some j). Hence, establishing L' sufficiently establishes L. Moreover, Varghese defines local correction as a function in the local state of a node as well as its neighbors. For each link between two nodes in the network, he introduces a *link predicate* representing the constraint that needs to be established for the link variables. The existence of a transformation over link predicates that do not cause interference between processes is a necessary condition for local correction. Most of Varghese's work captures protocols for message passing models. He demonstrates that any locally checkable protocol can be transformed into a stabilizing one using global correction in at most N steps where N is the number of nodes in the network. Moreover, Varghese devises a definition of local checking and correction for shared memory models.

Awerbuch *et al.* [119] provide a compiler that produces self-stabilizing protocols given a synchronous, deterministic, non-interactive input protocol in the message passing model. They suggest a *resynchronizer* to augment the input protocol by a global state checker; then they apply techniques for correction either by using local correction or a global reset as described in [120], [121]. To design non-locally checkable systems, Varghese [122] proposes a counter flushing technique, where a leader node systematically increments and flushes the value of a counter throughout the network.

9.3.2 Automated Synthesis from Specification

We demonstrate attempts to solve Church's problem in the context of closed-systems. Emerson and Clarke [123] address the design of synchronization skeletons[8] of concurrent programs from their branching time logic specification. Their approach uses a tableau-based proof of the satisfiability of a branching temporal logic formula and generates a shared-memory model of this formula accordingly. They represent the tableau by and-or nodes where each node is a subformula of the original specification. Every atom/leaf of the tableau proof has a corresponding model (satisfying tree). They introduce a step to build these models from the tableau proof into Kripke structures[9] that satisfy the temporal logic formula; i.e., the system's specification. If the temporal logic formula is unsatisfiable, the

[8]Synchronization skeletons are portions of concurrent programs executing interprocess communication and synchronization as opposed to functional parts.

[9]Kripke structures are finite models for modal logics and can be represented by a relation between a finite number of worlds. Each of these worlds maintains the truth of a finite set of logical atoms; i.e., propositions [124].

specification is inconsistent.

Manna and Wolper [125] follow a similar track to Emerson and Clarke's. However, they use LTL as a specification method and use communicating sequential processes [126] for their model of concurrent programs. They justify their use of LTL by arguing that BTL is not necessary for expressing concurrency. Moreover, they criticize the use of a shared memory model by Emerson and Clarke because all the variables have to be globally declared. The alternative to shared memory in their approach is that all processes should exchange messages; their interconnection network is a complete graph.

9.3.3 Automated Program Revision

Automated program revision considers modification of protocols/programs/dynamic systems in the context of fault tolerance [9], [13]. Despite that systems are closed, the role of the environment is captured by a *fault-model*. Faults are global transitions that perturb the state of the protocol/program/dynamic system, but have the specific property that they are non-deterministic. For instance, consider a *deadlock* state where a fault transition f is enabled; f may not execute forever, that is, such a state is a deadlock. In other words, computations involving fault transitions are not *maximal* (Chapter 2), unlike computations involving only program transitions. This way, some of the general properties of open systems are captured by closed system models.

Most of the work on automated program revision focuses on adding *safety* properties to distributed protocols [127] to render them fault-tolerant. Ebnenasir *et al.* devise a framework for the addition of fault tolerance using an enumerative representation of states and transitions [128]. Kulkarni and Ebnenasir prove that addition of *fail-safe* fault tolerance to a distributed protocol is NP-Complete in the size of the state space [129]. A protocol is *fail-safe* fault-tolerant if under a certain fault model, it will never violate the *safety* part of its specification [13]. Bonakdarpour and Kulkarni design and implement symbolic heuristics to add safety and liveness properties to distributed protocols [39]. They demonstrate that the addition of *liveness* to distributed UNITY programs is NP-Complete (in the size of the state space) [130]. Moreover, they devise a heuristic for the synthesis of *leads-to* properties in a distributed program.

Other automated techniques augment the input protocol transition system with recovery actions to enable the protocol's convergence to its legitimate behavior for a class of locally correctable protocols [131]. Abujarad and Kulkarni present a method for adding convergence to acyclic network topologies. It is known from Varghese *et al.* that acyclic topologies can be made stabilizing by local corrections [1]: for such topologies, non-progress cycles of illegitimate behavior can be avoided easily by spreading the state corruption to the leaf nodes of the network.

We consider a program revision approach for the specific problem of adding convergence/stabilization to distributed protocols. Our contribution is to create a set of heuristics that, together, provide help for the designer by generating candidate solutions that are correct-by-construction. We do not restrict our network topologies to any specific class. Moreover, we consider heuristics that cover cases that cannot be solved automatically by heuristics as in [128], [130], [131] like cyclic network topologies.

9.3.4 Automated Verification of Self-Stabilization

A few authors investigate verification of self-stabilization from either a model-checking or an automatic theorem proving perspectives.

Merz *et al.* [132] illustrate a mechanical proof using Isabelle [133] to Dijkstra's mutual exclusion algorithm [7]. They establish a series of lemmas and corollaries through mechanical verification by defining a decreasing function of network states to prove convergence. Similarly, Qadeer and Shankar [134] automatically verify a proof for Dijkstra's mutual exclusion algorithm using the PVS theorem prover [135]; they demonstrate the challenges of mechanically verifying stabilization as compared to manual proofs. Kulkarni *et al.* [136] illustrate how decomposition of a distributed protocol can help its mechanical verification. They use PVS to prove the correctness of Dijkstra's mutual exclusion algorithm by splitting it into a convergence component and a closure component.

Shukla *et al.* [137] develop a tool that reuses the SPIN model checker [26] for the purpose of verifying self-stabilization. Tsuchiya *et al.* exploit the SMV model checker to symbolically verify [138] several protocols for self-stabilization and discover a bug in an otherwise stabilizing protocol. Their results illustrate that, although symbolic model checking permits the exploration of a larger state space, the size of the verified protocols was still limited (8 to 10 processes). Their work is usually cited as an experimental witness of the hardness of verification of self-stabilization; let alone its synthesis.

To overcome the state explosion problem in verification of self-stabilization, Liveris *et al.* consider program abstraction for verification of parametrized[10] protocols [139]. They apply some sufficient conditions due to Kesten and Pnueli [140] for verifying liveness in parametrized systems by transforming each process with a parameterized number of

[10]A parametrized protocol is correct regardless of its number of processes. Here parameterization is with respect to the number of processes.

variables to one with fixed number of variables. They illustrate their approach on three case studies: leader election, coloring and spanning tree construction.

