AUTOMATIC VERIFICATION AND REVISION FOR MULTITOLERANT PROGRAMS

By

Jingshu Chen

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science - Doctor of Philosophy

2013

ABSTRACT

AUTOMATIC VERIFICATION AND REVISION FOR MULTITOLERANT PROGRAMS

By

Jingshu Chen

The notion of multitolerance is based on the observation that modern programs are often subject to multiple faults. And, the requirements in the presence of these faults vary based on the nature of the faults, their severity and the cost of providing fault-tolerance to them. Hence, assurance of multitolerant systems is necessary and challenging. This dissertation proposes to provide such assurance via automated verification and revision.

Regarding verification, we focus on verification of self-stabilization, which is the ability of the program to recover from arbitrary states. Most of literature on verification of fault-tolerance focuses on safety property; our work complements it by considering liveness properties. Hence, we envision verification of multitolerant programs by using existing approaches for verifying safety and using the results from this dissertation for verifying liveness. We propose a technique that is based on a bottleneck (fairness requirements) identified in existing work on verification of stabilization. Our approach uses the effectiveness of fairness along with symbolic model checking, and hence reduces the cost of verification substantially. We also propose a constraint-based approach that reduces the task of verifying self-stabilization into a well-studied problem of constraint solving, so that one can leverage existing highly optimized solutions (SAT/SMT solvers) to reduce the verification cost.

Regarding revision, we focus on revising existing programs to add multitolerance in an automatic way. Revising the program manually is expensive since it requires additional verification steps to guarantee correctness. Also, manual revision may violate existing re-

quirements. For these reasons, we propose an automatic approach to revise a given program to add multitolerance for the identified faults. We investigate the complexity of automatic revision for adding multitolerance. We also develop algorithms (and heuristics) for automatic revision for adding multitolerance to existing programs. We implement these algorithms in a model repair tool for automatically adding multitolerance. Additionally, we build a lightweight framework for automatically revising UML state diagram to add fault-tolerance. Specifically, this framework allows designers to revise an existing UML model to add fault-tolerance without a detailed knowledge of the formalism behind model repair algorithms.

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Dr. Sandeep Kulkarni, for the incredible guidance, support and patience he provided during my Ph.D. life. Sandeep introduced me to the research area of automatic program revision and verification. He was always there for support whenever I needed some fresh insights about how to improve a paper or a talk.

I am extremely grateful to Dr. Laura Dillon for the valuable guidance from her during years of my Ph.D. program. I also express my thanks to Dr. Abdol-Hossein Esfahanian and Dr. Rajesh Kulkarni, for their critical and insightful questions/comments during my defense.

Also, I would like to truly thank the Department of Computer Science and Engineering at Michigan State University for offering me financial support through teaching assistantships and fellowships. In particular, I express my thanks to Dr. Eric Torng for approving the department's financial support for my Ph.D. education.

Finally, I would like to use this opportunity to thank all the faculties and friends I met at Michigan State University.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Fault-tolerance

The past decade has witnessed the increasing deployment of software systems in our daily lives. Every call we receive, every email we send, every bank transaction we request, operating systems used for office and study, adaptive cruise control in vehicles, all rely on software systems. Ensuring the reliability of software systems is more critical than ever before.

Fault-tolerance is crucial to the reliability of software systems. Generally, there are two main requirements in providing fault-tolerance. The first requirement is that the program should preserve its safety properties in the presence of faults. The second is that the program should *recover* from faults so that its subsequent computation is correct. That is, the program should meets its safety and livenss properties after recovery. Intuitively, *safety* states that nothing bad ever happens in program computations, and *liveness* states that something good will eventually occur in every program computation. When both of these requirements are met in the presence of faults, we denote the corresponding program as *masking* fault-tolerant.

While masking fault-tolerance is ideal, due to feasibility and/or cost issues, one may choose to provide a weaker level of tolerance. One weaker level of fault-tolerance is *nonmasking*. In this level, the program provides recovery but may violate safety during recovery. Nonmasking fault-tolerance is desirable when the design of masking fault-tolerance is either expensive or impossible. For example, in [46], authors provide nonmasking fault-tolerance to memory safety bugs in Neutron, a version of the TinyOS operating system [147]. In this example, while one could technically design a masking fault-tolerant system, it is very expensive in terms of human effort. Other examples include algorithms for clock synchronization [151, 191], where clock values could be corrupted by faults, such as node failure, random re-starts and initial lack of synchronization. In these examples, it is impossible or expensive to guarantee the safety property (e.g., clock drift is always limited). Hence, nonmasking fault-tolerance is preferred so that the program will eventually recover to states where clocks remain synchronized. Other examples of nonmasking fault-tolerance include [47] [78] [215] [214].

Another weaker level of fault-tolerance is *failsafe*. In this level, the program always satisfies its safety properties in the presence of faults, but it may not satisfy its liveness properties. Failsafe fault-tolerance is applicable in situations where safety is much more important than liveness (e.g., safety-critical systems). Since liveness is not guaranteed, implementation costs of failsafe fault-tolerance are typically lower than those of masking fault-tolerance. Failsafe fault-tolerance is also utilized in systems at the component level, e.g., one may choose to ensure that in case of faults, a component guarantees its own safety constraints although it may not satisfy its liveness constraints. Upon noticing this, other components could ensure that safety and liveness are satisfied for the overall system. Examples of such approach

include [222], where the authors present a mechanism to prevent a single faulty node from monopolizing the communication bus in a distributed hard real-time system. In this example, failsafe fault-tolerance is imposed to enforce fail-silent behavior of the node. Other examples of failsafe systems include [166] [204] [106] [119].

1.2 Multitolerance

A program is often subject to multiple faults. Moreover, the level of expected tolerance to these faults is different. To illustrate this, consider a simple networking system, in which nodes communicate with each other by message passing. Two possible faults, message loss and node failure could affect the program. The program is designed to *mask* the occurrence of message loss, i.e., ensure that the program continues to satisfy its specification even if messages are lost. However, for more serious faults, e.g., node failure, the program may only provide *nonmasking* fault-tolerance, where the program eventually reorganizes itself to legitimate states while some other properties (e.g., safety properties, satisfaction of requests generated during recovery) may not be met during recovery.

Unfortunately, there exist situations where multiple faults occur *simultaneously*. By simultaneous, we mean that a fault from one class occurs before the system has recovered from a fault from another class. Considering the above example, clearly, if the program provides nonmasking fault-tolerance for node failure, it cannot guarantee masking fault-tolerance if message loss occurs during recovery from node failure. Hence, when these two faults occur simultaneously, it follows that the best one can do is to ensure that the program eventually recovers to states from where it satisfies its specification. In other words, the tolerance provided for the case where faults from both classes occur simultaneously is equal to the

'minimum' level of fault-tolerance provided to each class of faults. We denote such multitolerance as *strong* multitolerance. (See Chapter 5.8 for a precise definition.) In the above example, a strongly multitolerant program would ensure that nonmasking fault-tolerance would be provided if node failure and message loss occur simultaneously.

Instead of guaranteeing the minimum level of fault tolerance, another possible solution is to ensure that the program provides an appropriate level of fault-tolerance if faults from only one class occur. However, it may not provide any tolerance if faults from two classes occur simultaneously. This scenario is possible when recovery from node failure requires that message delivery is reliable and no messages are lost. We denote such multitolerance as *weak* multitolerance.

1.3 Contributions

The goal of this dissertation is to build a model-based framework that helps to provide multitolerance in an automatic way. To achieve this, our work addresses two important problems: (1) automatic program revision for adding multitolerance, and (2) efficient verification of multitolerance. Next, we will give a brief overview of the key contributions of this dissertation.

1.3.1 Automatic Revision for Adding Multitolerance

It is desirable to revise existing program designs for adding multitolerance when faults are newly identified. Manual revision is expensive because (1) manual revision requires another round of verification to ensure that the identified errors are fixed, (2) manual revision may not be successful and may introduce new human-induced errors to the existing program

design. It is a tedious and expensive task to repeat verification and manual revision if the verification step does not succeed. This motivates the need for automating the revision phase. We address the problem of automatic program revision for adding multitolerance. We characterize multitolerance and differentiate our proposed multitolerance from previous approaches. We investigate the complexity of automatic addition for multitolerance, and develop novel algorithms and heuristics for automatic addition of multitolerance to existing programs.

1.3.2 Verification of Stabilization

We also investigate the problem of verifying self-stabilization, which is the ability to recover from arbitrary state. Most of literature on verification of fault-tolerance focuses on guaranteeing that a program satisfies its safety properties, our work complements the existing literature by considering liveness properties. Thus, we envision verification of multitolerant programs by using existing approaches for verifying safety and using the results from this dissertation for verifying liveness. One key challenge in verification is how to avoid state space explosion. The problem of 'state explosion' is worsened in verification of stabilizing programs, because verifying stabilization requires considering all possible states.

