Automated Synthesis of Timed and Distributed Fault-Tolerant Systems

by

Fathiyeh Faghihekhorasani

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2015

Abstract

This dissertation concentrates on the problem of automated synthesis and repair of fault-tolerant systems. In particular, given the required specification of the system, our goal is to synthesize a fault-tolerant system, or repair an existing one. We study this problem for two classes of timed and distributed systems.

In the context of timed systems, we focus on efficient synthesis of fault-tolerant timed models from their fault-intolerant version. Although the complexity of the synthesis problem is known to be polynomial time in the size of the time-abstract bisimulation of the input model, the state of the art lacked synthesis algorithms that can be efficiently implemented. This is in part due to the fact that synthesis is in general a challenging problem and its complexity is significantly magnified in the context of timed systems. We propose an algorithm that takes a timed automaton, a set of fault actions, and a set of safety and bounded-time response properties as input, and utilizes a space-efficient symbolic representation of the timed automaton (called the zone graph) to synthesize a fault-tolerant timed automaton as output. The output automaton satisfies strict phased recovery, where it is guaranteed that the output model behaves similarly to the input model in the absence of faults and in the presence of faults, fault recovery is achieved in two phases, each satisfying certain safety and timing constraints.

In the context of distributed systems, we study the problem of synthesizing fault-tolerant systems from their intolerant versions, when the number of processes is unknown. To synthesize a distributed fault-tolerant protocol that works for systems with any number of processes, we use counter abstraction. Using this abstraction, we deal with a finite-state abstract model to do the synthesis. Applying our proposed algorithm, we successfully synthesized a fault-tolerant distributed agreement protocol in the presence of Byzantine fault. Although the synthesis problem is known to be NP-complete in the state space of the input protocol (due to partial observability of processes) in the non-parameterized setting, our parameterized algorithm manages to synthesize a solution for a complex problem such as Byzantine agreement within less than two minutes.

A system may reach a bad state due to wrong initialization or fault occurrence. One of the well-known types of distributed fault-tolerant systems are self-stabilizing systems. These are the systems that converge to their legitimate states starting from any state, and if no fault occurs, stay in legitimate states thereafter. We propose an automated sound and complete method to synthesize self-stabilizing systems starting from the desired topology and type of the system. Our proposed method is based on SMT-solving, where the desired specification of the system is formulated as SMT constraints. We used the Alloy solver to implement our method, and successfully synthesized some of the well-known self-stabilizing algorithms. We extend our method to support a type of stabilizing algorithm called ideal-stabilization, and also the case when the set of legitimate states is not explicitly known.

Quantitative metrics such as recovery time are crucial in self-stabilizing systems when used in practice (such as in networking applications). One of these metrics is the average recovery time. Our automated method for synthesizing self-stabilizing systems generate some solution that respects the desired system specification, but it does not take into account any quantitative metrics. We study the problem of repairing self-stabilizing systems (where only removal of transitions is allowed) to satisfy quantitative limitations. The metric under study is average recovery time, which characterizes the performance of stabilizing programs. We show that the repair problem is NP-complete in the state space of the given system.

Acknowledgements

I would like to express my special appreciation and thanks to my supervisor Professor Dr. Borzoo Bonakdarpour, who has been a tremendous mentor for me. I would like to thank him for encouraging my research and guiding me during the course of my PhD. He is a brilliant professor with solid theoretical and practical knowledge. Without his advice and support, I cannot imagine my success in this journey. I also would like to thank my co-supervisor Professor Dr. Krzysztof Czarnecki for his support and dedication.

I would like to thank the members of my dissertation committee, Professors Mohamed Gouda, Richard Trefler, David Toman, and Derek Rayside, who accepted to review my thesis and provided me with valuable suggestions.

I have been very fortunate to work with members of Waterloo Formal Methods (WatForm) Lab. In particular, I acknowledge Shahram Esmaeilsabzali, Amirhossein Vakili, Alma Juarez-Dominguez, Zarrin Langari, and Vajih Montaghami for their support and encouragements. I would also like to use this opportunity to thank all my friends at the university of Waterloo; because of them my stay here has been very enjoyable and memorable.

Finally, and most importantly, I thank my parents, for their endless love and support, without which I would not have succeeded. They always encouraged me to work hard towards my goals and they did everything to help me in this path. My special thanks and sincere appreciation are extended to the love of my life, my husband, Vahid Pourahmadi. His love and understanding encouraged me to work hard and pursue my PhD despite all difficulties during this journey. Last, but not least, I am thankful to my son Ali for giving me happiness, energy, and motivation during the last year of my studies.

Dedication

To my parents:

Fatemeh Seyedabrishami and
Mohammad Faghihekhorasani

and

To my beloved husband and lovely son:

Vahid and Ali Pourahmadi

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Dependability is a vital property of many computing systems, especially, embedded safety/ mission-critical systems. Avizienis et al. define dependability by the following attributes [13]:

- Availability: readiness for correct service
- Reliability: continuity of correct service
- Safety: absence of catastrophic consequences on the user(s) and the environment
- Integrity: absence of improper system alteration
- Maintainability: ability for a process to undergo modifications and repairs.

Most systems are exposed to a set of faults, which can have significant impact on the system's dependability. We can consider fault as a defect in the system, which may or may not lead to a system failure (specification violation). For instance, an exception may be thrown, but the system can catch and handle that, so that the overall system execution respects the specification. There have been several mechanisms in the literature to attain the various attributes of dependability, which are categorized in [13] into four major groups of fault prevention, fault tolerance, fault removal, and fault forecasting. Our focus in this thesis is to automatically provide fault-tolerance, and we consider any fault that can be represented as a change of state in the system, which include different types of faults (e.g., stuck-at, crash, fail-stop, timing, performance, Byzantine, message loss, etc.), and different natures of the faults (permanent, transient, or intermittent) [17].

The challenge in designing fault-tolerant systems is that new faults may be introduced during the systems' lifecycle due to the change of environment or incomplete specification.

Designing the system from scratch to make it fault-tolerant for the new faults is a complex task. Manual revision of the system, on the other hand, needs a round of verification to ensure that the revised system is indeed fault-tolerant for the new faults, and also all properties satisfied by the original system, still hold in the revised one. If the verification does not pass, we should go back to revision, and we may need to repeat this cycle several times before the system is successfully revised. Therefore, having access to an automated method for system revision to provide fault-tolerance is highly desirable. Such a method should be correct by construction, meaning that new cycles of verification-revision is not required, and hence, the system can be easily revised considering new faults incrementally. We study the problem of automated revision to provide fault-tolerance for two important classes of systems; real-time and distributed systems.

Synthesis of fault-tolerant systems incrementally is not desirable in some applications, as they should be fault-tolerant to any possible fault that takes the system to an arbitrary state. Such systems are called *self-stabilizing* and are required in applications, such as networking or robotics. Automated synthesis of such systems from a set of requirements is a complex, but highly desirable task. This is due to the fact that an automated synthesis results in a system that is correct by construction, and verification is no longer required. Also, there are cases where there does not exist any system satisfying a set of requirements. Proof of non-existence for these cases is generally highly complex. Having access to a complete automated synthesis method, we can automatically reach these impossibility results, which will save a lot of time and effort from system designers. Automated synthesis of a program from a set of requirements is generally a highly complex and challenging problem due to the high time and space complexity of its decision procedures. Synthesizing self-stabilizing distributed protocols involves an additional level of complexity, due to constraints caused by distribution. Examples of such constraints include read-write restriction of processes in the shared-memory model, timing models, and symmetry. These constraints result in combinatorial blowups in the search space of corresponding synthesis problems. With this motivation, we study automated synthesis of self-stabilizing systems, starting from a set of requirements.

1.1 Synthesizing Timed Fault-Tolerant Systems

Our first research direction is to design an automated technique for synthesizing fault-tolerant timed systems from fault-intolerant systems. What makes this problem difficult is the conflicting nature of *time-predictability* (as required in real-time systems) and unanticipated time and type of faults. In particular, fault-tolerance requires that the system should eventually return to its ideal behavior, and its real-time nature needs the recovery to be timely. While satisfying both requirements may not be possible, incorporating *two-phase fault recovery* [17] enables the system to first recover to a safe or acceptable state quickly, and then return to its ideal behavior. Also, the intermediate state could be useful for other purposes, e.g., for logging. For instance, in a traffic signal controller, if the controller detects a fault, all signals should first turn red immediately to prevent catastrophic consequences (phase 1) before final recovery to normal behavior (phase 2).

The other issue when synthesizing real-time systems is their infinite semantic models because of the infinity of time. In this research, we propose a time and space-efficient algorithm that takes as input (1) a fault-intolerant program in terms of a timed automaton, (2) a set of faults, and (3) specification of phased recovery, and generates as output a fault-tolerant timed automaton that respects the phased recovery specification in the absence and presence of faults. Our technique utilizes an abstraction data structure called *zone graphs*.

1.2 Synthesizing Untimed Distributed Fault-Tolerant Systems

The second type of systems we consider is distributed systems. There are two sources of complexity when dealing with synthesizing distributed systems. First, the time and space complexity is an obvious issue when dealing with distributed systems. Secondly, we need to consider classes of transitions or, what we call, *group transitions* when designing distributed systems. These classes exist in distributed systems, since each process has only a partial view of the system. For example, in a system with set of variables $\{a, b\}$, consider a process that can only read the variable a. If this process has a transition that changes the value of a from 1 to 2, then this transition occurs independent of the value of b. Hence, this transition is, in fact, an equivalence class, where the value of a changes from 1 to 2 and the value of b can be anything in the domain of b as long as this value remains the same in the source and target of the transition. We call this equivalence class, a *group* of transitions, meaning that if one is removed (or added) from the system, then the entire group should be removed (or added) as well.

The problem of synthesis of fault-tolerant distributed systems is previously studied in [21]. In this dissertation, our focus is on *parameterized synthesis*, where a distributed fault-tolerant system has n processes, where n is unknown a priori. Our research goal is to propose an algorithm that takes as input (1) a *process template*, (2) a set of faults, and (3) a specification, and generates as output a process template, such that each system instantiated from that template respects the recovery specification in the absence and presence of faults. For tackling this problem, our idea is to use a type of *finite abstraction*, called *counter abstraction* [72], as the underlying data structure for synthesis. We also prove that our algorithm is sound, meaning that the resulting system is fault-tolerant when instantiated for any number of processes.

A special type of distributed fault-tolerant systems are *self-stabilizing* systems, which have two key features:

- *Strong convergence.* When a fault occurs in the system and, consequently, reaches some arbitrary state, the system is guaranteed to recover proper behavior within a finite number of execution steps.
- *Closure.* Once the system reaches such good behavior, typically specified in terms of a set of *legitimate states*, it remains in this set thereafter in the absence of new faults.

Self-stabilization has a wide range of application domains, including networking [34] and robotics [70]. The concept of self-stabilization was first introduced by Dijkstra in the seminal paper [29], where he proposed three solutions for designing self-stabilizing token circulation in ring topologies. Twelve years later, in a follow up article [30], he published the correctness proof, where he states that demonstrating the proof of correctness of self-stabilization was more complex than he originally anticipated. Indeed, designing correct self-stabilizing algorithms is a tedious and challenging task, prone to errors. Also, complications in designing self-stabilizing algorithms arise, when there is no commonly accessible data store for all processes, and the system state is based on the valuations of variables distributed among all processes [29]. Thus, it is highly desirable to have access to techniques that can automatically generate self-stabilizing protocols that are correct by construction.

With this motivation, we focus on the problem of automated synthesis of self-stabilizing protocols. Based on the input specification and the type of output program, there are various synthesis techniques. Our technique to synthesize self-stabilizing protocols takes as input the following specification:

1. A *topology* that specifies (1) a finite set V of variables allowed to be used in the protocol and their respective finite domains, (2) the number of processes, and (3) read-set and write-set of each process; i.e., subsets of V that each process is allowed to read and write.
2. A set of *legitimate states* in terms of a Boolean expression over V.

3. The *timing model*; i.e., whether the synthesized protocol is synchronous or asynchronous.

4. *Symmetry*; i.e., whether or not all processes should behave identically.

5. *Type of stabilization*; i.e., *strong* convergence guarantees finite-time recovery, while *weak* convergence guarantees only the possibility of recovery from any arbitrary state.

Our approach is, in particular, SMT[1]-based. That is, given the five above input constraints, we encode them as a set of SMT constrains. If the SMT instance is satisfiable, then a witness solution to its satisfiability is a distributed protocol that meets the input specification. If the instance is not satisfiable, then we are guaranteed that there is no protocol that satisfies the input specification. To the best of our knowledge, unlike the work in [21, 35], our approach, is the first *sound* and *complete* technique that synthesizes self-stabilizing algorithms. That is, our approach guarantees synthesizing a protocol that is correct by construction, if theoretically, there exists one, thanks to the power of existing constraint solvers. It allows synthesizing protocols with different combinations of timing models along with symmetry and types of stabilization. In order to demonstrate the effectiveness of our approach, we conduct a diverse set of case studies for automatically synthesizing well-known protocols from the literature of self-stabilization. These case studies include Dijkstra's token ring [29] (for both three and four state machines), maximal matching [68], weak stabilizing token circulation in anonymous networks [28], and the three coloring problem [45]. Given different input settings (i.e., in terms of the network topology, type of stabilization, symmetry, and timing model), we report and analyze the total time needed for synthesizing these protocols using the constraint solver Alloy [49].

There are cases, where developing a formal predicate for legitimate states (LS) is not at all a straightforward task. For instance, the set of legitimate states for Dijkstra's token ring algorithm with three-state machines [29] for three processes is the following [2]:

$$
\begin{aligned}
&(((x_0(s) + 1 \bmod 3 = x_1(s)) \;\wedge\; (x_1(s) + 1 \bmod\; 3 \neq x_2(s)))) \vee \\
&(((x_1(s) = x_0(s)) \;\wedge\; (x_1(s) + 1 \bmod 3 \neq x_2(s)))) \vee \\
&((x_1(s) + 1 \bmod 3 = x_0(s)) \;\wedge\; ((x_1(s) + 1 \bmod 3 \neq x_2(s)))) \vee \\
&((x_0(s) + 1 \bmod 3 \neq x_1(s)) \;\wedge\; (x_1(s) + 1 \bmod 3 \neq x_0(s)) \wedge (x_1(s) + 1 \bmod 3 = x_2(s)))
\end{aligned}
$$

Developing such a predicate requires huge expertise and insight. Ideally, the designer should use the basic requirements (unique token and circulation of it) to identify the

[1] *Satisfiability Modulo Theories* (SMT) are decision problems for formulas in first-order logic with equality combined with additional background theories such as arrays, bit-vectors, etc.

[2] Note that this is just an example to demonstrate the complexity of a predicate for LS.

desired system, instead of somehow magically producing a complex predicate such as the one above.

In our next study, we propose an automated approach to synthesize self-stabilizing systems given (1) the network topology, and (2) the high-level specification of legitimate states in a fragment of linear temporal logic (LTL). Furthermore, we explore automated synthesis of *ideal-stabilizing* protocols [69]. These protocols always satisfy their specification, i.e., all states are legitimate. They address two drawbacks of self-stabilizing protocols, namely exhibiting unpredictable behavior during recovery and poor compositional properties. In the case of self-stabilizing systems, we successfully synthesize Dijkstra's [29] token ring and Raymond's [74] mutual exclusion algorithms without legitimate states as input. We also synthesize ideal-stabilizing leader election and local mutual exclusion (in a line topology) protocols.

In out next work, we study the requirements on the performance of self-stabilizing systems. In other words, our previous work only focuses on synthesis of *some* solution that respects only closure and convergence. However, some quantitative metrics such as recovery time are as crucial as correctness in practice (e.g., in developing stabilizing network protocols). With this motivation, we study the problem of repairing existing weak/strong-stabilizing programs under performance constraints. The constraint under investigation is, in particular, *average recovery time*. This metric can be measured by giving weights to states and transitions of a stabilizing program and computing the expected value of the number of steps that it takes the program to reach a legitimate state. These weights can be assigned by a uniform distribution (in the simplest case), or by more sophisticated probability distributions. This technique has been shown to be effective in measuring the performance of weak-stabilizing programs as well, where not all computations converge [40], as well as cases where faults hit certain variables or locations more often. In this thesis, we show that the complexity of repairing an existing weak-stabilizing protocol to obtain either a weak or strong stabilizing protocol, so that (1) only removal of transitions is allowed during repair, and (2) the repaired protocol satisfies a certain average recovery time, is NP-complete.

In summary, our research problem is automated synthesis and repair of fault-tolerant systems. To evaluate our work, we implement our proposed algorithms for the synthesis of fault-tolerant systems in different settings and do experiments on several well-known case studies.

1.3 Organization of the Document

The rest of this document is organized as follows. In Chapter 2, we present our work towards automated synthesis of fault-tolerant real-time systems, and Chapter 3 presents our work on synthesis of self-stabilizing systems. We discuss our complexity results on repairing self-stabilizing systems in Chapter 4. In Chapter 5, we present our research on parametrized synthesis of fault-tolerant systems, and in Chapter 6, we discuss the work related to our research goals. We make concluding remarks, and present a summary of our future goals for this dissertation in Chapter 7.

Chapter 2

Synthesis of Fault-Tolerant Real-Time Systems

2.1 Introduction

Dependability and *time-predictability* are two vital properties of most embedded (especially, safety/mission-critical) systems. Consequently, providing *fault-tolerance* and meeting *timing constraints* are two inevitable aspects of dependable real-time embedded systems. However, these two features have conflicting natures; fault-tolerance deals with unanticipated time and type of faults, while meeting time constraints requires time predictability. This conflict makes design and analysis of fault-tolerant real-time systems a tedious and error-prone task. Hence, it is highly desirable to have access to automated techniques that can generate fault-tolerant models that meet their timing constraints and are correct by construction.

In many commonly considered systems, fault recovery has to be achieved in multiple (possibly ordered) *phases*, each satisfying certain constraints. In particular, fault-tolerance requires that the system should eventually return to its ideal behavior, and its real-time nature needs the recovery to be quick. While satisfying both requirements may not be possible, having two-phase fault recovery enables the system to first recover to a *safe* or acceptable state quickly, and then returns to its *ideal* behavior. Also, the intermediate state could be useful for other purposes, e.g., for logging.

In the context of synthesizing timed models with bounded-time phased fault recovery, assume that when a fault occurs, the system is required to reach a state, where Q holds in phase 1; and in phase 2, the system execution should get to a state where P is satisfied. In [17], the authors showed that if Q is not required to be closed in the execution of recovery transitions, then synthesizing a timed automaton [4] with 2-phase recovery is NP-complete in the size of the detailed region graph [4] of the input automaton[1]. On the contrary, if the closure of Q is required and, moreover, $P \subseteq Q$, then the synthesis problem can be solved in polynomial time. The polynomial-time algorithm presented in [17] to solve the latter problem is only an evidence for proving the problem complexity and is not an efficient practical solution with potential for implementation. This is simply because the size of a region graph grows incredibly huge even for small models. With this motivation, in this research, we propose a time- and space-efficient algorithm for synthesizing timed automata that provide 2-phase recovery, where Q is required to be closed and $P \subseteq Q$, while no new behaviors are added in the absence of faults.

2.2 Preliminaries

In this section, we present the preliminary concepts on timed automata and specifications.

2.2.1 Timed Automata with Deadlines (TAD) [4, 22]

In this research, we adopt the notion of timed automata [4] with deadlines (TAD) [22] extended by discrete variables.

2.2.1.1 Syntax

Let $X = \{x_1, x_2, ..., x_m\}$ be a finite set of *clock variables* that range over real numbers $\mathbb{R}_{\geq 0} \cup \{-1\}$ [2]. The value -1 identifies a *disabled* clock variable. The set Φ of all *clock constraints* over X is inductively defined as follows:

$$p ::= x \sim n \mid p \wedge p \mid \neg p$$

where n is a non-negative integer, and $\sim \in \{<, \leq, >, \geq\}$. Let V be a set of finite-domain *discrete variables*. We denote the set of all *guards* (Boolean expressions) over V by G_D.

Definition 1. *A* timed automaton with deadline *is a tuple* $TAD = (L, l_0, V, U, X, E)$, *where*

- L *is a finite set of* locations

[1]A detailed region graph is a finite bisimilar representation of a timed automaton.
[2]We are using continuous time to model variables that evolve over time.

- $l_0 \in L$ *is the* initial location
- V *is a finite set of* discrete variables
- U *is a finite set of* update functions
- X *is a finite set of clock variables and*
- $E \subseteq L \times U \times G_D \times \Phi \times \Phi \times 2^X \times 2^X \times L$ *is a finite set of timed* switches.

A timed switch is of the form $(l, u, g_d, g_c, d, (X_{res}, X_{dis}), l')$*, where* X_{res} *is a set of clocks to be reset,* X_{dis} *is a set of clock variables being disabled, such that* $X_{res} \cap X_{dis} = \{\}$*,* $g_c \in \Phi$ *is a clock constraint, and* $d \in \Phi$ *is the transition delay, such that* $d \Rightarrow g_c$*.* [3] □

In Definition 1, delay d determines the urgency of a switch. There are three different types of delays [22]. Intuitively, when $d = g_c$, the switch is called *eager*. An enabled eager switch cannot be delayed and, hence, does not let time progress before its execution. If $d = \mathit{false}$, then the switch is *lazy*, meaning that whenever it gets enabled, its execution can be delayed by letting time progress. This delay may even result in disabling the transition. In a *delayable* switch, d is the falling edge of a right-closed guard g_c; *i.e.,* whenever a delayable switch is enabled, its execution can be delayed as long as the associated guard remains true.

2.2.1.2 Semantics

In the following, we use val_d to denote a function that maps each $v \in V$ to a value in its finite domain Dom_v, and is called a *valuation* of discrete variables. Likewise, val_c denotes a *clock valuation*, which is a function that maps each clock variable $x \in X$ to a value in $\mathbb{R}_{\geq 0} \cup \{-1\}$. An *update function* $u \in U$, is a function $Dom_{v_1} \times \ldots \times Dom_{v_{|V|}} \rightarrow Dom_{v_1} \times \ldots \times Dom_{v_{|V|}}$ that maps each valuation val_d to a valuation val'_d. We denote the fact that a (clock or discrete) valuation val satisfies a guard g by $val \models g$. Each element of a tuple denoting a switch e is presented by the name of the element subscripted by e. For example, u_e denotes the update function of the switch e. The *semantic model* of a TAD is a tuple $\mathcal{SM} = (S, s_0, T)$, where

- S is the *state space* of the semantic model. Each *state* is a tuple (l, val_d, val_c), where $l \in L$ is a location, and val_d and val_c are discrete and clock valuations, respectively.

[3] Note that $\Rightarrow$ denotes the logical implication.

- $s_0 = (l_0, (val_d)_0, \overrightarrow{0})$ is the *initial state*, where l_0 is the initial location, $(val_d)_0$ is a valuation in which all discrete variables are initialized to some value in their domains, and $\overrightarrow{0}$ denotes the clock valuation with all clocks being set to zero.

- T is the set of *transitions* on S. In order to define T, we first identify the clock valuations from where time can progress from a location l and valuation val_d. Let E_l be the set of switches originating from l. We define $c(l, val_d)$ as the set of clock valuations:

$$c(l, val_d) = \{val_c \mid \neg \bigvee_{e \in E_l} ((val_c \models d_e) \ \wedge \ (val_d \models (g_d)_e))\}$$

and is called the *time progress condition* of location l and valuation val_d. For $\delta \in \mathbb{R}_{\geq 0}$, we write $val_c + \delta$ to denote $val_c(x) + \delta$ for every clock variable $x \in X$, if $x \neq -1$ (*i.e.*, time does not advance for disabled clocks). The set T of transitions in the semantic model is classified as follows:

Immediate Transitions: A transition $(l, val_d, val_c) \rightarrow (l', val'_d, val_c[X_{res}, X_{dis}])$ exists in T iff there exists a switch $(l, u, g_d, g_c, d, (X_{res}, X_{dis}), l') \in E$, such that $(val_c \models g_c) \wedge (val_d \models g_d)$, where $u(val_d) = val'_d$, and $val_c[X_{res}, X_{dis}]$ is the valuation val_c, where

  - for each $x \in X_{res}$, we have $val_c(x) = 0$
  - for each $x \in X_{dis}$, we have $val_c(x) = -1$
  - the value of other clock variables are unchanged.

The set of immediate transitions is denoted by T_{imm}.

Delay transitions: A transition $(l, val_d, val_c) \rightarrow (l, val_d, val_c + \delta)$ exists in T iff $\forall t < \delta : (val_c + t) \in c(l, val_d)$. The set of delay transitions in T is denoted by T_d.

2.2.1.3 Example

We use the following running example to describe the concepts throughout this chapter. Consider two processes that execute in mutual exclusion using a shared memory location. To coordinate, one of the processes is the master process (illustrated in Fig. 2.1). The automaton has three locations, execution (initial location), cleanup, and waiting, a clock variable x, and a discrete variable *token* shared between the processes. The clock constraint of switches are placed in [] and a switch delay is identified by {}.

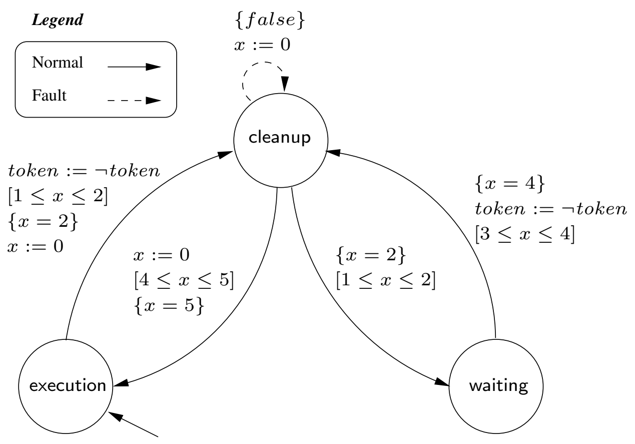


Figure 2.1: A timed automaton with deadline augmented with one fault switch

The master process stays in execution for 1 to 2 time units. Then, it resets x, toggles the value of *token*, and goes to cleanup, where it can spend another 1 to 2 time units for garbage collection. Changing the value of the shared variable allows the slave process (not shown here) to start execution. Then, the master process goes to location waiting, where it waits for the slave process execution to finish. When the value of x is between 3 to 4 time units, it again toggles the value of *token*, so that the slave process stops execution, and reaches location cleanup. In this location, the master process does the garbage collection for the slave, and also ensures that the slave process has noticed the change in *token*. The master process subsequently moves to location execution.

2.2.2 Specification

In this section, we present the notion of specification and what it means for a timed automaton to satisfy a specification.

Definition 2. *A* state predicate *SP of a semantic model $\mathcal{SM} = (S, s_0, T)$ is a subset of S, where in the corresponding Boolean expression, each clock variable is only compared with non-negative integers.* $\square$

In other words, a state predicate must be definable by the syntax of clock constraints as defined in Subsection 2.2.1.

Definition 3. *A* computation *of a semantic model $\mathcal{SM} = (S, s_0, T)$ is a finite or infinite sequence of states of the form: $\overline{s} = (s_0, \tau_0) \rightarrow (s_1, \tau_1) \rightarrow \ldots$ iff:*

- *for all $i \in \mathbb{Z}_{\geq 0}$: $(s_i, s_{i+1}) \in T$*
- *the sequence $\tau_0, \tau_1, \ldots$ (called the* global time*), satisfies the following conditions:*
  - monotonicity*: for all $i \in \mathbb{Z}_{\geq 0}, \tau_i \leq \tau_{i+1}$*
  - divergence*: if $\overline{s}$ is infinite, for all $t \in \mathbb{R}_{\geq 0}$, there exists $i \in \mathbb{Z}_{\geq 0}$, such that $\tau_i \geq t$*
  - consistency*: for all $i \in \mathbb{Z}_{\geq 0}$, if (s_i, s_{i+1}) is a delay transition in T, such that $s_i = (l, val_d, val_c)$, $s_{i+1} = (l, val_d, val_c + \delta)$, then $\tau_{i+1} - \tau_i = \delta$, and if (s_i, s_{i+1}) is an immediate transition in T, then $\tau_{i+1} = \tau_i$.* □

Definition 4. *A* specification *is a set of infinite computations that satisfy time-monotonicity and divergence [46].* □

Definition 5. *A state predicate SP is* closed *in a set of transitions T, iff*

- *if an immediate transition in T originates from SP, it terminates in SP*
- *if a delay transition in T with duration δ originates in state $s \in SP$, then for all $\delta' \leq \delta$, a delay transition with duration δ' that starts in s also terminates in a state in SP.* □

Definition 6. *Let TAD be a timed automaton with semantic model $\mathcal{SM} = (S, s_0, T)$, $SPEC$ be a specification, and SP be a state predicate of TAD. We write $TAD \models_{SP} SPEC$ (read TAD* satisfies *$SPEC$* from *SP), iff (1) SP is closed in T, and (2) every computation of TAD that starts from SP is in $SPEC$.* □

The reason for defining satisfaction 'from' a state predicate is due to the fact that when we add fault transitions to a model, the closure of its normal behavior is not ensured. This notion of normal behavior is captured by a state predicate called the set of *legitimate states* defined next.

Definition 7. *Let TAD be a timed automaton and LS be a nonempty state predicate of TAD. We say that LS is a set of* legitimate states *of TAD iff $TAD \models_{LS} SPEC$.* □

Definition 8. *Let P and Q be state predicates and $\delta \in \mathbb{R}_{\geq 0}$. A* bounded response *property is of the form $P \mapsto_{\leq \delta} Q$, and defines computations $\overline{s} = (s_0, \tau_0) \rightarrow (s_1, \tau_1) \rightarrow \ldots$, where for all $i \geq 0$, if $s_i \in P$, then there exists $j \geq i$, such that $s_j \in Q$ and $\tau_j - \tau_i \leq \delta$.* □

In this chapter, our notion of specification consists of two parts: (1) a *safety specification*, and (2) a *liveness specification* [3, 46]. Roughly speaking, our notion of safety is characterized by a set of unsafe timing independent transitions and a set of bounded-time response properties.

Definition 9. *A* safety specification *consists of two parts:*

1. Timing-independent Safety*: Specified by a set of immediate* bad transitions bt. *The specification in which each computation has no bad transitions is denoted by* $SPEC_{\overline{bt}}$.
2. Timing Constraint*: Denoted by* $SPEC_{\overline{br}}$ *is the conjunction* $\bigwedge_{i=1}^{m}(P_i \mapsto_{\leq \delta_i} Q_i)$. $\square$

A bad transition that can be specified by its target state only defines a set of *bad states*. In the context of our example, a state in which $token = 1$ and the model is in location $\mathsf{execution}$ is a bad state. Note that it is not always the case that bad transitions can be identified by bad states. For example, in traffic signal controller, a bad transition can be a transition originating from a state where the signal is initially red and becomes yellow in the target state.

Definition 10. *A* liveness specification $SPEC$ *is a set of computations with this condition: for each finite computation* $\overline{\alpha}$, *there exists a nonempty suffix* $\overline{\beta}$, *such that* $\overline{\alpha}\overline{\beta} \in SPEC$. $\square$

Following [3] and [46], liveness specification is included in all specifications and, hence, it is not repeated in the specification representation.

Example. Consider the timed automaton in Fig. 2.1. The timing independent safety specification for mutual exclusion between the two processes is characterized by:

$$bt = \{(s_0, s_1) \mid s_1 \models (\mathsf{execution} \wedge (token = 1))\}$$

which requires the master process not to be in location $\mathsf{execution}$, when the value of $token$ is 1. The set of legitimate states of this example is specified using the following expression:

$$\begin{aligned} LS \equiv\ & ((\mathsf{execution}) \Rightarrow ((x \leq 2) \wedge (token = 0))) \wedge \\ & ((\mathsf{cleanup}) \quad \Rightarrow (((x \leq 2) \wedge (token = 1)) \vee \\ & \qquad\qquad\qquad ((3 \leq x \leq 5) \wedge (token = 0))) \wedge \\ & ((\mathsf{waiting}) \quad \Rightarrow ((1 \leq x \leq 4) \wedge (token = 1)) \end{aligned}$$

It is straightforward to see that starting from any state in LS, execution of *normal* switches of the automaton in Fig. 2.1 results in a state in LS and a transition in $SPEC_{bt}$ will never execute.

2.3 Timed Automata with Faults and Strict 2-Phase Fault Recovery

In this section, we present the notions of faults and strict 2-phase fault recovery [17].

2.3.1 Fault Model

A *fault* is systematically represented as a transition. Fault representation with a transition is possible for different types of faults (e.g., stuck-at, crash, fail-stop, timing, performance, Byzantine, message loss, etc.), nature of the faults (permanent, transient, or intermittent), or the ability of the program to observe the effects of the faults [17].

Given a semantic model $\mathcal{SM} = (S, s_0, T)$, a set F of faults is a subset of all possible immediate transitions[4]. In other words, $F \subseteq (S \times S)_{imm}$, where

$$\begin{aligned}(S \times S)_{imm} = \{&(l, val_d, val_c) \to (l', val'_d, val_c[X_{res}, X_{dis}]) \mid \\ &(l, val_d, val_c), (l', val'_d, val_c[X_{res}, X_{dis}]) \in S \wedge X_{dis} = \emptyset\}\end{aligned}$$

Similar to the notion of legitimate states for a timed automaton in the absence of faults, we introduce the notion of *fault-span* to reason about the behavior of a timed automaton in the presence of faults.

Definition 11. *For a semantic model $\mathcal{SM} = (S, s_0, T)$, legitimate states LS, and a set F of faults, a state predicate FS is a* fault-span *or* F-span *of the model $\mathcal{SM}$ from LS iff (1) $LS \subseteq FS$, and (2) FS is closed in $T \cup F$.* □

Hence, a fault-span is a state predicate up to which (but not beyond which) faults can perturb the state of a system. In order to distinguish the transitions/switches defined in the given timed automaton and faults, in the remainder of this chapter, we call the former *normal* transitions/switches.

Example. In Figure 2.1, the fault switch introduced in location cleanup, resets clock variable x at any time. Notice that if x gets reset when $x \leq 2$, then this fault starts and ends within the legitimate states. However, if $3 \leq x \leq 5$ and x gets reset, then the fault leads the execution to a state outside the legitimate states. The delay of the fault switch is set to lazy, since it does not impose any constraints on time progress. Observe that, if a computation starts from a state in LS where $3 \leq x \leq 5$ and $token = 0$, when the fault occurs, after 1 to 2 time units, the computation goes to waiting and subsequently to cleanup where $token$ gets toggled (with value 1). The next transition of the computation is a bad transition, as the model goes to execution location, while $token = 1$. This clearly violates the safety specification.

[4]We note that while delay faults cannot be modeled explicitly due to the semantics of TADs, one can specify a delay fault by employing an additional location, where the delay occurs.

2.3.2 Strict 2-phase Fault Recovery

Intuitively, in *strict 2-phase recovery* [17], when the state of a system is perturbed by faults, the system is required to either directly return to its legitimate states LS within $\theta \in \mathbb{Z}_{\geq 0}$ time units, or, if direct recovery is not feasible, then it should first reach an *intermediate* recovery predicate Q within $\theta \in \mathbb{Z}_{\geq 0}$ (*i.e.,* phase 1), from where the system reaches LS within $\delta \in \mathbb{Z}_{\geq 0}$ time units (*i.e.,* phase 2).

Definition 12. *Let $\mathcal{SM} = (S, s_0, T)$ be the semantic model of a timed automaton with legitimate states LS, Q be a state predicate called* intermediate recovery predicate*, F be a set of faults, $SPEC$ be a specification, and $\theta, \delta \in \mathbb{Z}_{\geq 0}$. The* strict 2-phase recovery *specification for $\mathcal{SM}$ is $SPEC_{\overline{br}} = (\neg LS \mapsto_{\leq \theta} Q) \;\wedge\; (Q \mapsto_{\leq \delta} LS)$.* □

The other types of 2-phase recovery that are outside the scope of this research are specified by different $SPEC_{\overline{br}}$ [17]. For example, ordered-strict recovery is specified by $SPEC_{\overline{br}} = (\neg LS \mapsto_{\leq \theta} (Q - LS)) \wedge (Q \mapsto_{\leq \delta} LS)$. In order to define the notion of fault-tolerance using 2-phase recovery, we first characterize a notion where computations that can be produced in the presence of faults can be extended, such that they eventually meet the specification.

Definition 13. *A timed automaton TAD with semantic model $\mathcal{SM} = (S, s_0, T)$* maintains *$SPEC$ from state predicate SP iff*

- *SP is closed in T, and*
- *for every computation prefix $\overline{\alpha}$ of $\mathcal{SM}$ that starts in SP, there exists a computation suffix $\overline{\beta}$, such that $\overline{\alpha\beta} \in SPEC$.*

We say that TAD violates *$SPEC$ from SP iff it is not the case that TAD* maintains *$SPEC$ from SP.* □

Concerning Definitions 6 and 13, we note that if a timed automaton satisfies $SPEC$ from SP, then it maintains $SPEC$ from SP as well. However, the reverse direction does not always hold. Definition 13 is introduced for computations that TAD cannot produce, but can be extended to a computation in $SPEC$ by adding *recovery* computation suffixes.

Definition 14. *An automaton TAD with semantic model $\mathcal{SM} = (S, s_0, T)$ is F*-tolerant *to $SPEC$ from LS iff*

1. *$TAD \models_{LS} SPEC$,*

2. there exists an F-span FS of TAD from LS, st.

- *$(S, s_0, T \cup F)$ maintains $SPEC$ from FS, where $SPEC_{\overline{br}}$ is as defined in Definition 12,*
- *$(S, s_0, T \cup F)$ satisfies $FS \mapsto_{<\infty} LS$ from FS.* $\square$

The last condition is added to handle the case where response properties in $SPEC_{\overline{br}}$ are unbounded (since in this case, Definition 13 fails, as it only captures finite prefixes).

Example. Let Q be the set of states in which the automaton stays in waiting long enough to ensure that nothing bad happens; *i.e.*, $Q \equiv (\mathsf{waiting} \ \wedge \ (x \geq 5))$. The timing-independent safety property for this automaton in defined in Subsection 2.2.2. The timing constraint is defined as follows:

$$SPEC_{\overline{br}} = (\neg LS \mapsto_{\leq 6} Q) \ \wedge \ (Q \mapsto_{\leq 2} LS).$$

where the response times are chosen arbitrarily. The property $SPEC$ is the union of $SPEC_{bt}$ and $SPEC_{\overline{br}}$.

2.4 Problem Statement

Given are a fault-intolerant timed automaton TAD with semantic model $\mathcal{SM} = (S, s_0, T)$ and legitimates states LS, a set F of faults, and specification $SPEC$, such that $TAD \models_{LS} SPEC$. Our goal is to develop an algorithm for synthesizing an automaton TAD' with semantic model $\mathcal{SM}' = (S', s_0, T')$ and legitimate states LS' from TAD, such that TAD' is F-tolerant to $SPEC$ from LS'. We require that the algorithm for adding fault tolerance does not introduce new behaviors to TAD in the absence of faults. To this end, we define the notion of *projection*. Intuitively, the projection of transitions T on state predicate SP includes all immediate transitions that start and end in SP, and the delay transitions that start in SP and remain in SP continuously.