9.3.5 Parameterized Verification of Convergence

We summarize related work to the verification of properties of parameterized networks. In particular, we focus on the verification of convergence in arbitrary-sized networks and how it compares to the results of Chapters 5 and 6.

Parameterized Verification Using Cutoffs. Emerson *et al.* [51]–[54] reduce subclasses of the *parameterized verification problem* to the verification of small network sizes with fixed number of processes. Emerson and Namjoshi [51] assume a ring of symmetric processes that communicate by means of a token. They demonstrate that the verification of conjunctive properties over pairs of processes reduces to verification of rings with a fixed size. Moreover, the authors demonstrate that adopting value-carrying tokens renders the verification problem undecidable. In [52], Emerson and Namjoshi extend their approach by reducing arbitrary-sized, star-interconnected, synchronous protocols to abstract graphs of finite size. Emerson and Kahlon [54] demonstrate the existence of small model reductions to parameterized valued-token passing rings as long as every process passes the tokens for a finite number of times. In [53], they extend their reduction to arbitrary sized models whose actions have conjunctive or disjunctive guards. Their model can only capture processes all of which access the same set of variables.

Our model of a unidirectional ring is incomparable to Emerson *et al.*'s. First, we do not assume symmetry. Second, our model represents processes that communicate multi-valued tokens. However, our assumption of self-disabling processes is more restrictive than Emerson *et al.*'s. Nevertheless, a unidirectional ring with self-disabling processes captures a class of useful distributed protocols that is not covered by the aforementioned methods.

Verification by abstraction. A considerable amount of work adopt abstraction to handle the infinite number of states in parameterized verification. *Network invariants* are introduced by Wolper and Lavinfosse [141] to capture all possible behaviors of an arbitrary number of processes in the network. A property satisfied by a network invariant is satisfied by any instance of the network but not necessarily the converse; abstraction is hence necessarily incomplete. Kurshan and McMillan [142] demonstrate a general abstraction rule based on composition and induction over a sequence of processes. The generality of their approach is due to the abstract properties of their composition operators and partial order relations on processes. Kesten *et al.* [143] present yet another induction method using network invariants with a proof rule based on an abstraction relation and composition of processes.

The main drawback of abstraction methods with respect to convergence synthesis is their dependence on human ingenuity for generating abstractions; every protocol requires a different abstract network invariant that, in general, cannot be automatically computed. To overcome this drawback, Pnueli *et al.* [144] demonstrate a method where conjunctive sets of reachable states can be automatically deduced. They project the set of reachable states, for a specific network size, over a subset of "variables of interest" in some conjunct. Their method generalizes the projected conjunct for every process in the network. They provide a cutoff theorem, thereby reducing verification of an arbitrary-sized network to a finite number of protocol instances. Despite the inherent incompleteness of this method, it has proved that it is of practical values in automated verification of safety properties. A similar approach for verifying response properties by Fang *et al.* [145] abstracts out decreasing ranking functions for an arbitrary protocol instance. They generalize the convergence stairs likewise while using a cutoff theorems proper to response properties.

Namjoshi [146] illustrates that the cutoff method for verification of parameterized systems is complete for safety properties. That is, there always exists a maximum size for the number of symmetric processes that captures all the "behaviors of interest" in the network with respect to a given safety property. Furthermore, he provides a modification to the method by Pnueli *et al.* [144] to accommodate his completeness result.

Network grammars. Shtadler and Grumberg [147] introduce network grammars as a means to representing global states of arbitrary-sized networks of linear or ring topologies, as words generated by network grammars. For verification purposes, they compute an equivalent network invariant to the network grammar and apply finite state verification on the equivalent model/abstraction. As an extension, Clarke *et al.* [148] relax the equivalence relation between the model and its network invariant to a pre-order relation such that the network invariant abstracts out the grammar; this relaxation increases the possibility of finding an invariant at the cost of completeness. Kesten *et al.* [149] restrict network grammars to regular languages; however their approach extends verification to tree-like topologies by capturing their global states as accepted trees by a tree-automaton. Moreover, they represent reachable sets of states by finite automata, thereby reducing the verification of safety properties to automata-theoretic product and emptiness problems.

A follow-up of the aforementioned approaches generated a plethora of publications in what is now called *regular model checking*. Jonnson and Nilsson [150] describe how to derive a finite state transducer representing the transitive closure of the network's transition relation. A finite state transducer is a finite state automaton augmented with a function that maps the set of input alphabet to the set of output symbols. Subsequently, they illustrate how to verify safety properties using their derived transitive closure automaton. Bouajjani *et al.* [151] demonstrate different techniques to compute finite state transducers representing the set of reachable states and the transitive closure relation of a parameterized protocol, respectively. They illustrate how to make use of the transitive closure relation to verify liveness properties. Abdulla *et al.* [152] introduce an abstraction on regular model checking by assuming a preorder relation between words representing states. This relation eliminates transducers in verification of safety properties, thereby simplifying the computationally demanding automata-theoretic operations required by regular model checking. Due to the extensive literature on regular model checking, we direct the reader to a survey by Abdulla *et al.* [153].

Livelocks in Network Protocols. In one of the early investigations of livelocks, Kwong [41], [46] characterizes a livelock in a parallel program by an infinite computation where some process is never executed; a.k.a., starvation. Kwong demonstrates two techniques for proving livelock-freedom: (1) Sufficient conditions that guarantee the absence of livelocks and, (2) a well-founded set technique. Kwong's first method is amenable to automatic verification, however, the sufficient conditions he establishes are applicable only to deterministic protocols, with non-interfering[11] and independent local transitions. Gouda and Chang [47] identify a subset of networks that can be exactly abstracted by finite graphs and thus, whose livelock-freedom is decidable. Despite the fact that Gouda and Chang [47] capture unbounded channels, their approach is not applicable for our purpose of proving livelock-freedom in a parameterized ring.