1.3.3 Using Model Repair to Add Fault-tolerance in UML State Diagram

State diagrams in the Unified Modeling language (UML) are used to describe the states as well as the transitions between states of an object in the software system. Hence, UML state diagrams is widely used in modeling program behaviors for helping programmers to

better understand the intricacies of their codes. Since the language of state diagrams is not a computational one, using model repair to automatically revise the state diagram is a challenging task. Furthermore, the use of the model repair technique requires the knowledge of logic and automated reasoning, which makes it harder for designers to apply the model repair technique in some degree. Thus it is desirable if the details of model repair are hidden from designers so that they can enjoy the benefits of applying model revision and continue to work with a familiar language to specify the artifacts of the system.

With this motivation, we propose a lightweight repair framework for UML state diagrams, which takes a UML state diagram as input, utilizes the techniques of model repair to revise UML state diagram, and produces a fault-tolerant UML state diagram. Complicated details, such as the model repair process, and the transformation between UML state diagram and underlying computing models, are hidden from users in the framework. This feature reduces the learning curve for applying model repair in UML state diagram to add fault-tolerance. To demonstrate the feasibility of this framework, we applied the proposed approach in two scenarios, including the classic problem of Byzantine agreement and the adaptive cruise control systems in vehicles.

1.4 Organization

The rest of this dissertation is organized as follows. Chapter 2 is dedicated to formalizing the preliminary concepts which are used through this dissertation. Chapter 3 investigates the effectiveness of transition systems to model faults. Chapter 4 presents our approaches for automatic verification of multitolerant programs. Chapter 5 presents our work on automatic revision for adding multitolerance. Chapter 6 introduces our work on using the model repair

technique to revise UML state diagram for adding fault-tolerance. Finally, in Chapter 7, we provide concluding remarks and discuss future work.

Chapter 2

Preliminaries

This chapter is devoted to formalizing the fundamental concepts through this dissertation, namely, programs, specification, faults, and fault-tolerance. The notion of programs is adapted from Kulkarni and Arora [134]. The definition of specification is due to Alpern and Schneider [7]. The definition of faults and fault-tolerance are from Arora and Gouda [17], Kulkarni [133] and Bonakdarpour [35].

2.1 Models and Programs

This section formally defines models and programs. A model is described by an abstract program.

A program, $\mathcal{P}$, is described using a finite set of variables $V_{\mathcal{P}} = \{v_0, v_1, \ldots, v_n\}$, and a finite set of program actions $A_{\mathcal{P}} = \{ac_0, ac_1, \ldots, ac_m\}$, where $n \geq 0$ and $m \geq 0$. Each variable, $v_i \in V_{\mathcal{P}}$, is associated with a finite domain of values, D_i. Each action, $ac_i \in A_{\mathcal{P}}$, is defined as follows: $ac_i :: g_i \longrightarrow st_i$; where g_i is a Boolean formula involving program variables and st_i is a statement that updates a subset of program variables.

For such a program, we define the notion of state, state space and state predicate.

Definition 2.1.1. ***(State)*** *A state, s, of program $\mathcal{P}$ assigns each variable $v_i \in V_{\mathcal{P}}$ a value from its respective domain D_i.*

Definition 2.1.2. ***(State space)*** *The state space, $S_{\mathcal{P}}$, of a program $\mathcal{P}$ is the set of all possible states of $\mathcal{P}$.*

Definition 2.1.3. ***(State predicate)*** *A state predicate of $\mathcal{P}$ is a Boolean expression defined over the program variables $V_{\mathcal{P}}$. Thus, a state predicate C of $\mathcal{P}$ identifies the subset, $S_C \subseteq S_{\mathcal{P}}$, where C is* true *in a state s iff $s \in S_C$.*

Definition 2.1.4. ***(Enabled)*** *The action $a_i :: g_i \rightarrow st_i$, is enabled in a state s iff g_i is true in s.*

Observe that an action ac in a program corresponds to a set of transitions $(s_0,\ s_1)$, where s_0 is the initial state and s_1 is the next state that is obtained by executing the statement of the action that is enabled in s_0.

Definition 2.1.5. ***(Computation)*** *A* computation *of $\mathcal{P}$ is a finite or infinite state sequence: $\overline{\sigma} = \langle s_0, s_1,\ \ldots \rangle$ s.t. the following conditions are satisfied: (1) $\forall j : 0 < j < lengthof(\overline{\sigma}) : (s_{j-1}, s_j)$ is a transition of $\mathcal{P}$, (2) if $\overline{\sigma}$ is finite and terminates in s_f then there does not exist any state s such that (s_f, s) is a transition of $\mathcal{P}$.* □

A program that is described in terms of its variables $V_{\mathcal{P}}$ and actions $A_{\mathcal{P}}$ can alternatively be viewed in terms of its state space and its transitions. In particular, $V_{\mathcal{P}}$ and corresponding domain of each variable identifies the state space, $S_{\mathcal{P}}$. And, each action ac in $A_{\mathcal{P}}$ is a subset of $S_{\mathcal{P}} \times S_{\mathcal{P}}$. Since the subsequent definitions of specification, fault-tolerance and

multitolerance do not consider the structure of the program in terms of its variables and actions, we define program in an abstract fashion in terms of its state space and transitions.

Definition 2.1.6. ***(Program)*** *The program $\mathcal{P}$ is defined as the tuple $\langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$, where $S_{\mathcal{P}}$ is the state space, $\psi_{\mathcal{P}} \subseteq S_{\mathcal{P}} \times S_{\mathcal{P}}$.*

Remark 2.1.1. *In the above definition, $\psi_{\mathcal{P}}$ is the transitions of $\mathcal{P}$. In most cases, we need to refer to transitions of the program as state space is obvious from the context. Hence, we use $\mathcal{P}$ and its transitions $\psi_{\mathcal{P}}$ interchangeably.*

Remark 2.1.2. *In describing the program in subsequent chapters, for ease of understanding, we will describe it in terms of its variables and actions. The conversion from this notation to state space and transitions is straightforward.*

Remark 2.1.3. *In our work on verification of multitolerant programs, we need a more detailed structure of the program transitions. Specifically, we need to split transitions $\psi_{\mathcal{P}}$ into set of transitions that correspond to individual actions, ac_1, ac_1, ..., such that $ac_1 \cup ac_2 \cup \ldots = \psi_{\mathcal{P}}$.*

2.2 Specification

In this section, we formally present the notions of specification and the related concepts that are used in this work.

Definition 2.2.1. *(**Safety Specification**) The* safety specification *for program $\mathcal{P}$ is specified as a set of bad transitions [133], i.e., for program $\mathcal{P}$, its safety specification is a subset of $S_{\mathcal{P}} \times S_{\mathcal{P}}$.* □

We say a transition (s_0, s_1) violates the safety specification *sspec* iff (if and only if) $(s_0, s_1) \in$ *sspec* . A sequence $\overline{\sigma} = \langle s_0,\ s_1,\ \ldots\rangle$ satisfies *sspec* iff $\forall j : 0 < j < lengthof(\overline{\sigma}) : (s_{j-1}, s_j) \notin sspec$.

Definition 2.2.2. *(**Liveness Specification**) A liveness specification is specified in terms of a set of infinite sequences.* □

A sequence $\overline{\sigma} = \langle s_0,\ s_1,\ \ldots\rangle$ satisfies a liveness specification *lspec* iff some suffix of $\overline{\sigma}$ is in *lspec*.

A specification *spec* for program $\mathcal{P}$ is of the form $\langle sspec, lspec\rangle$, where *sspec* is a safety specification of $\mathcal{P}$, and *lspec* is a liveness specification of $\mathcal{P}$. A sequence satisfies the specification *spec* iff it satisfies the corresponding safety and liveness specification. Hence, for ease of understanding, we say that a specification is an intersection of a safety specification and a liveness specification.

We now define what it means for a program $\mathcal{P}$ to satisfy a specification.