Definition 15. *The* projection *of a set T of transitions on a state predicate SP is defined as follows:*

$$\begin{aligned} T \mid SP = \{&(s_0, s_1) \in T_{imm} \mid s_0, s_1 \in SP\} \cup \\ \{&(l, val_d, val_c) \to (l, val_d, val_c + \delta) \in T_d \mid \\ &((l, val_d, val_c) \in SP) \ \wedge \ (\forall \epsilon \in \mathbb{R}_{\geq 0} : ((\epsilon \leq \delta) \Rightarrow \\ &\qquad (l, val_d, val_c + \epsilon) \in SP))\} \quad \square \end{aligned}$$

Recall that T_{imm} and T_d are the sets of immediate and delay transitions in T, respectively. Using this definition, we clarify our requirement of not adding new behavior to TAD in the absence of faults. If LS' contains a state that is not included in LS, then TAD' may have a computation that reaches a state that is not reachable in TAD in the absence of faults. This may falsify $TAD' \models_{LS'} SPEC$ and, hence, we require $LS' \subseteq LS$. Likewise, if $T' \mid LS'$ contains a transition that is not included in $T \mid LS'$, then there may exist a computation in the synthesized model that is not in the original model in the absence of faults. Hence, we also require $(T' \mid LS') \subseteq (T \mid LS')$.

We assume there exists a clock for each bounded response property. The clock is needed to measure time when the first predicate in the property becomes true. Also, for simplicity and without loss of generality, we assume when a fault occurs, no fault happens until the system goes back to LS'. In [19], the authors present an algorithm based on region graph that can deal with the case where faults occur in the fault-span as well.

Problem statement. Given a fault-intolerant timed automaton TAD with semantic model $\mathcal{SM} = (S, s_0, T)$, a set F of faults, intermediate predicate Q, where $LS \subseteq Q$, and specification $SPEC$, such that $TAD \models_{LS} SPEC$, our goal is to propose an algorithm for synthesizing an automaton TAD' with $\mathcal{SM}' = (S', s'_0, T')$, and legitimate states LS' from TAD, such that:

1. $LS' \subseteq LS$,
2. Q is closed in T',
3. $(T' \mid LS') \subseteq (T \mid LS')$, and
4. TAD' is F-tolerant to $SPEC$ from LS'.

The constraint on closure of Q and $LS \subseteq Q$ are included, because otherwise the problem becomes NP-complete [17] in the size of time-abstract bisimulation of TAD. In this research, our focus is on devising a zone-based algorithm for the case where the problem can be solved in polynomial time in the size of time-abstract bisimulation of TAD. Througout this chapter, we refer to the input program as the *fault-intolerant* timed automaton.

2.5 The Synthesis Algorithm

In this section, we present our zone-based algorithm for solving the problem of synthesizing a fault-tolerant TAD' from a given TAD as stated in Section 4.3.

2.5.1 Zone Graphs

Since the state space of a timed automaton is infinite, in order to formally analyze a timed automaton, we use an equivalent space-efficient finite symbolic transition system, called a *zone graph* [32]. A *clock zone* ξ is inductively defined as $\xi ::= x \preceq n \mid x - y \preceq n \mid \xi \wedge \xi$, where x and y are clock variables, n is a constant integer, and $\preceq \in \{<, \leq\}$. As an example, Fig. 2.2 represents the clock valuations encoded in the zone ξ, where we have two clock variables x and y.

Let ξ be a clock zone on the set of m clock variables and $[\![\xi]\!] = \{\_val_c \in \mathbb{R}^m_{\geq 0} \mid \_val_c \models \xi\}$. The operators *up* and *resdis* are defined on clock zones as follows:

- $up(\xi) = \{val_c + \delta \mid val_c \in [\![\xi]\!] \ \wedge \ \delta \in \mathbb{R}_{\geq 0}\}$
- $resdis(\xi, (X_{res}, X_{dis})) = \{val_c[X_{res}, X_{dis}] \mid val_c \in [\![\xi]\!]\}$

Observe that operator *up* has no effect on disabled clock variable. A *zone* z is a tuple $z = \langle l, \_val_d, [\![\xi]\!]\rangle$, where l is a location, $\_val_d$ is a valuation of discrete variables, and ξ is a clock zone.

Definition 16. *Let $TAD = (L, l_0, V, U, X, E)$ be a timed automaton. The* zone graph *of TAD is defined as a transition system $\mathcal{Z}(TAD) = (Z, z_0, \rightsquigarrow)$, where*

- *Z is the set of zones defined on TAD*
- *$z_0 = \langle l_0, (\_val_d)_0, up(\overrightarrow{0}) \ \cap \ c(l_0, (\_val_d)_0)\rangle$*
- *$\rightsquigarrow$ is the relation defined on zones by: $\langle l, \_val_d, \xi\rangle \rightsquigarrow \langle l', \_val'_d, \xi'\rangle$, if there exists $(l, u, g_d, g_c, d, (X_{res}, X_{dis}), l') \in E$, such that $\_val_d \models g_d$, $u(\_val_d) = \_val'_d$, and $\xi' = up(resdis(\xi \wedge g_c, (X_{res}, X_{dis})) \ \cap \ c(l', \_val'_d)$.* □

Example. Fig. 2.3 shows the zone graph of the automaton in Fig. 2.1.

We use the following zone operators [16, 24] in our algorithm:

- *and*(ξ_1, ξ_2) returns the conjunction of the constraints in ξ_1 and ξ_2.
- *down*(ξ) returns the weakest precondition of ξ with respect to delay, which is the set of clock assignments that can reach ξ by some delay δ:

$$down(\xi) = \{\_val_c \mid \_val_c + \delta \in \xi \ \wedge \ \delta \in \mathbb{R}_{\geq 0}\}$$

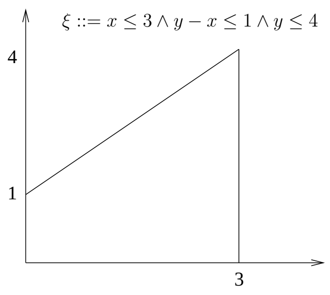


Figure 2.2: An example of a zone

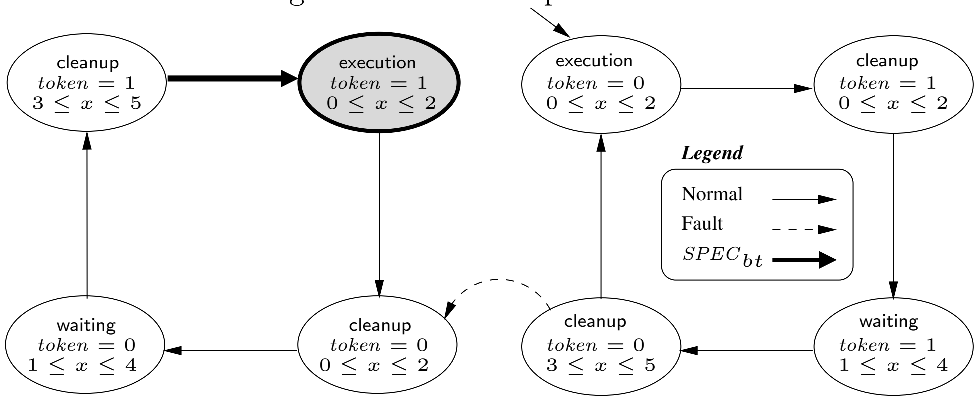

cleanup $token = 1$ $3 \leq x \leq 5$
execution $token = 1$ $0 \leq x \leq 2$
execution $token = 0$ $0 \leq x \leq 2$
cleanup $token = 1$ $0 \leq x \leq 2$
Legend
Normal
Fault
$SPEC_{bt}$
waiting $token = 0$ $1 \leq x \leq 4$
cleanup $token = 0$ $0 \leq x \leq 2$
cleanup $token = 0$ $3 \leq x \leq 5$
waiting $token = 1$ $1 \leq x \leq 4$

Figure 2.3: Zone graph of the timed automaton in Fig. 2.1

- $\mathit{free}(\xi, x)$ removes all constraints on the clock x:

$$\mathit{free}(\xi, x) = \{\_val_c[x = \delta] \mid \_val_c \in \xi \ \wedge \ \delta \in \mathbb{R}_{\geq 0}\}$$

- $\mathit{Pred}_e(\xi)$ computes the set of clock valuations that after some delay δ can take switch e, and reach ξ, and is formally defined as

$$\begin{aligned}\mathit{Pred}_e(\xi) = \{ \ &\_val_c \mid (\_val_c + \delta) \models g_c \wedge \\ &(\_val_c + \delta)[X_{res}, X_{dis}] \in \xi \wedge \\ &e = (l, u, g_d, g_c, d, (X_{res}, X_{dis})) \wedge \\ &\delta \in \mathbb{R}_{\geq 0}\}.\end{aligned}$$

2.5.2 Algorithm Sketch

Our zone-based algorithm consists of the following steps (Algorithm 1):

1. *Automaton enhancement:* The input model is enhanced with a new location ("sink"), and a number of switches entering it, which prune computations that violate the given specification. As a result, the corresponding zone graph will be more efficient. Also, the model is augmented with two clocks, and delay transitions that can be utilized for adding 2-phase recovery within specific delays.

2. *Zone graph generation:* Next, the zone graph of the enhanced input automaton is generated. We utilize an existing algorithm from the literature of verification for this step.

3. *Adding recovery behavior:* To enable 2-phase recovery, we add possible transitions among the zones of the zone graph. In this step, new zones may be added to the zone graph.

4. *Backward zone generation:* For the newly added zones in the last step, we identify the backward reachable zones to ensure that the new zones do not introduce terminating computations.

5. *Cycle removal:* Since adding recovery transitions may create cycles, the algorithm removes the possible cycles to ensure correct recovery.

6. *Zone graph repair:* The zone graph is modified, so that it satisfies the safety property in the presence of faults, and also does not contain any deadlock states.

Finally, one can generate an automaton from the repaired zone graph. We consider this step as a black box, which gets a zone graph and returns a timed automaton corresponding to that semantic model.

2.5.3 Algorithm Description

The main algorithm (Algorithm 1) takes a timed automaton TAD, with legitimate states LS, fault transitions F, and intermediate recovery predicate Q such that $LS \subseteq Q$ as input. The specification consists of the timing-independent safety specification (the set BT of bad transitions) and timing constraints (as the recovery time δ and intermediate recovery time θ).

2.5.3.1 Steps 1, 2: Automaton Enhancement / Zone Graph Generation

Algorithm Zone_based_Synthesis starts by automaton invoking function Enhance_Automaton (see Function 2). The entire $\neg LS$ is (often) too large and impractical to build and explore.

Hence, function Enhance_Automaton uses a heuristic to build a weak enough fault-span (rather than considering the entire $\neg LS$), such that we generate the zones only reachable using (1) the program switches, and (2) any possible delay, when the state of the model is in $\neg Q$. We exclude $Q - LS$, since adding delay transitions may violate the closure of Q. The clocks x_f and x_q are added to keep track of the time elapsed since a computation reaches $\neg LS$ and Q, respectively (Line 1). A new location, called sink (Line 2), along with the added switches leading to sink are used to prune the computations violating the specification.

Algorithm 1 Zone_based_Synthesis

Input: A timed automaton TAD, with legitimate states LS, fault switches F, bad transitions BT, intermediate recovery predicate Q st. $LS \subseteq Q$, recovery and intermediate recovery times δ and θ.
Output: If successful, a fault-tolerant TAD' with legitimate states LS'.

```
1: $TAD'' \leftarrow$ Enhance_Automaton($TAD, LS, F, Q, \delta, \theta, BT$)
2: $(Z, z_0, \rightsquigarrow) \rightarrow$ Construct_Zone_Graph ($TAD''$)
3: $(Z', z'_0, \rightsquigarrow), waiting \leftarrow$ Add_Trans($(Z, z_0, \rightsquigarrow), BT$)
4: $(Z', z'_0, \rightsquigarrow) \leftarrow$ Backward_Zones($(Z', z'_0, \rightsquigarrow'), \rightsquigarrow, waiting$)
5: $(Z', z'_0, \rightsquigarrow) \leftarrow$ Cycle_Removal ($Z', z'_0, \rightsquigarrow'$)
6: $nz \leftarrow \{z_0 \mid \exists z_1, z_2 \dots z_n \cdot (\forall j \mid 0 \leq j < n : (z_j, z_{j+1}) \in F'^z) \wedge (z_{n-1}, z_n) \in BT'^z\}$;
7: $Z_1 \leftarrow Z' - nz$
8: $LS_1^z \leftarrow LS'^z - nz$
9: $mz \leftarrow \{(z_0, z_1) \mid (z_1 \in nz) \vee (z_0, z_1) \in BT'^z\}$;
10: $\rightsquigarrow \leftarrow \rightsquigarrow - mz$
11: repeat
12:     $Z_2, LS_2^z \leftarrow Z_1, LS_1^z$;
13:     $nz \leftarrow \{z_0 \mid \nexists z_1 : (z_0, z_1) \in \rightsquigarrow\}$;
14:     $Z_1 \leftarrow Z_1 - nz$;
15:     $LS_1^z \leftarrow LS_1^z - nz$;
16:     $mz \leftarrow \{(z_0, z_1) \mid z_1 \in nz\}$;
17:     $\rightsquigarrow \leftarrow \rightsquigarrow - mz$;
18:     $nz' \leftarrow \{z_0 \mid (z_0, z_1) \in mz \cap F^z\}$;
19:     $Z_1 \leftarrow Z_1 - nz'$;
20:     $LS_1^z \leftarrow LS_1^z - nz'$;
21:     if ($Z_1 = \emptyset \vee LS_1^z = \emptyset$) then
            print "no fault-tolerant program found";exit;
22:     end if
23: until ($Z_1 = Z_2 \wedge LS_1^z = LS_2^z$)
24: $TAD' \leftarrow Construct\_Automaton((Z_1, z_0, \rightsquigarrow), LS_1^z)$
25: return $TAD'$
```

The first set of pruned computations are those violating timing-independent safety specification in terms of bad states BS (Line 3). Computations reachable from a bad state can be pruned, and, hence, eager switches E_0 are used not to let time progress after we reach a bad state. The second set of states that can be used to prune the zone graph are the ones that violate timing constraints of 2-phase recovery:

- A computation cannot stay in $\neg LS - Q$ for more than θ time units. Hence, the set E_1 of switches are added to ensure that every computation that stays more than θ time units in $\neg LS - Q$ will be pruned (Line 4). Note that switches in E_1 are eager.

Function 2 Enhance_Automaton

Input: A timed automaton $TAD = (L, l_0, V, U, X, E)$, with legitimate states LS, fault switches F, intermediate recovery predicate Q, recovery time δ, intermediate recovery time θ, and bad transitions BT
Output: An enhanced automaton $TAD' = (L', l_0, V, U, X', E')$

```
1: $X' \leftarrow X \cup \{x_f, x_q\}$
2: $L' \leftarrow L \cup \{\mathsf{sink}\}$
3: $E_0 \leftarrow \{(l, \mathfrak{u}, g_d, true, true, (\emptyset, X'), \mathsf{sink}) \mid \forall\_val_d \models g_d : (l, val_d) \in BS\}$
4: $E_1 \leftarrow \{(l, \mathfrak{u}, true, x_f = \theta, x_f = \theta, (\emptyset, X'), \mathsf{sink}) \mid l \in L\}$
5: $E_2 \leftarrow \{(l, \mathfrak{u}, true, x_q = \delta, x_q = \delta, (\emptyset, X'), \mathsf{sink}) \mid l \in L\}$
6: $F' \leftarrow \{(l_1, u, g_d, \phi, false, (r_1 \cup x_f, r_2), l_2) \mid \forall(l_1, u, g_d, \phi, false, (r_1, r_2), l_2) \in F\}$
7: $E_3 \leftarrow \{(l, u, g_d, \phi \wedge x_f \geq 0, true, (x_q, x_f), l) \mid \forall\_val_d \models g_d : \forall\_val_c \models \phi : (l, val_d, val_c) \in Q - LS\}$
8: $E_4 \leftarrow (l_1, u, g_d, g_c \wedge (x_f < 0), d, (r_1, r_2), l_2) \mid \forall(l_1, u, g_d, g_c, d, (r_1, r_2), l_2) \in E\}$
9: $E_5 \leftarrow (l_1, u, g_d, g_c \wedge (x_f \geq 0), false, (r_1, r_2), l_2) \mid \forall(l_1, u, g_d, g_c, d, (r_1, r_2), l_2) \in E\}$
10: $E' \leftarrow E \cup F' \cup \bigcup_{i=0}^{5} E_i$
11: return $TAD' = (L', l_0, V, U, X', E')$
```

- Similarly, we respect the recovery time δ by adding the switches in E_2, which do not let time progress when the value of $x_q = \delta$ (Line 5).

Note that all added switches to the sink location disable all clocks. Also, a unique update function $\mathfrak{u}$ is used to set the value of discrete variables. This is done to avoid having multiple sink states with different clock valuations or discrete variables valuations in the semantic model.

The set F' of switches (Line 6) corresponds to the set F of faults, where the urgency is set to *false* (as the fault transitions may not be taken in the computation), and with the clock c_f being added to the set of clocks to be reset. E_3 are eager switches that are triggered as soon as a state in $Q - LS$ is reached, where x_f is disabled and x_q is reset (Line 7). E_4 and E_5 are added, so that the switches of the program are lazy when the computation is not in Q, while they have the specified urgency when the computation in Q. This way, we allow any possible delay in $\neg Q$ for generating the weak enough fault-span (Lines 8 and 9).

Function 3 Add-Trans

Input: A zone graph $(Z, z_0, \rightsquigarrow)$, a set of legitimate zones LS^z, intermediate recovery zones Q^z, a set of bad transitions BT^z
Output: A zone graph $(Z', z'_0, \rightsquigarrow')$, with recovery transitions being added, and a set of new subzones *waiting*

```
 1: $waiting \leftarrow \emptyset$
 2: $Z' \leftarrow Z$
 3: $\rightsquigarrow' \leftarrow \rightsquigarrow$
 4: FindZonesRanking $(Z, z_0, \rightsquigarrow)$
 5: ConnectZones($Z - Q^z$, $Z$)
 6: ConnectZones($Q^z - LS^z$, $Q^z$)
 7: ConnectZonesRes($Z - Q^z$, $Z$)
 8: ConnectZonesRes($Q^z - LS^z$, $Q^z$)
 9: return $(Z', z'_0, \rightsquigarrow'), waiting$
10: function ConnectZones($Z_1, Z_2$: Set of zones){
11: for all $z \in Z_1$, $z' \in Z_2$ st. $(z, z') \notin (\rightsquigarrow \cup BT^z)$ do
12:     if $(rank(z) < \infty)$ break
13:     Let $z = (l, (\_val_d), \xi)$ and $z' = (l', (\_val'_d), \xi')$
14:     $\xi'' \leftarrow \mathfrak{to}(\xi, \xi')$
15:     if $(\xi'' = \emptyset)$ continue
16:     $con(z) = 1$
17:     if $(\xi'' = \xi)$ then
18:         $rank(z) = rank(z') + 1$
19:         $\rightsquigarrow' \leftarrow \rightsquigarrow' \cup \{(z, z')\}$
20:     else
21:         $z'' \leftarrow (l, (\_val_d), \xi'')$
22:         $waiting \leftarrow waiting \cup \{(z'', z)\}$
23:         $Z' \leftarrow Z' \cup z''$
24:         $\rightsquigarrow' \leftarrow \rightsquigarrow' \cup (z'', z')$
25:     end if
26: end for}
27: operator $\mathfrak{to}(\xi_1, \xi_2$: Clock zone) {
28: for all $x \in X$ do
29:     if $ub(\xi_1, x) < lb(\xi_2, x)$ then
30:         return $\emptyset$
31:     end if
32: end for
33: return $and\,(\xi_1, down(\xi_2))$ }
34: function ConnectZonesRes($Z_1, Z_2$: Set of zones){
35: for all $z \in Z_1$ st. $\neg con(z) \wedge loc(z) \neq \mathsf{sink} \wedge \nexists z''' : (z, z''') \in \rightsquigarrow'$ , $z' \in Z_2$ st. $(z, z') \notin BT^z$ do
36:     if $(rank(z) < \infty)$ break
37:     Let $z = (l, (\_val_d), \xi)$ and $z' = (l', (\_val'_d), \xi')$
38:     $\xi'' \leftarrow \mathfrak{tores}(\xi, \xi')$
39:     if $(\xi'' = \emptyset)$ continue
40:     if $(\xi'' = \xi)$ then
41:         $rank(z) = rank(z') + 1$
42:         $\rightsquigarrow' \leftarrow \rightsquigarrow \cup (z, z')$
43:     else
44:         $z'' \leftarrow (l, (\_val_d), \xi'')$
45:         $waiting \leftarrow waiting \cup \{(z'', z)\}$
46:         $Z' = Z' \cup z''$
47:         $\rightsquigarrow' \leftarrow \rightsquigarrow' \cup (z'', z')$
48:     end if
49: end for}
```

```
operator $\mathfrak{tores}(\xi_1, \xi_2$: Clock zone) {
Let $X' = \emptyset$
for all $x \in X$ do
    if $lb(\xi_2, x) = 0$ then
        $X' = X' \cup \{x\}$
    end if
end for
for all $x \in X$ do
    if $x \notin X' \wedge ub(x, \xi_1) < lb(x, \xi_2)$ then
        return $\emptyset$
    end if
end for
$\xi_3 = and\,(\xi_1, free(down(\xi_2), X'))$
return $\xi_3$
}
```

Function 4 Backward_Zones

Input: A zone graph $(Z', z'_0, \rightsquigarrow')$, the original set of transitions $\rightsquigarrow$, and a set of pairs of zones *waiting*.
Output: A zone graph $(Z', z'_0, \rightsquigarrow')$, with newly added zones being traced backward.

```
while $waiting \neq \emptyset$ do
    Let $(z_0, z_1)$ be a pair in $waiting$
    $waiting \leftarrow waiting - \{(z_0, z_1)\}$
    for all $z$ st. $(z, z_1) \in \rightsquigarrow$ do
        Let $e$ be the original switch for transition $(z, z_1)$
        Let $(l, (\_val_d), \xi) = z$ and $(l_1, (\_val_d)_1, \xi_1) = z_1$
        $\xi' \leftarrow Pred_e(\xi_1)$
        $z' = (l, (\_val_d), \xi')$
        Let $waiting_0$ denote the set of first elements in $waiting$
        if $(z' \notin Z' \cup waiting_0)$ then
            $waiting = waiting \cup (z', z)$
        end if
        $\rightsquigarrow' = \rightsquigarrow' \cup (z', z_0)$
    end for
end while
return $(Z', z'_0, \rightsquigarrow')$
```

Example. Fig. 2.4 shows the result of applying our algorithm on the running example (Fig. 2.1). The dashed zones are in $\neg LS$, and the dashed transitions corresponds to the fault. Zone 5 is generated by switch E_5. Adding this switch lets the states in $\neg LS - Q$ have any possible delay. Zone 11 is the "sink" zone, which is used to prune computations leading to violate the specification.

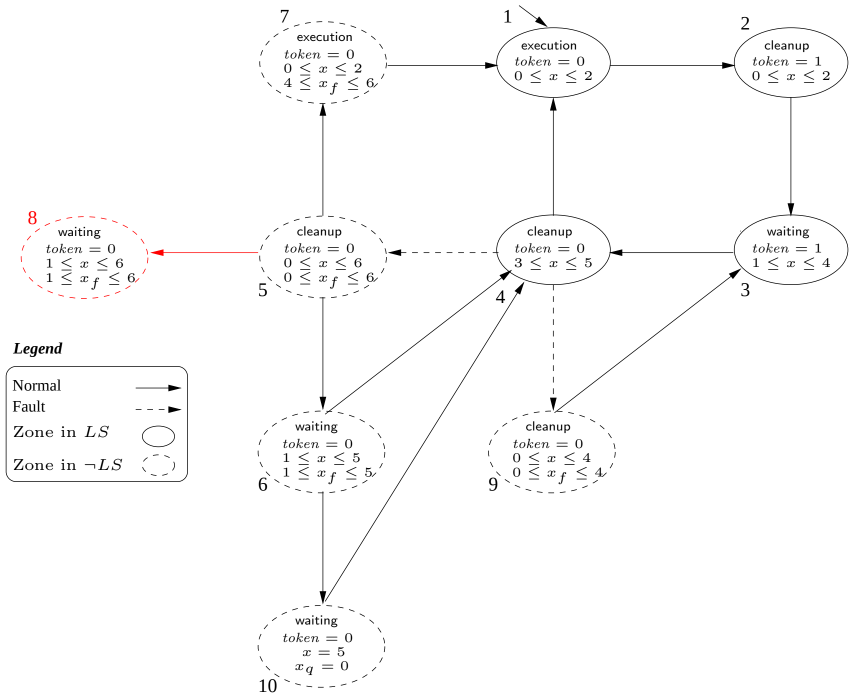


Figure 2.4: Synthesized zone graph of the timed automaton in Fig. 2.1

2.5.3.2 Step 3: Adding Recovery Paths

After generating the enhanced automaton (Line 1), Algorithm 1 calls Function 3 (Add_Trans) to add recovery transitions (Line 3 of Algorithm 1). In order to reduce the complexity of this step, our idea is to first find the ranking of each zone in $\neg LS - Q$ (respectively, $Q - LS$) based on the length of the shortest path to a zone in Q (respectively, LS), and then dynamically update this ranking during the recovery addition step. As soon as the ranking of a zone in $\neg LS$ is less than infinity (there is a path for it to LS), we stop finding a recovery transition from that zone. Adding recovery transitions in Function 3 is achieved by applying two strategies: (1) connecting existing zones to each other (Lines 5–6), and (2) connecting zones by resetting clocks for deadlock zones that cannot get connected using strategy 1 (Lines 7–8).

2.5.3.2.1 Strategy 1 After initializations (Lines 1-3 of Function 3), we add recovery transitions from zones in $\neg LS - Q$ to any possible zone, and also from zones in $Q - LS$ to any possible zone in Q (Lines 5 and 6, respectively) by calling function ConnectZones (defined in Lines 10–26). For adding the transitions between zones, one has to ensure that an added transition respects the clock constraints of source and target zones. To this end, we introduce the operator $\mathfrak{to}$ (defined in Lines 27-33) for finding the subset of a zone which can be connected to another zone. Two conditions for connecting two zones are:

- The upper bound of each clock variable in the first zone should be larger than its lower bound in the second zone. If this condition does not hold, then there is a time gap between the two zones.
- The time monotonicity condition should hold between them. For checking this condition, the intersection of the clock valuations that can reach the target zone, and the source zone is calculated. The result is a subzone of the source zone that can be connected to the target zone, which can be empty or the original source zone.

If zone z is connected to zone z', we set the variable $con(z)$ to 1 to remember that a subset of this zone has been connected to another zone (Line 16). In case a new subzone z'' is created (Line 21), since ξ'' does not include all clock valuations of ξ, we need to ensure that all incoming computations to z'' respect time monotonicity. To this end, all new subzones are added to a waiting set (Line 22), which will be processed in Line 4 of Algorithm 1. Each member of the waiting list is a tuple with the first element being the new subzone, and the second being the original zone from which the subzone is formed.

Example. In Fig. 2.4, zone 9 is added when Function 3 attempts to connect zone 5 to zone 3 in strategy 1. Likewise, zone 6 is added when trying to connect zone 8 to zone 4. The transition from zone 7 to zone 1 is also added in this step.

2.5.3.2.2 Strategy 2 Next, Function Add_Trans handles deadlock zones that could not be connected to other zones (Lines 7 and 8 of Function 3) by calling function ConnectZones-Res (Lines 34–49). This strategy is identical to strategy 1, except it uses operator $\mathfrak{tores}$ (instead of $\mathfrak{to}$). This operator (defined in Lines 50–64) finds a subzone of the first zone that can be connected to the second zone by resetting a set of clock variables. Again applying this operator may result in creation of new subzones that are added to the set *waiting* for later backward[5] zone generation processing.

[5]The clocks that their lower bound is 0 in the target zone can be reset.

2.5.3.3 Step 4: Backward Zone Generation

Since addition of recovery transitions in Step 3 may create new subzones (returned in *waiting* by Function 3), if all incoming transitions of the original superzone are added to the new subzone naively, we may introduce terminating computations. This happens when there are valuations in the predecessor zones of the original zone that cannot reach the new subzone. As an example, consider three zones $\xi_1 ::= 1 \leq x \leq 5$, $\xi_2 ::= 3 \leq x \leq 5$, and $\xi_3 ::= 1 \leq x \leq 4$. There is a transition from ξ_1 to ξ_2. Assume that in Step 3, we tried to add a transition from ξ_2 to ξ_3, and as a result, a new zone $\xi_2' ::= 3 \leq x \leq 4$ is generated, and the transition (ξ_2', ξ_3) is added. Now, if we add a transition from the predecessor of ξ_2, which is ξ_1, to ξ_2', a terminating computation is generated. The problem arises when the computation is in ξ_1, and the clock x has the value $4 < x \leq 5$; it cannot take the added transition to get to the zone $\xi_2' ::= 3 \leq x \leq 4$, since the clock value should decrease to make this happen. To address this case, Function 4 (Backward_Zones) is invoked for backward generation of predecessor zones for each new subzone in *waiting* (called in Line 4 of Algorithm 1).

In Function 4, for each new zone in *waiting*, the switches (including faults) leading to the original zone are considered (Lines 4 and 5), and for each switch, the previous zone of the new zone using this switch is calculated using the $Pred_e$ operator (Line 7). If the previous zone is not already included in the set of zones nor in the waiting list, it will be added to *waiting* (Line 11). Function 4 repeats these steps until all backward reachable zones are explored and the appropriate transitions leading to the new zone are added (Line 13).

Example. In this step, zone 6 is traced backward using the switch corresponding to the transition from zone 5 to zone 8. The result is the added transition from zone 5 to zone 6. Zone 9 is likewise traced backward using the fault switch (corresponding to the transition from zone 4 to zone 5), and as a result, the transition from zone 4 to zone 9 is added.

2.5.3.4 Step 5: Removing Cycles

Adding recovery transitions may lead to introducing a cycle in the zone graph, which violates the bounded response requirement. Thus, the possible added cycles are removed (Line 5 of Algorithm 1). Observe that our assumption on closure of Q will not allow any cycles to be formed between $Q - LS$ and $\neg LS - Q$. Hence, the only possibility of introducing a cycle is between zones in $\neg LS - Q$ and in $Q - LS$.

Removing the cycles can be implemented by applying classic graph-theoretic algorithms. Note that we have the rank of each zone in $\neg LS - Q$ (respectively, $Q - LS$) based on the length of the shortest path to a zone in Q (respectively, LS). For each transition in $\neg LS - Q$

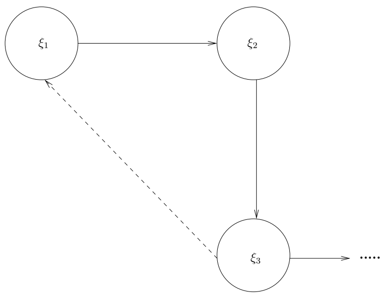


Figure 2.5: An example of cycle removal

(respectively, $Q - LS$), if the rank of the source is less than the rank of the target, then the transition will be removed, as it does not contribute in synthesizing a solution. This transition removal ensures cycle-freedom in the fault span. As an example, consider a part of a zone graph shown in Fig. 2.5. Assume that all three zones are in $\neg LS$, and as shown, the recovery paths of all three zones ξ_1, ξ, and ξ_3 are through the outgoing path from ξ_3. It is obvious that the rank of ξ_3 is less than the rank of ξ_1, and hence, the transition between them will be removed. By removing this transition, the cycle among these three zones will be removed as well.

2.5.3.5 Step 6: Zone Graph Repair

In order to ensure that the synthesized zone graph does not violate timing-independent safety, in Lines 6–10, Algorithm 1 identifies and removes the set of zones/transitions from where faults alone can lead a computation to a state from where safety can be violated (since occurrence of faults cannot be prevented). The rest of the algorithm (Lines 11–23 of Algorithm 1) removes deadlock zones and ensures the closure of legitimate states in the zone graph using a straightforward fixpoint computations. Finally, in Line 24 of Algorithm 1, it generates the output automaton out of the repaired zone graph.

Example. Zone 8 and 11 are deadlock zones and, hence, get removed in this step. The automaton in Fig. 2.6 can be generated out of the repaired zone graph in Fig. 2.4.

Theorem 1. Zone_based_Synthesis *algorithm is sound.*

Proof. We show that any output of algorithm Zone_based_Synthesis is sound. In other words, it meets the four conditions of the problem statement in Section 4.3. We distinguish four cases:

1. By construction, $LS' \subseteq LS$ trivially holds, as no state is added to LS. LS' might have some states removed compared to LS and those are the ones removed in Step

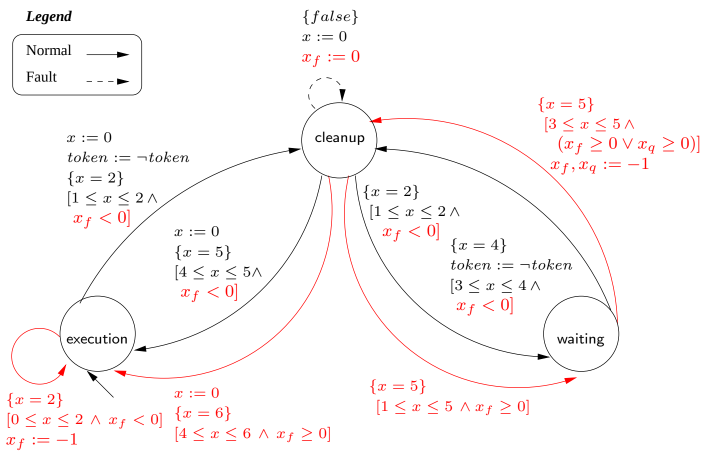


Figure 2.6: Repaired automaton of the TAD in Fig. 2.1

6. Also, observe that clock variables x_f and x_q are disabled in LS and, hence, their values are irrelevant in LS.

2. Q is closed in T'. Recall that Q is closed in the original model. The only switches we add to the automaton in Step1 originating from Q are the ones leading the states that do not satisfy the safety properties to the sink location. Note that the sink zone and all its incoming transitions will be removed in Step 6. Finally, in adding recovery transitions in Step 3, no transition is added from Q to $\neg Q$.

3. By construction, $(T' \mid LS') \subseteq (T \mid LS')$ also trivially holds, as no transition originating from LS is added.

4. TAD' is F-tolerant to $SPEC$ from LS'. To prove this condition, we distinguish two cases:

   - First, we have to show that $TAD' \models_{LS} SPEC$. By construction, and following cases 1 and 3, as well as the fact that the algorithm removes all deadlock states, it follows that the set of computations of TAD' is a subset of computations of TAD' in the absence of faults. Hence, we have $TAD' \models_{LS'} SPEC$.
   - We now need to show that there exists an F-span from where TAD' maintains $SPEC$ in the presence of faults. To this end, notice that if a computation reaches a state in $\neg LS$, by construction, no suffix of this computation includes a transition in BT. Hence, TAD' in the presence of faults maintains $SPEC_{\overline{bt}}$. Moreover, any computation that reaches a state in $\neg LS$ is guaranteed to reach Q and LS within θ and δ time units. This is ensured by Step 1 (by adding

eager switches that do not let x_f and x_q exceed the allowed bounds), Step 4 (by not letting terminating computations being added to the synthesized model), Step 5 (by removing cycles), and Step 6 (by removing deadlocks and ensuring the closure of fault-span and LS). Observe that since all computations are guaranteed to reach LS, liveness is automatically preserved. □

Theorem 2. *Zone_based_Synthesis algorithm is terminating.*

Proof. Now, we show that algorithm Zone_based_Synthesis is terminating. Steps 1 and 2 are clearly terminating, as automaton enhancement has no loops and zone graph generation for finite set of location is always guaranteed to terminate [16].

In step 3, zones are first ranked based on their shortest path to LS. This is done by a slightly modified version of Dijkstra's shortest path algorithm. Then, recovery transitions are added among the zones, which is, in the worst case, quadratic in the size of the zone graph, and hence, terminating. In the next step, zones reachable backward from the newly added zones are calculated. Since only a finite number of backward reachable zones could be generated, this step is also terminating.

The cycle removal step is done by checking the ranks of the source and target zones of transitions in $\neg LS$, and hence, in the worst case, is quadratic in the size of the zone graph. In the last step, the zone graph is repaired by removing the bad states and deadlocks. Deadlock removal is done in a loop, which terminates when a fixpoint is reached, or all zones are removed. It will always reach a fixpoint (if the zone graph does not get empty), since in each iteration the deadlock zones and their incoming transitions are removed. Transition removal might make more zones deadlock, which will be removed in the next iteration. If no new deadlock is formed in an iteration, a fixpoint is reached, and the loop terminates. Since the zone graph is finite, it will eventually reach a fixpoint, or the zone graph gets empty. □

2.6 Implementation and Experimental Results

We have implemented our algorithm to evaluate the efficiency of our synthesis method. We leveraged the IF toolset [23] for zone graph generation. IF provides an intermediate representation for specification of timed automata with urgency. It implements and evaluates different semantics of time, and various types of real-time constructs. We use the intermediate representation syntax to model a timed automaton with faults and automatically add switches to the the input model (Step 1 of Algorithm 1). Then, we utilize the IF API to generate the zone graph of the enhanced automaton. The generated zone graph

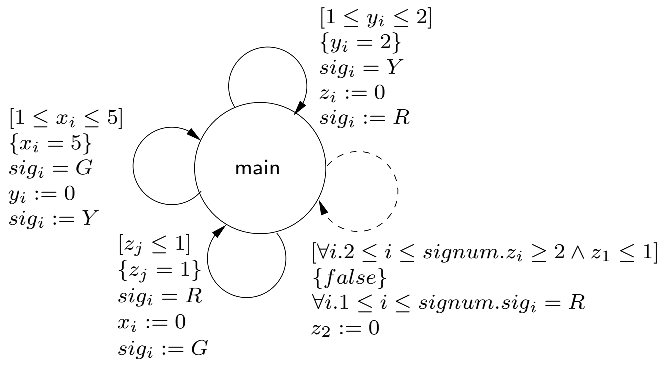


Figure 2.7: Automaton for traffic controller i.

is stored in a graph data structure with zones being marked with LS, $Q - LS$, and $\neg Q$. Then, the rest of the algorithm (Steps 2 – 6) are performed on the generated zone graph. The result is a synthesized zone graph, which can be used to generate the fault-tolerant timed automaton. To evaluate our algorithm, we conducted two case studies.