Static Analysis of Livelocks Leue *et al.* [48], [154], [155] devise the notion of cycle dependency for control flow analysis. Intuitively, a cycle dependency is a relation between local cycles C_1 and C_2 such that C_2 continues to execute if C_1 is executed repeatedly. The authors demonstrate a sufficient test for livelock-freedom by transforming the input protocol to an integer linear program. Their knowledge of cycle dependencies often results in better scaling of their exploration for livelocks. Ouaknine *et al.* [49] provide a sound but incomplete syntactic check on CSP[12] protocols for livelock-freedom. Blieberger *et al.* [50] conservatively check livelock-freedom in Ada programs by examining the communication patterns between potential loops. In a sense, our feasible propagation is a formalization of the notion of dependency between local cycles captured by Leue *et al.* and Blieberger *et al.* [50], [155], respectively.

We summarize in Table 9.1 the direct contribution of each approach to the design and verification of self-stabilization.

We summarize the complexity results of different categories in Table 9.2.

[11]Non-interfering local transitions do not disable each other

[12]Communicating Sequential Processes

Table 9.1
Contribution of approaches to design and verification of self-stabilization

Problem		Supervisory Control	Open Systems	Closed Systems
Model-Based Verification	*LTL*	N/A	No	Yes
	CTL	N/A	No	Yes
Spec-Based Synthesis	*Centralized*	Yes	No	No
	Distributed	No	No	No
Program Revision	*Centralized*	N/A	No	Yes
	Distributed	N/A	No	No[a]

[a]This is where we contribute

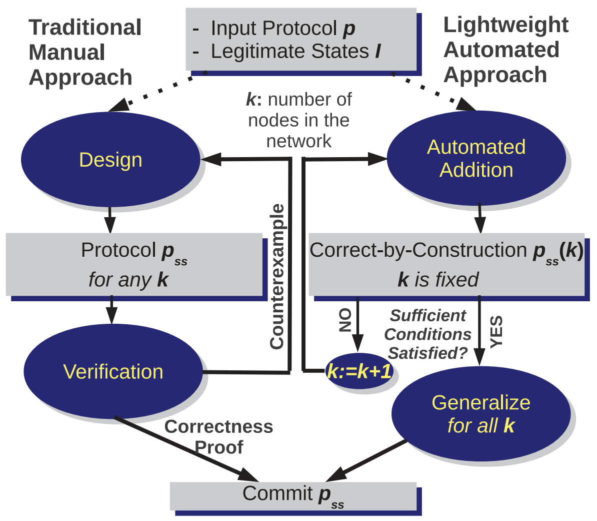
Table 9.2
Summary of Complexity Results

Problem		Open Systems	Closed Systems
Model-Based Verification	*LTL*	PSPACE (l)[a]	PSPACE (l)
	CTL	EXPTIME-Complete (s)	PTIME (s)
Spec-Based Synthesis	*Centralized*	PTIME (s)[b]	PTIME (s)
	Distributed	Undecidable	PSPACE (s)
Program Revision	*Centralized*	NP-Complete (s)[c]	PTIME (s)
	Distributed	N/A	NP-Complete (s)[d]

[a]l: length of the specification. s: size of the state space
[b]However, time complexity increases exponentially with the number of accepting states in an intermediately generated automaton
[c]For a subset of invariant properties, repair as a game takes a polynomial time in the size of the state space
[d]Addition of convergence is in NP but we are not aware of neither polynomial-time synthesis algorithm, nor a proof of completeness.

9.4 Discussion

We proposed an approach as an extension of program revision for synthesis of convergence/stabilization in closed systems. We considered a transient fault model that can non-deterministically corrupt the state of a distributed program/protocol without causing any permanent damage. We also assumed that faults will eventually stop occurring so recovery can be guaranteed. We explored properties of transition systems of stabilizing



Traditional Manual Approach
- Input Protocol p
- Legitimate States I
Lightweight Automated Approach
k: number of nodes in the network
Design
Automated Addition
Protocol p_{ss} for any k
Counterexample
Correct-by-Construction $p_{ss}(k)$ k is fixed
NO
Sufficient Conditions Satisfied?
YES
Verification
k:=k+1
Generalize for all k
Correctness Proof
Commit p_{ss}

Figure 9.2: Synthesize in Small Scale and Generalize

protocols. We provided algorithmic methods (heuristics) to intelligently and efficiently transform the non-stabilizing transition system into a stabilizing one. As contrasted to Awerbuch *et al.*'s compiler, our approach automatically produced a tailored stabilizing solution for each given protocol (like Dijkstra's solution in [7]) without imposing global snapshot algorithms or global resets which are, in general, expensive. Unlike Awerbuch *et al.*'s compiler, our heuristics could handle nondeterministic, asynchronous and interactive protocols. Our synthesized solution is in the worst case, as expensive as the approach in Awerbuch *et al.* [119].

Our approach is significantly different from Varghese's as we handled shared-memory models. We did not assume local predicates on links; i.e., predicates on the messages sent over links, and hence, Varghese's theorem for linear recovery time does not apply to our case.

We devised sound but incomplete algorithms (i.e., heuristics) in Chapter 3 for automating the design of convergence while mitigating the entangled cycle and deadlock resolution problems. Our heuristics may fail to find a design while there exists one, but if they find one, this design is self-stabilizing by construction. Moreover, we kept our approach *lightweight*: we obtained solutions for input instances having a few number of components and inductively reasoned about them by gradually increasing the number of components as far as our computational resources permitted. In fact, we leveraged the power of computer clusters to design convergence in symmetric protocols (Chapter 4). To circumvent the state explosion problem, we resorted to symbolic (implicit) methods for representing transition systems [19], [138].

To alleviate the difficulty of scaling-up our stabilizing solutions, we pursued an approach where we *design-in-small-scale* and *generalize*; Figure 9.2 summarizes how our paradigm differs from the traditional methods for designing self-stabilization. We founded a theory for the study of deadlocks and livelocks in unidirectional rings of arbitrary sizes. Specifically, we provided in Chapter 5 necessary and sufficient conditions – verifiable in the local state space of each process– for the deadlock-freedom of ring networks. Moreover, we illustrated our preliminary results about livelocks in unidirectional rings and derived thereof an algorithm sketch for designing convergence in unidirectional rings. Chapter 6 established an algebraic representation of livelocks in unidirectional rings.

Our local approach to verification and design reasons about a variety of possible solutions for a given conjunctive set of legitimate states closed in the input protocol. We investigated generalization in local state spaces, thereby enabling a method that combines design and verification instead of conceiving them as separate tasks. Thus, our approach differs from automated abstraction techniques like Fang *et al.*'s decreasing ranking functions [145], or any of the aforementioned regular model checking techniques.