Definition 2.2.3. ***(Satisfies)*** *Let $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}}\rangle$ be a program, S be a state predicate, and* spec *be a specification for $\mathcal{P}$. We write $\mathcal{P} \models_S$* spec *and say that $\mathcal{P}$* satisfies spec *from S iff (1) S is closed in $\psi_{\mathcal{P}}$, and (2) every computation of $\mathcal{P}$ that starts from a state in S satisfies* spec. □

Assumption 2.2.1. *For simplicity of subsequent definitions, if $\mathcal{P}$ satisfies spec from S, we assume that $\mathcal{P}$ includes at least one transition from every state in S. If $\mathcal{P}$ does not include a transition from state s then we add the transition (s, s) to $\mathcal{P}$. Note that this assumption is not restrictive in any way. It simplifies subsequent definitions, as one does not have to model terminating computations explicitly.*

Definition 2.2.4. ***(Invariant)*** *Let* $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ *be a program,* S *be a state predicate, and* spec *be a specification for* $\mathcal{P}$*. If* $\mathcal{P} \models_S$ spec *and* $S \neq \{\}$*, we say that* S *is an* invariant *of* $\mathcal{P}$ *for* spec*.* □

Whenever the specification is clear from the context, we shall omit it; thus, "S is an invariant of $\mathcal{P}$" abbreviates "S is an invariant of $\mathcal{P}$ for *spec*". Note that Definition 2.2.3 introduces the notion of satisfaction with respect to computations. In case of computation prefixes that are not necessarily maximal, we characterize them by determining whether they can be extended to an infinite computation that satisfies the specification.

Definition 2.2.5. ***(Maintains)*** *Program* $\mathcal{P}$ maintains spec *from* S *iff (1)* S *is closed in* $\psi_{\mathcal{P}}$*, and (2) for all computation prefixes* $\overline{\alpha}$ *of* $\mathcal{P}$ *starting in* S*, there exists a computation suffix* $\overline{\beta}$ *such that* $\overline{\alpha}\overline{\beta} \in$ spec*. We say that* $\mathcal{P}$ violates spec *from* S *iff* $\mathcal{P}$ *does not maintain* spec *from* S*.* □

We note that if $\mathcal{P}$ satisfies *spec* from S then $\mathcal{P}$ maintains *spec* from S as well, but the reverse direction does not hold. We, in particular, introduce the notion of *maintains* for computations that a (fault-intolerant) program cannot produce, but the computation can be extended to one that is in *spec* by adding *recovery* (see Section 2.3 for details).

2.3 Faults

Each class of fault f that a program is subject to is systematically represented by a set of transitions. Formally, a *fault class* for program $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ is a subset of $S_{\mathcal{P}} \times S_{\mathcal{P}}$. The effectiveness of this representation is presented in Chapter 3.

Definition 2.3.1. ***(Fault-span)*** *A state predicate T is an f-span (read as* fault-span *for f) of $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ from S iff the following conditions are satisfied: (1) $S \subseteq T$, and (2) T is closed in $\psi_{\mathcal{P}} \cup f$.* □

Observe that for all computations of $\mathcal{P}$ that start from states in S, T is a boundary in the state space of $\mathcal{P}$ up to which (but not beyond which) the states of $\mathcal{P}$ may be perturbed by the occurrence of the transitions in f. Subsequently, as we defined the computations of $\mathcal{P}$, one can define *computations of program $\mathcal{P}$ in the presence of faults f* by simply substituting $\psi_{\mathcal{P}}$ with $\psi_{\mathcal{P}} \cup f$ in Definition 2.1.5.

2.3.1 Fault-Tolerance

We now define what it means for a program to be failsafe/nonmasking/masking f-tolerant (read as fault-tolerant to fault class f).

Definition 2.3.2. ***(Masking f-tolerance)*** *A program $\mathcal{P}$ is masking f-tolerant from S for* spec*, iff the following conditions hold:*

1. $\mathcal{P} \models_S$ spec*;*
2. *There exists T such that:*
   (a) *T is an f-span of $\mathcal{P}$ from S;*
   (b) *$\langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \cup f \rangle$ maintains* spec *from T;*
   (c) *Every computation of $\langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ that starts from a state in T eventually reaches a state of S.* □

Thus, if program $\mathcal{P}$ is masking f-tolerant from S for *spec* then S is closed in $\psi_{\mathcal{P}}$ and every computation of $\mathcal{P}$ that starts from a state in S satisfies *spec* in the absence of faults.

Additionally, in the presence of faults, there is a fault-span predicate T ($S \subseteq T$) that is closed in $\psi_{\mathcal{P}} \cup f$.

Definition 2.3.3. ***(Failsafe f-tolerance)*** *A program $\mathcal{P}$ is failsafe f-tolerant from S for* spec*, iff conditions 1, 2a and 2b in Definition 2.3.2 hold.* □

Definition 2.3.4. ***(Nonmasking f-tolerance)*** *A program $\mathcal{P}$ is nonmasking f-tolerant from S for* spec*, iff conditions 1, 2a and 2c in Definition 2.3.2 hold.* □

Notation. Whenever the specification *spec* and the invariant S are clear from the context, we omit them; thus, "f-tolerant" abbreviates "f-tolerant from S for *spec*".

2.4 Model Revision for Adding Fault-tolerance

In this section, we define the model revision problem in the context of adding fault-tolerance.

Problem 2.4.1. ***The Model Revision Problem*** *for Adding Fault-tolerance*

Given a program $\mathcal{P}$, a safety specification ϕ, an invariant $\mathcal{I}$ of $\mathcal{P}$ from where $\mathcal{P}$ satisfies ϕ and a set of fault actions $\mathcal{F}$: Does there exist a $\mathcal{P}'$ with an invariant $\mathcal{I}'$ such that

- *(C1)* $\mathcal{I}' \subseteq \mathcal{I}$,
- *(C2)* $(s_0, s_1) \in \mathcal{P}' \wedge s_0 \in \mathcal{I}' \Rightarrow (s_0, s_1) \in \mathcal{P}$, *and*
- *(C3)* $\mathcal{P}'$ *is fault tolerant to* $\mathcal{F}$ *from* $\mathcal{I}'$ *for* ϕ. □

Since the goal of this problem is to add fault-tolerance, the revision for adding fault-tolerance is not permitted to add new behaviors in the absence of faults. To meet this requirement, we include two constraints $C1$ and $C2$. Specifically, constraint $C1$ states that

$\mathcal{I}'$ is a subset of $\mathcal{I}$. If $C1$ is not true then it implies that the fault-tolerant program could begin in a state from where the original fault-intolerant program violates its specification. Hence, we cannot conclude correctness of the program behavior (in the absence of faults) if it starts from a state in $\mathcal{I}' - \mathcal{I}$. Thus, constraint $C1$ is required. Likewise, if $\mathcal{P}'$ has new transitions in $\mathcal{I}'$ then it would imply that $\mathcal{P}'$ could have behaviors that are not in $\mathcal{P}$. In other words, the computation of $\mathcal{P}'$ may generate new ways to satisfy *spec* in the absence of faults. Hence, constraint $C2$ is required.

Based on the above problem statement, we can view transitions of $\mathcal{P}'$ in two parts: (1) transitions of $\mathcal{P}$ that are preserved and (2) new transitions that are added to provide recovery from faults. Furthermore, since a transition in a UML model may correspond to several transitions in the underlying computation model. For this reason, the first part can be subdivided into transitions that are preserved as is and transitions that are preserved in part, i.e., they are restricted to execute under certain conditions. Thus, the transitions of the fault-tolerant program can be partitioned into three types:

1. **O**riginal Transitions $\mathcal{T}_O$. $\mathcal{T}_O$ corresponds to transitions that are preserved as is in the fault-tolerant program.

2. **S**trong Transitions $\mathcal{T}_S$. $\mathcal{T}_S$ corresponds to transitions of the original program that are restricted to execute under certain constraints. In other words, these transitions are strengthened version of the original actions.

3. **R**ecovery Transitions $\mathcal{T}_R$. $\mathcal{T}_R$ does not have the counterpart actions in the input program. $\mathcal{T}_R$ is added by model revision to provide recovery in the presence of faults.

Chapter 3

Effectiveness of Transition Systems to Model Faults

In this chapter, we present an overview of our fault modeling approach, that is, using transition systems. In particular, we focus on whether/how faults from various categories can be modeled using transition systems. We also discuss the practicability of such modeling approach in two contexts: model checking and model repair, which are important techniques for automatic verification and revision. Besides, we evaluate the completeness of using transition systems to model faults introduced from recent literatures.

3.1 Motivation

This research has been motivated in part due to questions raised by readers and reviewers about fault modeling approach used in our (and other similar) work. In particular, the formal methods community typically takes this approach for granted (e.g., [57, 85, 90, 140, 240]). However, some researchers argue that this approach would have limited application with

respect to different classes of faults. Our goal in this chapter is to answer questions such as these and identify feasibility of modeling fault-tolerant systems using transition systems. In particular, we focus on three questions. The first question deals with the feasibility and limitation of using transition systems to model faults. Specifically, this includes the following concerns.

- The fault definition in Section 2.3 assumes that faults are a set of new transitions that are *added* to the given program. How can faults such as stuck-at faults be modeled as *added transitions* that perturb the given program, since stuck-at faults intuitively remove transitions from the original program. The same question is also extended to other types of faults such as node failure since these faults do not seem to add any new transitions to the original program.