2.6.1 Case Study 1: Circular Traffic Controller

The first case study (adopted from [17]) is an automaton for a circular traffic controller (Fig. 2.7), with *signum* number of signals. In this automaton, $j = (i + 1) \mod 2$. The dashed switch is the fault and solid switches are the ones given in the input model. For each signal, a discrete variable sig_i ranges over $\{R, G, Y\}$. Also, there are three clock variables for each signal, x_i, y_i, and z_i, that act as timers to change the signal phase. For instance, when a signal i is green, then it goes yellow at least after one time unit and at most within 5 time units (*i.e.,* $1 \le x_i \le 5$). Such change of phase resets clock y_i, which keeps track of time elapsed since signal i has turned yellow. All signals operate identically. One possible set of legitimate states for this model is the following predicate:

$$\begin{aligned} LS \equiv\ & \forall i \in [0, signum). \\ & [(sig_i = G) \Rightarrow ((sig_j = R) \wedge (x_i \le 5) \wedge (z_i > 1))] \wedge \\ & [(sig_i = Y) \Rightarrow ((sig_j = R) \wedge (y_i \le 2) \wedge (z_i > 1))] \wedge \\ & [(sig_i = R) \wedge (sig_j = R) \Rightarrow ((z_i \le 1) \oplus (z_j \le 1))] \end{aligned}$$

where $\oplus$ denotes the exclusive-or operator. A bad transition is one that reaches a state where more than one signal is not red:

$$bt = \{(s_0, s_1) \mid s_1 \models (\exists i, j.\ (i \ne j) \wedge (sig_i \ne R) \wedge (sig_j \ne R))\}$$

Table 2.1: Results for traffic controller of 3-11 signals

	3	**5**	**7**	**9**	**11**
Steps 1,2 (sec)	0.02	0.06	0.62	8.92	265.059
Steps 3-6 (sec)	0.02	0.02	0.07	0.10	0.15
Total synthesis time (sec)	0.04	0.08	0.69	9.02	265.209
Zone Graph Generation of Intolerant Model	0.0 s	$2h, 38m$	$> 3h$	$> 3h$	$> 3h$
Zone Graph Size of Enhanced Automaton	47	59	71	83	95
Zone Graph Size of Intolerant Model	309	1279032	$> 10^6$	$> 10^6$	$> 10^6$

The fault (as can be seen in Fig. 2.7) can reset (for instance) z_2, when all z-clocks, except for z_1, are greater than 2, due to a circuit malfunction. We consider the following bounded response property for this model:

$$SPEC_{\overline{br}} = (\neg LS \mapsto_{\leq 2} Q) \ \wedge \ (Q \mapsto_{\leq 3} LS)$$

Table 2.1 shows the breakdown of the time spent in different steps of our synthesis algorithm in the first three rows for 3-11 traffic signals. As expected, synthesis time increases as we increase the number of traffic signals. However, observe that the bottleneck of our algorithm turns out to be zone graph generation and not the synthesis steps. Thus, to better evaluate our algorithm, we compare it with its corresponding verification time. Notice that zone graph generation of the enhanced automaton (first row) significantly outperforms zone graph generation time for the original automaton with faults (fourth row). This is because the fault leads to bad states and a significant number of reachable zones are eliminated by our pruning switches added in Function 2, and hence, the number of zones in the zone graph of the enhanced automaton is less than the original zone graph. Having a smaller zone graph also assists in increasing the efficiency of the next steps of our algorithm.

2.6.2 Case Study 2: Train Signal Controller

Our second case study is a railway signal controller, consisting of *signum* signals operating in a circular manner for controlling m trains (Fig. 2.8). In this automaton, $k = (i + 1)$ mod 2. Train j is modeled by a discrete variable tr_j that ranges from 1 to *signum*, which shows the location of the train (*i.e.,* the signal ahead of the train). When a train passes a signal, it changes phase from green or yellow to red. When a signal $i + 2$ turns red, its previous signal $i + 1$, which is also red, turns yellow. Then, if the previous signal i is yellow, it may turn green. It takes a train 5 time units to travel from one signal to the

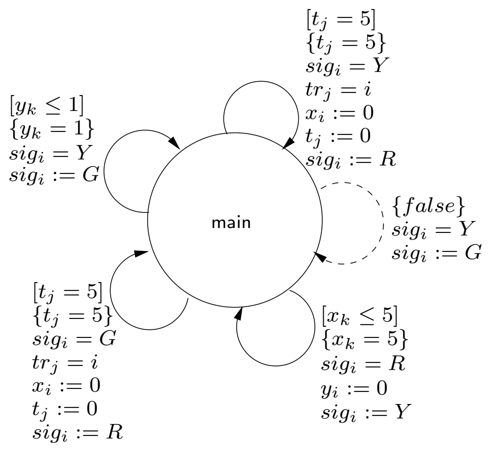


Figure 2.8: Automaton for train signal controller

next. All signals operate identically and, hence, the entire model of the train controller is the parallel composition of *signum* timed automata illustrated in Fig. 2.8.

The safety specification of this model requires that no two trains can be in the same location at the same time:

$$bt = \{(s_0, s_1) \mid s_1 \models (\exists i, j.\ (i \neq j) \wedge (tr_i = tr_j))\}$$

The fault in our case study occurs when the first signal changes phase from yellow to green due to circuit problems. This fault does not cause the computation to violate the specification, but it may result in a deadlock computation, where trains cannot proceed due to deadlocked signals. The bounded response property considered for this model is the following:

$$SPEC_{\overline{br}} = (\neg LS \mapsto_{\leq 2} Q)\ \wedge\ (Q \mapsto_{\leq 1} LS)$$

One possible set of legitimate states is the set of states reachable from the initial state, where no two trains are in the same location, by the switches of the timed automaton (Fig. 2.8).

Table 2.2 presents the results for 4-6 signals and constant number of two trains. As can be seen, the bottleneck is mostly in the step for adding transitions among zones. This is due to the fact that in this model, the fault does not lead the computation to reach bad states and, hence, our pruning strategy is not necessarily helpful. Comparison between the number of zones in the original model and the enhanced one shows that there is an increase in the zone graph size. This is due to adding switches E_5 in Function 2, which let any possible delay in states out of Q. We note that our idea for ranking the zones and updating the ranks dynamically has significantly made this step more efficient. However,

Table 2.2: Results for train signal controller

	4	**5**	**6**
Steps 1,2 (sec)	0.78	1.89	3.38
Step 3 (sec)	9.87	12.95	43.39
Step 4 (sec)	3.16	2.52	9.18
Step 5 (sec)	1.31	1.72	3.67
Step 6 (sec)	2.22	2.56	7.18
Total synthesis time (sec)	17.34	21.64	66.8
Zone Graph Generation of Intolerant Model	1.0 s	1.0 s	1.0 s
Zone Graph Size of Enhanced Automaton	893	1424	1942
Zone Graph Size of Intolerant Model	442	792	1112

we believe that using heuristics, we can still make this step more efficient at the cost of losing completeness.

Our conclusion is that in that in some case studies (such as our first one), the proposed algorithm competes with the verification time (model checking of the original model). In this case study, as the algorithm bottleneck is the zone graph generation time, we can claim that its scalability is as well as the zone graph generation in the underlying tool (IF in our case studies). In the case studies that our pruning strategy does not help (such as our second case study), the bottleneck of the algorithm is mostly in the recovery addition phase. Our idea for ranking the zones and updating the ranks dynamically has helped significantly to make this step more efficient.

Chapter 3

Synthesis of Distributed Self-Stabilizing Systems

3.1 Introduction

Self-stabilization is a versatile technique for forward fault recovery. A self-stabilizing system has two key features:

- *Strong convergence.* When a fault occurs in the system and, consequently, reaches some arbitrary state, the system is guaranteed to recover proper behavior within a finite number of execution steps.
- *Closure.* Once the system reaches such good behavior, typically specified in terms of a set of *legitimate states*, it remains in this set thereafter in the absence of new faults.

Self-stabilization has a wide range of application domains, including networking [34] and robotics [70]. The concept of self-stabilization was first introduced by Dijkstra in the seminal paper [29], where he proposed three solutions for designing self-stabilizing token circulation in ring topologies. Twelve years later, in a follow up article [30], he published the correctness proof, where he states that demonstrating the proof of correctness of self-stabilization was more complex than he originally anticipated. Indeed, designing correct self-stabilizing algorithms is a tedious and challenging task, prone to errors. Also, complications in designing self-stabilizing algorithms arise, when there is no commonly accessible data store for all processes, and the system state is based on the valuations of variables distributed among all processes [29]. Thus, it is highly desirable to have access to techniques that can automatically generate self-stabilizing protocols that are correct by construction.

Program *synthesis* (often called the holy grail of computer science) is an algorithmic technique that takes as input a logical specification and automatically generates as output a program that satisfies the specification. Automated synthesis is generally a highly complex and challenging problem due to the high time and space complexity of its decision procedures. For this reason, synthesis is often used for developing small-sized but intricate components of systems. Synthesizing self-stabilizing distributed protocols involves an additional level of complexity, due to constraints caused by distribution. Examples of such constraints include read-write restriction of processes in the shared-memory model, timing models, and symmetry. These constraints result in combinatorial blowups in the search space of corresponding synthesis problems. For instance, in [54], the authors show that adding stabilization behaviors to a non-stabilizing protocol is NP-complete in the size of the state space, which itself is exponential in the size of variables of the protocol. Also, Ebnenasir and Farahat [35] propose a heuristic automated method to synthesize self-stabilizing algorithms, which is an incomplete technique (i.e., it may fail to find a solution even if there exists one). In bounded synthesis [42], the authors propose a method for synthesizing synchronous distributed protocols that interact with the environment. While this method is quite general, it is not clear how it perform in the context of self-stabilizing protocols.

With this motivation, we focus on the problem of automated *synthesis* of self-stabilizing protocols. Based on the input specification and the type of output program, there are various synthesis techniques. Our technique to synthesize self-stabilizing protocols takes as input the following specification:

1. A *topology* that specifies (1) a finite set V of variables allowed to be used in the protocol and their respective finite domains, (2) the number of processes, and (3) read-set and write-set of each process; i.e., subsets of V that each process is allowed to read and write.

2. A set of *legitimate states* in terms of a Boolean expression over V.

3. The *timing model*; i.e., whether the synthesized protocol is synchronous or asynchronous.

4. *Symmetry*; i.e., whether or not all processes should behave identically.

5. *Type of stabilization*; i.e., *strong* convergence guarantees finite-time recovery, while *weak* convergence guarantees only the possibility of recovery from any arbitrary state.

Our synthesis approach is based on constraint solving and consists of three steps: (1) encoding of the synthesis problem as a set of constraints, (2) quantifier elimination, and (3) complete search over the possible solutions. Our approach is, in particular, SMT[1]-based. That is, given the five above input constraints, we encode them as a set of SMT constrains. We note that quantifier elimination in SMT-solvers even in finite instances is not completely automated. If the SMT instance is satisfiable, then a witness solution to its satisfiability is a distributed protocol that meets the input specification. If the instance is not satisfiable, then we are guaranteed that there is no protocol that satisfies the input specification. To the best of our knowledge, unlike the work in [21, 35], our approach, is the first *sound* and *complete* technique that synthesizes self-stabilizing algorithms[2]. That is, our approach guarantees synthesizing a protocol that is correct by construction, if theoretically, there exists one, thanks to the power of existing constraint solvers. It allows synthesizing protocols with different combinations of timing models along with symmetry and types of stabilization.

Our technique for transforming the input specification into an SMT instance consists in developing the following two sets of constraints:

- *State and transition constraints* capture requirements from the input specification that are concerned with each state and transition of the output protocol. For instance, read-write restrictions constrain transitions of each process; i.e., in all transitions, a process should only read and write variables that it is allowed to. Timing models, symmetry, and designation of legitimate states are constraints applied to states and transitions. Encoding these constraints in an SMT instance is relatively straightforward.

- *Temporal constraints* in our work are only concerned with ensuring closure as well as weak and strong convergence. Our approach to encode weak and strong convergence in an SMT instance is inspired by *bounded synthesis* [42]. In bounded synthesis, temporal logic properties are first transformed into a universal co-Büchi automaton. This automaton is subsequently used to synthesize the next-state function or relation, which in turn identifies the set of transitions of each process.

Solving the satisfiability problem for the conjunction of all above state, transition, and temporal properties results in synthesizing a stabilizing protocol. In order to demonstrate

[1] *Satisfiability Modulo Theories* (SMT) are decision problems for formulas in first-order logic with equality combined with additional background theories such as arrays, bit-vectors, etc.

[2] In [55], the authors independently develop another sound and complete solution. We discussed this work in Chapter 6 in detail.

the effectiveness of our approach, we conduct a diverse set of case studies for automatically synthesizing well-known protocols from the literature of self-stabilization. These case studies include Dijkstra's token ring [29] (for both three and four state machines), maximal matching [68], weak stabilizing token circulation in anonymous networks [28], and the three coloring problem [45]. Given different input settings (i.e., in terms of the network topology, type of stabilization, symmetry, and timing model), we report and analyze the total time needed for synthesizing these protocols using the constraint solver Alloy [49].

The shortcoming of this work is that an *explicit* description of the set of legitimate states is needed as an input to the synthesis algorithm. The problem here is developing a formal predicate for legitimate states is not at all a straightforward task. For instance, the set of legitimate states for Dijkstra's token ring algorithm with three-state machines [29] for three processes is the following:

$$\begin{aligned}
&(((x_0(s)+1 \bmod 3 = x_1(s)) \;\wedge\; (x_1(s)+1 \bmod 3 \neq x_2(s)))) \vee \\
&(((x_1(s) = x_0(s)) \;\wedge\; (x_1(s)+1 \bmod 3 \neq x_2(s)))) \vee \\
&((x_1(s)+1 \bmod 3 = x_0(s)) \;\wedge\; ((x_1(s)+1 \bmod 3 \neq x_2(s)))) \vee \\
&((x_0(s)+1 \bmod 3 \neq x_1(s)) \;\wedge\; (x_1(s)+1 \bmod 3 \neq x_0(s)) \wedge (x_1(s)+1 \bmod 3 = x_2(s)))
\end{aligned}$$

where variable x_i belongs to process i, $s \in LS$, and $x_i(s)$ denotes the value of x_i in state s. Developing such a predicate requires huge expertise and insight. Ideally, the designer should use the basic requirements (unique token and circulation of it) to identify the desired system, instead of somehow magically producing a complex predicate such as the one above. To the best of our knowledge, there exists no automated sound and complete method that can synthesize self-stabilizing systems from their high-level specification.

In our next study, we propose an automated approach to synthesize self-stabilizing systems given (1) the network topology, and (2) the high-level specification of legitimate states in a fragment of linear temporal logic (LTL). Furthermore, we explore automated synthesis of *ideal-stabilizing* protocols [69]. These protocols always satisfy their specification, i.e., all states are legitimate. They address two drawbacks of self-stabilizing protocols, namely exhibiting unpredictable behavior during recovery and poor compositional properties.

In order to keep the input specification as implicit as possible, the input LTL formula may include a set of uninterpreted predicates. In designing ideal-stabilizing systems, the transition relation of the system and interpretation function of uninterpreted predicates must be found such that the specification is satisfied in every state. Our synthesis approach for these problems is also SMT-based. In order to demonstrate the effectiveness of our approach, we conduct a diverse set of case studies using the constraint solver Alloy [49]. In the case of self-stabilizing systems, we successfully synthesize Dijkstra's [29] token ring and Raymond's [74] mutual exclusion algorithms without legitimate states as input. We also synthesize ideal-stabilizing leader election and local mutual exclusion (in a line topology) protocols.

3.2 Preliminaries

In this section, we present the preliminary concepts on distributed programs in the shared-memory model, self-stabilization [31], and concrete/uninterpreted local/global predicates.

3.2.1 Distributed Programs

Throughout this chapter, let V be a finite set of discrete *variables*, where each variable $v \in V$ has a finite domain D_v. A *state* is a valuation of all variables; i.e., a mapping from each variable $v \in V$ to a value in its domain D_v. We call the set of all possible states the *state space*. A *transition* in the state space is an ordered pair (s_0, s_1), where s_0 and s_1 are two states. A *state predicate* is a set of states and a *transition predicate* is a set of transitions. We denote the value of a variable v in state s by $v(s)$.

Definition 17. *A* process π *over a set V of variables is a tuple $\langle R_\pi, W_\pi, T_\pi \rangle$, where*

- *$R_\pi \subseteq V$ is the* read-set *of π; i.e., variables that π can read,*
- *$W_\pi \subseteq R_\pi$ is the* write-set *of π; i.e., variables that π can write, and*
- *T_π is the transition predicate of process π, such that $(s_0, s_1) \in T_\pi$ implies that for each variable $v \in V$, if $v(s_0) \neq v(s_1)$, then $v \in W_\pi$.* □

Notice that Definition 17 requires that a process can only change the value of a variable in its write-set (third condition), but not blindly (second condition). We say that a process $\pi = \langle R_\pi, W_\pi, T_\pi \rangle$ is *enabled* in state s_0 if there exists a state s_1, such that $(s_0, s_1) \in T_\pi$.

Definition 18. *A* distributed program *is a tuple $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$, where*

- *$\Pi_\mathcal{D}$ is a set of processes over a common set V of variables, such that:*
  - *for any two distinct processes $\pi_1, \pi_2 \in \Pi_\mathcal{D}$, we have $W_{\pi_1} \cap W_{\pi_2} = \emptyset$*
  - *for each process $\pi \in \Pi_\mathcal{D}$ and each transition $(s_0, s_1) \in T_\pi$, the following* read restriction *holds:*

$$\begin{aligned} \forall s_0', s_1' : \ & ((\forall v \in R_\pi : (v(s_0) = v(s_0') \ \wedge \ v(s_1) = v(s_1'))) \ \wedge \\ & (\forall v \notin R_\pi : v(s_0') = v(s_1'))) \implies (s_0', s_1') \in T_\pi \end{aligned} \tag{3.1}$$

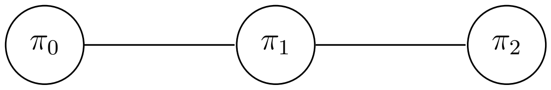


Figure 3.1: Example of a maximal matching problem

- *$T_{\mathcal{D}}$ is a transition predicate.* □

Intuitively, the read restriction in Definition 18 imposes the constraint that for each process π, each transition in T_π depends only on reading the variables that π can read (i.e. R_π). Thus, each transition in $T_{\mathcal{D}}$ is in fact an equivalence class in $T_{\mathcal{D}}$, which we call a *group* of transitions. The key consequence of read restrictions is that during synthesis, if a transition is included (respectively, excluded) in $T_{\mathcal{D}}$, then its corresponding group must also be included (respectively, excluded) in $T_{\mathcal{D}}$. Also, notice that $T_{\mathcal{D}}$ is defined in an abstract fashion. In Section 3.4, we will discuss what transitions are included in $T_{\mathcal{D}}$ based on the timing model and symmetry of process in $\Pi_{\mathcal{D}}$.

Example 3.2.1. We use the problem of distributed self-stabilizing *maximal matching* as a running example to describe the concepts throughout this chapter. In an undirected graph a maximal matching is a maximal set of edges, in which no two edges share a common vertex. Consider the graph in Fig. 3.1 and suppose each vertex is a process in a distributed program. In particular, let $V = \{match_0, match_1, match_2\}$ be the set of variables and $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ be a distributed program, where $\Pi_{\mathcal{D}} = \{\pi_0, \pi_1, \pi_2\}$. We also have $D_{match_0} = \{1, \bot\}$, $D_{match_1} = \{0, 2, \bot\}$, and $D_{match_2} = \{1, \bot\}$. The intuitive meaning of the domain of each variable is that each process can be either matched to one of its adjacent processes (i.e., π_0 can be matched to π_1, π_1 can be matched to either π_0 or π_2, and π_2 can be match to π_1) or to no process (i.e., the value $\bot$). Each process π_i can read and write variable $match_i$ and read the variables of its adjacent processes. For instance, $\pi_0 = \langle R_{\pi_0}, W_{\pi_0}, T_{\pi_0} \rangle$, with $R_{\pi_0} = \{match_0, match_1\}$ and $W_{\pi_0} = \{match_0\}$. Notice that following Definition 18 and read/write restrictions of π_0, (arbitrary) transitions

$$t_1 = ([match_0 = match_2 = \bot, match_1 = 0], [match_0 = 1, match_1 = 0, match_2 = \bot])$$
$$t_2 = ([match_0 = \bot, match_1 = 0, match_2 = 1], [match_0 = match_2 = 1, match_1 = 0])$$

have the same effect as far as π_0 is concerned (since π_0 cannot read $match_2$). This implies that if t_1 is included in the set of transitions of a distributed program, then so should t_2. Otherwise, execution of t_1 by π_0 will depend on the value of $match_2$, which, of course, π_0 cannot read.

Definition 19. *A* computation *of $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ is an infinite sequence of states $\overline{s} = s_0 s_1 \cdots$, such that: (1) for all $i \geq 0$, we have $(s_i, s_{i+1}) \in T_{\mathcal{D}}$, and (2) if a computation* *reaches a state s_i, from where there is no state $\mathfrak{s} \neq s_i$, such that $(s_i, \mathfrak{s}) \in T_{\mathcal{D}}$, then the computation stutters at s_i indefinitely. Such a computation is called a* terminating computation*.* $\Box$

As an example, in maximal matching, computations may terminate when a matching between processes is established.

3.2.2 Predicates

Let $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ be a distributed program over a set V of variables. The *global state space* of $\mathcal{D}$ is the set of all possible global states of $\mathcal{D}$: $\Sigma_{\mathcal{D}} = \prod_{v \in V} D_v$. Likewise, for a process $\pi \in \Pi_{\mathcal{D}}$, the *local state space* of π is the set of all possible local states of π: $\Sigma_\pi = \prod_{v \in R_\pi} D_v$.

Definition 20. *Let $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ be a distributed program. An* interpreted global predicate *is a subset of $\Sigma_{\mathcal{D}}$ and an* interpreted local predicate *is a subset of Σ_π, for some $\pi \in \Pi$.* $\Box$

Definition 21. *Let $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ be a distributed program. An* uninterpreted global predicate *up is an uninterpreted Boolean function from the set of all states. An* uninterpreted local predicate *lp is an uninterpreted Boolean function from the local state space of π, for some $\pi \in \Pi$. We denote the uninterpreted local predicate for the process π by lp_π.* $\Box$

The interpretation of an uninterpreted global predicate is a Boolean function from the set of all states:

$$up_{\mathrm{I}} : \Sigma_D \mapsto \{true, false\}$$

up_{I} represents an interpreted global predicate that includes all states mapped to true. Similarly, the interpretation of an uninterpreted local predicate for the process π is a Boolean function:

$$lp_{\mathrm{I}} : \Sigma_\pi \mapsto \{true, false\}$$

lp_{I} represents an interpreted local predicate that includes all local states mapped to true. Throughout this chapter, we use 'uninterpreted predicate' to refer to either uninterpreted global and local predicate, and use global (local) predicate to refer to interpreted global (local) predicate.

3.2.3 Topology

We now define the notion of *topology*. Intuitively, a topology specifies only the architectural structure of a distributed program (without its set of transitions). The reason for defining topology is that one of the inputs to our synthesis problem is a topology based on which a distributed program is synthesized as output.

Definition 22. *A* topology *is a tuple* $\mathcal{T} = \langle V_{\mathcal{T}}, |\Pi_{\mathcal{T}}|, R_{\mathcal{T}}, W_{\mathcal{T}} \rangle$*, where*

- $V_{\mathcal{T}}$ *is a finite set of finite-domain discrete variables,*
- $|\Pi_{\mathcal{T}}| \in \mathbb{N}_{\geq 1}$ *is the number of processes,*
- $R_{\mathcal{T}}$ *is a mapping* $\{0 \dots |\Pi_{\mathcal{T}}| - 1\} \mapsto 2^V$ *from a process index to its read-set,*
- $W_{\mathcal{T}}$ *is a mapping* $\{0 \dots |\Pi_{\mathcal{T}}| - 1\} \mapsto 2^V$ *that maps a process index to its write-set, such that* $W_{\mathcal{T}}(i) \subseteq R_{\mathcal{T}}(i)$*, for all* i $(0 \leq i \leq |\Pi_{\mathcal{T}}| - 1)$. □

Example 3.2.2. The topology of our matching problem is a tuple $\langle V, |\Pi_{\mathcal{T}}|, R_{\mathcal{T}}, W_{\mathcal{T}} \rangle$, where

- $V = \{match_0, match_1, match_2\}$, with domains $D_{match_0} = \{1, \bot\}$, $D_{match_1} = \{0, 2, \bot\}$, and $D_{match_2} = \{1, \bot\}$,
- $|\Pi_{\mathcal{T}}| = 3$,
- $R_{\mathcal{T}}(0) = \{match_0, match_1\}$, $R_{\mathcal{T}}(1) = \{match_0, match_1, match_2\}$, $R_{\mathcal{T}}(2) = \{match_1, match_2\}$, and
- $W_{\mathcal{T}}(0) = \{match_0\}$, $W_{\mathcal{T}}(1) = \{match_1\}$, and $W_{\mathcal{T}}(2) = \{match_2\}$.

Definition 23. *A distributed program* $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ has *topology* $\mathcal{T} = \langle V_{\mathcal{T}}, |\Pi_{\mathcal{T}}|, R_{\mathcal{T}}, W_{\mathcal{T}} \rangle$*, if and only if*

- *each process* $\pi \in \Pi_{\mathcal{D}}$ *is defined over* $V_{\mathcal{T}}$
- $|\Pi_{\mathcal{D}}| = |\Pi_{\mathcal{T}}|$
- *there is a mapping* $g : \{0 \dots |\Pi_{\mathcal{T}}| - 1\} \mapsto \Pi_{\mathcal{D}}$ *such that*

$$\forall i \in \{0 \dots |\Pi_{\mathcal{T}}| - 1\} : (R_{\mathcal{T}}(i) = R_{g(i)}) \wedge (W_{\mathcal{T}}(i) = W_{g(i)})$$ □

3.2.4 Self-Stabilization

Pioneered by Dijkstra [29], a *self-stabilizing system* is one that always recovers a good behavior (typically, expressed in terms of a set of *legitimate states*), even if it starts execution from any arbitrary initial state. Such an arbitrary state may be reached due to wrong initialization or occurrence of transient faults.

Definition 24. *A distributed program $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D}\rangle$ is* self-stabilizing *for a set LS of* legitimate states *if and only if the following two conditions hold:*

- Strong convergence: *In any computation $\overline{s} = s_0 s_1 \cdots$ of $\mathcal{D}$, where s_0 is an arbitrary state of $\mathcal{D}$, there exists $i \geq 0$, such that $s_i \in LS$. That is, the* computation-tree logic *(CTL) [36] property*[3]*:*
$$SC \;=\; \mathbf{A}\Diamond LS \tag{3.2}$$

- Closure: *For all transitions $(s_0, s_1) \in T_\mathcal{D}$, if $s_0 \in LS$, then $s_1 \in LS$ as well. That is, the CTL property:*
$$CL \;=\; LS \Rightarrow \mathbf{A} \bigcirc LS \tag{3.3}$$

□

Notice that the strong convergence property ensures that starting from any state, any computation will converge to a legitimate state of $\mathcal{D}$ within a finite number of steps. The closure property ensures that starting from any legitimate state, execution of the program remains within the set of legitimate states. Also, since all states in a self-stabilizing distributed program are considered as initial states, CTL Formula 3.3 (i.e., closure *CL*) is evaluated over all possible states. This is why the formula is not of form $\mathbf{A}\Box(LS \Rightarrow \mathbf{A} \bigcirc LS)$.

Example 3.2.3. In our maximal matching problem, the set of legitimate states is:

$$\begin{aligned} LS = \{ & [match_0(s) = 1, match_1(s) = 0, match_2(s) = \perp], \\ & [match_0(s) = \perp, match_1(s) = 2, match_2(s) = 1]\} \end{aligned}$$

There exist several results on impossibility of distributed self-stabilization (e.g., in token circulation and leader election in anonymous networks [47]). Thus, less strong forms of stabilization have been introduced in the literature of distributed computing. One example is *weak-stabilizing* distributed programs [44], where there only exists the *possibility* of convergence.

Definition 25. *A distributed program $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D}\rangle$ is* weak-stabilizing *for a set LS of* legitimate states *if and only if the following two conditions hold:*

[3]In CTL '**A**' denotes 'for all computations', '**E**' denotes 'there exists a computation', '$\Diamond$' denotes 'eventually', and $\bigcirc$ denotes 'next state'.

- Weak convergence: *For each state s_0 in the state space of $\mathcal{D}$, there exists a computation $\overline{s} = s_0 s_1 \cdots$ of $\mathcal{D}$, where there exists $i \geq 0$, such that $s_i \in LS$. That is, the CTL property:*
$$WC \;=\; \mathbf{E}\Diamond LS \tag{3.4}$$
- Closure: *As defined in Definition 24.* $\square$

Notice that unlike strong self-stabilizing programs, in a weak-stabilizing program, there may exist execution cycles outside the set of legitimate states. In the rest of the chapter, we use 'strong self-stabilization' (respectively, 'strong convergence') and 'self-stabilization' (respectively, 'convergence') interchangeably.

Notation. We denote the fact that a distributed program $\mathcal{D}$ satisfies a temporal logic property φ by $\mathcal{D} \models \varphi$. For example, $\mathcal{D} \models SC$ means that distributed program $\mathcal{D}$ satisfies strong convergence.

3.3 Ideal-Stabilization

Before giving the definition of *ideal-stabilization*, we briefly introduce LTL_R, which is a subset of LTL.

3.3.1 Linear Temporal Logic (LTL)

Linear temporal logic (LTL) is a well-known language for specifying temporal properties of concurrent programs. In this study, we use a fragment of LTL formulation, denoted LTL_R, where nested temporal operators are not allowed.

Definition 26 (LTL_R syntax)**.** *The set of* LTL_R *properties are formed by one of the following definitions:*

$$\begin{aligned} P &::= \mathit{true} \mid p \mid \mathit{up} \mid \neg P \mid P_1 \wedge P_2 \\ Q &::= \mathbf{X}\,P \mid P_1\,\mathbf{U}\,P_2 \mid \neg Q \mid Q_1 \wedge Q_2 \end{aligned}$$

where p is a predicate, up is an uninterpreted predicate, and $\mathbf{X}$ *and* $\mathbf{U}$ *are temporal operators.* $\square$

Definition 27 (LTL_R semantics)**.** *Let $\overline{s} = s_0 s_1 \cdots$ be a computation, i be a non-negative integer,* $\{\mathrm{I}\}$ *be a set of interpretation functions (one for each uninterpreted predicate), and*

$up_{\mathrm{I}} \in \{\mathrm{I}\}$ *denote the interpretation function for up. The* satisfaction relation *(denoted* $\models$*) in* $\mathrm{LTL_R}$ *is inductively defined as below:*

$$\begin{array}{rll}
\overline{s}, i \models \mathit{true} & & \\
\overline{s}, i \models p & \mathit{iff} & s_i \models p \;\; (s_i \in p) \\
\overline{s}, i, up_{\mathrm{I}} \models up & \mathit{iff} & s_i \models up_{\mathrm{I}} \;\; (s_i \in up_{\mathrm{I}}) \\
\overline{s}, i, \{\mathrm{I}\} \models \neg P & \mathit{iff} & \overline{s}, i, \{\mathrm{I}\} \not\models P \\
\overline{s}, i, \{\mathrm{I}\} \models P_1 \wedge P_2 & \mathit{iff} & \overline{s}, i, \{\mathrm{I}\} \models P_1 \;\wedge\; \overline{s}, i, \{\mathrm{I}\} \models P_2 \\
\overline{s}, i, \{\mathrm{I}\} \models \mathbf{X}\, P & \mathit{iff} & \overline{s}, i+1, \{\mathrm{I}\} \models P \\
\overline{s}, i, \{\mathrm{I}\} \models P_1 \,\mathbf{U}\, P_2 & \mathit{iff} & \exists k \geq i : \overline{s}, i, \{\mathrm{I}\} \models P_2 \;\wedge\; \forall j : i \leq j < k : \overline{s}, i, \{\mathrm{I}\} \models P_1
\end{array}$$

And, $\overline{s}, \{\mathrm{I}\} \models P \quad \mathit{iff} \quad \overline{s}, 0, \{\mathrm{I}\} \models P.$ □

We note that $\mathbf{F}\, P$ ('eventually' P) is an abbreviation of $\mathit{true}\,\mathbf{U}\,P$. Observe that if up is an uninterpreted local predicate for the process π, then $s_i \models up_{\mathrm{I}}$, iff the projection of s_i on the read-set of π is a member of up_{I}.

Notation. If there exists an interpretation function for each uninterpreted predicate, such that all computations of a distributed program $\mathcal{D}$ satisfy an $\mathrm{LTL_R}$ formula P, then we say that $\mathcal{D}$ satisfies P and write $\mathcal{D} \models P$.

Example 3.3.1. Consider the problem of token passing exclusion in a ring topology (*i.e.*, token ring), where each process π_i has a variable x_i with the domain $D_{x_i} = \{0, 1, 2\}$. This problem has two requirements:

Safety The *safety* requirement for this problem is that in each state, there is exactly one enabled process (*i.e.*, only one process can execute). To formulate this requirement, we assume each process π_i is associated with a local uninterpreted predicate tk_i, which shows whether π_i is enabled. Let LP be the set of all uninterpreted predicates for a ring of size n, i.e., $LP = \{tk_i \mid 0 \leq i < n\}$. A process π_i can execute a transition, if and only if tk_i is true. The $\mathrm{LTL_R}$ formula, $\varphi_{\mathbf{ME}}$, expresses the above requirement for a ring of size n:

$$\varphi_{\mathbf{ME}} = (\forall i \in \{0 \cdots n-1\} : tk_i \iff (\forall val \in \{0,1,2\} : (x_i = val) \Rightarrow \mathbf{X}\,(x_i \neq val)))$$

Using the set of uninterpreted predicates, the safety requirement can be expressed by the following $\mathrm{LTL_R}$ formula:

$$\psi_{\mathbf{safety}} = \exists i \in \{0 \cdots n-1\} : (tk_i \;\wedge\; \forall j \neq i : \neg tk_j)$$

Fairness This requirement implies that for every process π_i and starting from each state, the computation should reach a state, where π_i is enabled. One way to guarantee this requirement is that processes get enabled in a clockwise order in the ring [4], which can be formulated as follows:

$$\psi_{\mathbf{fairenss}} = \forall i \in \{0 \cdots n-1\} : (tk_i \Rightarrow \mathsf{X}\, tk_{(i+1 \bmod n)})$$

Thus, the requirements of the token ring protocol can be formulated as $\psi_{\mathbf{TR}} = \psi_{\mathbf{safety}} \wedge \psi_{\mathbf{fairness}}$. We note that based on Definition 19, since computations are infinite, property $\psi_{\mathbf{TR}}$ automatically ensures deadlock-freedom as well.

Example 3.3.2. As another example, consider the problem of local mutual exclusion on a line topology, where each process π_i has a Boolean variable c_i. The requirements of this problem are as follows:

Safety In each state, *(i)* at least one process is enabled (*i.e.*, deadlock-freedom), and *(ii)* no two neighbors are enabled (*i.e.*, mutual exclusion). To formulate this requirement, we associate each process π_i with a local uninterpreted predicate tk_i, which is true when π_i is enabled:

$$\varphi_{\mathbf{ME}} = \forall i \in \{0 \cdots n-1\} : tk_i \iff ((c_i \Rightarrow \mathsf{X}\, \neg c_i) \wedge (\neg c_i \Rightarrow \mathsf{X}\, c_i))$$

Thus, $LP = \{tk_i \mid 0 \leq i < n\}$ and the safety requirement can be formulated by the following LTL_R formula:

$$\psi_{\mathbf{safety}} = (\exists i \in \{0 \cdots n-1\} : tk_i) \wedge (\forall i \in \{0 \cdots n-2\} : \neg(tk_i \wedge tk_{(i+1)}))$$

Fairness Each process π_i is eventually enabled: $\psi_{\mathbf{fairenss}} = \forall i \in \{0 \cdots n-1\} : (\mathsf{F}\, tk_i)$

The requirements of the local mutual exclusion protocol on a line topology is $\psi_{\mathbf{ME}} = \psi_{\mathbf{safety}} \wedge \psi_{\mathbf{fairness}}$.

3.3.2 Formal Characterization of Ideal-Stabilization

In self-stabilizing systems, the program behaviour during recovery is unpredicted and this is undesirable for some applications. To overcome this limitation, *ideal-stabilization* is introduced in [69]. In an ideal stabilizing system, every state is legitimate, and hence, every computation starting from any state satisfies the system specification.

[4]Note that this is a sufficient requirement for fairness, but it is not necessary.

Definition 28. *Let ψ be an* $\mathrm{LTL_R}$ *specification and $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ be a distributed program. We say that $\mathcal{D}$ is* ideal stabilizing *for ψ iff starting from any arbitrary state $\mathcal{D} \models \psi$.* □

Ideal-stabilization can be defined for an ideal specification, or a non-ideal specification. An ideal specification is satisfied in every possible computation, and a specification is non-ideal otherwise. Since ideal-stabilization to non-ideal specifications is more useful in practice, in this study, we only consider this type of ideal stabilization. To design an ideal stabilizing system for a non-ideal specification, we should find a transition predicate and an interpretation function for every uninterpreted predicate (if included in the specification), such that the system satisfies the specification. Note that there is a specification for every system to which it ideally stabilizes [69], and that is the specification that includes all of the system computations. In this thesis, we do the reverse; meaning that getting a specification $\mathcal{SP}$, we synthesize a distributed system that ideally stabilizes to $\mathcal{SP}$.

3.4 Timing Models and Symmetry in Distributed Programs

Our synthesis problem takes as input the type of timing model as well as symmetry requirements among processes. These constraints are defined in Subsections 3.4.1 and 3.4.2.

3.4.1 Timing Models

Two commonly-considered timing models in the literature of distributed computing are *synchronous* and *asynchronous* programs [63]. In an asynchronous distributed program, every transition of the program is a transition of one and only one of its processes.

Definition 29. *A distributed program $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ is* asynchronous *if and only if the following condition holds:*

$$\begin{aligned} ASYN \ &= \forall (s_0, s_1) \in T_\mathcal{D} : ((\exists \pi \in \Pi_\mathcal{D} : (s_0, s_1) \in T_\pi) \vee \\ &\quad ((s_0 = s_1) \ \wedge \ \forall \pi \in \Pi_\mathcal{D} : \forall \mathfrak{s} : (s_0, \mathfrak{s}) \notin T_\pi)) \end{aligned} \tag{3.5}$$

Thus, the transition predicate of an asynchronous program is simply the union of transition predicates of all processes. That is,

$$T_\mathcal{D} = \bigcup_{\pi \in \Pi_\mathcal{D}} T_\pi$$

An asynchronous distributed program resembles a system, where process transitions execute in an *interleaving* fashion.

In a synchronous distributed program, on the other hand, in every step, all enabled processes have to take a step simultaneously.