Our local algebraic characterization is necessary and sufficient for livelocks in unidirectional rings whose processes are self-disabling. In spite of our restricted model of computation, we managed to capture many protocols of interest that have self-disabling processes. A relaxation of self-disablement renders our characterization only necessary but insufficient for livelock-freedom. A more powerful model of computation on a unidirectional ring converts livelock-freedom verification to an undecidable problem. As such, an acceptable characterization of livelocks in more general models of computation (as in Suzuki [45]) essentially has non-identical necessary and sufficient conditions for livelock-freedom. We believe that our exploration of the local state space laid a starting point for efficient reasoning about convergence of distributed protocols, thereby alleviating the hardness of global state space exploration.

Chapter 10

Concluding Remarks

We present in this chapter a summary of the contributions of this dissertation. Section 10.1 is a recall of the main results we covered throughout our exposition. In Section 10.2, we summarize a set of related open problems that still require further investigation.

10.1 Summary

In Chapters 3 and 4, we developed heuristics for synthesizing convergence into non-stabilizing protocols that satisfy closure. The first heuristic transforms a non-converging protocol to a converging protocol, while the second heuristic ensures that the output protocol is symmetric provided the symmetry of the input protocol. We capture a protocol's atomic execution steps by actions. Our heuristics keep adding convergence actions to resolve deadlocks while ensuring livelock-freedom in the set of illegitimate global states; i.e., the heuristic discards actions forming global non-progress cycles. Our heuristics include convergence actions enabled only outside the set of legitimate configurations to ensure closure. For cycle resolution, we implemented a Strongly Connected Component (SCC) detection algorithm by Gentitlini *et al.* [23].

Our heuristics automatically generated solutions to Dijkstra's token passing protocol (for the first time). They also generated solutions for maximal matching on a ring adapted from [15]. Our heuristics corrected a pitfall in a solution for matching on a ring. This further motivates the need for automated design methods. Our heuristics also generated symmetric and non-symmetric solutions to ring coloring, leader election and agreement protocols with network sizes up to 40 nodes [10].

Our experiments on global state space always hit the bottleneck of state space explosion. Moreover, the local nature of information about the global state in the process of a distributed system imposed transition grouping constraints during the addition of convergence. Complete algorithms for the addition of convergence could not ignore the combinatorial search problem for a stabilizing solution in the global state space. Furthermore, solutions found by global state space exploration are not necessarily generalizable; even if our heuristics succeed in finding a solution for a specific network size, there are no guarantees that generalizations of the obtained solution will preserve convergence for an arbitrary number of processes.

In Chapters 5 and 6, we demonstrated a theoretical foundation for reasoning locally about global properties of network protocols. Specifically, we illustrated how a protocol on a ring topology is provably deadlock-free by examining the local state space of each process in the ring. Interestingly enough, we proved the existence of locally verifiable necessary and sufficient conditions for livelock-freedom of protocols on unidirectional rings. We illustrated the power of our theory by providing, in Section 6.3, yet another proof of livelock-freedom of Dijkstra's celebrated token ring protocol [7]. A pleasant outcome of our theory about livelock-freedom in unidirectional rings is that it holds for protocols irrespective of their concurrency execution semantics, or fairness assumptions.

In Chapter 7, we presented an application of the design of convergence in wireless sensor nodes. We demonstrated how to add recovery to the TinyOS scheduler in order to maintain an intact linked list of tasks identifiers. We illustrated that our approach surpasses traditional error detecting and correcting codes in terms of time and space usage.

10.2 Extensions

We conjecture that local reasoning is very promising in that it eliminates the need for global state space exploration, thereby obviating state space explosion and enabling proofs of properties that hold for arbitrary number of processes. In order to demonstrate the impact of our theory, we shall develop algorithms and tools for the design of convergence in local state space, in the same vein as the methodology that we have illustrated in Section 5.4.

Deadlock-Freedom and Local Safety Properties. We will extend our theory for deadlock-freedom in two directions. First, we will consider formulating a theory for deadlocks in arbitrary network topologies. It turns out that the continuation relation on local states is definable for arbitrary topologies and not just for rings. It remains to demonstrate how the recursive addition of nodes/processes to a distributed protocol affects the continuation relation of each of its processes, thereby affecting the finiteness of the continuation graph for arbitrary network sizes. Second, we will equally consider reachability properties of subsets of local states. Specifically, we will investigate the possibility of designing general safety properties in arbitrary networks by exploring the local state spaces of their processes in a way similar to local reasoning about deadlocks.

Livelock-Freedom and Non-Self-Disablement. Despite the fact that the verification of livelocks in unidirectional rings is undecidable in general [45], we consider formulating non-identical necessary and sufficient conditions for the livelock-freedom of general protocols on unidirectional rings. We shall develop similar conditions for arbitrary networks too. It turns out that, in arbitrary networks of self-disabling processes, the absence of sub-rings of processes where the Flow Equation, as defined in Section 6.2 does not hold, is sufficient for proving global livelock-freedom of the ring. On the other hand, having an *isolated* sub-ring of processes in an arbitrary network, where both the Flow and Reconstruction Equations of Chapter 6 hold, sufficiently prove the existence of a livelock. A sub-ring is *isolated* if and only if every local transition participating in the global livelock does not access variables other than those in the designated sub-ring. We shall investigate the possibility of further weakening (strengthening) these sufficient (necessary) conditions for livelock-freedom. Ultimately, we plan to develop synthesis algorithms and corresponding tools that reason about deadlocks, livelocks and local safety properties for arbitrary distributed protocols.

References

[1] G. Varghese, “Self-stabilization by local checking and correction,” Ph.D. dissertation, Masschussets Institute of Technology, 1993.

[2] E. Rosen, “Vulnerabilities of network control protocols: An example,” *ACM Special Interest Group on Data Communications Computer Communication Review*, vol. 11, no. 3, pp. 10–16, 1981.

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

[4] C. Cassandras and S. Lafortune, *Introduction to discrete event systems*. Springer, 2008.

[5] M. Schneider, “Self-stabilization,” *ACM Computing Surveys*, vol. 25, no. 1, pp. 45–67, 1993.