- How can faults such as buffer overflow be modeled in terms of new transitions, as these faults are essentially (intentional or unintentional) bugs in the software.

- A permanent fault such as byzantine fault causes a process to behave arbitrarily. Moreover, this fault is permanent in that the malicious process can perturb the program *several* times. How can such faults be modeled using added transitions?

- How to model faults such as physical degradation, which are continuous in nature, and hence, may not suitable for being modeled as *discrete* transition systems.

The second question regarding fault modeling is: *Even if it is* possible *to model faults using transition systems, is it* appropriate *and* reasonable. Specifically, this question compares the complexity of verification of a fault-intolerant program with the complexity of verification of the corresponding fault-tolerant program (or the complexity of the corresponding repair

for adding fault-tolerance) that uses transition systems to model faults. The answer to this question will identify circumstances where modeling of faults in terms of transition systems may be feasible but expensive thereby making it difficult (or impossible) to utilize it.

The third question regarding fault modeling is: *How can we know if we have represented all types of faults that may affect a computer system?* This question is crucial since it characterizes the completeness of transitions systems to be used to model faults. While answering this question is beyond the scope of formal methods, we can consider this question in the context of a fault classification for practical systems. In particular, we can begin with a classification of faults from the perspective of practitioners and then evaluate whether faults from those models can be effectively represented using transition systems.

Based on these questions, in this work, we focus on the *feasibility* and *practicality* of modeling faults as transition systems. We begin with the third question. Specifically, we utilize the classification from seminal paper by Avizienis et al. [24]. This classification provides different causes of faults and their effects. Since this classification is based on practitioner's viewpoint, it provides the basis for answering the first two questions. Specifically, in this work, we focus on how/if each category of fault identified here can be modeled in terms of transitions. From the point of view of formal methods, it is necessary to model the effect of the fault as opposed to the cause of the fault. Hence, our work will focus on how effect of faults from each category can be modeled.

Additionally, we also evaluate the complexity introduced by modeling faults as transition systems. We consider the complexity in terms of two objectives: *model checking* and *model revision*. Specifically, model checking [110, 174] is one of the most successful strategies for providing assurance for model of hardware and software design. It focuses on deciding

whether a given model of system, say M satisfies the given property pr. Since model checking computes (directly or indirectly) all computations of M to determine whether pr is satisfied, it is especially useful in providing assurance about a system developed from that model. The related problem of model revision [36,37] focuses on scenarios where model checking produces a counterexample or where an existing model needs to be revised to add new properties (such as safety, liveness and timing constraints). Thus, the goal in model revision is to modify the given model M so that it satisfies the given property pr. Since the revised model is correct by construction, it can assist us in obtaining a correct model of system when model checking ends up finding a counterexample.

We view the fault modeling in *online* setting where faults occur *during* system execution. In *offline* setting, there is often no need to model faults explicitly. This is due to the fact that in the offline setting, faults have already occurred at the beginning and the goal of the system is to provide acceptable service even in the presence of faults. Hence, offline setting often requires us to model a degraded version of the original system itself rather than modeling faults. In online setting, however, system may initially execute without faults. Then, one or more faults could occur and change subsequent system behavior. Hence, in online setting, it is often necessary to model faults explicitly.

3.2 A Taxonomy of Faults

In this section, we recall the terminology of fault classification from [24]. The classification is based on eight basic viewpoints about faults. Table 3.1, 3.2 and 3.3 illustrate this classification as well as results in this chapter about the ability and effect of modeling faults from the respective category.

Fault Characteristics		Categories							
		1	2	3	4	5	6	7	8
Phase	Development	√	√	√	√	√	√	√	√
	Operational								
System Boundaries	Internal	√	√	√	√	√	√	√	√
	External								
Phenomenological Cause	Human-made	√	√	√	√	√	√	√	√
	Natural								
Dimension	Software	√	√	√	√	√			
	Hardware						√	√	√
Objective	Non-Malicious	√	√	√	√			√	√
	Malicious					√	√		
Intent	Non-Deliberate	√	√					√	√
	Deliberate			√	√	√	√		
Capability	Accidently	√		√		-	-	√	
	Incompetence		√		√	-	-		√
Persistence	Persistent	√	√	√	√	√	√	√	√
	Transient								

Fault Characteristics		Categories							
		9	10	11	12	13	14	15	16
Phase	Development	√	√	√					
	Operational				√	√	√	√	√
System Boundaries	Internal	√	√	√	√	√			
	External						√	√	√
Phenomenological Cause	Human-made	√	√	√	√	√	√	√	
	Natural								√
Dimension	Software								
	Hardware	√	√	√	√	√	√	√	√
Objective	Non-Malicious	√	√	√	√	√	√	√	√
	Malicious								
Intent	Non-Deliberate	√	√			√	√	√	√
	Deliberate			√	√				
Capability	Accidently	√		√		√	√	√	√
	Incompetence		√		√				
Persistence	Persistent	√	√	√	√		√		
	Transient					√		√	√

Table 3.1: Fault classification (1).

Fault Characteristics		Categories							
		17	18	19	20	21	22	23	24
Phase	Development								
	Operational	√	√	√	√	√	√	√	√
System Boundaries	Internal								
	External	√	√	√	√	√	√	√	√
Phenomenological Cause	Human-made	√	√	√	√	√	√	√	√
	Natural								
Dimension	Software								√
	Hardware	√	√	√	√	√	√	√	
Objective	Non-Malicious	√	√	√	√	√			
	Malicious						√	√	√
Intent	Non-Deliberate	√							
	Deliberate		√	√	√	√	√	√	√
Capability	Accidently			√			-	-	-
	Incompetence	√	√		√	√	-	-	-
Persistence	Persistent	√			√		√		√
	Transient		√	√		√		√	

Fault Characteristics		Categories							
		25	26	27	28	29	30	31	-
Phase	Development								-
	Operational	√	√	√	√	√	√	√	-
System Boundaries	Internal								-
	External	√	√	√	√	√	√	√	-
Phenomenological Cause	Human-made	√	√	√	√	√	√	√	-
	Natural								-
Dimension	Hardware								-
	Software	√	√	√	√	√	√	√	-
Objective	Non-Malicious			√	√	√	√	√	-
	Malicious	√	√						-
Intent	Non-Deliberate			√	√	√			-
	Deliberate	√	√				√	√	-
Capability	Accidently	-	-	√			√		-
	Incompetence	-	-		√	√		√	-
Persistence	Persistent	√			√			√	-
	Transient		√	√		√	√		-

Table 3.2: Fault classification (2).

Section	Categories	Effect in Model Verification and Revision	Feasibility Practicability
3.3.5	1-11	Fault prevention is more suitable. No need to model faults Explicitly.	Not feasible
3.3.2	12-13	Significant increase in reachable states. Need for modeling continuous behavior.	Feasible Not Practical
3.3.4	14,17, 20, 27, 30	Small increase in reachable states. Potential use for modeling blind writes.	Feasible Practical
3.3.1	5,6, 22-25	Small increase in reachable states. Need for modeling read/ write restriction.	Feasible Practical
3.3.3	15-21, 26-31	Small increase in reachable states.	Feasible Practical

Table 3.3: Fault classification (3).

Next, we briefly describe the viewpoints considered in this classification.

One viewpoint is based on how the fault occurs. In this viewpoint, there are two possibilities: **development faults** and **operational faults**. The former corresponds to the case where fault occurs due to mistakes during development. This category includes faults such as buffer overflows, incorrect results of certain floating point division caused by Pentium FDIV bug etc. The latter corresponds to the case where fault occurs while the system is deployed.

The second viewpoint is based on whether the fault occurs inside system boundary (**internal faults**) or whether it occurs outside system boundary, i.e., in the environment (**external faults**). The former corresponds to faults such as physical deterioration of brakes in the vehicle and logic bomb. And, the latter corresponds to faults such as bit flip in memory caused by cosmic ray and wrong parameter configuration.

The third viewpoint is based on the cause of the fault. In this viewpoint, a fault can be either **natural fault** or **human made fault**. The former corresponds to random events that may occur naturally. Examples of such faults include internet and telecoms connectivity disrupted by Taiwan earthquake. The latter corresponds to (intentional or otherwise) mistakes caused by humans. Examples of such faults include development failure of the AAS system [181].

The fourth viewpoint is based on how the fault affects the system. In this viewpoint, a fault can be either **hardware fault** or a **software fault**. Examples of former include loss of network switch and deflated car tie whereas examples of latter include Y2038 problem of software system and Trojan horses.

The fifth viewpoint is based on the objective of the fault. In this viewpoint, the fault can be either a **malicious fault** or a **non-malicious fault**. In the former case, the goal of the (human responsible for the) fault is to intentionally disrupt the system execution. Examples of such faults include attacker and worm. In the latter case, there is no malicious objective. Examples of such faults include physical deterioration and heating/cooling caused by natural environments.