Definition 30. *A distributed program $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ is* synchronous *if and only if the following condition holds:*

$$\begin{aligned} SYN = &\forall (s_0, s_1) \in T_\mathcal{D} : \forall \pi \in \Pi_\mathcal{D} : \\ &(\exists \mathfrak{s} : ((s_0, \mathfrak{s}) \in T_\pi) \wedge \forall v \in W_\pi : v(s_1) = v(\mathfrak{s})) \vee \\ &(\forall \mathfrak{s} : ((s_0, \mathfrak{s}) \notin T_\pi) \wedge \forall v \in W_\pi : v(s_0) = v(s_1)) \end{aligned} \quad (3.6)$$

$\square$

In other words, a distributed program is synchronous, if and only if each transition $(s_0, s_1) \in T_\mathcal{D}$ is obtained by execution of all enabled processes (the ones that have a transition starting from s_0). Hence, the value of the variables in their write-sets change in s_1 accordingly. Also, for all non-enabled processes, the value of the variables in their write-sets do not change from s_0 to s_1.

3.4.2 Symmetry

Symmetry [39] in distributed programs refers to similarity of behavior of different processes.

Definition 31. *A distributed program $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ is called* symmetric *if and only if for any two distinct processes $\pi, \pi' \in \Pi_\mathcal{D}$, there exists a bijection $f : R_\pi \to R_{\pi'}$, such that the following condition holds:*

$$\begin{aligned} SYM = &\forall (s_0, s_1) \in T_\pi : \exists (s'_0, s'_1) \in T_{\pi'} : \\ &(\forall v \in R_\pi : (v(s_0) = f(v)(s'_0))) \wedge (\forall v \in W_\pi : (v(s_1) = f(v)(s'_1))) \end{aligned} \quad (3.7)$$

$\square$

In other words, in a symmetric distributed program, the transitions of a process can be determined by a simple variable mapping from another process. A distributed program is called *asymmetric* if it is not symmetric.

3.5 Problem Statement

In this chapter, we propose synthesis solutions for three problems. The goal of the first problem is to synthesize strong and weak self-stabilizing distributed programs by starting from the description of its set of legitimate states and the architectural structure of processes. In the second and third problems, we get as input the desired system topology, and two LTL_{R} formulas φ and ψ that involve a set LP of uninterpreted predicates. For instance, in Example 3.3.1, $\psi_{\mathbf{TR}}$ includes safety and fairness, which should hold in the set of legitimate states, while $\varphi_{\mathbf{TR}}$ is a general requirement that we specify on every uninterpreted predicate tk_i. The goal of the second and third problems is to synthesize self-stabilizing and ideal-stabilizing systems, respectively. Note that in ideal-stabilizing systems, all states are legitimate. In the second problem, we don't get LS as a set of states (global predicate), and hence, we refer to this problem as "synthesis of self-stabilizing systems with implicit LS". The three problem statements are more formally presented below.

Problem statement 1 (self-stabilization). Given is

1. a topology $\mathcal{T} = \langle V, |\Pi_{\mathcal{T}}|, R_{\mathcal{T}}, W_{\mathcal{T}} \rangle$
2. a set LS of legitimate states
3. the specification of the timing model, type of self-stabilization, and symmetry of the resulting system.

The synthesis algorithm is required to generate as output a distributed program $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$, such that, based on the given input specification: (1) $\mathcal{D}$ has topology $\mathcal{T}$, (2) $\mathcal{D} \models SC \wedge CL$ or $\mathcal{D} \models WC \wedge CL$, and (3) $T_{\mathcal{D}}$ respects $ASYN$ or SYN, and if symmetry is required, it also respects SYM.

Problem statement 2 (self-stabilization with implicit LS). Given is

1. a topology $\mathcal{T} = \langle V, |\Pi_{\mathcal{T}}|, R_{\mathcal{T}}, W_{\mathcal{T}} \rangle$;
2. two $\mathrm{LTL_R}$ formulas φ and ψ that involve a set LP of uninterpreted predicates.
3. the specification of the timing model, type of self-stabilization and symmetry of the resulting system.

The synthesis algorithm is required to identify as output

1. a distributed program $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$
2. an interpretation function for every local predicate $lp \in LP$
3. the global state predicate LS

such that (1) $\mathcal{D}$ has topology $\mathcal{T}$, (2) $\mathcal{D} \models \varphi$, (3) $\mathcal{D} \models (LS \Rightarrow \psi)$, (4) $\mathcal{D}$ is self-stabilizing for LS ($\mathcal{D} \models SC \wedge CL$ or $\mathcal{D} \models WC \wedge CL$), and (5) $T_{\mathcal{D}}$ respects $ASYN$ or SYN, and if symmetry is required, it also respects SYM.

Problem statement 3 (ideal-stabilization). Given is

1. a topology $\mathcal{T} = \langle V, |\Pi_{\mathcal{T}}|, R_{\mathcal{T}}, W_{\mathcal{T}} \rangle$
2. two $\mathrm{LTL_R}$ formulas φ and ψ that involve a set LP of uninterpreted predicates.
3. the specification of the timing model and symmetry of the resulting system.

The synthesis algorithm is required to generate as output

1. a distributed program $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$
2. an interpretation function for every local predicate $lp \in LP$

such that (1) $\mathcal{D}$ has topology $\mathcal{T}$, (2) $\mathcal{D} \models (\varphi \wedge \psi)$, and (3) $T_{\mathcal{D}}$ respects $ASYN$ or SYN, and if symmetry is required, it also respects SYM.

3.6 SMT-based Synthesis Solution

In this section, we propose a technique that transforms the synthesis problem stated in Section 3.5 into an SMT solving problem. An SMT instance consists of two parts: (1)

a set of *entity* declarations (in terms of sets, relations, and functions), and (2) first-order modulo-theory *constraints* on the entities. An SMT-solver takes as input an SMT instance and determines whether or not the instance is satisfiable; i.e., whether there exists concrete SMT entities (also called an *SMT model*) that satisfy the constraints. We transform the input to our synthesis problem into an SMT instance. If the SMT instance is satisfiable, then the witness generated by the SMT solver is the answer to our synthesis problem. We describe the SMT entities obtained in our transformation in Subsection 3.6.1. Constraints that appear in all SMT instances regardless of the timing model, type of symmetry and stabilization are presented in Subsection 3.6.2, while constraints depending on these factors are discussed in Subsection 3.6.4.

3.6.1 SMT Entities

Recall that the inputs to our problem are a topology $\mathcal{T} = \langle V, |\Pi_\mathcal{T}|, R_\mathcal{T}, W_\mathcal{T}\rangle$, a set LS of legitimate states, and the program type. Let $D = \langle \Pi_\mathcal{D}, T_\mathcal{D}\rangle$ denote the distributed program to be synthesized that has topology $\mathcal{T}$ and legitimate states LS. In our SMT instance, we include:

- A set D_v for each $v \in V$, which contains the elements in the domain of v.
- A set called S, whose cardinality is

  $$\left|\prod_{v\in V} D_v\right|$$

  (i.e., the Cartesian product of all variable domains). This set represents the state space of the synthesized distributed program. Recall that in a self-stabilizing program, any arbitrary state can be an initial state and, hence, we need to include the entire state space in the SMT instance.
- An uninterpreted function v_val for each variable v, $v\_val : S \mapsto D_v$ that maps each state in the state-space to a valuation of that variable.
- A relation $T_\mathcal{D}$ that represents the transition relation of the synthesized distributed program (i.e., $T_\mathcal{D} \subseteq S \times S$). Obviously, the main challenge in synthesizing $\mathcal{D}$ is identifying $T_\mathcal{D}$, since variables (and, hence, states) and read/write-sets of $\Pi_\mathcal{D}$ are given by topology $\mathcal{T}$.
- An uninterpreted function γ, from each state to a natural number ($\gamma : S \mapsto \mathbb{N}$). We will discuss this function in detail in Subsection 3.6.4.1.

- In the case of problem statements 2 and 3: An uninterpreted function lp_val for each uninterpreted predicate $lp \in LP$; i.e, $lp\_val : S \mapsto$ **Boolean**.
- In the case of problem statement 1: A Boolean function $LS : S \mapsto \{0, 1\}$. $LS(s)$ is true if and only if s is a legitimate state.
- In the case of problem statement 2: An uninterpreted function $LS : S \mapsto$ **Boolean**.

Example 3.6.1. In our maximal matching problem (problem statement 1), the SMT entities are as follows:

- $D_{match_0} = \{\perp, 1\}$, $D_{match_1} = \{\perp, 0, 2\}$, $D_{match_2} = \{\perp, 1\}$
- set S, where $|S| = 12$
- $match_0\_val : S \mapsto D_{match_0}$, $match_1\_val : S \mapsto D_{match_1}$, $match_2\_val : S \mapsto D_{match_2}$
- $T_{\mathcal{D}} \subseteq S \times S$
- $\gamma : S \mapsto \mathbb{N}$
- $LS : S \mapsto \{0, 1\}$

3.6.2 General Constraints

In this section, we present the constraints that correspond to all problem statements and appear in all SMT instances regardless of the timing model and type of symmetry and stabilization.

3.6.2.1 State Distinction

As mentioned, we specify the size of the state space in the model. The first constraint in our SMT instance stipulates that any two distinct states differ in the value of some variable:

$$\forall s_0, s_1 \in S : (s_0 \neq s_1) \implies (\exists v \in V : v\_val(s_0) \neq v\_val(s_1)) \tag{3.8}$$

Example 3.6.2. In our maximal matching problem, the state distinction constraint is:

$$\begin{aligned}\forall s_0, s_1 \in S : (s_0 \neq s_1) \implies & (match_0\_val(s_0) \neq match_0\_val(s_1)) \vee \\ & (match_1\_val(s_0) \neq match_1\_val(s_1)) \vee \\ & (match_2\_val(s_0) \neq match_2\_val(s_1))\end{aligned}$$

3.6.2.2 Read Restrictions

To ensure that $\mathcal{D}$ meets the read restrictions given by $\mathcal{T}$, we add the following constraint for each process index $i \in \{0, \dots, |\Pi_{\mathcal{T}}| - 1\}$:

$$\forall s_0, s_1 \in S \ : \Big((s_0, s_1) \in T_{\mathcal{D}} \ \wedge \ \exists v \in W_{\mathcal{T}}(i) : \ v\_val(s_0) \neq v\_val(s_1)\Big) \implies \Big(\forall s_0', s_1' \in S \ : \ \Big(\forall v' \in R_{\mathcal{T}}(i) : v'\_val(s_0) = v'\_val(s_0') \ \wedge \ \forall v' \in W_{\mathcal{T}}(i) : v'\_val(s_1) = v'\_val(s_1')\Big) \implies (s_0', s_1') \in T_{\mathcal{D}}\Big) \tag{3.9}$$

Note that Constraint 3.9 is formulated differently from the definition of read restriction in Condition 3.1. The reason is that Definition 18 corresponds to an asynchronous system. To cover both synchronous and asynchronous systems, we formalize read restrictions as Constraint 3.9, which can be used in addition to Constraint 3.23 to synthesize asynchronous systems. This will be discussed in Subsection 3.6.4.3.

Example 3.6.3. In our maximal matching problem, the read restriction for process 0 is the following constraint:

$$\begin{aligned}
&\forall s_0,\ s_1 \in S : \big((s_0, s_1) \in T_{\mathcal{D}} \ \wedge \ match_0\_val(s_0) \neq match_0\_val(s_1)\big) \implies \\
&\forall s_0', s_1' \in S : \ \big(match_0\_val(s_0) = match_0\_val(s_0') \ \wedge \\
&\qquad\qquad match_1\_val(s_0) = match_1\_val(s_0') \ \wedge \\
&\qquad\qquad match_0\_val(s_1) = match_0\_val(s_1')\big) \implies (s_0', s_1') \in T_{\mathcal{D}}
\end{aligned}$$

3.6.3 General Constraints for Problem Statements 1 & 2

In this section, we present the general constraints (independent of timing model and type of symmetry and stabilization) that are added in the case of problem statements 1 and 2 (synthesizing a self-stabilizing system).

3.6.3.1 Constraints for Behavior of the Synthesized Program in the Absence of Faults

In the case of synthesizing a self-stabilizing system, our synthesis problem can also take as input a set of actions that the synthesized program must include inside the legitimate states. This constraint is beneficial in cases where the behavior of a self-stabilizing protocol in the absence of faults is already known and/or is important to preserve. Given a set of transitions that start in LS, denoted by T_{LS}, Constraint 3.10 is added to the SMT instance:

$$\begin{aligned}&(\forall(s,s') \in T_{LS} : (s,s') \in T_p) \wedge \\ &(\forall(s,s') \in T_p : LS(s) \wedge LS(s') \implies (s,s') \in T_{LS})\end{aligned} \tag{3.10}$$

While the first conjunct ensures that the synthesized model includes all transitions in T_{LS}, the second conjunct guarantees that no new behavior is added in the fault-free scenarios. Notice that the constraint can be easily changed to synthesize a model, where the set of transitions in LS is a nonempty subset of T_{LS}.

3.6.3.2 Closure (CL)

The formulation of the CL constraint in our SMT instance is as follows:

$$\forall s, s' \in S \ : \ (LS(s) \wedge (s,s') \in T_{\mathcal{D}}) \implies LS(s') \tag{3.11}$$

3.6.4 Program-specific Constraints

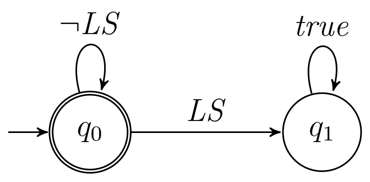
We now present the model constraints that depend on the specific timing model, type of symmetry, and stabilization (i.e., strong and weak-stabilization, asynchronous and symmetric programs).

3.6.4.1 Strong Convergence (SC)

Our formulation of the SMT constraints for SC is an adaptation of the concept of *bounded synthesis* [42]. Inspired by bounded model checking techniques [26], the goal of bounded synthesis is to synthesize an implementation that realizes a set of linear-time temporal logic (LTL) properties, where the size of the implementation is bounded (in terms of the number of states). We emphasize that although strong convergence (Constraint 3.2) is stated in CTL, it can also be stated by an equivalent LTL property:

$$\mathcal{D} \models \mathbf{A}\Diamond LS \ \Leftrightarrow \ \mathcal{D} \models \Diamond LS$$

for any distributed program $\mathcal{D}$. One difficulty with bounded model checking and synthesis is to make an estimate on the size of reachable states of the program under inspection. This difficulty is not an issue in the context of synthesizing self-stabilizing systems, since it is assumed that any arbitrary state is either reachable or can be an initial state. Hence,



$Q = \{q_0, q_1\}$, $Q_0 = \{q_0\}$, $\Delta = \{(q_0, q_0), (q_0, q_1), (q_1, q_1)\}$, $G(q_0, q_0) = \{\neg LS\}$, $G(q_0, q_1) = \{LS\}$, $G(q_1, q_1) = \{true\}$

Figure 3.2: Universal co-Büchi automaton for strong convergence $\varphi = \Diamond LS$.

the bound will be equal to the size of the state space; i.e., the size is a priori known by the input topology. The bounded synthesis technique for synthesizing a state-transition system from a set of LTL properties consists in two steps [42]:

- **Step 1: Translation to universal co-Büchi automaton.** First, we transform each LTL property φ into a universal co-Büchi automaton B_φ using the method in [61]. Roughly speaking, a universal co-Büchi automaton [42, 61] is a tuple $B_\varphi = \langle Q, Q_0, \Delta, G\rangle$, where Q is a set of states, $Q_0 \subseteq Q$ is the set of initial states, $\Delta \subseteq Q \times Q$ is a set of transitions, and G maps each transition in Δ to propositional conditions. Each state could be accepting (depicted by a circle), or rejecting (depicted by a double-circle). In particular, Fig. 3.2 shows the universal co-Büchi automaton for the strong convergence property $SC = \Diamond LS$. This property is, in fact, the only property for which we use bounded synthesis.

  Let $ST = \langle S, S_0, T_\mathcal{D}\rangle$ be a state-transition system, where S is a set of states, $S_0 \subseteq S$ is the set of initial states, and $T_\mathcal{D} \subseteq S \times S$ is a set of transitions. We say that B_φ accepts ST if and only if on every infinite path of ST running on B_φ, there are only finitely many visits to the set of rejecting states in B_φ [61]. For instance, if a state-transition system is self-stabilizing for the set LS of legitimate states, all its infinite paths visit a state in $\neg LS$ only finitely many times. Hence, the automaton in Fig. 3.2 accepts such a system.

- **Step 2: SMT encoding.** In this step, the conditions for the co-Büchi automaton to satisfy a state-transition system are formulated as a set of SMT constraints. To this

[4]Observe that the 'run graph of ST on B_φ' is a subset of the cross product of the automata B_φ and ST, with the initial state (q_0, s_0), such that for each transition $((q, s), (q', s'))$, three conditions hold; $(q, q') \in \Delta$, $(s, s') \in T_\mathcal{D}$, and $G(q, q')(s)$ evaluates to *true*. B_φ accepts ST, if every infinite path in the corresponding run graph, starting from the initial state, only encounters finitely many rejecting states of B_φ.

end, we utilize the technique proposed in [42] for developing an *annotation function* $\lambda : Q \times S \mapsto \mathbb{N} \cup \{\bot\}$, such that the following three conditions hold:

$$\forall q_0 \in Q_0 : \forall s_0 \in S_0 \;:\; \lambda(q_0, s_0) \in \mathbb{N} \tag{3.12}$$

If (1) $\lambda(q,s) \neq \bot$ for some $q \in Q$ and $s \in S$, (2) there exists $q' \in Q$ such that q' is an accepting state and $(q,q') \in \Delta$ with the condition $g \in G$, and (3) g is satisfied in the state s, then

$$\forall s' \in S : (s,s') \in T_{\mathcal{D}} \implies (\lambda(q',s') \neq\bot \;\wedge \lambda(q',s') \geq \lambda(q,s)) \tag{3.13}$$

and if q' is a rejecting state in the co-Büchi automaton, then

$$\forall s' \in S \;:\; (s,s') \in T_{\mathcal{D}} \implies (\lambda(q',s') \neq\bot \;\wedge \lambda(q',s') > \lambda(q,s)) \tag{3.14}$$

It is shown in [42] that the acceptance of a finite-state state-transition system by a universal co-Büchi automaton is equivalent to the existence of an annotation function λ. The natural number assigned to (q,s) by λ is meant to represent the maximum number of rejecting states that occur on some path of $B_\varphi \times ST$ that reaches (q,s) (i.e., when running the state-transition system ST on the universal co-Büchi automaton B_φ).

To ensure that the synthesized distributed program $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ satisfies strong convergence, we use the bounded synthesis technique explained above. In the first step, we construct the universal co-Büchi automaton for the LTL property $\Diamond LS$ (see Fig. 3.2). The annotation constraints for the transitions in $T_{\mathcal{D}}$ with the set of states S for the automaton in Fig. 3.2 are as follows:

$$\forall s \in S \;:\; \lambda(q_0, s) \neq\bot \tag{3.15}$$

$$\begin{aligned}&\forall s, s' \in S :\; (\lambda(q_0,s) \neq\bot \;\wedge LS(s) \;\wedge\; (s,s') \in T_{\mathcal{D}}) \implies \\ &\qquad (\lambda(q_1,s') \neq\bot \;\wedge \lambda(q_1,s') \geq \lambda(q_0,s))\end{aligned} \tag{3.16}$$

$$\begin{aligned}&\forall s, s' \in S \;:\; (\lambda(q_1,s) \neq\bot \;\wedge \mathit{true} \;\wedge\; (s,s') \in T_{\mathcal{D}}) \implies \\ &\qquad (\lambda(q_1,s') \neq\bot \;\wedge \lambda(q_1,s') \geq \lambda(q_1,s))\end{aligned} \tag{3.17}$$

$$\begin{aligned}&\forall s, s' \in S \;:\; (\lambda(q_0,s) \neq\bot \;\wedge \neg LS(s) \;\wedge\; (s,s') \in T_{\mathcal{D}}) \implies \\ &\qquad (\lambda(q_0,s') \neq\bot \;\wedge \lambda(q_0,s') > \lambda(q_0,s))\end{aligned} \tag{3.18}$$

Notice that Constraint 3.15 is obtained from Constraint 3.12 (since in a self-stabilizing system, every state can be an initial state). Similarly, Constraints 3.16 and 3.17 are instances of Constraint 3.13 for transitions (q_0,q_1) and (q_1,q_1), respectively. Also, Constraint 3.18 is an instance of Constraint 3.14 for transition (q_0,q_0) (see Fig 3.2). We now claim that Constraints 3.16 and 3.17 can be eliminated.

Lemma 1. *There always exists a non-trivial annotation function λ, which evaluates Constraints 3.16 and 3.17 as true.*

Proof. We show that we can always find an annotation function that satisfies Constraints 3.16 and 3.17 without violating the other constraints. To this end, assume that there is an annotation that satisfies all properties except for the Constraint 3.16. Hence, we have:

$$\exists s, s' \in S \;:\; LS(s) \;\wedge\; (s, s') \in T_{\mathcal{D}} \;\wedge (\lambda(q_1, s') = \perp \;\vee \lambda(q_1, s') < \lambda(q_0, s))$$

We can simply assign $\lambda(q_0, s)$ to $\lambda(q_1, s')$, without violating Constraints 3.15 and 3.18. This assignment can be done in a fixpoint iteration, until no more violation exists. We can develop a similar proof for Constraint 3.17. Intuitively, for each state s, we assign to $\lambda(q_1, s)$, the maximum number assigned to $\lambda(q_1, s')$, for every state s' in any path reaching s. $\square$

Following Lemma 1, since Constraints 3.16 and 3.17 can be removed from the SMT instance, all constraints involving λ will have q_0 as their first argument. This observation results in replacing λ by a simpler annotation function γ as follows:

- Function γ takes only one argument, since the state of the co-Buchi automaton is always q_0.
- Due to Constraint 3.15, the value $\perp$ is irrelevant in the range of the annotation functions. Hence, we define our annotation function as:

$$\gamma \;:\; S \mapsto \mathbb{N} \tag{3.19}$$

As a result, one can simplify Constraints 3.15-3.18 as follows:

$$\forall s, s' \in S \;:\; \neg LS(s) \;\wedge\; (s, s') \in T_{\mathcal{D}} \implies \gamma(s') > \gamma(s) \tag{3.20}$$

The intuition behind Constraints 3.19 and 3.20 can be understood easily. If we can assign a natural number to each state, such that along each outgoing transition from a state in $\neg LS$, the number is strictly increasing, then the path from each state in $\neg LS$ should finally reach LS or get stuck in a state, since the size of state space is finite. Also, there can not be any loops whose states are all in $\neg LS$, as imposed by the annotation function.

Finally, the following constraint ensures that there is no deadlock state in $\neg LS$:

$$\forall s \in S \;:\; \neg LS(s) \implies \exists s' \in S \;:\; (s, s') \in T_{\mathcal{D}} \tag{3.21}$$

3.6.4.2 Weak Convergence (WC)

To synthesize a weak self-stabilizing system, the SMT instance should encode property $WC = \exists\Diamond LS$ rather than SC. Notice that WC is not an LTL property and, hence, cannot be transformed into an SMT constraint using the 2-step approach introduced in Subsection 3.6.4.1. To this end, we refine the constraints developed for strong convergence as follows. Since in weak convergence, for each state in $\neg LS$, a path should exist to a state in LS, we utilize the following constraint:

$$\forall s \in S \ : \ \neg LS(s) \implies \exists s' \in S \ : \ (s, s') \in T_{\mathcal{D}} \ \wedge \ \gamma(s') > \gamma(s) \tag{3.22}$$

where γ is the annotation Function 3.19. It is straightforward to prove using induction that if a transition system satisfies Constraint 3.22, then for each state in $\neg LS$, there exists a path to a state in LS.

3.6.4.3 Constraints for an Asynchronous System

The transition relation obtained using the constraints introduced in the previous Subsection does not impose any requirements on which process can execute in each state. In fact, since $T_{\mathcal{D}}$ encodes a next-state function, all processes that can execute a local transition while respecting the read-write restrictions would take a step. Such a program stipulates a synchronous program, where all processes execute a local transition at the same time (if there exists one). To synthesize an asynchronous distributed program, instead of a transition function $T_{\mathcal{D}}$, we introduce a transition relation T_i for each process index $i \in \{0, \dots, |\Pi_{\mathcal{T}}| - 1\}$ $T_{\mathcal{D}} = T_0 \cup \dots \cup T_{|\Pi_{\mathcal{T}}|-1})$, and add the following constraint for each transition relation:

$$\forall (s_0, s_1) \in T_i \ : \forall v \notin W_{\mathcal{T}}(i) \ : \ v\_val(s_0) = v\_val(s_1) \tag{3.23}$$

Constraint 3.23 ensures that in each relation T_i, only process π_i can execute. By introducing $|\Pi_{\mathcal{T}}|$ transition relations, we consider all possible interleaving of processes execution.

Example 3.6.4. To synthesize an asynchronous version of our maximal matching example, we define three relations T_0, T_1, and T_2 and add a constraint for each to the SMT instance. For example, the constraint for T_0 is:

$$\begin{aligned} \forall (s_0, s_1) \in T_0 \ : & \ (match_1\_val(s_0) = match_1\_val(s_1)) \wedge \\ & (match_2\_val(s_0) = match_2\_val(s_1)) \end{aligned}$$

3.6.4.4 Constraints for Symmetric Systems

To synthesize a symmetric distributed program, processes should have a symmetric topology as well, meaning that the number of read variables and write variables, as well as their domains, should be similar in all processes (see Constraint 3.7). Let us assume that the size of the read-set and write-set of all processes are $|R_p|$ and $|W_p|$, respectively. Also, assume R_p and W_p to be a set of variables with the same domains as the read-set and write-set of each process. We define an uninterpreted relation T_p that represents how processes execute in a symmetric distributed program:

$$T_p \subseteq (\prod_{(v \in R_p)} D_v) \times (\prod_{(v \in W_p)} D_v) \tag{3.24}$$

Let

$$V\_val : S \mapsto \prod_{v \in V} D_v$$

be the set of all state valuations for the variables in V for a given state. We define a function

$$f : \mathbb{N} \mapsto V\_val$$

that gets a process index and maps it to the valuation function of all variables in the read-set of the process. Likewise, we define a function

$$g : \mathbb{N} \mapsto V\_val$$

which does a similar task for the variables in the write-set of each process. We add Constraint 3.25 for each process index $i \in \{0, \dots, |\Pi_{\mathcal{T}}| - 1\}$ to ensure that all processes act symmetrically:

$$\forall s_0, s_1 \in S : ((s_0, s_1) \in T_i \iff (f(i)(s_0), g(i)(s_1)) \in T_p) \tag{3.25}$$

Note that synthesis of symmetric systems does not need the read restriction constraints. The reason is that the next value of write variables of a process π is specified by a relation (T_p) based on the values of read variables of the process π. We should also mention that Constraint 3.25 corresponds to an asynchronous system. The constraint could be easily rewritten for a synchronous system, where there is only one transition relation.

Example 3.6.5. In order to synthesize a symmetric program for our matching problem, we assume there are four processes π_0, π_1, π_2, and π_3 on a ring, and that all domains are similar and equal to $\{l, s, r\}$, meaning that a process can be matched to its left or right neighbor, or to itself (no matching). Thus, we define the uninterpreted relation

$T_p \subseteq \{l,s,r\} \times \{l,s,r\} \times \{l,s,r\} \times \{l,s,r\}$. Function f maps each process to the values of matching of its right neighbor, itself, and left neighbor, and function g maps each process to value of the only variable in the write-set. For each process, we add an instance of Constraint 3.25 to the SMT instance. For example, for process π_0, the following constraint is added to the SMT instance:

$$\forall s_0, s_1 \in S \ : \ (s_0, s_1) \in T_0 \iff \\ ((match_4\_val(s_0), match_0\_val(s_0), match_1\_val(s_0)), match_0\_val(s_1)) \in T_p$$

3.6.5 Constraints for Problem Statements 2 & 3

In this section, we present the constraints that are added in the case of problem statements 2 and 3.

3.6.5.1 Local Predicates Constraints

Let LP be the set of uninterpreted predicates used in formulas φ and ψ. For each uninterpreted local predicate lp_π, we need to ensure that its interpretation function is a function of the variables in the read-set of π. To guarantee this requirement, for each $lp_\pi \in LP$, we add the following constraint to the SMT instance:

$$\forall s, s' \in S : \ (\forall v \in R_\pi : (v(s) = v(s')) \implies (lp_\pi(s) = lp_\pi(s'))$$

Example 3.6.6. As an example, in Example 3.3.1, we add the following constraint for process π_1:

$$\forall s, s' \in S : \ (x_0(s) = x_0(s')) \wedge (x_1(s) = x_1(s')) \wedge (x_2(s) = x_2(s')) \implies (tk_1(s) = tk_1(s'))$$

Before presenting the rest of constraints, we present the formulation of an $\mathrm{LTL_R}$ formula as an SMT constraint. We use this formulation to encode the ψ and φ formulas (given as input) as ψ_{SMT} and φ_{SMT}, and add them to the SMT instance (as discussed in Sections 3.6.5.3 and 3.6.5.4).

3.6.5.2 SMT Formulation of an $\mathrm{LTL_R}$ Formula

An $\mathrm{LTL_R}$ formula consists of a set of predicates, logical operators, and temporal operators. Formulation of predicates and logical operators is straightforward in an SMT instance. Below, we discuss the formulation of the two temporal operators:

3.6.5.2.1 SMT formulation of $\mathbf{X}$ Operator: A formula of the form $\mathbf{X}\,P$ is translated to an SMT constraint as below:

$$\forall s, s' \in S \; : \; (s, s') \in T_{\mathcal{D}} \implies P(s') \tag{3.26}$$

3.6.5.2.2 SMT formulation of $\mathbf{U}$ Operator: Inspired by *bounded synthesis* [42], for each formula of the form $P\,\mathbf{U}\,Q$, we define an uninterpreted function $\gamma_i : S \mapsto \mathbb{N}$ and add the following constraints to the set of SMT constraints:

$$\forall s, s' \in S \; : \; \neg Q(s) \;\wedge\; (s, s') \in T_{\mathcal{D}} \implies (P(s) \;\wedge\; \gamma_i(s') > \gamma_i(s)) \tag{3.27}$$

$$\forall s \in S \; : \; \neg Q(s) \implies \exists s' \in S \; : \; (s, s') \in T_{\mathcal{D}} \tag{3.28}$$

The intuition behind Constraints 3.27 and 3.28 can be understood easily. If we can assign a natural number to each state, such that along each outgoing transition from a state in $\neg Q$, the number is strictly increasing, then the path from each state in $\neg Q$ should finally reach Q or get stuck in a state, since the size of state space is finite. Also, there cannot be any loops whose states are all in $\neg Q$, as imposed by the annotation function. Finally, Constraint 3.28 ensures that there is no deadlock state in $\neg Q$ states.

3.6.5.3 Synthesis of Self-Stabilizing Systems with Implicit LS

In this section, we present the constraints specific to Problem Statement 1. As mentioned in Section 3.5, one of the inputs in problem statement 2 is an LTL_R formula, φ describing the role of uninterpreted predicates. Considering φ_{SMT} to be the SMT formulation of φ, we add the following SMT constraint to the SMT instance:

$$\forall s \in S \; : \; \varphi_{SMT} \tag{3.29}$$

Another input to our problem is the LTL_R formula, ψ that includes requirements, which should hold in the set of legitimate states. We formulate this formula as an SMT constraint using the method discussed in 3.6.5.2.2. Considering ψ_{SMT} to be the SMT formulation of the ψ formula, we add the following SMT constraint to the SMT instance:

$$\forall s \in S \; : \; LS(s) \implies \psi_{SMT} \tag{3.30}$$

Example 3.6.7. Continuing with Example 3.3.1, we add the following constraints to encode $\varphi_{\mathbf{TR}}$:

$$\forall s \in S \; : \; \forall i \in \{0 \cdots n-1\} : tk_i(s) \iff (\forall s' \in S \; : \; (s, s') \in T_{\mathcal{D}} \Rightarrow x_i(s) \neq x_i(s'))$$

The other requirements of the token ring problem are $\psi_{\mathbf{safety}}$ and $\psi_{\mathbf{fairness}}$, which should hold in the set of legitimate states. To guarantee them, the following SMT constraints are added to the SMT instance:

$$\forall s \in S \ : \ LS(s) \implies (\exists i \in \{0 \cdots n-1\} : (tk_i(s) \ \wedge \ \forall j \neq i : \neg tk_j(s)))$$
$$\forall s \in S \ : \ LS(s) \implies \forall i \in \{0 \cdots n-1\} : (tk_i(s) \wedge (s, s') \in T_{\mathcal{D}}) \Rightarrow tk_{(i+1 \bmod n)}(s')$$

3.6.5.4 Synthesis of Ideal-Stabilizing Systems

We now present the constraints specific to problem statement 3. The only such constraints is related to the two LTL_R formulas φ and ψ. To this end, we add the following to tour SMT instance:

$$\forall s \in S \ : \ \varphi_{SMT} \wedge \psi_{SMT} \tag{3.31}$$

Example 3.6.8. For Example 3.3.2, to ensure $\varphi_{\mathbf{ME}}$ and $\psi_{\mathbf{ME}}$ we add the following constraint to the SMT instance:

$$\forall s \in S \ : \ \forall i \in \{0 \cdots n-1\} : tk_i(s) \iff (\forall s' \in S \ : \ (s, s') \in T_{\mathcal{D}} \Rightarrow c_i(s) \neq c_i(s'))$$

$\varphi_{\mathbf{ME}}$ is guaranteed by adding the following two constraints to the SMT instance:

$$\forall s, s' \in S \ : \ \forall i \in \{0, \ldots, |\Pi_{\mathcal{T}}| - 1\} \ : \ \neg tk_i(s) \ \wedge \ (s, s') \in T_{\mathcal{D}} \implies \gamma_i(s') > \gamma_i(s)$$
$$\forall s \in S \ : \ \forall i \in \{0, \ldots, |\Pi_{\mathcal{T}}| - 1\} \ : \ \neg tk_i(s) \implies \exists s' \in S \ : \ (s, s') \in T_{\mathcal{D}}$$

Note that adding two constraints to an SMT instance is equivalent to adding their conjunction.

3.7 Case Studies and Experimental Results

We evaluate our synthesis method using several case studies from well-known distributed self-stabilizing problems. We consider cases where synthesis succeeds and cases where synthesis fails to find a solution for the given topology. Failure of synthesis is normally due to impossibility of self-stabilization for certain problems. We emphasize that although our case studies deal with synthesizing a small number of processes (due to high complexity of synthesis), having access to a solution for a small number of processes can give key insights to designers of self- or ideal-stabilizing protocols to generalize the protocol for any number of processes. For example, our method can be applied in cases where there exists a *cut-off* *point* [51]. We should also mention that the maximum number of processes in the system we could synthesize differs from problem to problem. This number solely depends on the complexity of the input specification and, hence, the SMT instance. That means there is no fixed maximum number of processes that this method can handle. We note that the size of the SMT instance grows linearly with the number of processes for each case study. This is because the number of entities (i.e., variables, relations, etc) and constraints (e.g., read-write restrictions as well as temporal) grow linearly in the number of processes. Note that concrete states do not appear in an SMT instance. As described in Section 3.6, we only include a set S of state whose size is known.

We used the Alloy [49] model finder tool for our experiments. Alloy solver performs the relational reasoning over quantifiers, which means that we did not have to unroll quantifiers over their domains. All experiments in this section are run on a machine with Intel Core i5 2.6 GHz processor with 8GB of RAM. We conducted experiments using Z3 [5] and Yices [6] SMT solvers as well. The main reason that we used Alloy is that it simply shows better performance than Z3 and Yices in the majority of our case studies. While the reason is unknown to us, we believe it can be due to the fact that our problem is to find a transition *relation* that satisfies the constraints of a self-stabilizing system and Alloy seems to work better for relational models. Unfortunately, the internals of off-the-shelf SMT solvers remain a mystery, as such solvers use many heuristics and even randomizations, which we cannot explain their internals. We note that since our synthesis method and its implementation in Alloy is deterministic, we do not replicate experiments for statistical confidence[7].

3.7.1 Case Studies for Problem Statement 1

In this section, we present the case studies we have conducted for our solution to problem statement 1.

3.7.1.1 Maximal Matching

Our first case study is our running example, distributed self-stabilizing *maximal matching* [48, 68, 76]. Recall that each process maintains a *match* variable with domain of all its neighbors and an additional value $\perp$ that indicates the process is not matched to any of its

[5]http://research.microsoft.com/en-us/um/redmond/projects/z3/

[6]http://yices.csl.sri.com

[7]The reader can access the Alloy models at http://www.cas.mcmaster.ca/borzoo/Publications/15/TAAS/Alloy.zip.

neighbors. The set of legitimate states is the disjunction of all possible maximal matchings on the given topology. As an example, for the graph shown in Fig. 3.1, we have:

$$
\begin{aligned}
&(match_0(s) = 1 \ \wedge \ match_1(s) = 0 \ \wedge \ match_2(s) = \perp) \ \vee \\
&(match_0(s) = \perp \ \wedge match_1(s) = 2 \ \wedge \ match_2(s) = 1)
\end{aligned}
$$

Table 3.1 presents our results for different sizes of line and star topologies. Obviously, such topologies are inherently asymmetric. As expected, by increasing the number of processes, synthesis time also increases. Another observation is that synthesizing a solution for the star topology is in general faster than the line topology. This is because a protocol that intends to solve maximal matching for the star topology deals with a significantly smaller problem space. This is due to the fact that in a star topology, regardless of the size of the network, processes can only match to the process in the center. Also, synthesizing a weak-stabilizing protocol is faster than a self-stabilizing protocol, as the former has more relaxed constraints. The synthesized model (represented as *guarded commands*) for the case of 3 processes, with line topology, strong self-stabilization, and asynchronous timing model is as follows:

$$
\begin{array}{lrcl}
\pi_0 \ : & match_0 = \perp \ \wedge \ match_1 = 0 & \rightarrow & match_0 := 1 \\
 & match_0 = 1 \ \wedge \ match_1 = 2 & \rightarrow & match_0 := \perp \\
\pi_1 \ : & match_0 = \perp \ \wedge \ match_1 = \perp \wedge \ match_2 = 1 & \rightarrow & match_1 := 2 \\
 & match_0 = \perp \ \wedge \ match_1 = 2 \wedge \ match_2 = \perp & \rightarrow & match_1 := 0 \\
 & match_0 = 1 \ \wedge \ match_1 = \perp \wedge \ match_2 = \perp & \rightarrow & match_1 := 0 \\
 & match_0 = 1 \ \wedge \ match_1 = \perp \wedge \ match_2 = 1 & \rightarrow & match_1 := 0 \\
 & match_0 = 1 \ \wedge \ match_1 = 2 \wedge \ match_2 = \perp & \rightarrow & match_1 := \perp \\
\pi_2 \ : & match_1 = \perp \ \wedge \ match_2 = \perp & \rightarrow & match_2 := 1 \\
 & match_1 = 0 \ \wedge \ match_2 = 1 & \rightarrow & match_2 := \perp
\end{array}
$$

The synthesized model is depicted in Fig. 3.3.