[6] M. Gouda, “The triumph and tribulation of system stabilization,” in *Proceedings of the 9th International Workshop on Distributed Algorithms*. London, UK: Springer-Verlag, 1995, pp. 1–18.

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

[8] R. Keller, “Formal verification of parallel programs,” *Communications of the ACM*, vol. 19, no. 7, pp. 371–384, 1976.

[9] A. Arora and M. Gouda, “Closure and convergence: A foundation of fault-tolerant computing,” *IEEE Transactions on Software Engineering*, vol. 19, no. 11, pp. 1015–1027, 1993.

[10] A. Ebnenasir and A. Farahat, “A lightweight method for automated design of convergence,” in *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2011, pp. 219–230.

[11] ——, “Swarm synthesis of convergence for symmetric protocols,” in *2012 Ninth European Dependable Computing Conference (EDCC)*. IEEE, 2012, pp. 13–24.

[12] A. Farahat and A. Ebnenasir, "Local reasoning for global convergence of parameterized rings," in *Proceedings of the 32nd International Conference on Distributed Computing Systems*, June 2012.

[13] S. S. Kulkarni and A. Arora, "Automating the addition of fault-tolerance," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*. London, UK: Springer-Verlag, 2000, pp. 82–93.

[14] M. Gouda, "The theory of weak stabilization," in *5th International Workshop on Self-Stabilizing Systems*, ser. Lecture Notes in Computer Science, vol. 2194, 2001, pp. 114–123.

[15] M. G. Gouda and H. B. Acharya, "Nash equilibria in stabilizing systems," in *11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, 2009, pp. 311–324.

[16] E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1990.

[17] E. Emerson, "Temporal and modal logic, handbook of theoretical computer science (vol. b): formal models and semantics," 1991.

[18] A. Farahat and A. Ebnenasir, "A Lightweight Method for Automated Design of Convergence in Network Protocols," *To appear in ACM Transactions on Autonomous and Adaptive Systems*, 2012.

[19] R. Bryant, "Graph-based algorithms for boolean function manipulation." *IEEE Transactions On Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[20] S. Dolev and T. Herman, "Superstabilizing protocols for dynamic distributed systems," in *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 1995, p. 255.

[21] F. Stomp, "Structured design of self-stabilizing programs," in *Proceedings of the 2nd Israel Symposium on Theory and Computing Systems*, 1993, pp. 167–176.

[22] M. G. Gouda and N. Multari, "Stabilizing communication protocols," *IEEE Transactions on Computers*, vol. 40, no. 4, pp. 448–458, 1991.

[23] R. Gentilini, C. Piazza, and A. Policriti, "Computing strongly connected components in a linear number of symbolic steps," in *the 14th Annual ACM-SIAM symposium on Discrete algorithms*, 2003, pp. 573–582.

[24] A. Ebnenasir and A. Farahat, "Towards an extensible framework for automated design of self-stabilization," Michigan Technological University, Tech. Rep. CS-TR-10-03, May 2010, http://www.cs.mtu.edu/html/tr/10/10-03.pdf.

[25] E. Emerson and A. Sistla, "Symmetry and model checking," *Formal methods in system design*, vol. 9, no. 1, pp. 105–131, 1996.

[26] G. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[27] S.-T. Huang, "Leader election in uniform rings," *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 3, pp. 563–573, Jul. 1993.

[28] F. Somenzi, "CUDD: CU decision diagram package release 2.3. 0," 1998.

[29] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[30] T. Cottenier, A. van den Berg, and T. Elrad, "Motorola WEAVR: Aspect and model-driven engineering," *Journal of Object Technology*, vol. 6, no. 7, pp. 51–88, 2007.

[31] D. E. Knuth, *The art of computer programming: Volume 4, Fascicle 2. Generating all tuples and permutations*. Addison-Wesley, 2005.

[32] F. Somenzi, K. Ravi, and R. Bloem, "Analysis of symbolic SCC hull algorithms," in *Formal Methods in Computer-Aided Design*. Springer, 2002, pp. 88–105.

[33] A. Ebnenasir and A. Farahat, "Swarm synthesis of convergence for symmetric protocols," Michigan Technological University, Tech. Rep. CS-TR-11-02, May 2011, http://www.cs.mtu.edu/html/tr/11/11-02.pdf.

[34] F. Abujarad and S. S. Kulkarni, "Automated constraint-based addition of nonmasking and stabilizing fault-tolerance," *Journal of Theoretical Computer Science*, vol. 258, no. 2, pp. 3–15, 2011, in Press.

[35] G. J. Holzmann, R. Joshi, and A. Groce, "Swarm verification techniques," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–15, 2010.

[36] A. Arora, M. Gouda, and G. Varghese, "Constraint satisfaction as a basis for designing nonmasking fault-tolerant systems," *Journal of High Speed Networks*, vol. 5, no. 3, pp. 293–306, 1996, a preliminary version appeared at ICDCS'94.

[37] W. Leal and A. Arora, "Scalable self-stabilization via composition," in *IEEE International Conference on Distributed Computing Systems*, 2004, pp. 12–21.

[38] P. Godefroid, "Using partial orders to improve automatic verification methods," in *Computer-Aided Verification*. Springer, 1991, pp. 176–185.

[39] B. Bonakdarpour and S. S. Kulkarni, "Exploiting symbolic techniques in automated synthesis of distributed programs with large state space," in *Proceedings of the 27th International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, June 2007, pp. 3–10.

[40] S. Shukla, D. Rosenkrantz, and S. Ravi, "Developing self-stabilizing coloring algorithms via systematic randomization," in *Proceedings of the International Workshop on Parallel Processing*, 1994, pp. 668–673.

[41] Y. Kwong, "Livelocks in parallel programs," *International Journal of Computer Mathematics*, vol. 10, no. 2, pp. 121–135, 1981.

[42] L. Lamport, "A new solution of dijkstra's concurrent programming problem," *Communications of the ACM*, vol. 17, no. 8, pp. 453–455, 1974.

[43] F. Gartner, "Fundamentals of fault-tolerant distributed computing in asynchronous environments," *ACM Computing Surveys*, vol. 31, no. 1, 1999.

[44] K. Apt and D. Kozen, "Limits for Automatic Verification of Finit-State Concurrent Systems," *Information Processing Letters*, vol. 22, no. 6, pp. 307–309, 1986.