The sixth viewpoint is based on in the intent of the fault. In this viewpoint, the fault can be either a **deliberate fault** or a **non-deliberate fault**. The former case is due to bad decisions. Examples of such faults include wrong configuration that can affect security, networking, storage, middleware, etc [96]. The latter one is caused by mistakes. Examples of such faults include software flaws and physical production defects.

The seventh viewpoint is based on whether the fault is caused **accidently** or due to **incompetence**. And, finally, the eighth viewpoint is based on whether the fault is a **transient**

fault that occurs occasionally and does not persist for a long duration. Or a **persistent fault** where the fault persists for a long time (possibly forever).

3.3 Using Transition Systems to Model Faults

In this section, we discuss modeling of faults from different categories in Table 3.1 - 3.3. Based on the characteristics of the faults, we partition our discussion among Sections 3.3.1-3.3.5. For each section, we identify the abstract model of fault from one category first. Then we introduce how to mapping to the abstract model with concrete example. We also discuss variations of the same fault that are considered in the literature and evaluate its effect on modeling that fault. Subsequently, we identify related fault categories, i.e., fault categories where the modeling would be similar. Finally, we identify the effect of such modeling in verification and revisions.

3.3.1 Operational, External, Human-made, Persistent, Malicious and Software Faults

In this section, we focus on faults that are operational, external, human made, persistent, malicious and software. (This corresponds to category 24 in Table 3.2.) Thus, these faults occur during the system execution. Generally, they are caused by malicious users or attackers. And they persist permanently (or long enough time). It is expected that the system will continue to function even when the faults are present. While some attempts may be made to prevent such faults from occurring, it is possible that such preventions would not be fully successful. Hence, for assurance in the context of these faults, it is necessary that one considers the system execution where the faults (exhibited in terms of compromised hosts,

malicious users or attacks) continue to occur potentially frequently. Often, in such systems, assumptions are made about the number of faults that may exist at a given time. These assumptions ensure that sufficiently many 'good resources' are available to solve the problem at hand.

3.3.1.1 Abstract Model

To capture the impact that this type of fault has on the underlying system state, there are three types of abstract actions in the model for fault from this category.

- Access Actions. These actions allow user to get knowledge about the current system state. An example of access action is that an user get the data from remote server and make a local copy. We use variable v to denote user's copy of system state and variable s to denote the value of the current system state. Hence this type of actions can be modeled as follows:

  ACCESS action:

  $v := s$;

- Update Actions. These actions create a change to the system state. Examples of update actions include writing data to file in the server and updating value of a flag which is used to denote whether a file is changed. We use variable v to denote the system state and x to denote the value which is used to update system state by *UpdateActions*. Hence this type of actions can be modeled as follows:

  Update action:

$v := x$;

- Fault Actions. These actions, (which may be conducted by malicious users or attackers), perturb the system to a random state. An example of a fault action is that a malicious user changes the value of accessible data randomly. We introduce variable v to denote the state which is corrupted by fault actions. We also introduce an extra variable m to model whether malicious user exists in the environment. The fault actions can be modeled as follows:

  Fault action 1:

  $true \longrightarrow m := true;$

  Fault action 2:

  $m \quad \longrightarrow v := random();$

3.3.1.2 Mapping to Abstract Model - From a Concrete Example

Now, we discuss how to mapping raw actions in the real example to the abstract actions in the model described above.

A typical example of the faults from this category occurs in the context of distributed system where the system is organized as a network of nodes and some of these nodes may not work as expected and behave arbitrarily because of some reasons, such as attackers. One such example is Farsite [34]. Farsite is a serverless distributed file system that provides centralized file-system service. Farsite aims at providing secure, scalable and strongly consistent file storage service. In Farsite, the concept of servers in the system is virtual and the system

actually runs on a network of untrusted PCs some of which may be controlled by a malicious user. Hence, the whole infrastructure is highly susceptible to a fault that the virtual server behaves arbitrarily. This fact brings the challenge of guaranteeing the delivery of correct service in the presence of such faults.

Since our goal is to illustrate the modeling of such fault and not the details of Farsite, without loss of generality, we describe how one can model operations for a single file, say fl. In particular, labeling *UpdateAction* and *AccessAction* is straightforward. In this example, for these Read operations that allows users to get the data from servers, we mapping them as *AccessAction* whereas we mapping Write operation that allows users to write data to servers to *UpdateAction*. In each operation, the client identifies the list of server nodes that contain (or are likely to contain) a copy of the file. We use index j to quantify over the list of such servers. Furthermore, we use the term $data.j$ to denote data maintained by the server j for file fl and the term $copy.j$ to denote local copy of file fl.

In read operation, the client obtains a copy of the file from all (or a subset of) servers. In write operation, the data is written to the respective servers. In order to capture system semantics of fault tolerance mechanism designed in this example, we make extend of *AccessAction* in the basic abstract model. There are several copies of stored files. Since some of the copies may be incorrect due to malicious users, the client performs an agreement algorithm (e.g., majority computation) first, then return the agreed data to the user who requests file fl.

Thus, the operations in Farsite can be modeled as shown next.

Acess action:

$\forall j :: copy.j := data.j$;

$getResouce := agreement(copy.j)$;

return getResource;

Update action to write v to file:

$\forall j :: data.j := v$;

Obviously our above model in transition system can capture system semantic of this example.

Observe that the above model only describes the program actions. Next, we show that the actions of malicious users can also be modeled in terms of $faultaction$ in the abstract model. In particular, to denote whether process j is malicious, we introduce a variable $m.j$ that denotes whether j is malicious. The fault actions can be modeled as follows:

Fault action 1:

$true \longrightarrow m.j := true$;

Fault action 2:

$m.j \longrightarrow data.j := random()$;

Thus, Fault action 1 denotes that machine j is compromised. Fault action 2 denotes that the compromised machine can change the local data arbitrarily. Observe that along with the write action that assumes that data is written to all replicas, the fault action allows the malicious user to change the data stored in a compromised machine.

Next, we discuss the completeness of this approach. In particular, we argue that this approach is generic for faults from this category. Specifically, the faults are operational in nature, i.e., at the time system is created (and deployed), there are no faults perturbing the system. (Note that this is not true for development faults.) Thus, faults become operational

at some point after (possibly, before the system executes even one of its actions) the system is deployed. Any modeling must include an action by which the malicious components/processes appear in the system. In other words, there must be one action (similar to Fault action 1) where some components/process becomes faulty. Moreover, since the faults are permanent, it is required that it must be possible for the fault to persist forever. In other words, the actions modeling faults should be such that they can execute forever (such as that indicated by the guard of Fault action 2).

Finally, because the fault is human made and malicious, the faulty component/process can change the data that it controls arbitrarily (as in the statement of Fault action 2). Thus, any fault that is operational, human made, persistent and malicious, can be modeled using an approach similar to the one presented earlier.

3.3.1.3 Modeling Variations

There are several possible variations that one may consider in this category of faults, as follows.

- A malicious user may intend to remain hidden. In this case, Fault action 2 will have to be changed so that instead of changing the data arbitrarily, it will change it only in a way that allows prevents it from being discovered.

- The system may have a mechanism to clean up affected nodes (e.g., through antivirus programs etc.) If such a mechanism exists then it can be modeled by the dual of Fault action 1 where $m.j$ is changed from $true$ to $false$.

- Also, often assumptions are made about the number of malicious users that can exist in the system at a given time. For example, a standard assumption is that the number

of malicious replicas is less than a third of the total replicas. If such an assumption is desired, it can be achieved by changing Fault action 1 so that a node can become malicious only if the total number of malicious nodes will not exceed the bound. Note that this will require one to read the value m variable of all nodes. However, this is acceptable since we are simply recording the assumption.

3.3.1.4 Modeling Faults from Related Categories

Approach similar to the one in this section also applies for faults from other categories. For example, the above modeling can be applied when the fault is caused by hardware. Also, if one were to model the fault which persists for some duration and then disappears, we only need to slightly modify the aboved modeling (the case where fault is permanent (long-lasting) in nature), by adding the modeling of another fault action (similar to Fault action 1) where $m.j$ changes from $true$ to $false$. Based on this discussion, we can model faults from category 22-25 (cf. Table 3.2) using the approach in this section. A special case when the fault is transient and occurs only once, we can simplify the modeling of faults by the approach in Section 3.3.3.

Additionally, a similar approach could also be used for modeling malicious and deliberate development faults (category 5 and 6 in Table 3.1). Examples of faults from these categories include logic bomb. For faults from these categories, the modeling in this section may be used if there are several designer teams producing different parts of the system (e.g., with N-version programming) and sufficient redundancy exists to deal with such faults. However, for the case where one design team is creating the given system or where sufficient redundancy is unavailable, the only suitable approach is to use fault prevention where the goal is to ensure that the fault cannot happen. And, when fault prevention is used, there is no need to model

faults explicitly, as proving that fault prevention work requires one to only consider system behavior in the absence of faults.