3.7.1.2 Dijkstra's Token Ring with Three-State Machines

In the *token ring* problem, a set of processes are placed on a ring network. Each process has a so-called privilege (token), which is a Boolean function of its neighbors' and its own states. When this function is true, the process has the privilege.

Dijkstra [29] proposed three solutions for the token ring problem. In the *three-state token ring*, each process π_i maintains a variable x_i with domain $\{0, 1, 2\}$. The read-set of a

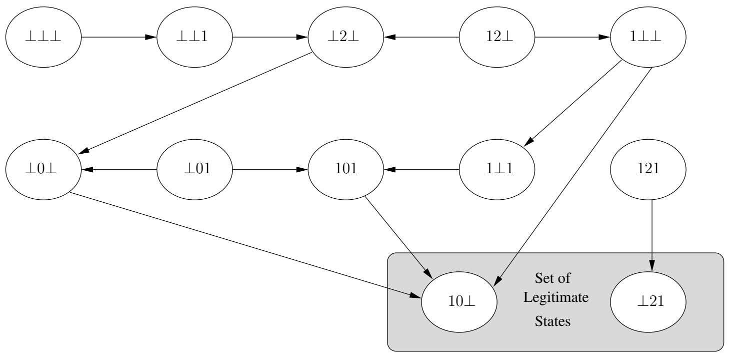


Figure 3.3: Synthesized model of maximal matching for line topology of size 3.

Topology	**# of Processes**	**Self-Stabilization**	**Timing Model**	**Time (sec)**
line	3	strong	asynchronous	0.16
line	3	strong	synchronous	0.44
line	4	strong	synchronous	5.18
line	4	weak	synchronous	3.29
line	5	weak	synchronous	340.62
star	4	strong	asynchronous	2.95
star	4	weak	asynchronous	2.93
star	5	strong	asynchronous	53.75
star	5	weak	asynchronous	41.80

Table 3.1: Results for synthesizing maximal matching for line and star topologies.

process is its own and its neighbors' variables, and its write-set contains its own variable. As an example, for process π_1, $R_{\mathcal{T}}(1) = \{x_0, x_1, x_2\}$ and $W_{\mathcal{T}}(1) = \{x_1\}$. Token possession is formulated using the conditions on a machine and its neighbors [29]. Briefly, in a state s, process π_0 (called the *bottom* process) has the token, when $x_0(s) + 1 \mod 3 = x_1(s)$, process $\pi_{(|\Pi_{\mathcal{T}}|-1)}$ (called the *top* process) has the token, when $(x_0(s) = x_{(|\Pi_{\mathcal{T}}|-2)}(s)) \wedge (x_{(|\Pi_{\mathcal{T}}|-2)}(s) + 1 \mod 3 \neq x_{(|\Pi_{\mathcal{T}}|-1)}(s))$, and any other process π_i owns the token, when either $x_i(s)+1 \mod 3$ equals to the x-value of its left or right process (i.e., x_{i-1} or $xi + 1$). The set of legitimate states are those in which exactly one process has the token. For example, for a ring of size three, the set of legitimate states is formulated by the following expression:

$$
\begin{aligned}
&(((x_0(s)+1 \bmod 3 = x_1(s)) \;\wedge\; (x_1(s)+1 \bmod 3 \neq x_2(s)))) \vee \\
&(((x_1(s) = x_0(s)) \;\wedge\; (x_1(s)+1 \bmod 3 \neq x_2(s)))) \vee \\
&((x_1(s)+1 \bmod 3 = x_0(s)) \;\wedge\; ((x_1(s)+1 \bmod 3 \neq x_2(s)))) \vee \\
&((x_0(s)+1 \bmod 3 \neq x_1(s)) \;\wedge\; (x_1(s)+1 \bmod 3 \neq x_0(s)) \wedge (x_1(s)+1 \bmod 3 = x_2(s)))
\end{aligned}
$$

Table 3.2 presents the result for synthesizing solutions for the three-state version. Obviously, symmetry is not studied, because the top and bottom processes do not behave similar to other processes. We note that the synthesized strong stabilizing programs using our technique are identical to Dijkstra's solution in [29]. Also, notice that the time needed to synthesize weak and strong stabilizing solutions for the same number of process is almost identical. This is due to the fact that the search space for solving the corresponding SMT instances are of the same complexity. We have also synthesized of a model given the fault-free scenario (i.e., transitions that start in LS) in Dijkstra's solution. The constraint for transitions that start in LS are as follows:

$$
\begin{aligned}
&\forall s \in S \;: ((x_0(s)+1 \bmod 3 = x_1(s)) \;\wedge\; (x_1(s)+1 \bmod 3 \neq x_2(s))) \implies \\
&\qquad (s, s_{(x_0 \leftarrow (x_0-1) \bmod 3)}) \in T_{\pi_0} \\
&\forall s \in S \;: \neg((x_0(s)+1 \bmod 3 = x_1(s)) \;\wedge\; (x_1(s)+1 \bmod 3 \neq x_2(s))) \implies \\
&\qquad \nexists s' \in S \;:\; (s, s') \in T_{\pi_0} \\
&\forall s \in S \;: ((x_1(s)+1 \bmod 3 = x_0(s)) \;\wedge\; ((x_1(s)+1 \bmod 3 \neq x_2(s)))) \implies \\
&\qquad (s, s_{(x_1 \leftarrow x_0)}) \in T_{\pi_1} \\
&\forall s \in S \;:\; ((x_0(s)+1 \bmod 3 \neq x_1(s)) \;\wedge\; (x_1(s)+1 \bmod 3 \neq x_0(s)) \wedge \\
&\qquad (x_1(s)+1 \bmod 3 = x_2(s))) \implies (s, s_{(x_1 \leftarrow x_2)}) \in T_{\pi_1} \\
&\forall s \in S \;:\; \neg(((x_1(s)+1 \bmod 3 = x_0(s)) \;\wedge\; ((x_1(s)+1 \bmod 3 \neq x_2(s)))) \vee \\
&\qquad ((x_0(s)+1 \bmod 3 \neq x_1(s)) \;\wedge\; (x_1(s)+1 \bmod 3 \neq x_0(s)) \wedge \\
&\qquad (x_1(s)+1 \bmod 3 = x_2(s)))) \implies \nexists s' \in S \;:\; (s, s') \in T_{\pi_1} \\
&\forall s \in S \;: ((x_1(s) = x_0(s)) \;\wedge\; (x_1(s)+1 \bmod 3 \neq x_2(s))) \implies \\
&\qquad (s, s_{(x_2 \leftarrow (x_1+1) \bmod 3)}) \in T_{\pi_2} \\
&\forall s \in S \;: \neg((x_1(s) = x_0(s)) \;\wedge\; (x_1(s)+1 \bmod 3 \neq x_2(s))) \implies \\
&\qquad \nexists s' \in S \;:\; (s, s') \in T_{\pi_2}
\end{aligned}
$$

where $\leftarrow$ denotes the assignment operator, and the subscript for the state s represents the state obtained by the given assignments in s. This constraint stipulates the behavior of the protocol in the absence of faults (i.e., in each state, only one process can execute and each process eventually get the chance to execute; mutual exclusion and non-starvation).

# of Proc.	Self-Stabilization	Timing Model	Symmetry	*LS* actions	Time (sec)
3	strong	asynchronous	asymmetric		1.26
3	strong	asynchronous	asymmetric	✓	4.68
3	weak	asynchronous	asymmetric		1.06
4	strong	asynchronous	asymmetric		63.02
4	strong	asynchronous	asymmetric	✓	225.54
4	weak	asynchronous	asymmetric		62.13

Table 3.2: Results for synthesizing three-state token ring

Comparing the results with and without given *LS* transition shows that the synthesis time increases when adding the constraints for the transitions inside the legitimate states. This may not be always the case, since adding constraints in some SMT solvers can limit the search space. In the case of given *LS* actions with strong self-stabilization, we can synthesize Dijkstra's protocol with the following actions [29]:

$$\begin{array}{llrcl}
\pi_0 & : & ((x_0 + 1) \bmod 3 = x_1) & \rightarrow & x_0 := (x_0 - 1) \bmod 3 \\
\pi_1 & : & ((x_1 + 1) \bmod 3 = x_0) & \rightarrow & x_1 := x_0 \\
 & & ((x_1 + 1) \bmod 3 = x_2) & \rightarrow & x_1 := x_2 \\
\pi_2 & : & (x_1 = x_0) \wedge ((x_1 + 1) \bmod 3 \neq x_2) & \rightarrow & x_2 := (x_1 + 1) \bmod 3
\end{array}$$

3.7.1.3 Dijkstra's Token Ring with Four-State Machines

In this Subsection, we consider Dijkstra's *four-state machine* solution for token ring [29]. Each process π_i has two Boolean variables; x_i and up_i, where $up_0 = true$ and $up_{(|\Pi_\mathcal{T}|-1)} = false$. Process π_0 is called the *bottom* and process $\pi_{(|\Pi_\mathcal{T}|-1)}$ is called the *top* process. The read-set and write-set of a process is similar to the three-state case in Section 3.7.1.2. Token possession is defined based on the variables of a process and its neighbors. Briefly, in a state, say s, the bottom process has the token, if $(x_0(s) = x_1(s)) \wedge (\neg up_1(s))$, the top process has the token, if $x_{(|\Pi_\mathcal{T}|-1)}(s) \neq x_{(|\Pi_\mathcal{T}|-2)}(s)$, and the condition for any other process π_i is $(x_i(s) \neq x_{(i-1)}(s)) \vee (x_i(s) = x_{(i+1)}(s) \wedge up_i(s) \wedge \neg up_{(i+1)}(s))$. The legitimate states are those where exactly one process has the token. For example, for a ring of three processes, LS is the defined by the following expression:

$$\begin{array}{l}((x_0(s) = x_1(s)) \wedge \neg up_1(s) \ \wedge (x_2(s) = x_1(s))) \ \vee \\ ((x_0(s) = x_1(s)) \wedge up_1(s) \ \wedge (x_2(s) \neq x_1(s))) \ \vee \\ (\neg up_1(s) \wedge (x_0(s) \neq x_1(s)) \ \wedge (x_2(s) = x_1(s))) \ \vee \\ (up_1(s) \wedge (x_0(s) = x_1(s)) \ \wedge (x_2(s) = x_1(s)))\end{array}$$

Our results on token ring with four-state machines are presented in Table 3.3. The table includes synthesis time when LS actions in Dijkstra's solution are given as input. Similar to the previous case study, the synthesis time has increased by adding the constraints for the LS actions. The constraints for transitions that start in LS are as follows:

$$\begin{array}{l}\forall s \in S \ : ((x_0(s) = x_1(s)) \wedge \neg up_1(s) \ \wedge (x_2(s) = x_1(s))) \implies (s, s_{(x_0 \leftarrow \neg x_0)}) \in T_{\pi_0} \\ \forall s \in S \ : \neg((x_0(s) = x_1(s)) \wedge \neg up_1(s) \ \wedge (x_2(s) = x_1(s))) \implies \nexists s' \in S \ : \ (s, s') \in T_{\pi_0} \\ \forall s \in S \ : (\neg up_1(s) \wedge (x_0(s) \neq x_1(s)) \ \wedge (x_2(s) = x_1(s))) \implies (s, s_{(x_1 \leftarrow \neg x_1, up_1 \leftarrow true)}) \in T_{\pi_1} \\ \forall s \in S \ : (up_1(s) \wedge (x_0(s) = x_1(s)) \ \wedge (x_2(s) = x_1(s))) \implies (s, s_{(up_1 \leftarrow false)}) \in T_{\pi_1} \\ \forall s \in S \ : \ \neg((\neg up_1(s) \wedge (x_0(s) \neq x_1(s)) \ \wedge (x_2(s) = x_1(s))) \vee \\ \qquad (up_1(s) \wedge (x_0(s) = x_1(s)) \ \wedge (x_2(s) = x_1(s)))) \implies \nexists s' \in S \ : \ (s, s') \in T_{\pi_1} \\ \forall s \in S \ : ((x_0(s) = x_1(s)) \wedge up_1(s) \ \wedge (x_2(s) \neq x_1(s))) \implies (s, s_{(x_2 \leftarrow \neg x_2)}) \in T_{\pi_2} \\ \forall s \in S \ : \neg((x_0(s) = x_1(s)) \wedge up_1(s) \ \wedge (x_2(s) \neq x_1(s))) \implies \nexists s' \in S \ : \ (s, s') \in T_{\pi_2}\end{array}$$

In the case of given LS actions with strong self-stabilization, we can synthesize Dijkstra's protocol with the following actions [29]:

$$\begin{array}{lrcl}\pi_0 \ : & (x_0 = x_1) \ \wedge \ \neg up_1 & \rightarrow & x_0 := \neg x_0 \\ \pi_1 \ : & (x_1 \neq x_0) & \rightarrow & x_1 := \neg x_1 \ ; \ up_1 := true \\ & (x_1 = x_2) \ \wedge \ up_1 \ \wedge \ \neg up_2 & \rightarrow & up_1 := false \\ \pi_2 \ : & (x_2 \neq x_1) & \rightarrow & x_2 := \neg x_2\end{array}$$

3.7.1.4 Token Circulation in Anonymous Networks

Token circulation in a *unidirectional* ring is one of the most studied self-stabilizing problems. Herman [47] showed that there is no non-probabilistic self-stabilizing algorithm for this problem in an anonymous network. In [28], Devismes et. al. proposed a weak self-stabilizing solution for this problem. We assume a similar topology to the one used in [28].

# of Proc.	Self-Stabilization	Timing Model	Symmetry	*LS* actions	Time (sec)
3	strong	asynchronous	asymmetric		0.86
3	strong	asynchronous	asymmetric	✓	44.71
3	weak	asynchronous	asymmetric		0.33
4	strong	asynchronous	asymmetric		30.32
4	strong	asynchronous	asymmetric	✓	51.79
4	weak	asynchronous	asymmetric		29.16

Table 3.3: Results for synthesizing four-state token ring

In a ring of size $|\Pi_{\mathcal{T}}|$, each process π_i has a variable dt_i with the domain $\{0, \ldots, m_{(|\Pi_{\mathcal{T}}|)}-1\}$, where $m_{(|\Pi_{\mathcal{T}}|)}$ is the smallest integer not dividing $|\Pi_{\mathcal{T}}|$. The read-set of a process is its own variable and the variable of its left neighbor, and its write-set contains its own variable. A process holds a token, if and only if $dt_i \neq dt_l + 1 \mod m_{(|\Pi_{\mathcal{T}}|)}$, where dt_l represents the variable of the left neighbor. A legitimate state is one where exactly one process has the token. For example, for a ring of size 3, *LS* can be formulated by the following expression:

$$\begin{aligned}
&(\ \neg(dt_0(s) = dt_2(s) + 1 \text{ mod } 2)\ \wedge (dt_1(s) = dt_0(s) + 1 \text{ mod } 2)\\
&\quad \wedge (dt_2(s) = dt_1(s) + 1 \text{ mod } 2)\ )\ \vee\\
&(\ \neg(dt_1(s) = dt_0(s) + 1 \text{ mod } 2)\ \wedge (dt_0(s) = dt_2(s) + 1 \text{ mod } 2)\\
&\quad \wedge (dt_2(s) = dt_1(s) + 1 \text{ mod } 2)\ )\ \vee\\
&(\ \neg(dt_2(s) = dt_1(s) + 1 \text{ mod } 2) \wedge (dt_0(s) = dt_2(s) + 1 \text{ mod } 2)\\
&\quad \wedge (dt_1(s) = dt_0(s) + 1 \text{ mod } 2)\ )
\end{aligned}$$

As can be seen in Table 3.4, for 3 processes, synthesizing a symmetric algorithm for strong self-stabilization is possible. However, For 4 and 5 processes, Alloy returns "unsatisfiable", which shows the impossibility of strong self-stabilizing for these topologies. For 4 and 5 processes, our method synthesized the same weak-stabilizing algorithm as the one proposed in [28], with the following action:

$$(dt \neq dt_l + 1 \text{ mod } m_{(|\Pi_{\mathcal{T}})}) \quad \rightarrow \quad dt := (dt_l + 1 \text{ mod } m_{(|\Pi_{\mathcal{T}})})$$

Using the "next instance" feature of Alloy, we could also synthesize another protocol for 4 processes in less than 3 seconds. We note that the size of the state space for 5 processes is less than the size for 4 process, as $m_{(|\Pi_{\mathcal{T}}|)}$ is 3 for 4, while it equals 2 for 5 processes. This is the reason why the synthesis time has decreased from 4 to 5 processes. Also, unlike the previous case study, synthesizing an asymmetric program in an anonymous network is not reasonable, because otherwise the network would lose its anonymity.

# of Processes	Self-Stabilization	Timing Model	Symmetry	Time (sec)
3	strong	asynchronous	symmetric	1.59
4	weak	asynchronous	symmetric	114.56
5	weak	asynchronous	symmetric	10.83

Table 3.4: Results for synthesizing token circulation in anonymous networks

3.7.1.5 Maximal Matching on Rings

Another case study is the maximal matching problem for ring topologies [45]. This is a special case of the case study in Section 3.7.1.1, where the topology allows synthesizing symmetric solutions. The *match* variable for each process has domain $\{l, r, s\}$, where values l and r represent matching with left and right processes, respectively, and value s shows that the process is self-matched. For example, the set of legitimate states for a ring of size 3 is defined by the following predicate:

$$\begin{aligned}&(match_0(s) = r \wedge match_1(s) = l \wedge match_2(s) = s) \vee\\&(match_0(s) = s \wedge match_1(s) = r \wedge match_2(s) = l) \vee\\&(match_0(s) = l \wedge match_1(s) = s \wedge match_2(s) = r)\end{aligned}$$

Table 3.5 shows the results of our experiments. Note that although the topology is symmetric, the synthesized protocol can be symmetric or asymmetric. We observe that synthesizing an asymmetric protocol is faster than a symmetric protocol, since for the latter, the SMT-solver has to search deeper in the state space to rule out asymmetric solutions. In other words, there exist more asymmetric protocols for the given input. One of the synthesized models for strong self-stabilization with asynchronous timing model in the symmetric case that works for both 3 and 4 processes is as follows (The model is represented by the set of actions for each process, and $match_l$ and $match_r$ refer to the variables of the left and right processes of the process, respectively):

# of Processes	Self-Stabilization	Timing Model	Symmetry	Time (sec)
3	strong	asynchronous	symmetric	1.53
3	weak	asynchronous	symmetric	1.85
4	strong	synchronous	asymmetric	106.6
4	strong	synchronous	symmetric	139.35
4	strong	asynchronous	symmetric	83.19
4	weak	asynchronous	symmetric	64.13
4	weak	asynchronous	asymmetric	42.29

Table 3.5: Results for synthesizing the maximal matching problem on a ring

$$\begin{array}{rcl}
(match = l) \wedge (match_l = l) \wedge (match_r = l) & \rightarrow & match := r \\
(match = l) \wedge (match_l = l) \wedge (match_r \neq l) & \rightarrow & match := s \\
(match = r) \wedge (match_l = l) \wedge (match_r = s) & \rightarrow & match := s \\
(match = r) \wedge (match_l = r) & \rightarrow & match := l \\
(match = r) \wedge (match_l = s) \wedge (match_r = s) & \rightarrow & match := l \\
(match = s) \wedge (match_l \neq r) \wedge (match_r = l) & \rightarrow & match := r \\
(match = s) \wedge (match_l = s) \wedge (match_r \neq l) & \rightarrow & match := l
\end{array} \tag{3.32}$$

Note that since the synthesized model is symmetric, we have similar set of actions for all processes, and hence, the process indexes are not specified for individual actions.

3.7.1.6 The Three-Coloring Problem

In the *three coloring problem* [45], we have a set of processes connected in a ring topology. Each process π_i has a variable c_i, with the domain $\{0, 1, 2\}$. Each value of the variable c_i represents a distinct color. A process can read and write its own variable. It can also read, but not write the variables of its left and right processes. For example, in a ring of four processes, the read-set and write-set of π_0 are $R_{\mathcal{T}}(0) = \{c_3, c_0, c_2\}$ and $W_{\mathcal{T}}(0) = \{c_0\}$, respectively. The set of legitimate states is those where each process has a color different from its left and right neighbors. Thus, for a ring of four processes, LS is defined by the following predicate:

$$\neg(c_0(s) = c_1(s) \ \vee \ c_1(s) = c_2(s) \ \vee \ c_2(s) = c_3(s) \ \vee \ c_3(s) = c_0(s))$$

# of Processes	Self-Stabilization	Timing Model	Symmetry	Time (sec)
3	strong	synchronous	asymmetric	0.56
3	strong	asynchronous	asymmetric	0.93
3	weak	synchronous	asymmetric	0.55
3	weak	asynchronous	symmetric	1.33
4	strong	synchronous	asymmetric	78.71
4	weak	synchronous	asymmetric	96.32
4	strong	asynchronous	asymmetric	35.09
4	strong	asynchronous	symmetric	60.35

Table 3.6: Results for synthesizing three-coloring

Our synthesis results for the three coloring problem are reported in Table 3.6. The results in this case study and the previous ones show that synthesizing asynchronous systems are generally faster compared to the synchronous ones, although we cannot claim if this is always the case. One of the synthesized models for strong self-stabilization with asynchronous timing model in the symmetric case that works for both 3 and 4 processes is as follows (The model is represented by the set of actions for each process, and c_l and c_r refer to the variables of the left and right processes of the process, respectively):

$$\begin{array}{lcl}
(c = 1) \wedge (c_l = 1) \wedge (c_r \neq 0) & \rightarrow & c := 0 \\
(c = 1) \wedge (c_l = 1) \wedge (c_r = 0) & \rightarrow & c := 2 \\
(c = 2) \wedge (c_l \neq 0) \wedge (c_r = 2) & \rightarrow & c := 0 \\
(c = 2) \wedge (c_l = 0) \wedge (c_r = 2) & \rightarrow & c := 1 \\
(c = 0) \wedge (c_l = 0) \wedge (c_r = 1) & \rightarrow & c := 2 \\
(c = 0) \wedge (c_l = 0) \wedge (c_r \neq 1) & \rightarrow & c := 1
\end{array}$$

3.7.1.7 One-bit Maximal Matching

One-bit maximal matching is a special case of maximal matching on a ring, where each process has only one Boolean variable x_i. Each process can read and write its own variable. It can also read, but not write the variables of its neighbors. A state is in LS, if and only if the following predicate holds for each process π_i:

$$(x_{(i-1)} \wedge \neg x_i) \;\vee\; (\neg x_{(i-1)} \wedge \neg x_i \wedge x_{(i+1)}) \;\vee\; (\neg x_{(i-1)} \wedge x_i \wedge \neg x_{(i+1)})$$

# of Processes	Self-Stabilization	Timing Model	Symmetry	Time (sec)
3	strong	asynchronous	asymmetric	0.61
3	weak	asynchronous	asymmetric	0.15
4	strong	asynchronous	asymmetric	0.92
4	weak	asynchronous	asymmetric	2.58
5	strong	asynchronous	asymmetric	7.33
5	weak	asynchronous	asymmetric	8.95

Table 3.7: Results for synthesizing one-bit maximal matching

Note that in the first clause, π_i is matched to its left neighbor, in the second clause, it is matched to itself, and in the last one, it is matched to its right neighbor. Our synthesis results for this problem are reported in Table 3.7. The actions for the synthesized strong self-stabilizing model in the case of 3 processes with asynchronous timing model are as follows:

$$\begin{array}{lrcl}
\pi_0 : & x_0 \wedge \neg x_1 \wedge x_2 & \rightarrow & x_0 := \neg x_0 \\
 & \neg x_0 \wedge \neg x_1 \wedge \neg x_2 & \rightarrow & x_0 := \neg x_0 \\
\pi_1 : & x_0 \wedge x_1 \wedge \neg x_2 & \rightarrow & x_1 := \neg x_1 \\
\pi_2 : & x_0 \wedge x_1 \wedge x_2 & \rightarrow & x_2 := \neg x_2 \\
 & \neg x_0 \wedge x_1 \wedge x_2 & \rightarrow & x_2 := \neg x_2
\end{array}$$

3.7.1.8 The Issue of Completeness

In order to demonstrate the issue of completeness, we focus on synthesizing a program that the approach in [35] is not able to handle, but our approach can.

In the *simplified four-state token ring problem* [55], there is a set of processes connected in a ring topology. Each process has two Boolean variables $\{t_i, x_i\}$. Each process can read and write its own variables. It can also read, but not write the variables of its left neighbor. Process π_0 is said to have a token, when $t_{(|\Pi_{\mathcal{T}}|-1)} = t_0$. For each process π_i, when $i \neq 0$, the token condition is $t_{(i-1)} \neq t_i$. The set of legitimate states are those, where exactly one process has a token. For example, for a ring of three processes, the set legitimate of states can be represented by the following predicate:

$$(t_2 = t_0 \ \wedge \ t_0 = t_1 \ \wedge \ t_1 = t_2) \vee (t_2 \neq t_0 \ \wedge \ t_0 \neq t_1 \ \wedge \ t_1 = t_2) \vee (t_2 \neq t_0 \ \wedge \ t_0 = t_1 \ \wedge \ t_1 \neq t_2)$$

We have successfully synthesized a strong self-stabilizing model for the case of three processes with asynchronous timing model in 36.72 sec. One of the models we have synthesized

# of Processes	Self-Stabilization	Timing Model	Symmetry	Time (sec)
3	strong	asynchronous	asymmetric	4.21
3	weak	asynchronous	asymmetric	1.91
4	strong	asynchronous	asymmetric	71.19
4	weak	asynchronous	asymmetric	73.55
4	strong	asynchronous	symmetric	178.6

Table 3.8: Results for synthesizing Dijkstra's three-state token ring.

is the one presented in [55], with the following actions:

$$\begin{array}{llcl}
\pi_0 \ : & \neg t_2 \wedge \neg t_0 & \rightarrow & t_0 := \mathit{true} \\
& t_2 \wedge \neg x_2 \wedge t_0 & \rightarrow & t_0 := \mathit{false};\ x_0 := \neg x_0 \\
& t_2 \wedge x_2 \wedge t_0 \wedge x_0 & \rightarrow & t_0 := \mathit{false}; \\
\pi_i (i \neq 0) \ : & \neg t_{(i-1)} \wedge t_i & \rightarrow & t_i := \mathit{false};\ x_i := \neg x_i \\
& t_{(i-1)} \wedge x_{(i-1)} \wedge \neg t_i & \rightarrow & t_i := \mathit{true} \\
& t_{(i-1)} \wedge \neg x_{(i-1)} \wedge (\neg t_i \vee x_i) & \rightarrow & t_i := \mathit{true}\ x_i := \mathit{false}
\end{array}$$

3.7.2 Case Studies for Problem Statement 2

In this section, we present the case studies we have conducted for our solution to problem statement 2.

3.7.2.1 Self-stabilizing Token Ring

Synthesizing a self-stabilizing system for Example 3.3.1 leads to automatically obtaining Dijkstra [29] *three-state* algorithm in a bi-directional ring. Each process π_i maintains a variable x_i with domain $\{0, 1, 2\}$. The read-set of a process is its own and its neighbors' variables, and its write-set contains its own variable. For example, in case of four processes for π_1, $R_\mathcal{T}(1) = \{x_0, x_1, x_2\}$ and $W_\mathcal{T}(1) = \{x_1\}$. Token possession and mutual exclusion constraints follow Example 3.3.1. Table 3.8 presents our results for different input settings. In the case of symmetric, we synthesizes protocols with symmetric middle (not top nor bottom) processes.

We present one of the solutions we found for the token ring problem in ring of three processes. First, we present the interpretation functions for the uninterpreted local predicates.

$$tk_0 \leftrightarrow x_0 = x_2$$
$$tk_1 \leftrightarrow x_1 \neq x_0$$
$$tk_2 \leftrightarrow x_2 \neq x_1$$

Next, we present the synthesized transition relations for each process:

$$\begin{array}{llcl}
\pi_0 : & (x_0 = x_2) & \rightarrow & x_0 := (x_0 + 1) \bmod 3 \\
\pi_1 : & (x_1 \neq x_0) & \rightarrow & x_1 := x_0 \\
\pi_1 : & (x_2 \neq x_1) & \rightarrow & x_2 := x_1
\end{array}$$

Note that our synthesized solution is similar to Dijkstra's k-state solution, although our topology is similar to the topology for Dijkstra's three-state solution. We could not synthesize the three-state solution, as in this protocol, the token does not always circulate in one direction (it changes its circulation direction), but we have this constraint in $\psi_{\mathbf{fairenss}}$, as presented in Example 3.3.1.

3.7.2.2 Mutual Exclusion in a Tree

In the second case study, the processes form a directed rooted tree, and the goal is to design a self-stabilizing protocol, where at each state of LS, one and only one process is enabled. Each process π_j has a variable h_j with domain $\{i \mid \pi_i \text{ is a neighbor of } \pi_j\} \cup \{j\}$. If $h_j = j$, then π_j has the token. Otherwise, h_j contains the process id of one of the process's neighbors. The holder variable forms a directed path from any process in the tree to the process currently holding the token. The problem specification is the following:

Safety We assume each process π_i is associated with an uninterpreted local predicate tk_i, which shows whether π_i is enabled. Thus, mutual exclusion is the following formula:

$$\psi_{\mathbf{safety}} = \exists i \in \{0 \cdots n-1\} : (tk_i \ \wedge \ \forall j \neq i : \neg tk_j)$$

Fairness Each process π_i is eventually enabled:

$$\psi_{\mathbf{fairenss}} = \forall i \in \{0 \cdots n-1\} : (\mathbf{F}\, tk_i)$$

The formula, $\psi_{\mathbf{R}}$ given as input is $\psi_{\mathbf{R}} = \psi_{\mathbf{safety}} \wedge \psi_{\mathbf{fairenss}}$

# of Processes	Self-Stabilization	Timing Model	Time (sec)
3	strong	synchronous	0.84
4	strong	synchronous	16.07
4	weak	synchronous	26.8

Table 3.9: Results for synthesizing mutual exclusion on tree (Raymond's algorithm).

Using the above specification, we synthesized a self-stabilizing systems, which resembles Raymond's mutual exclusion algorithm on a tree [74]. Table 3.9 shows the experimental results.

We present one of our solutions for token circulation on tree, where there is a root with two leaves. The interpretation functions for the uninterpreted local predicates is as follows:

$$\forall i \ : \ tk_i \leftrightarrow h_i = i$$

Another part of the solution is the transition relation. Assume π_0 to be the root process, and π_1 and π_2 to be the two leaves of the tree. Hence, the variable domains are $D_{h_0} = \{0, 1, 2\}$, $D_{h_1} = \{0, 1\}$, and $D_{h_2} = \{0, 2\}$. Below, each state is represented as (h_0, h_1, h_2), and we use s to show that the process is pointing to itself, and p to represent that a process is pointing to its parent.

$$\begin{gathered}
(2, p, p) \rightarrow (1, s, s) \\
(s, s, p) \rightarrow (2, p, s) \\
(s, s, s) \rightarrow (2, p, s) \\
(2, s, p) \rightarrow (1, p, s) \\
(2, s, s) \rightarrow (s, p, p) \\
(1, p, p) \rightarrow (1, s, p) \\
(1, p, s) \rightarrow (1, s, p) \\
(1, s, p) \rightarrow (s, p, p) \\
(1, s, s) \rightarrow (s, p, p) \\
(s, p, s) \rightarrow (1, p, s) \\
(2, p, s) \rightarrow (1, s, p) \\
(1, s, p) \rightarrow (s, p, p) \\
(s, p, p) \rightarrow (2, p, s)
\end{gathered}$$

3.7.3 Case Studies for Problem Statement 3

In this section, we present the case studies we have conducted for our solution to problem statement 3.

3.7.3.1 Leader Election

In leader election, a set of processes choose a leader among themselves. Normally, each process has a subset of states in which it is distinguished as the leader. In a legitimate state, exactly one process is in its leader state subset, whereas the states of all other processes are outside the corresponding subset.

We consider line and tree topologies, where each process has a variable c_i, with the domain of two or three values. To synthesize an ideal-stabilizing system for the problem of leader election, we associate an uninterpreted local predicate l_i for each process π_i, whose value shows whether or not the process is in its leader state. Based on the required specification, in each state of the system, there is one and only one process π_i, for which $l_i = \mathit{true}$. More formally, the following $\mathrm{LTL}_{\mathrm{R}}$ specification should hold for a system of n processes:

$$\psi_{\mathbf{safety}} = \exists i \in \{0 \cdots n-1\} : (l_i \ \wedge \ \forall j \neq i : \neg l_j)$$

The results for this case study are presented in Table 3.9. In the topology column, the structure of the processes along with the domain of variables is reported. In the case of 4 processes on a line topology and tree/2-state, no solution is found. The time we report in the table for these cases are the time needed to report unsatisfiability by Alloy.

We present the solution for the case of three processes on a line, where each process π_i has a Boolean variable c_i. Since the only specification for this problem is state-based (safety), there is no constraint on the transition relations, and hence, we only present the interpretation function for each uninterpreted local predicate l_i.

$$\begin{aligned} l_0 &= (c_0 \wedge \neg c_1) \\ l_1 &= (\neg c_0 \wedge \neg c_1) \ \vee \ (c_1 \wedge \neg c_2) \\ l_2 &= (c_1 \wedge c_2) \end{aligned}$$

3.7.3.2 Local Mutual Exclusion

Our next case study is local mutual exclusion, as discussed in Example 3.3.2. We consider a line topology in which each process π_i has a Boolean variable c_i. The results for this case study are presented in Table 3.11.

# of Proc.	Timing Model	Topology	Time (sec)
3	asynchronous	line/2-state	0.034
4	asynchronous	line/2-state	0.73
4	asynchronous	line/3-state	115.21
4	asynchronous	tree/2-state	0.63
4	asynchronous	tree/3-state	12.39

Table 3.10: Results for synthesizing ideal stabilizing leader election.

# of Proc.	Timing Model	Symmetry	Time (sec)
3	asynchronous	asymmetric	0.75
4	asynchronous	asymmetric	24.44

Table 3.11: Results for synthesizing ideal stabilizing local mutual exclusion.

The solution we present for the local mutual exclusion problem corresponds to the case of four processes on a ring. Note that for each process π_i, when tk_i is true, the transition T_i changes the value of c_i. Hence, having the interpretation functions of tk_i, the definition of transitions T_i are determined as well. Below, we present the interpretation functions of the uninterpreted local predicates tk_i.

$$\begin{aligned}
tk_0 &= (c_0 \wedge c_1) \vee (\neg c_0 \wedge \neg c_1) \\
tk_1 &= (\neg c_0 \wedge c_1 \wedge c_2) \vee (c_0 \wedge \neg c_1 \wedge \neg c_2) \\
tk_2 &= (\neg c_1 \wedge c_2 \wedge \neg c_3) \vee (c_1 \wedge \neg c_2 \wedge c_3) \\
tk_3 &= (c_2 \wedge c_3) \vee (\neg c_2 \wedge \neg c_3)
\end{aligned}$$

Chapter 4

Synthesizing Self-stabilizing Protocols under Average Recovery Time Constraints

4.1 Introduction

In the past few years, there has been an active area of research on synthesizing stabilizing programs automatically from their formal specification. These efforts range over complexity analysis [54] to efficient synthesis heuristics design [35] as well as less efficient but complete techniques, as our approach introduced in Chapter 3. However, none of these techniques take into account requirements on the performance (e.g., maximum or average convergence time) of the generated program. In other words, the existing approaches only synthesize *some* solution that respects only closure and convergence. We argue that this is a serious shortcoming, as some quantitative metrics such as recovery time are as crucial as correctness in practice (e.g., in developing stabilizing network protocols). Furthermore, it is common knowledge that designing correct distributed stabilizing programs is a challenging task and prone to errors. Adding recovery time constraints to the design process makes it even more daunting.

With this motivation, we study the problem of repairing weak/strong-stabilizing programs under performance constraints. The constraint under investigation is, in particular, *average recovery time*. Following the work in [40, 41], we argue that average recovery time is a more descriptive metric than the traditional asymptotic complexity measure (e.g., the big O notation for the number of rounds) to characterize the performance of stabilizing programs. Average recovery time can be measured by giving weights to states and transitions of a stabilizing program and computing the expected value of the number of steps that it takes the program to reach a legitimate state. These weights can be assigned by a uniform distribution (in the simplest case), or by more sophisticated probability distributions. This technique has been shown to be effective in measuring the performance of weak-stabilizing programs as well, where not all computations converge [40], as well as cases where faults hit certain variables or locations more often. In this chapter, we show that the complexity of *repairing* an existing weak-stabilizing protocol to obtain either a weak or strong stabilizing protocol, so that (1) only removal of transitions is allowed during repair, and (2) the repaired protocol satisfies a certain average recovery time, is NP-complete.

4.2 Average Recovery Time of Stabilizing Programs

As discussed in Chapter 3, a self-stabilizing program is one that starting from any arbitrary initial state reaches a legitimate state in a finite number of steps. Such an arbitrary state may be reached due to wrong initialization, or occurrence of transient faults. Upon reaching a legitimate state, the system is guaranteed to remain in such states thereafter in the absence of faults. Design and proof of correctness of self-stabilizing algorithms can be very tedious or in some cases impossible, such as token circulation and leader election in anonymous networks [47]. Thus, weaker forms of stabilization were introduced in the literature. Convergence is a strong property since it should be satisfied in all computations. This property is weakened in *weak-stabilizing* distributed programs [44], where the existence of a converging computation or *possibility* of convergence suffices. The computation existence condition in weak convergence, unlike strong convergence, allows for execution cycles outside the set of legitimate states.

Let us call the number of steps that a stabilizing program takes to reach a legitimate state the *recovery time* (or *convergence time*).

Definition 32. *Let $\sigma = s_0 s_1 \cdots$ be a computation of a stabilizing program that starts from initial state s_0 and reaches a legitimate state in LS. The* recovery time *of σ is the following:*

$$RT(\sigma_{s_0}) = \min\{j \mid s_j \in LS\},$$

where s_j is the j^{th} state in σ. □

Since this metric depends both on the initial state of the program and the computation that reaches a legitimate state, we are interested in calculating the *average* recovery time of stabilizing programs. Following the techniques introduced in [40, 41], such average can be computed through statistical expected value of recovery time from different initial states and through various computations. Thus, we need to calculate the expected recovery time defined in Definition 32 for all initial states $s_0 \in \Sigma$ and take the probabilistic average of them accounting for the impact of different recovery computations and different starting points.

Calculating the expected value of a discrete random variable requires valid probabilities for the occurrence of the elements in its domain. Here, we take *recovery time*, RT, as a discrete random variable with domain $D_{RT} = [0, \infty)$. Our final goal is to find the probability of RT having a specific value i which requires probability values for transitions. Yet, the transition relation of a distributed program as defined in Definition 18 lacks probability distribution. Without loss of generality, we assume a *uniform distribution* over the set of outgoing transitions of each state in the state space. Such assumption, basically, makes the transition system of distributed programs a *Markov chain*. On that account, one can define the *probability of a computation* in a distributed program as follows.