[45] I. Suzuki, "Proving properties of a ring of finite-state machines," *Information Processing Letters*, vol. 28, no. 4, pp. 213–214, 1988.

[46] Y. Kwong, "Livelocks in parallel programs," *International Journal of Computer Mathematics*, vol. 10, no. 3-4, pp. 201–212, 1982.

[47] M. Gouda and C. Chang, "Proving liveness for networks of communicating finite state machines," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 1, pp. 154–180, 1986.

[48] S. Leue and W. Wei, "Integer linear programming based property checking for asynchronous reactive system," *IEEE Transactions on Software Engineering*, no. 99, pp. 1–21, 2011.

[49] J. Ouaknine, H. Palikareva, A. Roscoe, and J. Worrell, "Static livelock analysis in CSP," *22nd International Conference on Concurrency Theory*, pp. 389–403, 2011.

[50] J. Blieberger, B. Burgstaller, and R. Mittermayr, "Static Detection of Livelocks in Ada Multitasking Programs," in *Proceedings of the 12th international conference on Reliable software technologies*. Springer-Verlag, 2007, pp. 69–83.

[51] E. Emerson and K. Namjoshi, "Reasoning about rings," in *Conference Record of the ACM Symposium on Principles of Programming Languages*, vol. 22. Association of Computer Machinery, 1995, pp. 85–94.

[52] ——, “Automatic verification of parameterized synchronous systems,” in *Computer Aided Verification*. Springer, 1996, pp. 87–98.

[53] E. Emerson and V. Kahlon, “Reducing model checking of the many to the few,” *Automated Deduction (CADE)*, pp. 236–254, 2000.

[54] ——, “Parameterized model checking of ring-based message passing systems,” in *Computer Science Logic*. Springer, 2004, pp. 325–339.

[55] J. L. W. Kessels, “An exercise in proving self-stabilization with a variant function,” *Information Processing Letters*, vol. 29, no. 1, pp. 39–42, 1988.

[56] M. Herlihy and N. Shavit, “The topological structure of asynchronous computability,” *Journal of the ACM (JACM)*, vol. 46, no. 6, pp. 858–923, 1999.

[57] A. Farahat and A. Ebnenasir, “Local reasoning for global convergence of parameterized rings,” Michigan Technological University, Tech. Rep. CS-TR-11-04, November 2011, http://www.cs.mtu.edu/html/tr/11/11-04.pdf.

[58] ——, “Exploiting computational redundancy for efficient recovery from soft errors in sensor nodes,” in *Proceedings of Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2011, pp. 619–624.

[59] R. W. Hamming, *Coding and Information Theory*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986.

[60] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer *et al.*, “TinyOS: An operating system for sensor networks,” *Ambient Intelligence*, pp. 115–148, 2005.

[61] F. Koushanfar, M. Potkonjak, and A. Sangiovanni-Vincentelli, “Fault tolerance techniques for wireless ad hoc sensor networks,” *sensors*, pp. 1491–1496, 2002.

[62] T. Clouqueur, P. Ramanathan, K. Saluja, and K. Wang, “Value-fusion versus decision-fusion for fault-tolerance in collaborative target detection in sensor networks,” in *Proceedings of Fourth International Conference on Information Fusion*. Citeseer, 2001.

[63] W. Heinzelman, J. Kulik, and H. Balakrishnan, “Adaptive protocols for information dissemination in wireless sensor networks,” in *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*. ACM, 1999, pp. 174–185.

[64] M. Arumugam and S. Kulkarni, “Self-stabilizing deterministic TDMA for sensor networks,” *Distributed Computing and Internet Technology*, pp. 69–81, 2005.

[65] S. S. Kulkarni and M. Arumugam, “Infuse: A TDMA based data dissemination protocol for sensor networks,” *International Journal on Distributed Sensor Networks (IJDSN)*, vol. 2, no. 1, pp. 55–78, 2006.

[66] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler, “The nesC language: A holistic approach to networked embedded systems,” in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, vol. 38, no. 5. ACM, 2003, pp. 1–11.

[67] G. C. Clark and J. B. Cain, *Error-Correction Coding for Digital Communications*. Springer, 1981.

[68] I. S. Reed and G. Solomon, “Polynomial codes over certain finite fields,” *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, pp. 300–304, 1960.

[69] Y. Aumann and M. Bender, “Fault tolerant data structures,” in *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*. Published by the IEEE Computer Society, 1996, pp. 580–589.

[70] I. Finocchi, F. Grandoni, and G. Italiano, “Resilient search trees,” in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2007, pp. 547–553.

[71] A. Jørgensen, G. Moruz, and T. Mølhave, “Priority queues resilient to memory faults,” *Algorithms and Data Structures*, pp. 127–138, 2007.

[72] B. Bonakdarpour and S. Kulkarni, “Sycraft: A tool for synthesizing distributed fault-tolerant programs,” *19th International Conference on Concurrency Theory*, pp. 167–171, 2008.

[73] T. Kam and R. K. Brayton, “Multi-valued decision diagrams,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M90/125, 1990. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1990/1671.html

[74] K. Ravi, R. Bloem, and F. Somenzi, “A comparative study of symbolic algorithms for the computation of fair cycles,” in *Formal Methods in Computer-Aided Design*. Springer, 2000, pp. 162–179.

[75] D. Chai, J. Jiang, Y. Jiang, Y. Li, A. Mishchenko, and R. Brayton, “Mvsis 2.0 user ’s manual,” *Department of Electrical Engineering and Computer Sciences*, 2003.

[76] D. Fudenberg and J. Tirole, *Game theory. 1991*. MIT Press, 1991.

[77] L. Lamport and N. Lynch, “Distributed computing: Models and methods,” in *Handbook of theoretical computer science (vol. B)*. MIT Press, 1991, pp. 1157–1199.

[78] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, Massachusetts: The MIT Press, 1999.

[79] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT Press, 2008.

[80] P. J. G. Ramadge and W. M. Wonham, “The control of discrete event systems,” *Proc. of the IEEE*, vol. 77, no. 1, pp. 81–98, Jan. 1989.

[81] F. Lin and W. Wonham, “Decentralized supervisory control of discrete-event systems* 1,” *Information Sciences*, vol. 44, no. 3, pp. 199–224, 1988.

[82] Ozveren, Willsky, and Antsaklis, “Stability and stabilizability of discrete event dynamic systems,” *JACM: Journal of the ACM*, vol. 38, 1991.