3.3.1.5 Effect during Verification of Fault-tolerance

If one were to verify a program that models malicious users in the above fashion, we can observe that the introduction of the variable $m.j$ increases the state space of the program. Additionally, to ensure that the variable $m.j$ is not used improperly, we need to syntactically check the program; specifically, we need to ensure that actions at machine j do not reference variable $m.k$, where $j \neq k$.

3.3.1.6 Effect during Revision for Adding Fault-tolerance

One important characteristic of above formulation is that variable $m.j$ is readable only to node j. Other nodes cannot read it. If model revision is used for a fault from this category, it is necessary that this restriction is continued to be satisfied in the revised model. Additionally, variable $m.j$ cannot be changed by any node; it may be changed only by a fault action. These restrictions have been shown to increase complexity from P to NP-complete in some instances [134].

3.3.2 Operational, Internal, Natural, Hardware, Non-malicious, Non-deliberate, Accidental and Persistent Faults

In this section, we focus on faults that are operational, internal, natural, hardware, non-malicious, non-deliberate, accidental and persistent. (This corresponds to category 12 (cf. Table 3.1).)

3.3.2.1 Abstract Model

Obviously, the fault from this category occurs during the system execution. Generally, they occur due to hardware degradation over time. And, it is expected that they last for a significant duration of time. Similar to the faults from Section 3.3.1, it is expected that the system will continue to function even when the faults are present. Periodic maintenance would be used to replace the hardware as necessary to correct this fault.

Taking these impacts that this category of fault has on the underlying system state into account, we introduce g to denote the hardware state and then the corresponding abstract action in our model is as follows, where ϵ is a small real number.

Fault action 1:

$g \geq \epsilon \longrightarrow g := g - \epsilon;$

3.3.2.2 Mapping to Abstract Model - From A Concrete Example

A typical example from this category occurs in the context of a braking control system where the brake may wear out due to physical deterioration thus system failures may occur with accidents and severe damage and human injuries as consequences.

The basic modeling of braking system includes the speed of the vehicle, the status of brakes (e.g., applied or not) and their reliability.

One such fault in this system can be caused by the case where wear and tear on brakes (or other factors) reduces their effectiveness. Modeling such fault into abstract action is straightforward. If we use g to denote the specified level of performance and integrity of the brake, we can apply abstract action in the basic model directly. To capture system semantics

of this example, we assume value of g is in the range $[0, 1]$ where $g = 1$ corresponds the ideal brake and $g = 0$ corresponds to a nonfunctioning brake in this case.

Besides, in order to model program actions, we use variable s to denote the speed of the vehicle. And, we use a Boolean variable b to denote whether the brake is pressed. (Similar to g, b could also be modeled as a continuous variable. However, this extension is straightforward and, hence omitted.) Thus, when the brake is pressed, the vehicle reduces the speed. We model this as a reduction in speed by a fixed value (denoted as C in the below program action) that depends on the reliability of brakes. Hence, one possible way to define such program action in this context is as follows:

Brake Action:

$b \ == \ true \ \longrightarrow \ s \ := \ MAX(s - g * C, 0);$

Next, we discuss the completeness of this approach. In particular, we argue that this approach is generic for faults from this category. Since the faults are operational in nature, faults occur at some point after the system is deployed and thus any modeling must include some action where some component/process becomes faulty. Also considering the faults are internal, natural, non-malicious and non-deliberate, the fault action should not be triggered by outsider. In other words, the guard of the fault action should be some status of fault component or process itself, like Fault action 1. Moreover, since the faults are permanent, it is required that it must be possible for the fault to persist forever (until the faulty component loses its capability totally).

Thus, any fault that is operational, internal, natural, hardware, nonmalicious, non-deliberate, accidental and persistent Faults, can be modeled using an approach similar to

the one presented earlier.

3.3.2.3 Modeling Variations

The approach proposed in this section is discrete in nature. It models that when the brake is pressed, speed reduces by a certain amount. A more accurate model would be to utilize timing based information for modeling both programs as well as faults. Examples of such approach include the timed automata [9] that combines transition systems with time. In particular, with such an approach, we could model speed reducing at a particular rate depending upon the reliability of brakes and the pressure applied on brakes.

3.3.2.4 Modeling Faults from Related Categories

Approach similar to the one in this section also applies for faults from other categories. The above modeling is for the case where fault is permanent in nature. If the faults persist for some duration and then disappears then we need to model another fault action (similar to the above fault action $g \geq \epsilon \longrightarrow g := g - \epsilon$) where g can be increased ϵ and reach at most 1 as range required. Based on this discussion, we can model faults from category 12-13 (cf. Table 3.1) using the approach in this section.

However, similar to Section 3.3.1, for the case where physical elements play a sole role in the structure of the whole system, when that elements totally loss the functions, e.g. a deflated tire of car, there is no need to model the fault explicitly. And, the only suitable approach is to use fault prevention to ensure such faults cannot happen.

3.3.2.5 Effect during Verification and Revision of Fault-tolerance

It is expected that the fault from this category will generally require one to consider hybrid models where both discrete and continuous variables are considered. The effect/effects of such hybrid model in verification is/are considered in [8]. Regarding revision, revising timed automata is considered in [35]. However, the cost of such revision is often higher (exponential in the constants involved in specifying timing constraints).

3.3.3 Operational, External, Natural, Hardware, Non-malicious, Non-deliberate, Accidental and Transient Faults

In this section, we focus on faults that are operational, external, natural, hardware, non-malicious, non-deliberate, accidental and transient. (This corresponds to category 15 in Table 3.1.)

3.3.3.1 Abstract Model

Obviously, the fault from this category occurs during the system execution. Generally, they occur due to unexpected transient issues that are unlikely to happen again. They are not malicious in nature but may cause system to behave in an incorrect manner. One approach for dealing with such faults is self-stabilization [59] where the system is guaranteed to recover from an arbitrary state to a legitimate state.

Modeling of the fault from this category in terms of transition systems is straightforward since the fault occurs once (or rarely) and the impact is to change the system state into random value. Specifically, we introduce x to denote system state then the fault can be modeled by the following action:

$true \longrightarrow x := random();$

3.3.3.2 Mapping to Abstract Model- From A Concrete Example

A typical example from this category includes faults that cause memory to change to an arbitrary state. A possible reason why such errors occur is cosmic rays, for example. Typically, it is assumed that such faults are not detectable and, hence, the system continues to operate in spite of them. Another related example in this class includes the use of uninitialized variables. Such a fault will result in the program starting in an arbitrary state. Moreover, other types of faults such as crash and message loss can exhibit a behavior that is similar to a fault from this category. Specifically, in [120], authors have shown that in the presence of these faults, the program may be (essentially) perturbed to an arbitrary state.

Mapping the raw actions in the real example to the abstract action in the model is straightforward. First, we map the variable (denoted as v) changed by the raw actions into x in the abstract action first. Second, we map the actual way to change the value of v to $random()$ in the abstract action. By these two steps, we can apply the basic abstract model to model fault actions in the real example directly.

Next, we discuss the completeness of this approach. In particular, we argue that this approach is generic for faults from this category. Consider the faults are operational in nature, there must be some fault action (similar to Fault action 1) occuring after the system is deployed. These faults may affect the entire system or a part of it. Since the faults are transient, this implies that their effect is temporary and the system continues to execute its actions after faults occur. Moreover, since the fault is accidental and not deliberate, the effect of fault on the affected part (like var x in Fault action 1) is random. Thus, any

fault that is operational, human made, persistent and malicious, can be modeled using an approach similar to the one presented earlier.

3.3.3.3 Modeling Variations

There are several possible variations that one may consider in this category. For example, the most common assumption used in the literature states that the fault will change the variable to only a value that is legitimate in some configuration. In other words, the fault will assign the variable a value that is from its domain; intuitively, this assumption is based on the fact that if the value assigned to the variable is outside the domain then it would be detected before the variable value is used. Also, often assumptions are made about inability of the fault to change the code itself, i.e., the fault only changes data. If faults are allowed to disrupt code then this can be modeled using the approach in Section 3.3.1. Alternatively, it could also be modeled using the approach in Section 3.3.4 if it is expected that the fault will render the corresponding component in a state from where it cannot continue its execution.

3.3.3.4 Modeling Faults from Related Categories

The approach similar to the one in this section also applies for faults from other categories. For example, the above modeling can also be applied when the fault is caused by human being's non-malicious action. Also, the modeling can be applied where fault is caused by software. Hence, we can model faults from category 15-16, 18-19, 21, 26, 28-29, 31 (cf. Table 3.2) using the approach in this section.