Definition 33. *Given a distributed program $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ with state space Σ, the probability of a computation $\sigma = s_0 s_1 \cdots$ is computed by the following formula:*

$$\mu(\sigma_{s_0}) = \prod_{i=0}^{\infty} \mathbf{P}(s_i, s_{i+1}) \tag{4.1}$$

where $\mathbf{P} : \Sigma \times \Sigma \to [0, 1]$ is the transition probability function such that for all $s \in \Sigma$:

- $$\sum_{s' \in S} \mathbf{P}(s, s') = 1, \tag{4.2}$$

- *Given $T(s) = \{(s, s') \mid \exists s' \in \Sigma : (s, s') \in T_{\mathcal{D}}\}$, the set of outgoing transitions of state s, a* uniform *probability distribution over the transitions can be obtained by:*

$$\mathbf{P}(s, s') = \frac{1}{|T(s)|} \tag{4.3}$$

□

Equation 4.3 assigns equal probability to all transitions originating from a state. We emphasize that this uniform probability distribution was introduced only to permit computing weighted average for recovery time. If, however, the distribution over the transition relation was known (e.g., for probabilistic programs or particular schedulers in which processes work in a random fashion with specific probability distribution), Equation 4.3 could be modified trivially while satisfying Equation 4.2.

Notation 1. *Let* $\eth^c_{s,n}$ *denote all* converging computations *originating from state* s *with recovery time* n*. That is,*

$$\eth^c_{s,n} = \{\sigma_s \mid RT(\sigma_s) = n\}.$$

For a computation σ starting from state $s \in \Sigma$, the probability of the event "σ having recovery time equal to n" can be calculated as follows:

$$\mathcal{P}(RT(\sigma_s) = n) = \sum_{\sigma_s \in \eth^c_{s,n}} \mu(\sigma_s) \tag{4.4}$$

In Equation 4.4, the probability of the computations with recovery time n starting from s are added together because they are *disjoint events*. In other words, two distinct computations cannot happen at the same time, so the probability of their union is the sum of their individual probabilities. We have now built the necessary background to demonstrate how to compute the expected recovery time of a stabilizing program.

The *expected recovery time* of a stabilizing program starting from state $s \in \Sigma$ is the following:

$$ERT(s) = E[RT(\sigma_s)] = \sum_{i=0}^{\infty}(i \times \mathcal{P}(RT(\sigma_s) = i))$$

Finally, we take the *average* of the expected recovery time values computed for all initial states in the state space. The reason behind this is to consider the fact that stabilizing programs may start executing from any arbitrary state. The *average recovery time* of a stabilizing system $\mathcal{D}$ with state space Σ and initial state distribution ι_{init} is obtained by the following formula:

$$AvgRT(\mathcal{D}, \Sigma) = \sum_{s \in S} \iota_{init}(s).ERT(s) \tag{4.5}$$

where $\iota_{init} : \Sigma \to [0, 1]$ is a probability distribution function such that

$$\sum_{s \in \Sigma} \iota_{init}(s) = 1.$$

Recall that a stabilizing program can reach any state due to the occurrence of transient faults. The appearance of this state (as the initial state) can follow a specific probability distribution. In cases where this distribution is not given, we decidedly assume uniform distribution; i.e., $\iota_{init}(s) = \frac{1}{|S|}$ for all $s \in \Sigma$. Otherwise, $\iota_{init}(s)$ can be any *real* value in $[0, 1]$ provided that it meets the condition in Equation 4.5. We stress that all definitions and computations presented in this section are valid for weak-stabilizing programs as well as self-stabilizing programs.

4.3 Problem Statement

In this section, we formally state the problem of repairing stabilizing programs whose average average recovery time is expected to be below a certain value. Given an existing stabilizing program and a real value *ert*, a repair algorithm generates another stabilizing program whose average recovery time is below *ert*. Moreover, the algorithm is required to preserve all properties of the input program. The latter can be achieved by allowing merely removing transitions from the original program. That is, we do not allow for adding transitions to avoid introducing new behavior to the program. Since the new transition set of the repaired program will be a subset of the set of transitions of the input program, the set of computations in the new program will be a subset of the set of computations of the original one as well. Hence, any universal property satisfied by the input program (even during convergence) will be satisfied by the repaired program as well. Formally, the decision problem we study is as follows:

Instance. A weak-stabilizing program $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$, and a real number *ert*.

Repair decision problem. Does there exist a stabilizing program $\mathcal{D}' = \langle \Pi_\mathcal{D}, T'_\mathcal{D} \rangle$, such that:

- $T'_\mathcal{D} \subseteq T_\mathcal{D}$, where $T'_\mathcal{D} \neq \emptyset$, and
- $\mathit{AvgRT}(\mathcal{D}') \leq \mathit{ert}$.

4.4 Complexity of Weak/Strong Repair

To prove that the complexity of the strong-stabilizing repair problem is NP-complete, we present a polynomial-time reduction from the *3-SAT problem.*

The 3SAT problem. For an input set of propositional variables $V = \{v_1, \dots, v_N\}$ and a propositional logic formula $P(v_1, ..., v_N) = Y_1 \wedge \cdots \wedge Y_M$, the *3-SAT decision problem* asks whether there exists an assignment of truth values to the input variables such that

- $P(v_1, \dots, v_N) = \mathit{true}$,
- $\forall v \in V : v = \mathit{true}$ or $v = \mathit{false}$.

where,

- $\forall\, 1 \leq j \leq M, Y_j$ is a disjunction of *three* literals,
- A *literal* is v or $\neg v$, where $v \in V$.

Proof. To prove the NP-completeness, we show that the problem is in NP and it is NP-hard.

Proof of membership to NP
A problem is in NP, if given a solution, we can verify its correctness in polynomial time. For our problem, given a system $\mathcal{D}' = \langle \Pi_{\mathcal{D}}, T'_{\mathcal{D}} \rangle$ as a solution, we should verify the following two conditions:

1. It is stabilizing.
2. $AvgRT(\mathcal{D}') \leq ert$

In order to prove that a program is stabilizing, we should verify strong convergence and closure. This verification can be achieved through simple graph exploration algorithms, such as BFS. Such algorithms have polynomial-time complexity in the number of states. Calculation of expected recovery time, which in essence is reachability analysis in Markov chains (discussed in Section 4.2) can be solved in polynomial time as well [14].

Proof of NP-hardness
We now present a mapping from an instance of 3-SAT to an instance of our problem, namely, a distributed stabilizing program $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$, with the following specifications:

$$ert = ((10M + 144N)/(64(M + 3N)))$$

Variables.

$$V = \{x, y, k, z, t, b, c\}$$

where the domain of c is $[1, (M + 3N)]$ and the rest are Boolean variables.

Processes. We declare two processes π_l and π_g.

Read/write restrictions.

$$R_{\pi_l} = \{x, y, k, c\}$$
$$W_{\pi_l} = \{x, k\}$$

	Y_j	v_i^1	v_i^2	v_i^3	v_i^4	v_i^5	v_i^6	v_i^7	v_i^8	v_i^9	v_i^{10}	v_i^{11}	v_i^{12}	v_i^{13}	v_i^{14}	v_i^{15}	v_i^{16}	v_i^{17}	v_i^{18}	v_i^{19}	v_i^{20}
x	F	F	T	T	F	F	F	T	T	F	T	F	F	F	F	T	T	T	F	F	T
y	F	T	T	T	T	T	F	F	T	T	T	T	T	T	T	T	T	F	F	T	T
k	F	F	F	F	T	T	T	F	F	T	F	T	F	T	T	F	F	F	T	T	F
z	F	T	T	T	T	F	F	F	F	F	F	T	F	F	F	F	T	T	T	T	T
t	F	T	T	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F	F	F
b	F	T	T	T	T	T	T	T	F	F	T	T	T	T	T	T	T	T	T	F	F
c	c_{Y_j}	$c_{v_i}^1$	$c_{v_i}^1$	$c_{v_i}^3$	$c_{v_i}^3$	$c_{v_i}^3$	$c_{v_i}^3$	$c_{v_i}^3$	$c_{v_i}^3$	$c_{v_i}^3$	$c_{v_i}^3$	$c_{v_i}^3$	$c_{v_i}^2$	$c_{v_i}^2$	$c_{v_i}^3$	$c_{v_i}^3$	$c_{v_i}^3$	$c_{v_i}^3$	$c_{v_i}^3$	$c_{v_i}^3$	$c_{v_i}^3$

Table 4.1: Valuation of variables in the Y_j state and v_i gadget

Process π_g can read all variables declared in the synthesis problem, and write all, except for the variables in the write-set of π_l. Hence,

$$R_{\pi_g} = \{x, y, k, z, t, b, c\}$$
$$W_{\pi_g} = \{y, z, t, b, c\}$$

States. The state space of our instance is set of all valuation of variables. For each 3-SAT formula, we specify the set of non-legitimate states. All other states are assumed to be in LS. For each variable v_i, $1 \leq i \leq N$, in the 3-SAT instance, there are 20 non-legitimate states labeled by $v_i^1, \cdots, v_i^{20}$ in the synthesis instance (see Fig. 4.1). These states, along with the transitions among them (described later), are called *gadget* v_i. All gadgets have identical structure.

Furthermore, for each clause Y_j, $1 \leq j \leq M$, there is one state in the set of non-legitimate states, with the same name. All other states are considered as legitimate states. Table 4.1 shows the value assignments of the variables in states of each gadget v_i. We note that there are three values of the variable c in each gadget, $c_{v_i}^1$, $c_{v_i}^2$, and $c_{v_i}^3$, which vary among different gadgets and Y_j states (that is why the domain of c has $M + 3N$ values).

For example, Fig. 4.1 shows gadget v_i, and the states for two clauses Y_1 and Y_2, where Y_1 contains v_i and Y_2 includes $\neg v_i$. We assume that for each variable v_i, there exists at least one clause including literal v_i and at least one clause containing literal $\neg v_i$.

Transition relation. The transitions between the states of each v_i gadget are the following (see Fig. 4.1 for an example):

$$\begin{aligned} T_{\pi_l} = \{&(v_i^1, v_i^2), (v_i^2, v_i^1), (v_i^3, v_i^4), (v_i^4, v_i^3), (v_i^6, v_i^7), (v_i^7, v_i^6), \\ &(v_i^8, v_i^9), (v_i^9, v_i^8), (v_i^5, v_i^{10}), (v_i^{10}, v_i^5), (v_i^{11}, v_i^{16}), \\ &(v_i^{16}, v_i^{11}), (v_i^{12}, v_i^{13}), (v_i^{13}, v_i^{12}), (v_i^{14}, v_i^{15}), \\ &(v_i^{15}, v_i^{14}), (v_i^{17}, v_i^{18}), (v_i^{18}, v_i^{17}), (v_i^{19}, v_i^{20}), (v_i^{20}, v_i^{19})\} \end{aligned}$$

$$
\begin{aligned}
T_{\pi_g} = \{&(v_i^2, v_i^3), (v_i^3, v_i^2), (v_i^4, v_i^5), (v_i^5, v_i^6), (v_i^6, v_i^5), (v_i^8, v_i^7),\\
&(v_i^7, v_i^8), (v_i^9, LS_1), (v_i^1 0, v_i^{16}), (v_i^{11}, v_i^5), (v_i^{13}, v_i^{14}),\\
&(v_i^{14}, v_i^{13}), (v_i^{15}, v_i^{16}), (v_i^{16}, v_i^{17}), (v_i^{17}, v_i^{16}), (v_i^{18}, v_i^{19}),\\
&(v_i^{19}, v_i^{18}), (v_i^{20}, LS_2)\} \cup\\
&\{(Y_j, v_i^1) \mid v_i \text{ is a literal in } Y_j\} \cup\\
&\{(Y_j, v_i^{12}) \mid \neg v_i \text{ is a literal in } Y_j\}
\end{aligned}
$$

There is also a self-loop in every state of LS. The transitions marked by ✓, $\times$, ♣, ♦ in Fig. 4.1 correspond to process π_l. In particular, we define $T^{\checkmark}$ and $T^{\times}$ as follows, and use them later in the proof:

$$
\begin{aligned}
T^{\checkmark} =&\{(v_i^3, v_i^4), (v_i^8, v_i^9), (v_i^{10}, v_i^5), (v_i^{16}, v_i^{11}), (v_i^{15}, v_i^{14}),\\
&(v_i^{20}, v_i^{19})\}\\
T^{\times} =&\{(v_i^4, v_i^3), (v_i^9, v_i^8), (v_i^5, v_i^{10}), (v_i^{11}, v_i^{16}), (v_i^{14}, v_i^{15}),\\
&(v_i^{19}, v_i^{20})\}
\end{aligned}
$$

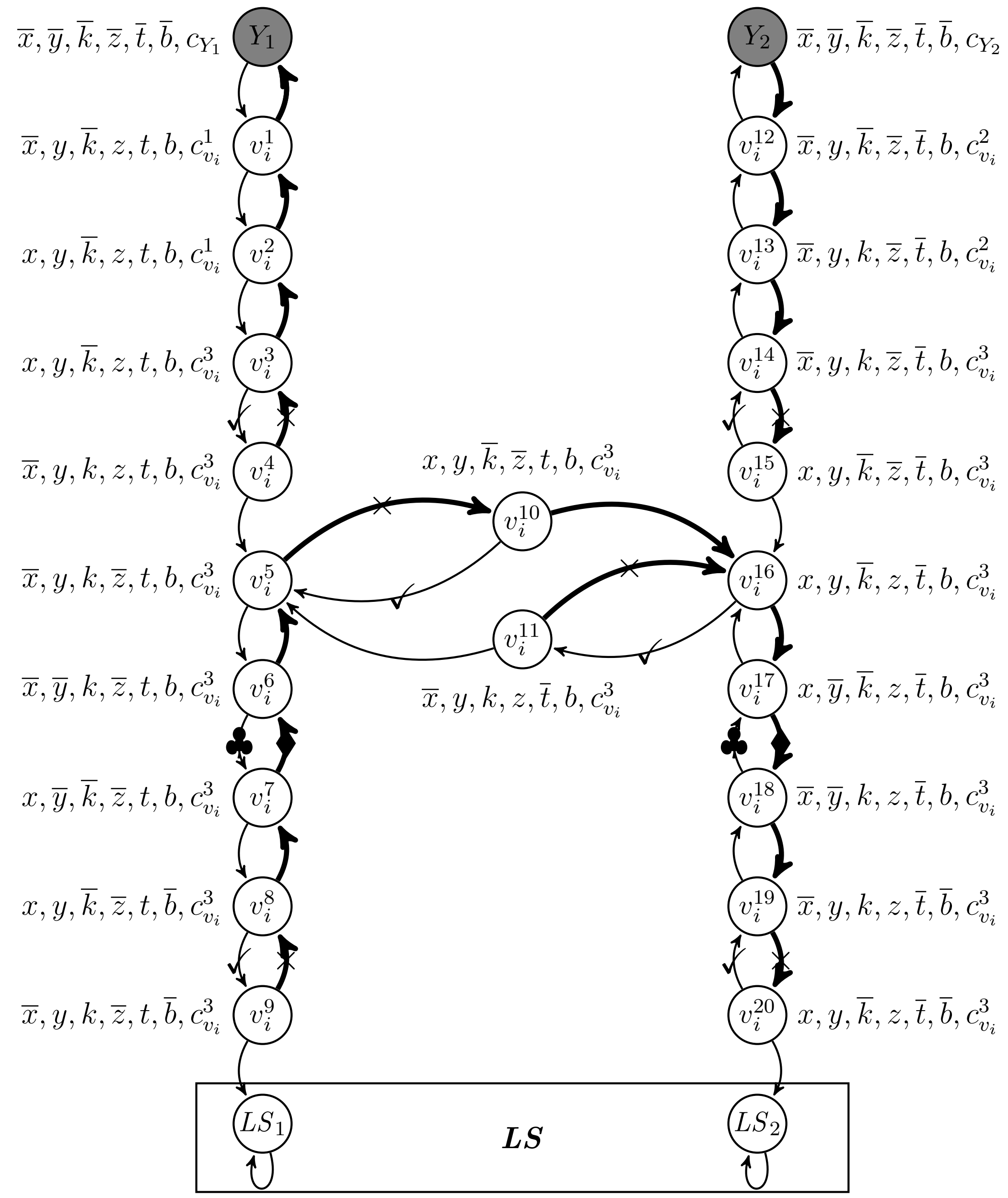

$\overline{x},\overline{y},\overline{k},\overline{z},\overline{t},\overline{b},c_{Y_1}$
Y_1
$\overline{x},y,\overline{k},z,t,b,c^1_{v_i}$
v_i^1
$x,y,\overline{k},z,t,b,c^1_{v_i}$
v_i^2
$x,y,\overline{k},z,t,b,c^3_{v_i}$
v_i^3
$\overline{x},y,k,z,t,b,c^3_{v_i}$
v_i^4
$\overline{x},y,k,\overline{z},t,b,c^3_{v_i}$
v_i^5
$\overline{x},\overline{y},k,\overline{z},t,b,c^3_{v_i}$
v_i^6
$x,\overline{y},\overline{k},\overline{z},t,b,c^3_{v_i}$
v_i^7
$x,y,\overline{k},\overline{z},t,\overline{b},c^3_{v_i}$
v_i^8
$\overline{x},y,k,\overline{z},t,\overline{b},c^3_{v_i}$
v_i^9
$x,y,\overline{k},\overline{z},t,b,c^3_{v_i}$
v_i^{10}
v_i^{11}
$\overline{x},y,k,z,\overline{t},b,c^3_{v_i}$
Y_2
$\overline{x},\overline{y},\overline{k},\overline{z},\overline{t},\overline{b},c_{Y_2}$
v_i^{12}
$\overline{x},y,\overline{k},\overline{z},\overline{t},b,c^2_{v_i}$
v_i^{13}
$\overline{x},y,k,\overline{z},\overline{t},b,c^2_{v_i}$
v_i^{14}
$\overline{x},y,k,\overline{z},\overline{t},b,c^3_{v_i}$
v_i^{15}
$x,y,\overline{k},\overline{z},\overline{t},b,c^3_{v_i}$
v_i^{16}
$x,y,\overline{k},z,\overline{t},b,c^3_{v_i}$
v_i^{17}
$x,\overline{y},\overline{k},z,\overline{t},b,c^3_{v_i}$
v_i^{18}
$\overline{x},\overline{y},k,z,\overline{t},b,c^3_{v_i}$
v_i^{19}
$\overline{x},y,k,z,\overline{t},\overline{b},c^3_{v_i}$
v_i^{20}
$x,y,\overline{k},z,\overline{t},\overline{b},c^3_{v_i}$
LS_1
LS_2
$\boldsymbol{LS}$

Figure 4.1: Gadget v_i in self-stabilizing instance

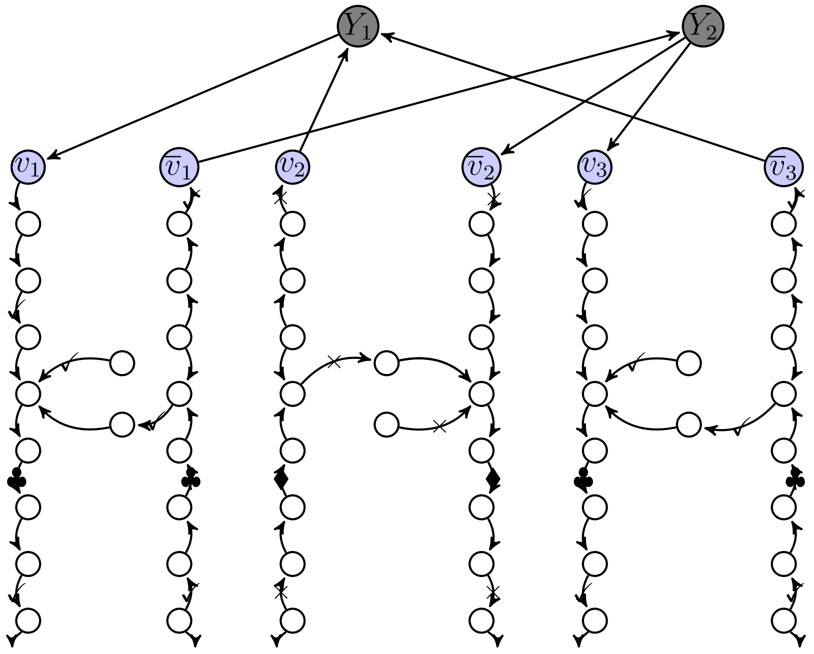


Figure 4.2: Self-stabilizing instance for formula $P = (v_1 \vee v_2 \vee \neg v_3) \wedge (\neg v_1 \vee \neg v_2 \vee v_3)$.

Moreover, for process π_g, we include the transition from Y_j to v_i^1 (respectively, v_i^{12}), if v_i (respectively, $\neg v_i$) exists in clause Y_j of the 3-SAT problem. Given the value assignments in Table 4.1, it is straightforward to determine the group of each transition. In particular, the transitions marked with the same symbol in Fig. 4.1 belong to one group. Note that since we include c in the read-set of π_l, there is no grouping between the transitions of π_l in different gadgets. Also, since π_g can read all variables, there is no grouping between the transitions of π_g. Hence, there is no grouping among transitions of different gadgets.

Now, we show that a 3-SAT problem has a solution, *if and only if* we can find a solution for the mapped synthesis problem. From now on we refer to the mapped synthesis problem as $\mathcal{SP}_1$.

- ($\Rightarrow$) First, we show that if the given 3-SAT instance is satisfiable, then there exists a strong stabilizing revision of the given program with an average recovery time within the given bound. Since the 3-SAT formula is satisfiable, there exists an assignment of truth values to all variables $v_1, \cdots, v_N$, such that each $Y_1, \cdots, Y_M$, is *true*. We now derive a program $\mathcal{D}'$ from $\mathcal{SP}_1$ with the required expected recovery time, obtained by removing a subset of the transitions from $\mathcal{D}$. For each variable v_i assigned to *true* in the solution of the 3-SAT instance, the following transitions are removed from gadget v_i (as highlighted in Fig. 4.1):

$$T^{\times} \cup \{(v_i^2, v_i^1), (v_i^3, v_i^2), (v_i^6, v_i^5), (v_i^7, v_i^6), (v_i^8, v_i^7),\\ (v_i^{10}, v_i^{16}), (v_i^{12}, v_i^{13}), (v_i^{13}, v_i^{14}), (v_i^{16}, v_i^{17}), (v_i^{17}, v_i^{18}),\\ (v_i^{18}, v_i^{19})\} \cup\\ \{(v_i^1, Y_j) \mid 1 \leq j \leq M\} \cup \{(Y_j, v_i^{12}) \mid 1 \leq j \leq M\}$$

Similarly, if v_i is assigned to *false*, then the following transitions are omitted:

$$T^{\checkmark} \cup \{(v_i^1, v_i^2), (v_i^2, v_i^3), (v_i^5, v_i^6), (v_i^6, v_i^7), (v_i^7, v_i^8),\\ (v_i^{11}, v_i^5), (v_i^{13}, v_i^{12}), (v_i^{14}, v_i^{13}), (v_i^{17}, v_i^{16}), (v_i^{18}, v_i^{17}),\\ (v_i^{19}, v_i^{18})\} \cup\\ \{(Y_j, v_i^1) \mid 1 \leq j \leq M\} \cup \{(v_i^{12}, Y_j) \mid 1 \leq j \leq M\}$$

Due to the symmetrical structure of each gadget, omitting these transitions has the same effect on average recovery time as removing the ones for when the variable is assigned to *true*.

Now, we show that the resulting synthesized program is strong stabilizing, and respect the required average recovery time *ert*:

- *(Closure)* The closure property of the original stabilizing program is preserved, since no new transitions were added in the synthesis procedure.
- *(Strong Convergence)* The transition removal does not violate the convergence of any of the states $v_i^1, \cdots, v_i^{20}$ (Fig. 4.2). For the Y_j states, since the corresponding 3-SAT instance is satisfiable, each Y_j includes a v_i, assigned to *true*, or a $\neg v_i$, assigned to *false*. In the former case, Y_j has a convergence path through v_i^1, and in the latter, it is reachable to LS through v_i^{12}. Also, there is no loop among states in $\neg LS$ in the synthesized program (Fig. 4.2).
- *(Average recovery time)* The average recovery time of the resulting program is $((10M + 144N)/(64(M + 3N)))$, which is equal to the bound specified in the synthesis problem (ert). We calculate the average recovery time of the resulting program using the methods introduced in Section 4.2. Note that since there are 6 Boolean variables and a variable with domain $M + 3N$ in $\mathcal{SP}_1$, $(64(M + 3N))$ is the total number of states in the state space.

• ($\Leftarrow$) Next, we prove that if there exists a solution for $\mathcal{SP}_1$, then the given 3-SAT formula is satisfiable by presenting a valid truth assignment derived from the solution of $\mathcal{SP}_1$. For each v_i, if the ✓ transitions are preserved in its associated gadget, v_i is assigned to *true*. But, if the $\times$ transitions are maintained, it is assigned to *false*.

Now, we prove that a variable v_i cannot be assigned to both *true* and *false* at the same time (both ✓ and $\times$ transitions preserved). In the solution for $\mathcal{SP}_1$, for each v_i, either the ✓ or $\times$ transitions should remain in their corresponding gadgets. The reason behind this is that if both groups of transitions are removed, the states below v_i^4 and v_i^{15}in the gadget (except for v_i^9 and v_i^{20}) cannot converge to LS. However, after removing only one group of transitions, each state Y_j will either converge through a path with ✓ transitions (implying $v_i = \mathit{true}$), or a path with $\times$ transitions (implying $v_i = \mathit{false}$). From Fig. 4.2 is obvious that the rest of the states also converge to LS.

On the other hand, both groups of transitions cannot exist simultaneously. If they do so, the recovery time of the solution will exceed the given bound *ert* because we chose *ert* to be equal to the average recovery time of the program when either ✓ or $\times$ transitions are removed (Fig. 4.2), while if both groups of transitions exist, average recovery time will be higher due to the loops in the transition system of the program. The other reason is that having both sets of transitions, there will be a loop in $\neg LS$, which violates the strong convergence.

Note that our requirement on the average recovery time leads to having a strong self-stabilizing solution. Hence, this proof works for repairing both weak and strong self-stabilizing protocols.

□

Chapter 5

Parameterized Synthesis of Fault-Tolerant Distributed Systems

5.1 Introduction

Kulkarni and Arora in [56] show that the problem of adding masking fault-tolerance [1] to distributed programs is NP-complete in the size of the input program's state space. A set of polynomial-time heuristics are introduced in [57] for the problem of adding masking fault-tolerance to distributed programs. The time and space complexity of the problem is high due to the two reason: 1) high space complexity of distributed system, and 2) read restriction of processes due to their partial visibility from the system's global state. Later, a more efficient symbolic heuristic is introduced in [21] for a similar problem. But still the algorithm can synthesize a fault-tolerant distributed system for a fixed number of processes. It is desirable to have an approach to synthesize fault-tolerant protocols that works for systems with any number of processes.

With this motivation, we propose an automated parameterized method for synthesizing masking fault-tolerant distributed protocols from their fault-intolerant version. Such protocols are parameterized by the number of processes. Our synthesis algorithm utilizes counter abstraction to construct a finite representation of the state space. Then, it performs fixpoint calculations to compute and exclude states that violate the safety specification in the presence of faults. To guarantee liveness, our algorithm ensures deadlock freedom and augments the input intolerant protocol with safe recovery paths. We demonstrate the effectiveness of our algorithm by synthesizing a fault-tolerant distributed agreement

[1]A masking fault-tolerant protocol is one that ensures constant satisfaction of safety and liveness specifications even in the presence of faults.

protocol in the presence of Byzantine fault. Although the synthesis problem is known to be NP-complete in the state space of the input protocol (due to partial observability of processes) in the non-parameterized setting, our parameterized algorithm manages to synthesize a solution for a complex problem such as Byzantine agreement within less than two minutes.

5.2 Preliminaries

In this section, we present the preliminary concepts on distributed programs in the shared-memory model, fault-tolerance, and counter abstraction.

5.2.1 Parameterized Distributed Programs

Before defining parameterized distributed programs, we need to introduce the notion of a process template. A *process template* is a tuple $\langle V, G\rangle$, where V is a finite set of *control variables* and G is a finite set of *guarded commands* — to be defined later — that specify the behavior of a process instantiated from the process template. We also use guards to model faults.

5.2.1.1 Variables.

The set of *control* variables V in a process template has two disjoint subsets: (1) a finite set V^ℓ of *local* variables; (2) a finite set V^s of *shared* variables. Each variable $v \in V^\ell \cup V^s$ has a finite domain D_v, and a set of initial values $I_v \subseteq D_v$. Intuitively, a process created from the process template can read and write any subset of V^ℓ in one atomic step; it can also read the shared variables of other processes created from this template in one atomic step.

5.2.1.2 Guarded Commands.

Let I be a set of index variables that range over natural numbers. The special index variable $\mathsf{id} \in I$ points to the process evaluating a guard.

First, define conditions as follows:

- *Index conditions* $j = a$ and $j = k$, where $j, k \in I$ are index variables, and $a \in \mathbb{N}$ is a constant index.

- *Variable conditions* $x[j] = d$ and $x[a] = d$, where $x \in V^\ell \cup V^s$ is a control variable, $d \in D_x$ is a value from x's domain, and $a \in \mathbb{N}$ and $j \in I$ are an index variable and a constant index respectively.

Having introduced conditions, we define guards as g, $\neg g$, $g \wedge g'$ and $g \vee g'$, where each of g, g' is either an index condition, or a variable condition. *Commands* are expressions $x[\mathsf{id}] := d$ and $x[\mathsf{id}] := y[a]$, if $x, y \in V$ are control variables with $D_x = D_y$, and $a \in \mathbb{N}$ and $d \in D_x$. A guarded command is an expression $\phi \rightarrow w_1; \dots; w_c$, if ϕ is a basic guard, and $w_1, \dots, w_c$ are commands that assign values to pairwise different variables.

To distinguish the commands that introduce faults from normal commands, we need additional definitions. We say that a variable condition accesses an external variable $x \in V$, if the condition is either of the form $x[j] = d$ and $j \in I \setminus \{\mathsf{id}\}$, or of the form $x[a] = d$. We call a guard ϕ *normal*, if all its variable conditions access only external variables from V^s.

5.2.1.3 Parameterized Distributed Programs

Having a process template $P = \langle V, G \rangle$, we can instantiate a process π_k (k is called the index of the process) by instantiating the set of control variables V_k from V, and the set of guarded commands G_k by replacing the index variable id with index k.

As the guards can refer to the variables of processes different from id by using a constant index, we identify the maximal index K that appears in index and variables conditions as well as in statements. That is, for every $j = a$, $x[b]$, occurring in the guards and actions, it holds that $K \geq a$ and $K \geq b$.

Given a process template P and a number $N \geq K$, one can construct a distributed program P^N by instantiating N processes with indices in $\mathcal{I}_N = \{1, \dots, N\}$. Each control variable $v \in V$ can then be thought as a vector of size N, where $v[k]$ is the variables instantiated in the process instance with index k. We, thus, have to deal with a parameterized family of programs $\mathcal{D} = \{P^N\}_{N \geq K}$, where each instance has a fixed number of processes. Informally, all but the first K processes behave similarly. Formal semantics of a distributed program P^N is captured below as a system instance.

Example. We use an algorithm for solving the problem of *Byzantine agreement* (denoted $\mathcal{BA}$) as a running example to describe the concepts throughout this chapter (see pseudo code 5). This problem consists of a fixed *general* process (i.e., π_1) and a parameterized set of $N - 1$ *lieutenant* processes.

Variables. The shared vector d has N control variables, with domain $D_{d[i]} = \{0, 1, \bot\}$, for all $i \in \mathcal{I}_N$. Each lieutenant process π_i, $2 \leq i \leq N$, is associated with variable $d[i]$ (i.e., lieutenant i can write into $d[i]$) and the general is associated with $d[1]$. Moreover, each

Function 5 Process Template of Byzantine Agreement

```
1: Template variables $V^{\ell} = \{b, f\}, V^s = \{d\}, V = V^{\ell} \cup V^s$:
       $b, f$ : **Boolean**; **init** $b = false, f = false$
       $d : \{0, 1, \bot\}$; **init** $d = \bot$
2: Template guards $G$:
       $(\mathsf{id} = 1) \wedge d[\mathsf{id}] = \bot \rightarrow \quad d[\mathsf{id}] := choose\ from\{0, 1\}, f[1] := true;$
       $(d[1] \neq \bot) \wedge (d[\mathsf{id}] = \bot) \wedge \neg f[\mathsf{id}] \wedge \neg b[\mathsf{id}] \rightarrow \quad d[\mathsf{id}] := d[1];$
       $(d[1] \neq \bot) \wedge (d[\mathsf{id}] \neq \bot) \wedge \neg f[\mathsf{id}] \wedge \neg b[\mathsf{id}] \rightarrow \quad f[\mathsf{id}] := true;$
```

process has the following local variables: (1) a Boolean variable f which shows whether or not the decision of the process is finalized, and (2) a Boolean variable b which shows whether or not the process is Byzantine (i.e., faulty).

Behavior. As can be seen in pseudo code 5, the general process (π_1), changes its decision to 0 or 1 arbitrarily, if its decision is $\bot$ (guarded commend labeled by g). As soon as the general process has a decision different from $\bot$, lieutenants can start execution. If it is undecided (i.e., its d variable equals $\bot$), and its finalization and Byzantine bits are both false, it may copy the decision of the general (guarded commend labeled by l_1). After copying the decision of the general, if the process Byzantine bit is still false, it may change its finalization bit to *true* (guarded commend labeled by l_2). Note that we assume that any system instance initializes with $d[k] = \bot$, $f[k] = false$, and $b[k] = false$, for all $k \in \mathcal{I}_N$.

Atomic Propositions. In order to specify behavior of parameterized distributed programs, we have to introduce atomic propositions that specify the values of the variables of some processes or all processes. We also should be able to specify the values of the variables of the fixed processes with indices up to K. To this end, similarly to [53], if ϕ is a guard with no index variables but j and ϕ' is a guard using only constant indices, the following three expressions are atomic propositions:

$$[\forall j.\ \phi] \text{ and } [\exists j.\ \phi] \text{ and } [\phi']$$

In the sequel, we denote the set of atomic propositions with respect to the set of variables V by AP.

5.2.1.4 System Instance

To define a system instance formally, we first introduce the notions of N-state valuation and of an index valuation.

Definition 34. *Given a number of processes N, an N-*state valuation* σ is a function $V \times \mathcal{I}_N \to \bigcup_{v \in V} D_v$ with the restriction that for each $v \in V$ and $j \in \mathcal{I}_N$, it holds that $\sigma(v, j) \in D_v$. We also define an* index valuation *as a partial function $\iota : I \nrightarrow \{1, \ldots, N\}$.*

For convenience, we also extend an N-state valuation σ on $\mathbb{N}$, i.e., for $d, a \in \mathbb{N}$, $\sigma(d, a) = d$. As usual, we define substitution $\sigma[e/(x, a)]$ as an N-state valuation σ' such that $\sigma'(y, b)$ is $\sigma(e)$, if $y = x$ and $b = a$, and $\sigma(y, b)$ otherwise. Similarly, we define substitution $\iota[a/j]$ for $j \in I$ and $a \in \mathcal{I}_N$.

Given a basic guard ϕ, a number $N \in \mathbb{N}$, an N-state valuation σ, and an index valuation ι, we define $\sigma, \iota \models \phi$ in a natural way, that is, for each $x \in V$, $j, k \in I$, and $a \in \mathbb{N}$:

- $\sigma, \iota \models j = a$ if and only if $\iota(j) = a$.
- $\sigma, \iota \models j = k$ if and only if $\iota(j) = \iota(k)$.
- $\sigma, \iota \models x[j] = d$ if and only if $\sigma(x, \iota(j)) = d$.
- The cases of $\neg\phi'$, $\phi' \wedge \phi''$, and $\phi' \vee \phi''$ are defined as usual, e.g., $\sigma, \iota \models \neg\phi'$ and $\sigma, \iota \models \phi' \wedge \phi''$ if and only if $\sigma, \iota \not\models \phi'$ and $\sigma, \iota \models \phi'$ and $\sigma, \iota \models \phi''$ respectively.

Given a quantified guard $\exists j_1 \ldots \exists j_m \forall k.\ \phi$, a number $N \in \mathbb{N}$, an N-state valuation σ, and an index valuation ι, we say that $\sigma, \iota \models \exists j_1 \ldots \exists j_m \forall k.\ \phi$ if and only if there exist indices $a_1, \ldots, a_m \in \mathcal{I}_N$ such that for all values $b \in \mathcal{I}_N$ it holds $\sigma, \iota' \models \phi$ for $\iota' = \iota[a_1/j_1, \ldots, a_m/j_m, b/k]$.

Definition 35. *Given a process template $P = \langle V, G \rangle$ and number of processes $N \in \mathbb{N}$, a system instance P^N is a labelled transition system $(\Sigma_N, \mathbf{I}_N, \Delta_N, \lambda_N)$ defined as follows:*

- *The set of global states Σ_N is the set of all N-state valuations.*
- *The set of initial states $\mathbf{I}_N \subseteq \Sigma_N$ that contains all global states $\sigma \in \Sigma_N$ satisfying $\forall v \in V.\ \forall j \in \mathcal{I}_N.\ \sigma(v, j) \in I_v$.*
- *The transition relation $\Delta_N \subseteq \Sigma_N \times \Sigma_N$. A pair $(\sigma, \sigma') \in \Delta_N$ if and only if there is a process index $a \in \mathcal{I}_N$ and a guarded command $\phi \to x_1[\mathsf{id}] := e_1; \ldots; x_m[\mathsf{id}] := e_m$ from G such that: The guard is satisfied when $\mathsf{id} = a$, that is, $\sigma, \{\mathsf{id} \mapsto a\} \models \phi$; Only $x_1, \ldots, x_m$ change, that is, $\sigma' = \sigma[\sigma(e_1, a)/(x_1, a), \ldots, \sigma(e_m, a)/(x_m, a)]$.*

- *The labelling function $\lambda_N : \Sigma_N \to 2^{AP}$. An atomic proposition $[\forall \mathsf{id}.\ \phi]$ belongs to $\lambda_N(\sigma)$ if and only if for all $i \in \mathcal{I}_N$, it holds that $\sigma, \{\mathsf{id} \mapsto i\} \models \phi$. The case of $[\exists \mathsf{id}.\ \phi]$ is defined similarly. Finally, when ϕ is using only constant process indices, it holds $[\phi] \in \lambda_N(\sigma)$ if and only if $\sigma, \{\} \models \phi$.*

5.2.2 Specification

We consider two categories of properties in a system specification: (1) state-based, and (2) transition-based. The state-based properties are those that can be specified based on the variables valuation in a state. Transition-based properties, on the other hand, are those that put a constraint on the transitions of the system (source and target states of transitions). Let V' be the set of variables, where each variable x' is a primed copy of variable x from V. Then a transition-based property is a guard over the variables V, V', and index variable id.