[83] K. M. Passino, A. N. Michel, and P. J. Antsaklis, “Lyapunov stability of a class of discrete event systems,” *IEEE Transactions on Automatic Control*, vol. 39, no. 2, pp. 269–279, Feb. 1994.

[84] K. Passino, K. Burgess, and A. Michel, “Lagrange stability and boundedness of discrete event systems,” *Discrete Event Dynamic Systems*, vol. 5, no. 4, pp. 383–403, 1995.

[85] K. Passino and K. Burgess, *Stability analysis of discrete event systems*. Wiley, 1998.

[86] A. Bacciotti and L. Rosier, *Liapunov functions and stability in control theory*. Springer Verlag, 2005.

[87] J. Oehlerking, A. Dhama, and O. Theel, “Towards automatic convergence verification of self-stabilizing algorithms,” *Self-Stabilizing Systems*, pp. 198–213, 2005.

[88] R. Kumar, V. Garg, and S. Marcus, “Language stability and stabilizability of discrete event dynamical systems,” *SIAM journal on control and optimization*, vol. 31, pp. 1294–1294, 1993.

[89] S. Young and V. Garg, “On self-stabilizing systems: an approach to the specification and design of fault tolerant systems,” in *Proceedings of the 32nd IEEE Conference on Decision and Control*. IEEE, 1993, pp. 1200–1205.

[90] B. Alpern and F. Schneider, “Defining liveness,” *Information processing letters*, vol. 21, no. 4, pp. 181–185, 1985.

[91] A. Church, “Applications of recursive arithmetic to the problem of circuit synthesis,” *Summaries of the Summer Institute of Symbolic Logic*, vol. 1, pp. 3–50, 1957.

[92] ——, “Logic, arithmetic and automata,” in *Proceedings of the international congress of mathematicians*, 1962, pp. 23–35.

[93] M.O. Rabin, “Automata on Infinite Objects and Church’s Problem,” *American Mathematical Society*, 1972.

[94] J. Buchi and L. Landweber, “Solving sequential conditions by finite-state strategies,” *Transactions of the American Mathematical Society*, vol. 138, no. 5, pp. 295–311, 1969.

[95] A. Pnueli and R. Rosner, “On the synthesis of a reactive module,” in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1989, pp. 179–190.

[96] ——, “On the synthesis of an asynchronous reactive module,” *Automata, Languages and Programming*, pp. 652–671, 1989.

[97] N. Piterman, A. Pnueli, and Y. Saar, “Synthesis of reactive designs,” in *Verification, Model Checking, and Abstract Interpretation*. Springer, 2006, pp. 364–380.

[98] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem, “Anzu: A tool for property synthesis,” in *Proceedings of the 19th international conference on Computer aided verification*. Springer-Verlag, 2007, pp. 258–262.

[99] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer, “Interactive presentation: Automatic hardware synthesis from specifications: a case study,” in *Proceedings of the conference on Design, automation and test in Europe*. EDA Consortium, 2007, pp. 1188–1193.

[100] A. Pnueli and R. Rosner, “Distributed reactive systems are hard to synthesize,” in *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*. IEEE, 1990, pp. 746–757.

[101] O. Kupferman and M. Vardi, “Synthesizing distributed systems,” in *Logic in Computer Science*. Published by the IEEE Computer Society, 2001, p. 0389.

[102] B. Jobstmann, A. Griesmayer, and R. Bloem, “Program repair as a game,” in *Computer Aided Verification*. Springer, 2005, pp. 226–238.

[103] M. Vardi and P. Wolper, “An automata-theoretic approach to automatic program verification,” in *Proceedings of the First Symposium on Logic in Computer Science*. Cambridge, UK, 1986, pp. 322–331.

[104] A. Griesmayer, R. Bloem, and B. Cook, “Repair of boolean programs with an application to C,” in *Computer Aided Verification*. Springer, 2006, pp. 358–371.

[105] A. Pnueli, “The temporal logic of programs,” in *18th Annual Symposium on Foundations of Computer Science*. IEEE, 1977, pp. 46–57.

[106] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems: Specification*. springer, 1992, vol. 1.

[107] ——, *Temporal verification of reactive systems: safety*. Springer Verlag, 1995, vol. 2.

[108] O. Kupferman and M. Vardi, “Module checking,” in *Computer Aided Verification*. Springer, 1996, pp. 75–86.

[109] ——, “Module checking revisited,” in *Computer Aided Verification*. Springer, 1997, pp. 36–47.

[110] M. Vardi, “Verification of open systems,” in *Foundations of Software Technology and Theoretical Computer Science*. Springer, 1997, pp. 250–266.

[111] L. Lamport, “Solved problems, unsolved problems and non-problems in concurrency,” *(SIGOPS) Operating Systems Review*, vol. 19, no. 4, pp. 34–44, 1985.

[112] J. Burns, M. Gouda, and R. Miller, “Stabilization and pseudo-stabilization,” *Distributed Computing*, vol. 7, no. 1, pp. 35–42, 1993.

[113] ——, “On relaxing interleaving assumptions,” in *Proceedings of the MCC Workshop on Self-Stabilizing Systems, MCC Technical Report No. STP-379-89*, 1989.

[114] S. Huang, L. Wuu, and M. Tsai, “Distributed execution model for self-stabilizing systems,” in *Proceedings of the 14th International Conference on Distributed Computing Systems*. IEEE, 1994, pp. 432–439.

[115] M. Gradinariu and S. Tixeuil, “Conflict managers for self-stabilization without fairness assumption,” in *27th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2007, pp. 46–46.

[116] A. F. Babich, “Proving total correctness of parallel programs,” *IEEE Transactions on Software Engineering*, vol. SE-5, no. 6, pp. 558–574, Jun. 1979.

[117] M. Gouda, “Multiphase Stabilization,” *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 201–208, 2002.

[118] S. Katz and K. J. Perry, “Self-stabilizing extensions for message-passing systems,” *Distributed Computing*, vol. 7, no. 1, pp. 17–26, 1993.

[119] B. Awerbuch and G. Varghese, “Distributed program checking: a paradigm for building self-stabilizing distributed protocols,” *Proceedings of 31st Annual IEEE Symposium on Foundations of Computer Science*, pp. 258–267, 1991.