3.3.3.5 Effect during Verification and Revision of Fault-tolerance

If one were to verify a system that models transient faults in the above fashion, then one has to consider system execution from arbitrary (respectively, large number of) states. In particular, if the fault can corrupt all program variables then this corresponds to considering execution from arbitrary states. The topic of such verification is considered in the context of self-stabilization; in [225], authors have shown the feasibility of verifying self-stabilization using model checker SMV [174].

There are no new difficulties introduced when applying model revision to address faults from this category. This is due to the fact that no auxiliary variables are needed to model this fault. An example where such fault modeling is used in model revision includes [1].

3.3.4 Operational, External, Hardware, Non-malicious and Persistent Faults

In this section, we focus on the class of faults that are operational, external, hardware, non-malicious and persistent. (They correspond to category 14, 17 and 20 (cf. Table 3.1 and 3.2).) These categories differ in terms of whether the fault is human-made or natural as well as whether the fault is deliberate or non-deliberate. However, the exact cause is not important in modeling effect of such faults.

3.3.4.1 Abstract Model

Unlike the faults from Section 3.3.1, where faults are malicious in nature, the faults in category 14, 17 and 20 are non-malicious. It is expected that these faults are persistent in nature. Examples of such faults include failure of nodes, failure of channels etc.

Modeling of such fault in terms of transition is similar to that in Section 3.3.1. Specifically, since the faulty component is permanently 'killed' we can model it with an auxiliary variable C that denotes whether the given component is correct or whether it has failed. Also, let *data* denote state of a node (including its buffered messages). Now, the fault action can be modeled as follows:

Fault action 1:

$C = true \longrightarrow C := false, data := empty;$

Additionally, all program actions would have to be modified so that they only execute when the switch is functioning correctly. In other words, all actions by which one node communicates with another would have to be modified so that it can occur only if the respective nodes have not failed.

3.3.4.2 Mapping to Abstract Model- From A Concrete Example.

A typical example of the faults from this category occurs in context of networking systems where a fault may cause one or more nodes to fail in a manner where it completely stops working.

One example of such a system is from [206] where authors have focused on developing a failstop processor: A failstop processor works correctly before a fault occurs and it performs no actions when it fails. Moreover, data maintained at the failed processor is lost.

Another example is SafetyNet [217]. SafetyNet is a lightweight global checkpoint/recovery scheme. It aims at providing stable and reliable system services even in the situation of either dropped coherence messages or the loss of an interconnection network switch (and its buffered

messages).

Mapping raw actions in the real example to abstract action in the model described above is very straightforward. The steps is similar with Section 3.3.3(, and hence omitted).

Next, we discuss the completeness of this approach. In particular, we argue that this approach is generic for faults from this category. Specifically, considering the faults are operational, it is required to include in the modeling such an action to denote faulty components/processes appear in the system. Also, since faults of this category are persistent, it is required that it must be possible for the fault to persist forever (such as that indicated by Fault action 1). Since the fault is external and non-malicious, the occurrence of fault is due to some change of environment condition and hence there must be some variable to denote this change in the modeling (like var c in the Fault action 1). Thus, any fault that is operational, external, persistent, hardware and non-malicious, can be modeled using an approach similar to the one presented earlier.

3.3.4.3 Modeling Variations

There are several possible variations that one may consider in this category of faults. For example, a 2D torus topology is considered in [217] to prevent a single point-of-failure by splitting each switch into two half-switches. Execution may resume after reconfiguration to route around the lost switch [118] although at reduced bandwidth. In this case, we can use two variables to denote the link status and buffered data of each half-switch independently. The link status of the whole switch can be modeled as disjunction of the link status of each half element. Also often redundancy is used in the networking system. For example, the configuration of a specific point-to-point path may consist of several available links. If such an assumption is desired, one can model the fault action for each available switch first.

Then, the link status of the whole configuration can be modeled as disjunction value of these available switches.

The fault modeling from this section assumes that the fault is permanent, i.e., the failed node remains failed forever. A variation of this model is one where the faulty node is repaired and integrated in the system. In this case, a dual of the fault action where C is set to true must be added. Depending upon how such an action restores the node, the data associated with the node would change. For example, if the restore is equivalent to reboot where the node starts from some fixed state, data will be changed accordingly.

3.3.4.4 Modeling Faults from Related Categories

Approach similar to the one in this section also applies for faults from other categories. For example, the above modeling can be used whether faults affect hardware or software. Hence, we can model faults from category 27 and 30 (cf. Table 3.2) using the approach in this section.

3.3.4.5 Effect during Verification and Revision of Fault-tolerance

The effect on verification and revision is similar to that in Section 3.3.1. There are some possible changes depending upon the assumptions about faults. In particular, variable C introduced in fault actions may or may not be readable by other processes depending upon whether one assumes that a fault is detectable or not. Furthermore, approaches such as failure detectors [220] could be used to model more fine tuned assumptions about delectability of the fault.

3.3.5 Development Faults

In this section, we focus on faults that occur during development. This corresponds to category 1-11 in the Table 3.1. Typical examples of such faults cause software flaws, logic bombs and hardware errata, etc. If the developer(s) or operator(s) did not realize at the time that the consequence of their decision was a fault (or conceal faulty actions like logic bombs with the malicious purpose), and furthermore, design or decision is accepted for use, these faults can be treated as intrinsic nature of system and the occurrence of the faults is unavoidable.

A typical example from this category includes Pentium FDIV bug in the Intel P5 Pentium floating point unit(FPU) that was caused by few missing entries in the lookup table used by the divide operation. Another example is that of buffer overflows where one copies a longer string into a shorter string thereby affecting other parts of memory.

We argue that for such faults, formal methods for modeling faults explicitly are either undesirable or impossible. For such faults, a more practical approach is fault prevention where the goal is to ensure that the fault does not occur. For example, a thorough analysis of code could be useful to ensure that logic bombs do not occur. Likewise, analysis of Pentium FPU with theorem provers [198] has been successful in identifying the missing table entries in Pentium. Likewise, approaches such as [226] can be used to prevent buffer overflows.

In other words, for faults from this category, one needs to consider *fault-free execution* to show that the fault does not occur. Since this requires consideration of only *fault-free* execution, it does not require one to model faults explicitly.

That said, in certain instances, one may consider these faults explicitly and tolerate them, as preventing them may be impossible. In such cases, one needs to consider the *effect* of

the development faults. We expect most such development faults to exhibit themselves as malicious faults (cf. Section 3.3.1) or as failstop (cf. Section 3.3.4) faults. In particular, we have discussed modeling of faults from category 5 and 6 in Section 3.3.1.

3.4 Practicability during Verification and Revision

In Section 3.3, we considered the *feasibility* of modeling faults in terms of transition systems. In this section, we utilize those results in identifying the *practicability* of such modeling. Specifically, our goal is to evaluate the *cost* of such modeling in two contexts: *model-checking* and *model revision*. In case of model checking, we want to compare the cost of verifying a *fault-intolerant* program with that of verifying *fault-tolerant* program. And, in case of model revision, we want to compare the cost of *verifying fault-intolerant* program with that of *revising it to add fault-tolerance*. Both these tasks are feasible only if we can model the faults during the verification and/or revision process.

3.4.1 Cost of Modeling Faults during Model Checking

Model checking focuses on deciding whether a given model of system, say M satisfies the given property pr. While the cost of model checking depends upon several factors, one important factor is the state space of the resulting model. Since modeling of faults in terms of transitions has the potential to increase the state space of the program, we evaluate the cost in terms of the increased state space. Specifically, we consider the increased cost of modeling faults from Sections 3.3.1-3.3.4.

Observe that for modeling persistent and malicious faults, in Section 3.3.1, we needed to add a variable $m.j$ for every user. Essentially, this would double the state space of that user.

Faults From	Increased Cost of Modeling Faults during Model Checking (in terms of the increased state space)
Section 3.3.1	Increased by a factor of 2^n.
Section 4.4.2	Increased by a constant factor if discrete degradation is considered. Potentially undecidable if continuous degradation is considered.
Section 3.3.3	Unchanged in statespace.
Section 3.3.4	Increased by a factor of 2^n.

Table 3.4: Cost of modeling faults during model checking.

Moreover, if there are n users in the system, then the total state space will be 2^n times more than that of the fault-intolerant system.

For faults from Section 3.3.2, the cost depends upon whether the physical degradation can be modeled using discrete values or whether continuous modeling is required. If the physical degradation is modeled using discrete values as in Section 3.3.2, the total state space will increase by a constant factor when compared with that of the fault-intolerant system. If the physical is modeled using continuous values, especially modeling in hybrid automata [8], the total reachability problem is potentially undecidable.