Example. As an example, the safety specification in the Byzantine agreement problem includes the following two state-based requirements:

- *Validity*, which means that if the general process is non-Byzantine, then the final decision of any non-Byzantine non-general process should be the same the decision of the general process.
- *Agreement*, which means that any two non-Byzantine non-general processes should finalize to the same decision.

More formally, the above specifications can be written as follows:

$$\begin{aligned}&([\neg b[1] \wedge d[1] = 0] \to [\forall j\,.\,\neg b[j] \wedge f[j] \to d[j] = 0]\\ &\wedge ([\neg b[1] \wedge d[1] = 1] \to [\forall j\,.\,\neg b[j] \wedge f[j] \to d[j] = 1])\end{aligned} \tag{$SPEC_1$}$$

$$\begin{aligned}&(([\exists i\,.\ \neg b[i] \wedge f[i] \wedge d[i] = 1] \to\ [\neg \exists j\,.\,\neg b[j] \wedge f[j] \wedge d[j] = 0])\\ &\wedge ([\exists i\,.\ \neg b[i] \wedge f[i] \wedge d[i] = 0] \to\ [\neg \exists j\,.\,\neg b[j] \wedge f[j] \wedge d[j] = 1]))\end{aligned} \tag{$SPEC_2$}$$

The following transition-baaed property prevents a non-general process with a finalized decision from changing its decision value or the finalization bit later:

$$\forall j\,.\ (f[j] \wedge d[j] = 1 \to (f'[j] \wedge d'[j] = 1)) \wedge (f[j] \wedge d[j] = 0 \to (f'[j] \wedge d'[j] = 0)) \tag{$SPEC_3$}$$

Byzantine agreement has to satisfy the specifications $SPEC_1$, $SPEC_2$, and $SPEC_3$.

5.2.3 Fault-Tolerance

Definition 36. *Let $P^N = (\Sigma_N, \mathbf{I}_N, \Delta_N, \lambda_N)$ be a distributed program, and SPEC be an* LTL *property. A Boolean formula LS is called the* legitimate states *of P^N from SPEC, if (1) LS is closed in P^N, (2) $P^N \models (LS \rightarrow SPEC)$, and (3) there is no deadlocked computation starting from LS.* □

Example. As an example, the legitimate states in the Byzantine agreement problem could be specified by the following predicate:

$$\begin{aligned}(\neg b[1] \ \rightarrow \ & (\forall j\,.\,\neg b[j] \rightarrow (d[j] = \bot \vee d[j] = d[1])) \wedge \\ & (\forall j\,.\,\neg b[j] \wedge d[j] = \bot \rightarrow \neg f[j])) \wedge \\ (b[1] \ \rightarrow \ & (\forall j\,.\,d[j] = 0) \vee (\forall j\,.\,d[j] = 1))\end{aligned}$$

A *process template with fault* is a tuple $P_F = \langle V, G, F \rangle$, where F is a set of guarded commands that specify the *faults* that may occur in the process execution. Hence, syntactically, a fault can be represented as a guarded command, where guards do not have to be normal. As introduced in Section 5.2.1, in a normal guarded command, the guard can only be on local variables, or the shared vector. However, in a fault guarded command, we don't have this constraint. Semantically, a system instance with N processes of the process template $P_F = \langle V, G, F \rangle$ is $P_F^N = (\Sigma_N, \mathbf{I}_N, \Delta_N \cup \Delta_F, \lambda_N)$, where $\Delta_F \subseteq \Sigma \times \Sigma$. We emphasize that fault representation with a transition is possible for different types of faults (stuck-at, crash, fail-stop, timing, performance, Byzantine, message loss, etc.), nature of the faults (permanent, transient, or intermittent), or the ability of the program to observe the effects of the faults [17].

Example. As an example, pseudo code 6 is the process template of the $\mathcal{BA}$ example with two faults. If there exists no faulty process, the first fault action can change the Byzantine bit of the process to *true*. When the Byzantine bit of a process is *true*, it can change its decision and (or) finalization bits freely to any value in their domains. As can be seen, the first guarded command is not normal, as the Byzantine bits are not part of the shared vector of the system. The reason for checking the Byzantine bit of other processes before going Byzantine is that we assume at each point of system execution, at most one process can be faulty.

Definition 37. *A computation Δ is called* finitely-faulty*, if it has the following condition: $\exists n \geq 0\,.\,\forall i \geq n\,.\,(s_i, s_{i+1}) \in \Delta \setminus \Delta_F$.* □

Function 6 Byzantine Agreement with Fault

```
1: Template variables $V^\ell = \{b, f\}, V^s = \{d\}, V = V^\ell \cup V^s$:
        $b, f$ : **Boolean**; **init** $b = \mathit{false}, f = \mathit{false}$
        $d : \{0, 1, \bot\}$; **init** $d = \bot$
2: Template guards $G$:
        $(\mathsf{id} = 1) \wedge d[\mathsf{id}] = \bot \quad \rightarrow d[\mathsf{id}] := \mathit{choose\ from}\{0, 1\}, f[1] := \mathit{true};$
        $(d[1] \neq \bot) \wedge (d[\mathsf{id}] = \bot) \wedge \neg f[\mathsf{id}] \wedge \neg b[\mathsf{id}] \quad \rightarrow d[\mathsf{id}] := d[1];$
        $(d[1] \neq \bot) \wedge (d[\mathsf{id}] \neq \bot) \wedge \neg f[\mathsf{id}] \wedge \neg b[\mathsf{id}] \quad \rightarrow f[\mathsf{id}] := \mathit{true};$
3: Faults $F$:
        $\forall k \,.\, \neg b[k] \quad \rightarrow b[\mathsf{id}] := 1;$
        $b[\mathsf{id}] \quad \rightarrow d[\mathsf{id}] := 0/1/\bot, f[\mathsf{id}] := \mathit{true}/\mathit{false};$
```

As a finitely-faulty computation has finitely many transitions from the set of faults, which is required to ensure *recovery* in the synthesized fault-tolerant system.

We can now define what it means for a distributed program to be *fault-tolerant*. When a fault occurs, the system execution might go out of its legitimate states, and the system specification might be also violated. Intuitively, a fault-tolerant distributed program satisfies the system specifications in the absence and presence of faults, and also, returns to its set of legitimate states in a finite number of steps. A more formal definition for fault-tolerance is given next.

Definition 38. *Consider $\models$ to denote satisfaction relation, and $\models_{\mathit{ff}}$ to denote satisfaction on finitely-faulty computations. A distributed program P^N with legitimate states LS, and specification $SPEC$ is* fault-tolerant *to the faults F, if and only if the following* LTL *properties hold:*

1. $P^N \models \mathbf{G}\,[(LS \rightarrow \mathbf{G}\,LS) \wedge (LS \rightarrow SPEC)]$
2. $P^N_F \models_{\mathit{ff}} LS \rightarrow \mathbf{G}\,SPEC$
3. $P^N_F \models_{\mathit{ff}} LS \rightarrow \mathbf{G}\,\mathbf{F}\,LS$ □

The first condition guarantees LS to be closed in P^N, and also every state in LS to satisfy the specification $SPEC$. The second condition requires any computation starting from LS

to satisfy $SPEC$. Finally, the last one ensures any system execution, starting from LS, to be in LS infinitely often. Note that a fault action might lead the system execution to a state outside LS. This condition requires the system to get back to the set of legitimate states in a finite number of states. Note that in the second and third conditions, P_F^N is used, which means that the computations in the conditions may include fault transitions (Δ_F) in addition to the system transitions (Δ).

5.3 Problem Statement

Problem statement. Given is a process template $P_F = \langle V, G, F \rangle$, a constant K, a specification $SPEC$, and an atomic proposition LS, such that for every $N \geq K$, and $P^N = (\Sigma_N, \mathbf{I}_N, \Delta_N, \lambda_N)$, $P^N \models \mathsf{G}\,[(LS \rightarrow \mathsf{G}\,LS) \wedge (LS \rightarrow SPEC)]$. Our goal is to propose an algorithm for synthesizing a process template P', and legitimate states LS' from P, such that for every $N \geq K$:

1. The set $\mathbf{I}'_N$ is not empty in P'^N.
2. For every LTL formula β, such that $P^N \models \beta$: $P'^N \models \beta$, and
3. P'^N_F is F-tolerant to $SPEC$ from LS'.

Hence, the input to our problem is a process template including a set of faults, such that in the absence of faults, it satisfies its specification. Our goal is to synthesize a process template from that such that for every instance of the synthesized template, it is fault-tolerant, and it also satisfies all the LTL properties that were already satisfied by the instances from the original template.

5.4 Parameterized Solution

In this section, we present our solution for synthesizing parameterized fault-tolerant distributed systems. The synthesis problem is known to be NP-complete for the concrete case [20]. For parameterized synthesis of fault-tolerant distributed systems, we propose a heuristic inspired by the algorithm introduced in [21] for the concrete case. For handling the parameterized case, we use an abstraction called *counter abstraction*, as introduced in Section 5.4.1. Another concept we need to introduce for algorithm description is *read restriction*, which we present in Section 5.4.2.

5.4.1 Counter Abstraction

In this section, we define $\{0, 1, \infty\}$-counter abstraction very much in the spirit of [72].

Definition 39. *Given a set of variables V, a local state valuation is a function $\rho : V \to \bigcup_{D_v | v \in V}$ with the restriction that for all $v \in V$, $\rho(v) \in D_v$.* □

With $\mathcal{P}_V$ we denote the set of local state evaluations defined with respect to V. As we consider finite sets of variables over finite domains, set $\mathcal{P}_V$ is finite. Further, given a system instance with N processes, we define function $\# : \Sigma_N \times \mathcal{P}_V \to \mathbb{N}_0$; for every $\sigma \in \Sigma_N$ and every $\rho \in \mathcal{P}_V$, the value $\#(\sigma, \rho)$ equals to cardinality of the set $\{i \mid K < i \leq N, \forall x \in V.\, \rho(x) = \sigma(x, i)\}$.

First, we introduce two sets of variables: set $A = \{x[a] \mid x \in V, a \in \mathcal{I}_K\}$; set $B = \{\kappa_\rho \mid \rho \in \mathcal{P}_V\}$, where $D_{\kappa_\rho} = \{0, 1, 2\}$. Set A keeps K copies of the process variables that correspond to the variables of the first K processes. Each variable κ_ρ from B plays the role of an abstract counter, whose values reflect the number of processes with their variables evaluated to ρ as follows: value 0 for zero processes; value 1 for exactly one process; value 2 for more than one process. Finally, we define $V_C = A \cup B$ and an *abstract state* as a valuation $\hat{\sigma} : V_C \to \bigcup_{v \in V_C} D_v$ with the restriction that for every $v \in V_C$, $\hat{\sigma}(v) \in D_v$. We denote the set of all such valuations with Σ_C.

To define the transition system of the counter abstraction, we introduce a parameterized abstraction function:

Definition 40. *Given the number of processes $N \geq K$, we define* state abstraction α_N *as a function $\Sigma_N \to \Sigma_C$ that for every $\sigma \in \Sigma_N$, returns an abstract state $\hat{\sigma} \in \Sigma_C$ that satisfies the following properties:*

- *For every $x \in V$ and every index $i \in \mathcal{I}_K$, it holds $\hat{\sigma}(x[i]) = \sigma(x, i)$.*
- *For every local state $\rho \in \mathcal{P}_V$, the following holds*

$$\hat{\sigma}(\kappa_\rho) = \begin{cases} 0, & \textit{if } \#(\sigma, \rho) = 0 \\ 1, & \textit{if } \#(\sigma, \rho) = 1 \\ 2, & \textit{if } \#(\sigma, \rho) \geq 2 \end{cases}$$

□

Having defined state abstraction, we define a $\{0, 1, \infty\}$-counter abstraction.

Definition 41. *Let (V, P) be a process template. A transition system $C = (\Sigma_C, \mathbf{I}_C, \Delta_C, \lambda_C)$ is a* counter abstraction *of the parameterized family $\{P^N\}_{N \geq K}$, if it satisfies the following conditions:*

- *As defined above, Σ_C is the set of all abstract states.*
- *The set of initial states $\mathbf{I}_C \subseteq \Sigma_C$ contains an abstract state $\hat{\sigma} \in \Sigma_C$ if and only if for every variable $x \in V$ it holds that the counter of each local state $\rho \in \mathcal{P}_V$ with $\rho(x) \notin I_x$ is set to zero, i.e., $\hat{\sigma}(\kappa_\rho) = 0$, and each fixed process $i \leq K$ respects the initial values, i.e., $\hat{\sigma}(x[i]) \in I_x$.*
- *$(\hat{\sigma}, \hat{\sigma}') \in \Delta_C$ if and only if there exists size $N \geq K$ and global states $\sigma, \sigma' \in \Sigma_N$ such that $\hat{\sigma} = \alpha_N(\sigma)$, $\hat{\sigma}' = \alpha_N(\sigma')$, and $(\sigma, \sigma') \in \Delta_N$.*
- *$p \in \lambda_C(\hat{\sigma})$ if and only if there exists size $N \geq K$ and a global state $\sigma \in \Sigma_N$ with $p \in \lambda_C(\sigma)$.*

□

5.4.2 Read Restriction

Read-set and write-set. Consider $\{\pi_1, \cdots, \pi_N\}$ to be the set of processes in a system instance with N processes. The *read-set* of each process π_i, where $1 \leq i \leq N$ (denoted R_{π_i}) is the set of all variables the process can read. Thus, $R_{\pi_i} = V^l_{\pi_i} \cup V^s$.The *write-set* of a process π (denoted W_{π_i}) is the set of variables a process is allowed to write into. Thus, $W_{\pi_i} = V^l_{\pi_i} \cup V^s_{\pi_i}$. Notice that the write-set of a variable is a subset of its read-set (i.e., $W_{\pi_i} \subseteq R_{\pi_i}$), and hence, a process cannot write into a variable blindly.

Example. As an example, in a system instance with N processes of our $\mathcal{BA}$ example, the read-set of each process π_i is $R_{\pi_i} = \{d, f[i], b[i]\}$, and its write-set is $W_{\pi_i} = \{d[i], f[i], b[i]\}$.

Read restriction is a constraint on the transition relation of distributed systems with shared memory model, and is imposed due to the fact that each process can read only a part of the system state.

Definition 42. *For a distributed system $P^N = (\Sigma_N, \mathbf{I}_N, \Delta_N, \lambda_N)$, the read restriction is defined as follows:*

$$\begin{aligned} \forall\,(s_1, s_1') \in \Delta_N \,.\, \exists\, 1 \leq j \leq N \,.\, \exists v \in W_{\pi_j} \,.\, s_1(v) \neq s_1'(v) \;\Rightarrow\; \\ \forall\, s_2, s_2' \in \Sigma_N \,.\, (\forall v \notin W_{\pi_j} : (v(s_1) = v(s_1') \,\wedge\, v(s_2) = v(s_2'))) \\ \wedge(\forall v \in R_{\pi_j} : (v(s_1) = v(s_2) \,\wedge\, v(s_1') = v(s_2'))) \\ \implies\; (s_2, s_2') \in \Delta_N \end{aligned} \tag{5.1}$$

□

In other words, read-restriction ensures that the action of each process does not depend on the variables it cannot read. If two states are the same, except for the values of variables not in the read-set of a process, and the process has a transition starting from one of these states, it should have similar transition starting from the other one, as well.

Example. Consider a $\mathcal{BA}$ system instance, P^3, with one general and two non-general processes ($N = 3$). Following Definition 42 and considering read/write restrictions of π_2, (an arbitrary) transition t_1:

$$([d = \{0, \bot, 0\}, b = \{0, 0, 1\}, f = \{1, 0, 1\}], \\ [d = \{0, 0, 0\}, b = \{0, 0, 1\}, f = \{1, 0, 1\}])$$

and transition t_2:

$$([d = \{0, \bot, 0\}, b = \{0, 0, 1\}, f = \{1, 0, 0\}], \\ [d = \{0, 1, 0\}, b = \{0, 0, 1\}, f = \{1, 0, 0\}])$$

have the same effect as far as π_2 is concerned (since π_2 cannot read $f[3]$). This implies that if t_1 is included in the set of transitions of a distributed program, then so should t_2. Otherwise, execution of t_1 by π_2 will depend on the value of $f[2]$, which, of course, π_2 cannot read.

Read restriction in abstract model. In a model in counter abstraction, the read restriction is different. The reason is that each state is determined based on the abstract number of processes in each local state, and hence, the exact information on the read-set of each process is not available. Before defining read restriction in counter abstraction, we need to have the definition of the abstract number of processes with some variable valuation. For a variable $v \in V$ and some valuation $val \in D_v$, the set of local states with variable v valued val is denoted by $\mathcal{P}_V^{(v=val)}$. More formally, $\mathcal{P}_V^{(v=val)} = \{\rho \in \mathcal{P}_V \mid \rho(v) = val\}$. For a set A with domain $0 \le i \le 2$, the abstract sum of the elements in A is defined as below:

$$\hat{\Sigma}(A) = \begin{cases} 0, & \text{if } \sum_{i \in A} i = 0 \\ 1, & \text{if } \sum_{i \in A} i = 1 \\ 2, & \text{if } \sum_{i \in A} i \ge 2 \end{cases}$$

We define the abstract number of a variable $v \in V$ with value val in an abstract state $\hat{\sigma} \in \Sigma_C$ as follows:

$$num(\hat{\sigma}, v, val) = \hat{\sum_{\rho \in \mathcal{P}_V^{(v=val)}}} \hat{\sigma}(\kappa_\rho)$$

Having the above definitions, we say that two abstract states $\hat{\sigma}_1$ and $\hat{\sigma}_2$ are *abstractly similar* ($\hat{\sigma}_1 \approx \hat{\sigma}_2$), if and only if the following condition holds:

$$\hat{\sigma}_1 \approx \hat{\sigma}_2 \iff (\forall\, v \in V^s \,.\, \forall val \in D_v \,.\, (num(\hat{\sigma}_1, v, val) = num(\hat{\sigma}_2, v, val)))$$

In other words, two abstract states are abstractly similar, if for each shared variable $v \in V^s$ and each value $val \in D_v$, the abstract number of the variables v with value val is the same in both abstract states.

Definition 43. *An abstract state $\hat{\sigma}_1$ is an* increment *of an abstract state $\hat{\sigma}_2$ with respect to the local state ρ, and is represented by $inc(\hat{\sigma}_1, \hat{\sigma}_2, \rho)$, if and only if: $\hat{\sigma}_1(k_\rho) = \hat{\sigma}_2(k_\rho) = 2 \;\vee\; \hat{\sigma}_2(k_\rho) = \hat{\sigma}_1(k_\rho) + 1$. An abstract state $\hat{\sigma}_1$ is a* decrement *of an abstract state $\hat{\sigma}_2$ with respect to the local state ρ, and is represented by $dec(\hat{\sigma}_1, \hat{\sigma}_2, \rho)$, if and only if: $inc(\hat{\sigma}_2, \hat{\sigma}_1, \rho)$.* □

Having the above definitions, we can define the read restriction constraint in counter abstraction as in equations 5.2 and 5.3:

$$\begin{aligned}
&\forall\,(\hat{\sigma}_1, \hat{\sigma}_1') \in \Delta_C \,.\, \forall\,\hat{\sigma}_2, \hat{\sigma}_2' \in \Sigma_C \,.\, (\hat{\sigma}_1 \approx \hat{\sigma}_2 \,\wedge \\
&(\forall v \in V^s \,.\, \forall\, 1 \le i \le K \,.\, \hat{\sigma}_1(v[k]) = \hat{\sigma}_1'(v[k]) \,\wedge\, \hat{\sigma}_2(v[k]) = \hat{\sigma}_2'(v[k])) \,\wedge \\
&(\exists \rho_1, \rho_2 \in \mathcal{P}_V \,.\, (inc(\hat{\sigma}_1, \hat{\sigma}_1', \rho_1) \wedge dec(\hat{\sigma}_1, \hat{\sigma}_1', \rho_2)) \,\wedge\, (inc(\hat{\sigma}_2, \hat{\sigma}_2', \rho_1) \wedge dec(\hat{\sigma}_2, \hat{\sigma}_2', \rho_2))))\wedge \\
&(\forall \rho \in \mathcal{P}_V \,.\, (\rho \neq \rho_1 \wedge \rho \neq \rho_2) \Rightarrow (\hat{\sigma}_1(k_\rho) = \hat{\sigma}_1'(k_\rho) \wedge \hat{\sigma}_2(k_\rho) = \hat{\sigma}_2'(k_\rho))) \\
&\quad \Longrightarrow\; (\hat{\sigma}_2, \hat{\sigma}_2') \in \Delta_C
\end{aligned} \tag{5.2}$$

$$\begin{aligned}
&\forall\,(\hat{\sigma}_1, \hat{\sigma}_1') \in \Delta_C \,.\, \forall\,\hat{\sigma}_2, \hat{\sigma}_2' \in \Sigma_C \,.\, (\hat{\sigma}_1 \approx \hat{\sigma}_2 \,\wedge \\
&\Big((\exists v \in V^s \,.\, \exists\, 1 \le i \le K \,.\, \hat{\sigma}_1(v[k]) \neq \hat{\sigma}_1'(v[k]) \,\wedge\, \hat{\sigma}_2(v[k]) \neq \hat{\sigma}_2'(v[k]) \,\wedge \\
&(\forall w \in V^s \,.\, \forall\, 1 \le j \le K \,.\, (w \neq v \vee j \neq i) \Rightarrow \hat{\sigma}_1(v[k]) = \hat{\sigma}_2(v[k]) \,\wedge\, \hat{\sigma}_2(v[k]) = \hat{\sigma}_2'(v[k]))\Big) \,\wedge \\
&(\forall \rho \in \mathcal{P}_V \,.\, \hat{\sigma}_2(k_\rho) = \hat{\sigma}_2'(k_\rho))) \\
&\quad \Longrightarrow\; (\hat{\sigma}_2, \hat{\sigma}_2') \in \Delta_C
\end{aligned} \tag{5.3}$$

In equation 5.2, the read restriction is imposed, where a process with index greater than K is executing, while equation 5.2 guarantees read restriction for the case a process with index less than K is executing. Note that we are considering asynchronous systems, and hence, in each step, only one process can take an action. In equation 5.2, the abstract number of a local state is decreasing in both transitions, and the abstract number of another is increasing, while the number of all other local states, as well as the variables of fixed processes do not change. In equation 5.3, a fixed process changes the value of one of its variables, and the number of all local states remain unchanged. In both cases, the two sources are pairwise abstractly similar.

Example. As an example, in the counter abstraction model of our $\mathcal{BA}$ example, a local state is a function $\rho : \{d, f, b\} \mapsto \{\mathit{true}, \mathit{false}, 0, 1, \bot\}$. Considering the domain of each variable, there are 12 possible local states ($|\mathcal{P}_V| = 12$). We represent an abstract state $\hat{\sigma}$ by a tuple t, consisting of a 12 values tuple t', and two values, d_g and b_g. d_g and b_g corresponds to the decision and Byzantine bits of the general process, and each element in t' corresponds to $\hat{\sigma}(\kappa_\rho)$, for some $\rho \in \mathcal{P}_V$. For a local state ρ, where $\rho(d) = d_1$, $\rho(f) = f_1$, and $\rho(b) = b_1$, $(d_1 \times 4) + (f_1 \times 2) + b_1$ is the index of $\hat{\sigma}(\kappa_\rho)$ in t'. For example, the abstract number of the processes in the local state $[0, 1, 1]$ is represented in $t'[3]$. Considering this notation, the transition from the state $[\{0, 0, 0, 2, 0, 0, 0, 1, 0, 0, 0, 0\}, d_g = 1, b_g = 0]$ to the state $[\{0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 0\}, d_g = 1, b_g = 0]$, and the transition from the state $[\{0, 0, 0, 2, 0, 1, 0, 0, 0, 0, 0, 0\}, d_g = 1, b_g = 0]$ to $[\{0, 0, 0, 2, 0, 1, 0, 1, 0, 0, 0, 0\}, d_g = 1, b_g = 0]$ are in the same group. The action corresponding to both transitions corresponds to a non-general process in the local state $[2, 0, 0]$, changing its decision to 1 (going to the local state $[1, 0, 0]$). The set of shared variables, V^s, includes the decision bits of all processes. The abstract number of non-generals with decision 0, decision 1, and decision 2 are the same in the source/destination of both transitions (their cardinality is 0,1,2 in the sources and 0,2,2 in the destinations, respectively), as well as the decision of the general process (it is 1 in both transitions).

5.4.3 Algorithm

Algorithm Sketch. We are given a process template $P_F = \langle V, G, F \rangle$, a constant K, a specification SPEC, and an atomic proposition LS. The goal of our algorithm is to synthesize a process template P', and legitimate states LS' from P, such that for every $N \geq K$, the set I'_N is not empty in P'^N, all LTL properties satisfied by P^{num} are also satisfied by P'^{num}, and P'^N_F is F-tolerant to SPEC from LS'. The algorithm consists of six steps (Algorithm 7).

In the first step, an abstraction of the parameterized system is generated using the counter abstraction method [72]. In the second step, the unreachable states are removed from the state-space. In Step 3, we check the state-space to identify the states where state-based safety specification is violated, and also the ones from where faults alone lead the execution to the state-based specification violation. Those states are removed from the state-space. In Step 4, we resolve deadlock states by adding recovery transitions to the model. The unresolved deadlock states are then removed in Step 5. Finally, in Step 6, we

Function 7 Parameterized_Synthesis

Input: A counter abstraction $C = (\Sigma_C, \mathbf{I}_C, \Delta_C, \lambda_C)$, a specification *SPEC*, and a set *LS* of legitimate states.

Output: If successful, a counter abstraction $C' = (\Sigma'_C, (\mathbf{I}_C)', \Delta'_C, \lambda'_C)$, and legitimate states LS'.

```
 1: $LS' = LS$
 2: $\Sigma'_C = \Sigma_C$
 3: $\Delta'_C = \Delta_C$
 4: repeat
 5:     $LS'' = LS'$
 6:     repeat
 7:         $\Delta''_C = \Delta'_C$
 8:         repeat
 9:             $\Sigma''_C = \Sigma'_C$
10:             $\Sigma'_C = =$ remove_unreachable $(\Sigma'_C, \Delta'_C)$
11:             $\Sigma'_C =$ remove_fte $(\Sigma'_C, \Delta'_C, fte)$
12:             $\Sigma'_C =$ remove_badstate $(\Sigma'_C, \Delta'_C)$
13:         until $\Sigma''_C = \Sigma'_C$
14:         add_recovery $(\Sigma'_C, \Delta'_C)$
15:         $(fte, ofd) =$ remove_deadlock $(\Sigma'_C, \Delta'_C, fte, ofd)$
16:     until $\Delta''_C = \Delta'_C$
17:     $LS' =$ construct_LS $(LS', ofd)$
18: until $LS'' = LS'$
19: $(\mathbf{I}_C)' = \{\sigma \mid \sigma \in \Sigma'_C \wedge \sigma \in \mathbf{I}_C\}$
20: return $(\Sigma'_C, (\mathbf{I}_C)', \Delta'_C, \lambda'_C)$
```

ensure that the invariant is closed in the synthesized program. We repeat steps 2-3, 2-5, and 2-6 until a fixpoint is reached, which means no more progress is possible. We represent the fixpoint computation by nested repeat-until loops in algorithm 7.

5.4.3.1 Step 1

In the first step, the abstract model $C = (\Sigma_C, \mathbf{I}_C, \Delta_C, \lambda_C)$ is generated from the fault-intolerant parameterized system. After generating the abstract model, we put a flag, *remove*, for each state and transition (initially set as *false*) that shows whether it has been removed during the synthesis or not. If we need to put the state or transition back, we simply change the value of *remove* to *false*. After generating the counter abstraction C, we call algorithm 7 on C, *SPEC*, and *LS*.

5.4.3.2 Step 2

In line 10 of Algorithm 7, a function named remove_unreachable is called to identify and remove the unreachable states of the model. An unreachable state is the one with no incoming transitions. For removing the unreachable states, we simply put their *remove* flags to *true*. The outgoing transitions from unreachable states are not removed to keep the maximum reachability in the model. Initially, this step might not change anything, but after removing transitions in the next steps of the algorithm, there will be unreachable states that should be removed by this step.

5.4.3.3 Step 3

In lines 11 and 12 of Algorithm 7, a set of states become unreachable. These states consist of:

- *fte*, *failed to eliminate* states, which are identified in Step 5. We will explain more about them later in this section.
- Bad states, which are the states that violate the state-based safety specification.

In order to remove bad states, we should make them unreachable, so that no system computation can reach them and violate the specification. Moreover, if a bad state is reachable only by faults, those states should also become unreachable, since we don't have control over the execution of faults.

5.4.3.4 Step 4

We don't allow deadlock states in the synthesized program. For abstract states out of LS, being deadlocked means violation of fault-tolerance, and for abstract states inside LS, being deadlocked means violating the second requirement of our problem statement. Hence, we don't want any deadlock states in our synthesized program. In this algorithm, deadlock states are resolved by either adding recovery paths (this step) or deadlock state elimination (next step).

In this step, we identify deadlock states outside LS, and for each, try to resolve it by adding a recovery transition to a state inside LS. We could add a recovery transition $(\hat{\sigma}_1, \hat{\sigma}_2)$, if the following conditions hold:

1. the asynchronous condition:

   $$(\exists j \in \mathcal{I}_K \,.\, \exists x \in V:\ \hat{\sigma}_1(x[j]) \neq \hat{\sigma}_2(x[j]) \wedge$$
   $$(\forall i \in \mathcal{I}_K \,.\, \forall y \in V\ :\ (i \neq j \vee y \neq x) \rightarrow \hat{\sigma}_1(y[i]) = \hat{\sigma}_2(y[i])) \wedge (\forall \rho \in \mathcal{P}_V:\ \hat{\sigma}_1(\kappa_\rho) = \hat{\sigma}_2(\kappa_\rho))) \vee$$
   $$(\forall i \in \mathcal{I}_K \,.\, \forall x \in V:\ \hat{\sigma}_1(x[i]) = \hat{\sigma}_2(x[i])) \wedge (\exists \rho_1, \rho_2 \in \mathcal{P}_V\ :\ inc(\hat{\sigma}_1, \hat{\sigma}_2, \rho_1) \wedge dec(\hat{\sigma}_1, \hat{\sigma}_2, \rho_2)) \wedge$$
   $$(\forall \rho \in \mathcal{P}_V:\ \rho \neq \rho_1 \wedge \rho \neq \rho_2 \rightarrow \hat{\sigma}_1(\kappa_\rho) = \hat{\sigma}_2(\kappa_\rho))$$

   This condition is required, since we are synthesizing an asynchronous system, and hence, there could be only one process that is changing from one local state to another. Based on the above condition, either a fixed process changes its local state, and all other processes remain unchanged, or all fixed processes are unchanged, and a parameterized process changes its local state. The latter is identified by changing the cardinality of one or two local states.

2. $(\hat{\sigma}_1, \hat{\sigma}_2) \models \psi$

   This condition is required, as the synthesized program should satisfy the given specification.

3. It does not create a *bad cycle* in the system.

Based on the first condition, when a parameterized process is taking an action, either there is a local state which cardinality is decreasing by one, and one local state which cardinality is increasing by one, or there is one local state decreasing (increasing) by one, and at least one unchanged local state with cardinality 2. When two local states are changing, we could find out which action is being taken by a process, and check it with the transition-based specification (second condition). If only one local state changes, then the local states with value 2 are the candidates for the other changing state. For each candidate, we check the corresponding action to see if it satisfies the second condition. If it does, then it could be a candidate to be checked for third condition.

Fault-tolerance is violated (third condition in Definition 38) if there exists a cycle outside the set of legitimate states. In counter abstraction, not all cycles will lead to loops in the corresponding instances.

Definition 44. *A* bad cycle *in a counter abstraction is a cycle consisting of only abstract states in* $\neg LS$*, with the condition that there exists a system instance for which the bad cycle is instantiated to a cycle.*

Below, we first give a brief introduction to *justice properties* [72] and then explain how we have used them in detecting bad cycles in our synthesis problem.

Definition 45. Justice requirements *are a set $J = \{J_1, \cdots, J_k\}$, such that each $J_i \in J$ is an assertion to ensure that each computation visits infinitely many J_i-states (states satisfying J_i).* $\square$

Definition 46. *Let φ be an assertion over the abstract state variables. We say that φ* suppresses the justice requirement *J, if for every two states σ_1 and σ_2 in a system instance with N processes, such that $(\sigma_1, \sigma_2) \in \Delta_N$, and both abstract states $\alpha_N(\sigma_1)$ and $\alpha_N(\sigma_2)$ satisfy φ, then $\sigma_1 \models \neg J$ implies $\sigma_2 \models \neg J$.* $\square$

Definition 47. *An assertion φ is called* justice suppressing *if, for every state σ such that $\alpha_N(\sigma)$ satisfies ϕ, there exists a justice requirement J such that $\sigma \models \neg J$ and φ suppresses J.* $\square$

It is shown in [72] that if we can find justice suppressing properties, we can safely add their negations to the set of justice properties (without violating the soundness of checking properties). In [72], a set of guidelines are presented to find justice suppressing properties in counter abstractions. As an example, one of the guidelines is as follows:

G1. If the system instance has the justice requirement $\neg(\pi[i] = l)$, then the assertion $k_l = 1$ is a justice suppressing property.

As an example, assume each local state ρ in $\mathcal{BA}$ is shown by a triple $[d, f, b]$, where d, f, and b are the variables for decision, finalization, and Byzantine bit respectively. We can find a justice suppressing property for $\mathcal{BA}$, using the above guideline. In $\mathcal{BA}$, $\neg(\pi[i] = [1, 0, 0])$ is a justice requirement, as each non-Byzantine process with a decision 1 should have a chance to finalize, unless it gets Byzantine. Hence, a process can not stay in that state infinitely often. Having this justice requirement for system instances, one justice suppressing property for the Byzantine agreement counter abstraction is $k_{[1,0,0]} = 1$, and hence, $\neg(k_{[1,0,0]} = 1)$ is a justice property for the counter abstraction of $\mathcal{BA}$.

Using the guidelines in [72], we can find a set of justice suppressing properties for the system under study. For each added transition, if the first two conditions are satisfied, we check if it is syntactically forming a new cycle in counter abstraction. If so, we check the abstract states in the cycle with our justice properties. If there exists a property for which all states in the new cycle do not satisfy it, we can safely add the transition. We mark each added transition with the decreasing and increasing indices, so at the end we can modify the guarded commands of the original program by adding the new transitions (actions) to them.

5.4.3.5 Step 5

In this step, the deadlock states that could not be resolved using the previous step, are removed. This is done by making the deadlock states unreachable. There are two types of transitions reaching a state; program transitions, and faults. If the state is reachable by a fault, we should backtrack, and make the source of the fault unreachable. The reason is that we cannot avoid a fault from happening. Such states are added to the list of "failed to eliminate" states (*fte*). If the source of the fault is in LS, it is added to the *ofd* list (to be removed in the next step), and if not, it is added to the set of deadlock states to be removed in the next rounds of this step. To remove a state from the set of deadlock states, all incoming transitions to each deadlock state, along with the transitions in its group, are removed from the counter abstraction. If this removal makes a new deadlock state (called *nds*) in the counter abstraction, we put all the transitions back to the model, and add *nds* to the set of deadlock states to be removed in the next rounds of this step.

5.4.3.6 Step 6

In this step, we make the states in *ofd* unreachable. If this removal leads to creating new deadlock states in LS, we make them unreachable as well to ensure the satisfaction of all LTL properties satisfied by the original program. If all initial states get removed in this step, the algorithm will terminate with failure.

5.5 Proof of Soundness

To prove the soundness of our algorithm, we show that all the requirements in our problem statement are satisfied by the algorithm.

1. The first condition requires that the set of initial states is not empty. This is guaranteed, as $I_N \subseteq LS$, and legitimate states can be removed in Step 6 of the algorithm. In this step, we check for the emptiness of the set of initial states, and it that happens, the algorithm returns with failure.

2. The second condition guarantees that all LTL formulas satisfied by the original system in the absence of faults are also satisfied by the synthesized system. This is true, since no state nor transition is added to LS in the synthesis. Also, no new deadlock states are added to the model.

3. The last condition is that the resulting system is F-tolerant. We show the satisfaction of this requirement by showing that each condition in the definition of fault-tolerance is respected:

- LS in the resulting system is closed due to the fact that no behaviour is added to LS, and also the last step of the algorithm. Every state in LS also satisfies ψ, since no state is added to LS, and the original program satisfies ψ.
- Every computation starting from LS, and taking program transitions or faults satisfies the specification ψ. This is correct since the original program satisfies ψ, and no state is added to the system. Hence, all state-based properties are preserved. The transition-based properties are also preserved, since when adding each recovery transition, the corresponding action is checked with respect to the that property. We should prove that if no prohibited transition is added in the abstract level, then no prohibited transition is added in any of the concrete models as well. We can prove that by proof by contradiction. Assume that there exists a prohibited transition in a concrete model. Then, there should exists at least one corresponding prohibited transition in the abstract model, and we know that no prohibited transition is added in the abstract level. Hence, the condition is satisfied.
- The last condition guarantees convergence to LS, starting from any state in LS, and taking any transition in program transitions or faults. This is also satisfied, since there are finite number of states in the state-space of the program, and none of states in $\neg LS$ are deadlocked. Since there is no cycle in $\neg LS$ either, all computations starting from a state in $\neg LS$ eventually get to LS.