[120] B. Awerbuch, B. Patt-Shamir, and G. Varghese, “Self-stabilization by local checking and correction,” in *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science.* IEEE, 1991, pp. 268–277.

[121] B. Awerbuch, B. Patt-Shamir, G. Varghese, and S. Dolev, “Self-stabilization by local checking and global reset,” *Distributed Algorithms*, pp. 326–339, 1994.

[122] G. Varghese, “Self-stabilization by counter flushing,” in *The 13th Annual ACM Symposium on Principles of Distributed Computing*, 1994, pp. 244–253.

[123] E. Emerson and E. Clarke, “Using branching time temporal logic to synthesize synchronization skeletons,” *Science of Computer programming*, vol. 2, no. 3, pp. 241–266, 1982.

[124] S. Kripke, “Semantical analysis of modal logic I: Normal modal propositional calculi,” *Mathematical Logic Quarterly*, vol. 9, no. 5-6, pp. 67–96, 1963.

[125] Z. Manna and P. Wolper, “Synthesis of communicating processes from temporal logic specifications,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 6, no. 1, pp. 68–93, 1984.

[126] C. Hoare, “Communicating sequential processes,” *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[127] A. Ebnenasir, “Automatic synthesis of fault-tolerance,” Ph.D. dissertation, Michigan State University, May 2005.

[128] A. Ebnenasir, S. S. Kulkarni, and A. Arora, “FTSyn: A framework for automatic synthesis of fault-tolerance,” *International Journal on Software Tools for Technology Transfer*, vol. 10, no. 5, pp. 455–471, 2008.

[129] S. S. Kulkarni and A. Ebnenasir, “The complexity of adding failsafe fault-tolerance,” *International Conference on Distributed Computing Systems*, 2002.

[130] B. Bonakdarpour and S. S. Kulkarni, “Revising distributed UNITY programs is NP-complete,” in *12th International Conference on Principles of Distributed Systems (OPODIS)*, 2008, pp. 408–427.

[131] F. Abujarad and S. Kulkarni, “Multicore constraint-based automated stabilization,” *Stabilization, Safety, and Security of Distributed Systems*, pp. 47–61, 2009.

[132] S. Merz, “On the verification of a self-stabilizing algorithm,” Technical Report, University of Munich, Tech. Rep., 1998.

[133] L. Paulson, *Isabelle: A generic theorem prover*. Springer, 1994, vol. 828.

[134] S. Qadeer and N. Shankar, “Verifying a self-stabilizing mutual exclusion algorithm,” in *IFIP International Conference on Programming Concepts and Methods (PROCOMET98)*, 1998, pp. 424–443.

[135] S. Owre, J. Rushby, and N. Shankar, “PVS: A prototype verification system,” *Automated Deduction (CADE11)*, pp. 748–752, 1992.

[136] S. Kulkarni, J. Rushby, and N. Shankar, “A case-study in component-based mechanical verification of fault-tolerant programs,” in *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilizing Systems*. IEEE, 1999, pp. 33–40.

[137] S. Shukla, D. Rosenkrantz, and S. Ravi, “A simulation and validation tool for self-stabilizing protocols,” in *The SPIN verification system: the second Workshop on the SPIN Verification System: proceedings of a DIMACS workshop, August 5, 1996*, vol. 32. Amer Mathematical Society, 1997, p. 153.

[138] K. McMillan, *Symbolic model checking*. Kluwer Academic Publishers Norwell, MA, USA, 1993.

[139] N. Liveris, H. Zhou, R. Dick, and P. Banerjee, “State space abstraction for parameterized self-stabilizing embedded systems,” in *Proceedings of the 8th ACM international conference on Embedded software*. ACM, 2008, pp. 11–20.

[140] Y. Kesten and A. Pnueli, “Control and data abstraction: The cornerstones of practical formal verification,” *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2, no. 4, pp. 328–342, 2000.

[141] P. Wolper and V. Lovinfosse, “Verifying properties of large sets of processes with network invariants,” in *Automatic Verification Methods for Finite State Systems*. Springer, 1990, pp. 68–80.

[142] R. Kurshan and K. McMillan, “A structural induction theorem for processes,” in *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*. ACM, 1989, pp. 239–247.

[143] Y. Kesten, A. Pnueli, E. Shahar, and L. Zuck, “Network invariants in action,” *International Conference on Concurrency Theory (CONCUR)*, pp. 217–264, 2002.

[144] A. Pnueli, S. Ruah, and L. Zuck, “Automatic deductive verification with invisible invariants,” *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 82–97, 2001.

[145] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck, “Liveness with invisible ranking,” in *Verification, Model Checking, and Abstract Interpretation*. Springer, 2004, pp. 109–132.

[146] K. Namjoshi, “Symmetry and completeness in the analysis of parameterized systems,” in *Verification, Model Checking, and Abstract Interpretation*. Springer, 2007, pp. 299–313.

[147] Z. Shtadler and O. Grumberg, “Network grammars, communication behaviors and automatic verification,” in *Automatic Verification Methods for Finite State Systems*. Springer, 1990, pp. 151–165.

[148] E. Clarke, O. Grumberg, and S. Jha, “Verifying parameterized networks using abstraction and regular languages,” *CONCUR’95: Concurrency Theory*, pp. 395–407, 1995.

[149] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar, “Symbolic model checking with rich assertional languages,” in *Computer Aided Verification*. Springer, 1997, pp. 424–435.

[150] B. Jonsson and M. Nilsson, “Transitive closures of regular relations for verifying infinite-state systems,” *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 220–235, 2000.

[151] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili, “Regular model checking,” in *Computer Aided Verification*. Springer, 2000, pp. 403–418.

[152] P. Abdulla, G. Delzanno, N. Henda, and A. Rezine, “Regular model checking without transducers (on efficient verification of parameterized systems),” *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 721–736, 2007.

[153] P. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena, “A survey of regular model checking,” *CONCUR 2004–Concurrency Theory*, pp. 35–48, 2004.

[154] S. Leue, A. Ştefănescu, and W. Wei, “A livelock freedom analysis for infinite state asynchronous reactive systems,” *CONCUR 2006–Concurrency Theory*, pp. 79–94, 2006.

[155] ——, “Dependency analysis for control flow cycles in reactive communicating processes,” *Model Checking Software*, pp. 176–195, 2008.