5.6 Case Study and Experimental Results

We have applied our algorithm on parameterized Byzantine agreement problem as presented in Section 5.2.1.3. Our experiment is run on a machine with Intel Core i5 2.6 GHz processor with 8GB of RAM, and the synthesis time is 72 seconds. Below we present the added and removed transitions after applying our synthesis method:

Added transitions:

$$(d[1] = 1) \wedge (d[\mathsf{id}] = \bot) \ \wedge \ \neg f[\mathsf{id}] \wedge \neg b[\mathsf{id}] \wedge (\exists i, j \,.\, i \neq j \wedge d[i] = 0 \wedge d[j] = 0) \wedge$$
$$\neg(\exists i, j \,.\, i \neq j \wedge d[i] = 1 \wedge d[j] = 1) \rightarrow d[\mathsf{id}] := 0 \wedge f[\mathsf{id}] := \mathit{true}/\mathit{false}$$

$$(d[1] = 0) \wedge (d[\mathsf{id}] = \bot) \ \wedge \ \neg f[\mathsf{id}] \wedge \neg b[\mathsf{id}] \wedge (\exists i, j \,.\, i \neq j \wedge d[i] = 1 \wedge d[j] = 1) \wedge$$
$$(\nexists i \,.\, d[i] = 0) \rightarrow d[\mathsf{id}] := 1 \wedge f[\mathsf{id}] := \mathit{true}$$

$$(d[1]=0) \wedge (d[\mathsf{id}]=\bot) \ \wedge\ \neg f[\mathsf{id}] \wedge \neg b[\mathsf{id}] \wedge (\exists i,j \,.\, i \neq j \wedge d[i]=1 \wedge d[j]=1) \wedge$$
$$\neg(\exists i,j \,.\, i \neq j \wedge d[i]=0 \wedge d[j]=0) \rightarrow d[\mathsf{id}] := 1$$

$$(d[\mathsf{id}]=1) \ \wedge\ \neg f[\mathsf{id}] \wedge \neg b[\mathsf{id}] \wedge (\nexists i \,.\, i \neq \mathsf{id} \wedge d[j]=1) \wedge$$
$$\big[\big(\exists i,j \,.\, d[i]=0 \wedge d[j]=\bot \wedge (d[1]=0)\big) \vee$$
$$\big((\exists i,j \,.\, i \neq j \wedge d[i]=0 \wedge d[j]=0) \wedge \neg(\exists i,j \,.\, i \neq j \wedge d[i]=\bot \wedge d[j]=\bot)\big)\big]$$
$$\rightarrow d[\mathsf{id}] := 0 \wedge f[\mathsf{id}] := \mathit{true}$$

$$(d[\mathsf{id}]=1) \ \wedge\ \neg f[\mathsf{id}] \wedge \neg b[\mathsf{id}] \wedge (\nexists i \,.\, i \neq \mathsf{id} \wedge d[j]=1) \wedge$$
$$\big[\big(\exists i,j \,.\, d[i]=0 \wedge d[j]=\bot \wedge \nexists k \,.\, (k \neq i \wedge k \neq j \wedge (d[k]=0 \vee d[k]=\bot)) \wedge (d[1]=0)\big) \vee$$
$$\big(\exists i,j \,.\, i \neq j \wedge d[i]=0 \wedge d[j]=0\big)\big] \rightarrow d[\mathsf{id}] := 0$$

$$(d[\mathsf{id}]=0) \ \wedge\ \neg f[\mathsf{id}] \wedge \neg b[\mathsf{id}] \wedge (d[1]=1) \wedge (\nexists i \,.\, i \neq \mathsf{id} \wedge d[j]=0) \wedge$$
$$(\exists i,j \,.\, d[i]=1 \wedge d[j]=\bot) \rightarrow d[\mathsf{id}] := 1 \wedge f[\mathsf{id}] := \mathit{true}$$

$$(d[\mathsf{id}]=0) \ \wedge\ \neg f[\mathsf{id}] \wedge \neg b[\mathsf{id}] \wedge (d[1]=1) \wedge (\nexists i \,.\, i \neq \mathsf{id} \wedge d[j]=0) \wedge (\exists i \,.\, d[j]=\bot) \wedge$$
$$\neg(\exists i,j \,.\, i \neq j \wedge d[i]=1 \wedge d[j]=1) \rightarrow d[\mathsf{id}] := 1$$

$$(d[\mathsf{id}]=0) \wedge \neg f[\mathsf{id}] \wedge \neg b[\mathsf{id}] \wedge (\nexists i \,.\, i \neq \mathsf{id} \wedge d[j]=0) \wedge \neg(\exists i,j \,.\, i \neq j \wedge d[i]=\bot \wedge d[j]=\bot) \wedge$$
$$(\exists i,j \,.\, i \neq j \wedge d[i]=1 \wedge d[j]=1) \rightarrow d[\mathsf{id}] := 1 \wedge f[\mathsf{id}] := \mathit{true}/\mathit{false}$$

$$(d[\mathsf{id}]=0) \wedge \neg f[\mathsf{id}] \wedge \neg b[\mathsf{id}] \wedge$$
$$\big[\big((d[1]=0) \wedge (\exists i \,.\, i \neq \mathsf{id} \wedge d[i]=0) \wedge (\exists i \,.\, d[i]=1 \wedge \nexists j \,.\, j \neq i \wedge d[j]=1)\big) \vee$$
$$\big((d[1]=1) \wedge (\nexists i \,.\, i \neq \mathsf{id} \wedge d[i]=0) \wedge (\exists i,j \,.\, i \neq j \wedge d[i]=1 \wedge d[j]=1)\big) \vee$$
$$\big((d[1]=1) \wedge (\nexists i \,.\, i \neq \mathsf{id} \wedge d[i]=0) \wedge (\exists i,j \,.\, i \neq j \wedge d[i]=\bot \wedge d[j]=\bot) \wedge (\nexists i \,.\, d[i]=1)\big)\big]$$
$$\rightarrow d[\mathsf{id}] := \bot$$

$$(d[\mathsf{id}]=1) \wedge \neg f[\mathsf{id}] \wedge \neg b[\mathsf{id}] \wedge (d[1]=0) \wedge (\nexists i \,.\, d[i]=0) \wedge (\nexists i \,.\, i \neq \mathsf{id} \wedge d[i]=1) \wedge$$
$$(\exists i,j \,.\, i \neq j \wedge d[i]=\bot \wedge d[j]=\bot) \rightarrow d[\mathsf{id}] := \bot$$

Removed transitions:

$$(d[\mathsf{id}]=\bot) \wedge \neg f[\mathsf{id}] \wedge \neg b[\mathsf{id}] \wedge (d[1]=0) \wedge \neg(\exists i,j \,.\, i \neq j \wedge d[i]=0 \wedge d[j]=0) \wedge$$
$$(\exists i,j \,.\, i \neq j \wedge d[i]=1 \wedge d[j]=1) \rightarrow d[\mathsf{id}] := 0$$

$$(d[\mathsf{id}] = \bot) \wedge \neg f[\mathsf{id}] \wedge \neg b[\mathsf{id}] \wedge (d[1] = 1) \wedge (\exists i, j \,.\, i \neq j \wedge d[i] = 0 \wedge d[j] = 0) \wedge \\ \neg(\exists i, j \,.\, i \neq j \wedge d[i] = 1 \wedge d[j] = 1) \rightarrow d[\mathsf{id}] := 1$$

$$(d[\mathsf{id}] = 0) \wedge \neg f[\mathsf{id}] \wedge \neg b[\mathsf{id}] \wedge \big[\big(\exists i \,.\, d[i] = \bot\big) \vee \\ \big((\nexists i \,.\, i \neq \mathsf{id} \wedge d[i] = 0) \wedge (\exists i, j \,.\, i \neq j \wedge d[i] = 1 \wedge d[j] = 1) \wedge (\nexists i \,.\, d[i] = \bot)\big) \vee \\ \big((\exists i \,.\, i \neq \mathsf{id} \wedge d[i] = 0) \wedge (\exists i \,.\, d[i] = 1 \wedge \nexists j \,.\, j \neq i \wedge d[j] = 1) \wedge (\nexists i \,.\, d[i] = \bot) \wedge d[1] = 1\big)\big] \\ \rightarrow f[\mathsf{id}] := \mathit{true}$$

$$(d[\mathsf{id}] = 1) \wedge \neg f[\mathsf{id}] \wedge \neg b[\mathsf{id}] \wedge \big[\big((\nexists i \,.\, i \neq \mathsf{id} \wedge d[i] = 1) \wedge (\exists i \,.\, d[i] = \bot)\big) \vee \\ \big((\exists i \,.\, d[i] = 0 \wedge \nexists j \,.\, j \neq i \wedge d[j] = 0) \wedge (\exists i \,.\, i \neq \mathsf{id} \wedge d[i] = 1)\big) \vee \\ \big((\exists i, j \,.\, i \neq j \wedge d[i] = 0 \wedge d[j] = 0) \wedge (\nexists i \,.\, i \neq \mathsf{id} \wedge d[i] = 1) \wedge (\nexists i \,.\, d[i] = \bot)\big) \vee \\ \big((\nexists i \,.\, d[i] = 0) \wedge (\exists i \,.\, i \neq \mathsf{id} \wedge d[i] = 1) \wedge (\exists i, j \,.\, i \neq j \wedge d[i] = \bot \wedge d[j] = \bot) \wedge d[1] = 0\big)\big] \\ \rightarrow f[\mathsf{id}] := \mathit{true}$$

Chapter 6

Related Work

In this chapter, we review the literature related to the research in this dissertation. We are focusing on synthesis in two contexts; (1) adding specifications to an abstract model, (2) designing correct programs from specification.

6.1 Comprehensive Synthesis

In comprehensive synthesis, the input is a specification given in terms of a temporal property, and the goal is to synthesize a model that satisfies the given specification. Our work in synthesizing self-stabilizing systems is in nature close to these approaches.

Concurrent programs can be divided into two parts; (1) *synchronization part* that ensures the relative timing of processes execution, and (2) *functional part* that are the program computations and data manipulations. As an example, in a mutual exclusion program, the part that ensures the mutual exclusion between sections of code is the synchronization part, and the code that is being mutually exclusive is the functional part [67]. Most existing work in comprehensive synthesis are concerned with the synchronization part of concurrent programs. The reason for considering synchronization part in synthesis is that it is an intricate yet manageable task, since it needs attention to lots of details, but most of them do not need insight into lots of mathematical theories [67].

The seminal work in this area is the one by Emerson and Clark presented in [37]. In this work, the goal is to synthesize a *synchronization skeleton* from a given CTL formula. A synchronization skeleton is defined as an abstraction of the program where any detail irrelevant to synchronization is suppressed. The synthesis is based on the *bounded finite model property*, which asserts that if a formula in an appropriate propositional temporal logic is satisfiable, there exits a finite state model satisfying it, and the model size is bounded by a function of the length of the formula. A tableau-based decision procedure is proposed in [37] that takes a CTL formula f_0; if unsatisfiable, returns "NO, f_0 is unsatisfiable", and if satisfiable, returns "YES, f_0 is satisfiable", along with a finite model satisfying f_0. The procedure starts by building a tableau from the CTL formula, which is a finite directed AND/OR graph. In each step, an unreduced AND-node or OR-node is reduced. The steps continues as long as possible. If the root is removed, the CTL formula is not satisfiable. Otherwise, the remaining unreduced nodes are unraveled to a finite model.

Manna and Wolper in [67] propose a method to synthesize the synchronization part of communing processes from propositional linear temporal logic (PTL). In this work, concurrent computations are abstracted into sequences of events, and described using PTL. A tableau-based decision procedure is then used to synthesize the synchronization part of processes from the specification in PTL. The major distinction of this work and the one presented by Emerson and Clark in [37] is the underlying logic. The other difference is that the work in [37] focuses on the shared-memory model for concurrent programs, while in [67], the communication among processes are done through a synchronizer process, using the message-passing method.

The two approaches mentioned above suffer from *state explosion* problem. The reason is that the number of states for a concurrent system grows exponentially with the increase in the number of processes. In [10], Attie and Emerson propose an extension to these approaches to synthesize a concurrent system with K similar processes, where K is an arbitrarily large number. The idea is to reduce the problem of synthesizing a system with K similar processes to the problem of constructing the product of a pair of sequential processes, which is called a *pair-system*. They prove that two properties of the pair-systems are preserved by the K-processes systems: (1) a propositional invariant, and (2) a temporal *leads-to* property (if condition 1 is true, then condition 2 will be eventually true, as well). The other correctness properties such as liveness are not considered in this paper.

The methods proposed in [37] and [67] synthesize systems with computation models that are often unrealistic; a highly centralized topology in [67], and access to the global information about the system state in [37]. In [12], Attie and Emerson propose a method to synthesize a concurrent model in shared memory model, where operations are only atomic reads or atomic writes of a single variable. The proposed approach is to first synthesize a program with multiple assignment operations. Then, the operations are decomposed into sequences of atomic read/write operations. After decomposition, the program is checked to ensure that it still satisfies the given specification.

Comprehensive synthesis is also studied in the context of real-time systems. In [5], a method was introduced to synthesize timed automata from real-time temporal logic MITL formula. In [64], a simpler approach is proposed for the same goal.

6.2 Automated Synthesis of Fault-Tolerance

The formal characterization of notion of faults and fault-tolerance are first introduced by Arora and Gouda in [7]. Based on this formalization, the faults that a program is subject to can be systematically represented by a transition predicate. Representation of faults as transitions is possible for different types of faults (e.g., stuck-at, crash, fail-stop, timing, performance, Byzantine, etc.), nature of the faults (permanent, transient, or intermittent), or the ability of the program to observe the effects of the faults [17]. Examples of different types of faults (e.g., fail-stop, Byzantine, state corruption, message loss, etc) modeled in a transition system are presented in [21]. Using the fault and fault-tolerance characterization in [7], there has been a series of work on adding fault-tolerance to various types of systems with different levels of fault-tolerance, which we present in this chapter for two categories of untimed and real-time systems.

6.2.1 Synthesis of Fault-Tolerant Untimed Systems

Automated synthesis of fault-tolerance is pioneered by Attie, Arora, and Emerson in [11]. Their approach is mainly based on the comprehensive synthesis method in [37]. Given is a program specification in CTL, a fault specification, a problem-fault coupling specification, which is also a CTL formula, and a level of fault-tolerance. The algorithm starts by finding the finite model using the tableau-based approach proposed in [37]. Then, the fault actions are added to the synthesized model to find the new reachable states. After that, recovery transitions to the program invariant are added, and at the end, the augmented global state-transition diagram is pruned to find each process.

In a series of papers [56–60], automatic addition of fault-tolerance to fault-intolerant untimed programs is studied, where the problem requirement is that no new behaviors are added to the original program in the absence of faults. Note that since a fault-intolerant program is given as input, these methods can be used to add fault-tolerance incrementally. However, such reuse is not feasible in comprehensive synthesis, since for each new specification, the synthesis procedure should be performed from scratch. Similar weakness is valid for the work in [11], as it is based on comprehensive synthesis. This line of work is pioneered by Kulkarni and Arora in [56]. In this paper, the authors introduce polynomial-time sound and complete algorithms for adding all levels of fault-tolerance (failsafe, nonmasking, and masking) to centralized programs. They also show that the problem of adding masking fault-tolerance to distributed programs is NP-complete in the size of the input program's state space. A set of polynomial-time heuristics are introduced in [57] for the problem of adding masking fault-tolerance [1] to distributed programs. Later, a more efficient symbolic heuristic is introduced in [21] for a similar problem. The other work in synthesis of fault-tolerant distributed systems is the one presented in [33]. In this work, the fault-tolerance specification is given as a CTL^* formula, and the goal is to determine whether a fault-tolerant implementation exists for that specification in a fully connected topology (each pair of processes is connected by a communication link). If the answer is “yes”, it automatically generates the system. They also prove that the problem of fault-tolerant synthesis from CTL^* specifications for a fully connected topology is $\mathsf{2EXPTIME}$-complete.

The other line of research is to enhance the level of fault-tolerance in a program. In [58], the authors propose an algorithm that takes a nonmasking fault-tolerant program, and enhances its level of fault-tolerance to masking. This requires to add safety in the presence of fault, and preserve recovery to the legitimate states. The problem is studied for both centralized and distributed systems. The algorithm proposed for the centralized case is sound and complete, whereas the one for distributed systems in only claimed to be sound.

6.2.2 Synthesis of Fault-Tolerant Real-Time Systems

In [18], the notion of faults and fault-tolerance introduced in [7] are extended in the context of real-time systems. The authors also introduce various levels of fault-tolerance for real-time systems in the presence of faults, which include nonmasking, failsafe, and masking. For failsafe and masking fault-tolerance, they introduce two additional levels, namely soft and hard, based on satisfaction of timing constraints in the presence of faults. In particular, both soft and hard fault-tolerant programs are required to satisfy their timing constraints in the absence of faults. However, in the presence faults, a soft fault-tolerant program is not needed to satisfy its timing constraints, while it is a requirement in a hard fault-tolerant program.

Bonakdarpour and Kulkarni in [17] introduce the notion of *2-phase fault recovery*. Fault-tolerance requires that the system should eventually return to its ideal behavior, and the real-time nature of the system under study needs the recovery to be timely, and satisfying both requirements may not be possible. The idea in [17] is to enable the system to first recover to a safe or acceptable state quickly, and then return to its ideal behavior (2-phase fault recovery). For instance, in a traffic signal controller, if the controller detects a fault, all signals should first turn red immediately to prevent catastrophic consequences (phase 1) before final recovery to normal behavior (phase 2). The complexity of adding different levels of fault-tolerance is studied in [17], where the synthesis problem we are interested in,

[1]A masking fault-tolerant protocol is one that ensures constant satisfaction of safety and liveness specifications even in the presence of faults.

is shown to be polynomial in the size of the time-abstract bisimulation of the input model. The algorithm in [17] for solving this problem is just proposed for the purpose of showing complexity, and it is essentially impractical due to the use of *region graphs* as the semantic model for timed automata.

6.3 Synthesis of Self-Stabilizing Systems

The concept of self-stabilization was first introduced by Dijkstra in the seminal paper [29], where he proposed three solutions for designing self-stabilizing token circulation in ring topologies. Twelve years later, in a follow up article [30], he published the correctness proof, where he states that demonstrating the proof of correctness of self-stabilization was more complex than he originally anticipated. Indeed, designing correct self-stabilizing algorithms is a tedious and challenging task, prone to errors. Also, complications in designing self-stabilizing algorithms arise, when there is no commonly accessible data store for all processes, and the system state is based on the valuations of variables distributed among all processes [29]. Thus, it is highly desirable to have access to techniques that can automatically generate self-stabilizing protocols that are correct by construction.

In [54], the authors show that adding strong convergence is NP-complete in the size of the state space, which itself is exponential in the size of variables of the protocol. Ebnenasir and Farahat [35] also proposed an automated method to synthesize self-stabilizing algorithms. Our work is different in that the method in [35] is not complete for strong self-stabilization. This means that if it cannot find a solution, it does not necessarily imply that there does not exist one. However, in our method, if the SMT-solver declares "unsatisfiability", it means that no self-stabilizing algorithm that satisfies the given input constraints exists. Also, using our approach, one can synthesize synchronous and asynchronous programs, while the method in [35] synthesizes asynchronous systems only. Finally, our method is based on the constantly-evolving technique of SMT solving. We expect our technique to become more efficient as more efficient SMT solvers emerge.

The synthesis technique introduced in [55] uses a backtracking-based synthesis algorithm. It is complete and shows better scalability than our SMT-based technique. Having said that, SMT-solvers provide us with enormous power and allow us to push the boundaries of synthesis to cases where the description of the set of legitimate states is not given explicitly. Our results show that synthesis of self-stabilizing protocols using such implicit descriptions is indeed possible using SMT-solvers. The other limitation in [55] is that it needs the set of actions on the underlying variables in the legitimate states. This is not required in our approach, although if a developer prefers to specify the set of actions in LS, our synthesis method allows that using the constraint presented in Section 3.6.3.1. This limitation makes it impossible to synthesize protocols that are live in the legitimate states and whose behavior is too complicated for superposition to capture such as Dijkstra's token ring, while our approach has no limitation on that.

In [1], the authors propose a constraint-based automated addition of self-stabilization to hierarchical programs. To deal with transient faults, their technique adds recovery actions while ensuring interference freedom among the recovery actions added for satisfying different constraints. This method can successfully synthesize stabilizing Raymond's mutual exclusion algorithm [74] and stabilizing diffusing computation [8]. This is another instance of a heuristic that works only a class of protocols, namely, hierarchical distributed algorithms.

Gascón and Tiwari [43] propose a method to synthesize Dijkstra's four-state token ring protocol. The solution is based on solving the $\forall\exists$ game for $\Diamond\Box\varphi$ (i.e., 'eventually always') properties using a QBF-solver, as $\Diamond\Box\varphi$ essentially expresses strong self-stabilization. The authors suggest that since synthesizing a protocol for $\Diamond\Box\varphi$ properties is difficult, they replace $\Diamond\Box\varphi$ with $X^c\Box\varphi$ (i.e., always within c steps). They further simplify $\Box\varphi$ to $\varphi \wedge X\varphi$. Due to the bound on c and replacement of $\Box\varphi$ with $\varphi \wedge X\varphi$, the synthesized solution may not be sound. Thus, they verify it. This loop iterates until verification is positive. In our approach, synthesis is achieved in one shot.

6.4 Controller Synthesis

Controller synthesis is in spirit close to our work in adding fault-tolerance to systems. A Discrete Event System (DES) is a dynamic system that evolves according to the occurrence of physical events. Controller synthesis for DES is pioneered by Ramadge and Wonham in [73]. The discrete controller synthesis is formulated as the problem of getting two languages $\mathcal{U}$ and $\mathcal{D}$, where $\mathcal{U}$ is called the plant system, and $\mathcal{D}$ is the desired system. The goal is to synthesize a language $\mathcal{C}$, such that $\mathcal{U} \cap \mathcal{C} \subseteq \mathcal{D}$. Considering $\mathcal{U}$ to be the fault-intolerant program, $\mathcal{D}$ be the desired specification, and $\mathcal{C}$ be the subset of program transitions that satisfy the specification, we can see that controller synthesis can solve a similar problem to that in synthesizing fault-tolerant systems. There are a number of differences between our work and controller synthesis; first, there is no method of adding *recovery transitions*, as used in our algorithm, in controller synthesis; second, the computation model considered in controller synthesis is prioritized synchronization, whereas the model in our synthesis work on distributed fault-tolerant systems is based on interleaving. Finally, we model distribution by specifying *read/write restrictions*, whereas decentralized plants are modeled through *partial observability*.

Maler, et al. [66] propose an algorithm for synthesizing timed automata formulated by the notion of timed games. The idea is to define a predecessor function that finds the configurations from which the automaton can be forced to the desirable set of configurations, and the algorithm is a fixed-point iteration of this function. In [9], a similar problem is tackled, with the difference that the controller has the option of doing nothing and let the time pass, in addition to choosing among actions. An on-the-fly algorithm on synthesizing timed models using zone graphs is proposed in [25], which is implemented in the tool UPPAAL-TIGA [15]. The algorithm is a symbolic extension of the algorithm suggested by Liu and Smolka [62]. The main idea of this work is (1) to use a combination of forward algorithm, and backward propagation, which helps the algorithm to terminate as soon as a winning strategy is identified, and (2) to use zone graph as the underlying structure of the algorithm. We use similar ideas in the area of fault-recovery for timed models. The distinction of our work with these work (and also with [65]) is handling bounded response properties and, more importantly, adding recovery paths that the original model does not contain.

In bounded synthesis [42], given is a set of LTL properties, which are translated to a universal co-Büchi automaton, and then a set of SMT constraints are derived from the automaton. Our work in synthesis of self-stabilizing systems is inspired by this idea for finding the SMT constraints for convergence. For distribution and timing models, we use a different approach from bounded synthesis, as they are not temporal properties. The other difference is that the main idea in bounded synthesis is to put a bound on the number of states in the resulting state-transition systems, and then increasing the bound if a solution is not found. In our work, since the purpose is to synthesize a self-stabilizing system, the bound is the number of all possible states, derived from the given topology.

6.5 Game Theory

Game theory is another research line that is close to the automated synthesis of fault-recovery. In game-theoretic approaches [71], the synthesis of controllers and reactive programs are considered as a two-player game between the program and the environment [77]. The interaction of the program and the environment is through a set of interface variables. Hence, the environment is only allowed to change the value of interface variables, while in the synthesis of fault-recovery, faults can change the value of any variable. The other difference is that in a two-player game model, the set of states from where the first player (program) can make a move is disjoint from the set of states from where those that the second player can move from [78], while in our work, faults can occur in any state of the system. Similar to discrete controller synthesis, the other distinction of our work with game theoretic approaches is that the latter do not address the issue of addition of recovery. Also, in game theory, the notion of distribution is modeled by partial observability.

6.6 Parameterized Synthesis

There is extensive research on parameterized systems in the context of verification. Parameterized verification is known to be undecidable [6], although it can be decided for some restricted cases [27, 38]. Parametrized model checking of fault-tolerant distributed systems is studied in [52]. In this paper, fault-tolerant distributed systems are considered as message-passing systems of n processes, out of which at most t may be faulty. To verify these systems and avoid state explosion problem, *PIA counter abstraction* is used, and as a result, the parameterized problem is reduced to finite-state model checking. In counter abstraction, a counter is associate to each local state that stores how many processes are in the corresponding state. A global state is the union of all these counters. In PIA counter abstraction, the domain of each counter is a set of intervals, which is selected based on the system we want to model check. We use similar idea in synthesis of fault-tolerant distributed systems. The abstraction we use is $\{0, 1, \infty\}$-counter abstraction [72], in which the domain of each variable could be zero, one, or two, where two represents all values more than one.

In [50], the problem of synthesis of a reactive system from a parameterized temporal logical specification is studied. The problem is known to be undecidable for system topologies in which processes are incomparable with respect to their information about the environment. In [50], the problem is restricted to topologies with identical processes, and the considered specifications are in $\mathsf{LTL}_{\text{-}\mathsf{X}}$. This problem is still undecidable, and hence, bounded synthesis [42], which is a semi-decision procedure is used to tackle it. Using this approach, a simple parameterized arbiter is synthesized in reasonable time.

Chapter 7

Conclusion

In this chapter, we present a summary of our contributions in this thesis, as well as some of the related open problems and future research directions.

7.1 Summary

To summarize, we have done research in two directions: (1) synthesis of fault-tolerant real-time systems, and (2) synthesis and repair of distributed fault-tolerant systems. In our first research direction, we focused on synthesizing fault-tolerant timed models from their intolerant version. The type of fault-tolerance under investigation is *strict 2-phase recovery*, where upon occurrence of faults, the system is expected to recover in two phases, each satisfying certain constraints. Our contribution is a synthesis algorithm that adds 2-phase strict fault recovery to a given timed model, while not adding new behaviors in the absence of faults. The latter is ensured with the fact that our algorithm adds no transition originating from a legitimate state of the input model. We used an space-efficient representation of timed models, known as the *zone graph*. To our knowledge, this is the first instance of such an algorithm. Our experiments showed that the proposed algorithm can compete with model checking, where the synthesis time is proportional to the corresponding verification time (zone graph generation time for the input model using the IF toolset). Note that synthesis is a significantly more complex problem compared to mode checking (*i.e.* satisfiability vs. verification).

The other type of systems we focused on are distributed systems. Synthesis of distributed fault-tolerant systems is studied in [21]. One of the major shortcomings of the algorithm proposed in [21] is the time complexity. Hence, using the proposed algorithm, a system with a bounded number of processes can be synthesized and, thus, the solution cannot be generalized to any number of processes. Our goal was to propose an automated parameterized method for synthesizing masking fault-tolerant distributed protocols from their fault-intolerant version. Such protocols are parameterized by the number of processes.

We have designed a synthesis algorithm that utilizes *counter abstraction* [72] to construct a finite representation of the state space. Then, it performs fixpoint calculations to compute and exclude states that violate the safety specification in the presence of faults. To guarantee liveness, our algorithm ensures deadlock freedom and augments the input intolerant protocol with safe recovery paths, while ensuring that no cycle is added outside of legitimate states. In counter abstraction, not every cycle is a real cycle, which can violate the liveness specifications. Hence, to find out if an added recovery transition is actually forming a cycle outside the legitimate states, we use *justice properties* as introduced in [72]. We use the guidelines presented in [72] to find justice properties for our case studies, and use them to distinguish cycles violating recovery to the legitimate states. We demonstrate the effectiveness of our algorithm by synthesizing a fault-tolerant distributed agreement protocol in the presence of Byzantine fault. Although the synthesis problem is known to be NP-complete in the state space of the input protocol (due to partial observability of processes) in the non-parameterized setting, our parameterized algorithm manages to synthesize a solution for a complex problem such as Byzantine agreement within less than two minutes.

Self-stabilizing systems are a well-known type of distributed fault-tolerant systems, in which, the system converges to its set of legitimate states, starting from any state (due to wrong initialization or fault occurrence). We proposed an automated technique for synthesis of finite-sized self-stabilizing algorithms using SMT-solvers. The first benefit of our technique is that it is sound and complete; i.e., it generates distributed programs that are correct by construction and, hence, no proof of correctness is required, and if it fails to find a solution, we are guaranteed that there does not exist one. The latter is due to the fact that all quantifiers range over finite domains and, hence, finite memory is needed for process implementations. This assumptions basically ensures decidability of the problem under investigation. Secondly, our method is fully automated and can save huge effort from designers, specially when there is no solution for the problem. Third, the underlying technique is based on SMT-solving, which is a fast evolving area, and hence, by introducing more efficient SMT-solvers, we expect better results from our proposed method. We have also extended our approach to support cases where the legitimate states is not explicitly given as a set of states, and also synthesis of ideal-stabilizing systems. We reported highly encouraging results of experiments on a diverse set of case studies on some of the well-known problems in self-stabilization.

When synthesizing self-stabilizing systems, what may be of interest to the user is the *average recovery time* of the system, which is the expected number of steps the system takes before reaching its set of legitimate states. We focused on automated repair of self-stabilizing systems under recovery time constraints. Our investigation shows that the computational complexity of the problem is NP-complete in the size of the input program state space. This result is shown by a reduction from the 3-SAT problem to our synthesis problem.

7.2 Future Work

In this section, we present some future research directions in the context of synthesizing fault-tolerant systems (Section 7.2.1) and synthesizing self-stabilizing systems (Section 7.2.2).

7.2.1 Open Problems Related to Synthesis of Fault-Tolerant Systems

Synthesis of Fault-Tolerant Real-Time Systems. In the context of synthesizing fault-tolerant real-time systems, an open problem is to investigate a method for improving the efficiency of our proposed synthesis method. As discussed in Section 2.6, a possible bottleneck of our method is on the step of adding recovery transitions. Our idea of ranking the zones and updating the ranks dynamically has significant effect on the efficiency of this step. But we believe that this phase may still be improved by introducing heuristics for edge selection. Note that applying heuristics can help the efficiency of the method in adding recovery transitions in the cost of losing completeness in this step. The other research direction is to consider other cases of synthesizing fault-tolerant real-time systems with 2-phase fault recovery, which are known to be NP-complete in the size of the detailed region graph of the input automaton [17]. The goal is to propose an efficient heuristic for synthesis in these cases using zone graphs, and conduct experiments to see if the synthesis can work efficiently in practice, despite the high complexity in theory.

Parameterized Synthesis of Fault-Tolerant Distributed Systems. In the context of parameterized synthesis of fault-tolerant distributed systems, one research direction is to investigate the complexity of the problem. Although we have proposed a heuristic for solving the problem, the complexity of synthesizing a distributed fault-tolerant system that works for any number of processes is unknown to us.

Other Research Directions. Another open problem in the context of synthesizing fault-tolerant systems is to investigate the synthesis of the most general version of the Byzantine generals problem where multiple faults can occur. The problem cannot be solved using the method proposed in [21]. One can investigate both parameterized and non-parameterized cases for solving this problem. Another research direction is to investigate the possibility of using SMT-solvers in synthesis of fault-tolerant systems to improve the efficiency of methods. It could be done by either using an SMT-based approach for the whole process, or to find the bottleneck steps of heuristics and see if SMT-solvers can help in those steps.

7.2.2 Open Problems Related to Synthesis of Self-Stabilizing Systems

Synthesis of Self-Stabilzing Systems *Inductive synthesis* is the process of generating a system from input-output examples. For each input-output example, the system is refined, until convergence is reached. The examples used for debugging can be negative or positive. The negative ones can be counter-examples found during checking the correctness of the program. A future research direction is to investigate the technique of counter-example guided inductive synthesis (CEGIS) [75] along with our SMT-based method for synthesizing self-stabilizing systems. That may be an interesting solution to the problem of scaling the synthesis process for larger number of processes.

The other research direction is to design an algorithm to synthesize parameterized self-stabilizing systems using SMT-solvers. We believe that using the abstraction methods, such as counter abstraction, we are able to automatically synthesize parameterized self-stabilizing systems.

Another research direction related to our SMT-based synthesis approach is to use SMT-solvers other than Alloy, such as Z3 or Yices, and see if there are cases where these solvers work better than Alloy. One way to improve the efficiency while working with these solvers is to unroll all quantifiers. This leads to having very big SMT instances, but it may help the efficiency of finding a solution significantly.

Repair of Self-Stabilzing Systems Following our NP-completeness results on repairing self-stabilizing systems, one future research direction is to investigate if there exists any heuristic with constant approximation ratio for repairing self-stabilizing systems under recovery time constraints, and if there exists none, provide an impossibility proof.

A heuristic is proposed in [2] to synthesize self-stabilizing systems under average recovery time constraints. One research direction is to investigate if there exists any better heuristic that works more efficiently in practice. The other research direction is to propose heuristics for synthesis and repair of self-stabilizing systems under average recovery time constraints in the parameterized case. More formally, given a parameterized self-stabilizing system, how we can repair the protocol such that the average recovery time of the resulting system is less than a specific number, when instantiated for any number of processes.

References

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

[2] Saba Aflaki, Fathiyeh Faghih, and Borzoo Bonakdarpour. Synthesizing self-stabilizing protocols under average recovery time constraints. In *International Conference on Distributed Computing Systems (ICDCS)*, 2015, To appear.

[3] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[4] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[5] R. Alur, T. Feder, and T.A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, 1996.

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

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

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

[9] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *IFAC Symposium on System Structure and Control*, pages 469–474, 1998.

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

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

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

[13] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[14] C. Baier and J-.P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[15] Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. UPPAAL-Tiga: Time for playing games! In *Computer-Aided Verification (CAV)*, pages 121–125, 2007.

[16] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124, 2003.

[17] B. Bonakdarpour and S. S. Kulkarni. Synthesizing bounded-time 2-phase recovery. In *Springer journal of Formal Aspects of Computing (FAOC)*. To appear.

[18] B. Bonakdarpour and S. S. Kulkarni. Automated incremental synthesis of timed automata. In *International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, LNCS 4346, pages 261–276, 2006.

[19] B. Bonakdarpour and S. S. Kulkarni. Incremental synthesis of fault-tolerant real-time programs. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 122–136, 2006.

[20] B. Bonakdarpour and S. S. Kulkarni. Revising distributed UNITY programs is NP-complete. In *Principles of Distributed Systems (OPODIS)*, pages 408–427, 2008.

[21] B. Bonakdarpour, S. S. Kulkarni, and F. Abujarad. Symbolic synthesis of masking fault-tolerant programs. *Springer Journal on Distributed Computing*, 25(1):83–108, March 2012.

[22] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *Compositionality: The Significant Difference (COMPOS)*, pages 103–129, 1997.

[23] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, and Laurent Mounier. IF: An intermediate representation and validation environment for timed asynchronous systems. In *World Congress on Formal Methods*, pages 307–327, 1999.

[24] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *International Conference on Concurrency Theory (CONCUR)*, pages 66–80, 2005.

[25] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *International Conference on Concurrency Theory (CONCUR)*, pages 66–80, 2005.

[26] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[27] Edmund M. Clarke, Muralidhar Talupur, Tayssir Touili, and Helmut Veith. Verification by network decomposition. In *International Conference on Concurrency Theory (CONCUR)*, pages 276–291, 2004.

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

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

[30] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, 1986.

[31] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ., 1990.

[32] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *International workshop on Automatic verification methods for finite state systems*, pages 197–212, 1990.

[33] R. Dimitrova and B. Finkbeiner. Synthesis of fault-tolerant distributed systems. In *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 321–336, 2009.

[34] Shlomi Dolev and Elad Schiller. Self-stabilizing group communication in directed networks. *Acta Informatica*, 40(9):609–636, 2004.

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

[36] E. A Emerson. *Handbook of Theoretical Computer Science*, volume B, chapter 16: Temporal and Modal Logics. Elsevier Science Publishers B. V., Amsterdam, 1990.

[37] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

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

[39] E. Allen Emerson and Richard J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 142–156, 1999.

[40] N. Fallahi and B. Bonakdarpour. How good is weak-stabilization? In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 148–162, 2013.

[41] N. Fallahi, B. Bonakdarpour, and S. Tixeuil. Rigorous performance evaluation of self-stabilization using probabilistic model checking. In *Proceedings of the 32nd IEEE International Conference on Reliable Distributed Systems (SRDS)*, pages 153 – 162, 2013.

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

[43] A. Gascón and A. Tiwari. Synthesis of a simple self-stabilizing system. In *Proceedings of the 3rd Workshop on Synthesis (SYNT)*, pages 5–16, 2014.

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

[45] M. G. Gouda and H. B. Acharya. Nash equilibria in stabilizing systems. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 311–324, 2009.

[46] T. A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43(3):135–141, 1992.

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

[48] Su-Chu Hsu and Shing-Tsaan Huang. A self-stabilizing algorithm for maximal matching. *Information Processing Letters*, 43(2):77–81, 1992.

[49] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press Cambridge, 2012.

[50] S. Jacobs and R. Bloem. Parameterized synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 362–376, 2012.

[51] S. Jacobs and R. Bloem. Parameterized synthesis. *Logical Methods in Computer Science*, 10(1), 2014.

[52] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Parameterized mdel checking of fault-tolerant distributed algorithms by abstraction. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 201–209, 2013.

[53] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 201–209, 2013.

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

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

[56] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 82–93, 2000.

[57] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of Byzantine agreement. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 130–140, 2001.

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

[59] S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. *International Conference on Distributed Computing Systems*, 2003.

[60] S. S. Kulkarni and A. Ebnenasir. Automated synthesis of multitolerance. In *International Conference on Dependable Systems and Networks (DSN)*, pages 209–219, 2004.

[61] O. Kupferman and M. Y. Vardi. Safraless decision procedures. In *Proceedings of 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 531–542, 2005.

[62] Xinxin Liu and Scott A. Smolka. Simple linear-time algorithms for minimal fixed points (extended abstract). In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 53–66, 1998.

[63] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[64] O. Maler, D. Nickovic, and A. Pnueli. From MITL to timed automata. In *Formal Modeling and Analysis of Timed Systems (FORMATS)*, pages 274–289, 2006.

[65] O. Maler, D. Nickovic, and A. Pnueli. On synthesizing controllers from bounded-response properties. In *Computer Aided Verification (CAV)*, pages 95–107, 2007.

[66] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 229–242, 1995.

[67] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(1):68–93, 1984.

[68] F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A new self-stabilizing maximal matching algorithm. *Theoretical Computer Science*, 410(14):1336–1345, 2009.

[69] Mikhail Nesterenko and Sébastien Tixeuil. Ideal stabilisation. *International Journal of Grid and Utility Computing*, 4(4):219–230, 2013.

[70] F. Ooshita and S. Tixeuil. On the self-stabilization of mobile oblivious robots in uniform rings. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 49–63, 2012.

[71] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Principles of Programming Languages (POPL)*, pages 179–190, 1989.

[72] A. Pnueli, J. Xu, and L. Zuck. Liveness with (0,1,inf)-counter abstraction. In *Computer Aided Verification (CAV)*, pages 107–122. Springer, 2002.

[73] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[74] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.

[75] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415, 2006.

[76] G. Tel. Maximal matching stabilizes in quadratic time. *Information Processing Letters*, 49(6):271–272, 1994.

[77] W. Thomas. On the synthesis of strategies in infinite games. In *Theoretical Aspects of Computer Science (STACS)*, pages 1–13, 1995.

[78] N. Wallmeier, P. Hütten, and W. Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In *Implementation and Application of Automata (CIAA)*, pages 11–22, 2003.