If one were to verify a system that models transient fault as in Section 3.3.3, then one has to consider all possible states that could be subtantially larger. However, the total state space will be unchanged compared with that of fault-intolerant system.

Since the modeling approach of faults from Section 3.3.4 is similar to that in Section 3.3.1, the increased state space on verification of modeling faults from Section 3.3.4 is similar to that of Section 3.3.1.

Based on the above discussion, we summarize the increased cost of modeling faults from Sections 3.3.1-3.3.4 in Table 3.4.

According to the results in Table 3.4, we argue that the increased cost due to modeling faults in 18 categories (identifed in Section 3.3.1, 3.3.3 and 3.3.4) is reasonable. However, the increased cost due to modeling faults in 2 categories (identified in Section 3.3.2) is high.

3.4.2 Cost of Modeling Faults During Model Revision

Since model checking computes (directly or indirectly) all computations of M to determine whether pr is satisfied, it is especially useful in providing assurance about a system developed from that model. The related problem of model revision [36, 37] focuses on scenarios where model checking produces a counterexample or where an existing model needs to be revised to add new properties (such as safety, liveness and timing constraints). Thus, the goal in model revision is to modify the given model M so that it satisfies the given property pr. Since the revised model is correct by construction, it can assist us in obtaining a correct model of system when model checking ends up finding a counterexample.

Considering read-write restriction must be continued to be satisfied in the revised model, for modeling persistent and malicious faults in Section 3.3.1, variable $m.j$ is readable only to node j. Other nodes cannot read it. Besides, variable $m.j$ cannot be changed by any node; it may be changed only by a fault action. These restrictions have been shown to increase complexity from P to NP-complete in some instances [134].

Regarding revision for the modeling of faults from Section 3.3.2, revising timed automata is considered in [35]. However, the cost of such revision is often higher (exponential in the constants involved in specifying timing constraints. In some circumstance, the problem is even undecidable.

For modeling transient faults in Section 3.3.3, it does not introduce new difficulties in

Faults from	Complexity of Modeling Faults during Model Revision (in terms of the increased state space)
Section 3.3.1	From P to NP-complete in size of statespace.
Section 3.3.2	Undecidable in certain circumstance.
Section 3.3.3	For centralized system, unchanged in complexity class. For distributed system, conjectured to NP-complete.
Section 3.3.4	From P to NP-complete in size of state space.

Table 3.5: Complexity of modeling faults during model revision.

model revision. This is due to the fact that no auxiliary variables are needed to model this fault. An example where such fault modeling is used in model revision includes [2].

Similarly to modeling of faults from Section 3.3.1, the complexity of modeling faults from Section 3.3.4 is increased from P to NP-complete in certain circumstance.

As discussed above, we summarize the complexity issues caused by modeling faults from Sections 3.3.1-3.3.4 during model revision in Table 3.5.

Based on these results in Table 3.5, the complexity increases substantially for model revision. However, efficient heuristics have been found to mitigate the complexity for modeling faults identified in Section 3.3.1, 3.3.3 and 3.3.4. Hence, we argue that model revision for faults from these categories is practical. And for faults discussed in Section 3.3.2, the increased complexity may be too high.

3.5 Relative Completeness with Recent Literature

In this section, we evaluate relative completeness of modeling approach proposed in this dissertation with recent literature. In particular, we study the papers from two premier

conferences in the area of fault tolerance: the International Conference On Distributed Computing Systems (ICDCS) and the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) from 2007 to 2010.

Of 135 fault-tolerant relevant papers, we evaluate each paper whether modeling of those faults is feasible using the approaches mentioned in our work. We find that faults mentioned from 51 papers can be modeled in the approach proposed in Section 3.3.1. Faults mentioned from 13 papers can be modeled in the approach proposed in Section 3.3.2. The fault model in the Section 3.3.3 can be applied to modeling faults from 40 papers. And the fault model in the Section 3.3.4 can be used in modeling faults from 31 papers.We summarize our evaluation that how these faults can be modeled in the our proposed approaches in Table 3.6 and 3.7.

Appropriate Model	Faults proposed in the publications from DSN and ICDCS *since 2007*
Approach in Section 3.3.1	DDoS Attacks in [84], [179] Selfishness of Node/Host in distributed system in [45], [178], [155], [219], [212], [182], [162], [121], [245], [29], [203], [259], [234], [184], [177], [201], [208], [108], [27] Attacks in [161], [239], [99], [170], [40], [223], [183], [144], [65], [210], [252], [192], [11], [12], [81], [248], [185], [255], [127], [129], [218], [44], [230], [4], [250], [30] Worms in [242], [82], [241],
Approach in Section 3.3.2	Power degradation of Battery in MANETs mentioned in [235], Energy consumption in [207], [153], [236], [148], [149], [91], [157], [126], [249], [224], [100], Aging-related bug in [98],

Table 3.6: A classification of faults proposed in DSN & ICDCS 2007-2010 (1).

Appropriate Model	Faults proposed in the publications from DSN and ICDCS *since 2007*
Approach in Section 3.3.3	Data Incoherency in [32], [28], [160], [243], [244], [49], [232], [5], [247], [152], [39], [229], [233], [97], [3], [73], [14], [122] Noise in WSN [258], User's misbehavior in [251], [252], [128], [253] Uncertain data in [62], [48], [25], [197], [187] Transient bugs in [256], [117], [190], [202], [211], [257], [195], [13], [109], [95], [193], [216], [94], [86]
Approach in Section 3.3.4	Message Loss in [231], [75], [176], [180] Crash Failure in [103], [83], [237], [55], [165], [238], [104], [124], [196], [209], [74], [246], [131], [168], [150], [175], [172], [43], [228], [167], [94] Disconnection in distributed system in [28], [112], [254], [205] [221], [154], [105].

Table 3.7: A classification of faults proposed in DSN & ICDCS 2007-2010 (2).

3.6 Summary

The main contributions of this chapter are as follows: i) starting from a taxonomy of faults in [24], which classifies faults into 31 categories, we showed that, faults from 20 categories of these 31 categories can be modeled using transition systems. These faults include Byzantine actions in a networked system, physical deterioration of brake in a braking control system and so on. And, we showed that faults from 11 categories cannot (or should not be) represented using transition systems. These include faults such as buffer overflow, hardware errata and so on. ii) We also investigated the feasibility and practicability of using transition systems to model faults. We showed that (1) the modeling of faults from 18 categories as transition systems is practical and feasible and (2) the modeling of faults from 2 categories as transition systems is not practical although feasible. iii) We investigated the relative completeness of

the proposed approach with recent literature.

Our approach for modeling faults is analogous with the approach for modeling fault-free behavior in model checking, and the latter has been shown to be one of the most successful strategies for analyzing fault-free models. Also, our approach has been used in verifying and revising fault-tolerant programs in the context of specific instances of faults. Hence, we expect the results in this chapter to bridge the gap between theory and practice in providing assurance about fault-tolerant system design. Moreover, our fault-modeling approach is beneficial in the situation where the system is subject to multiple faults from different classes (e.g., node crash, and message loss). In what follows, we are going to use this fault modeling approach to assist in analyzing the system that is subject to multiple faults.

Chapter 4

Automatic Verification of Self-Stabilizing Programs

4.1 Introduction

In this chapter, we describe our approach for automatic verification of self-stabilizing programs. Self-stabilization [60], an ability to converge to a legitimate state from arbitrary initial state(s), enables a program to automatically recover from the occurrence of (transient) faults. In particular, if a self-stabilizing program is perturbed by a transient fault then the program is guaranteed to recover to a legitimate state after faults stop. This property is especially useful in a large distributed network, where it is difficult or impossible to predict the exact dynamic situation. Hence, several algorithms such as routing, leader election, mutual exclusion [60, 63, 89] are designed to be self-stabilizing. Also, self-stabilization is an effective way to accommodate faults from multiple classes. In particular, a stabilizing design is able to tolerate multiple classes of faults that require nonmasking fault-tolerance.

However, verification of self-stabilizing programs is a challenging task [64, 189, 225]. Contrary to traditional verification that considers only a subset of reachable states, verification of stabilization requires one to consider all possible states that could be substantially larger, leading to worsening of the fundamental problem of 'state explosion'.

We explore the automatic verification approach that utilizes the power of model checking, which is one of the most successful automatic verification techniques and aims at automatically verify whether a given program meets the given property. Previously, Tsuchiya et al [225] have proposed an approach for model checking of self-stabilizing programs. This work addresses the problem of state space explosion with the help of symbolic model checking techniques, which use Ordered Binary Decision Diagrams (OBDDs) [41]. This work demonstrates feasibility of applying model checking for verifying self-stabilizing programs. Unfortunately, it only works for programs with a small number of processes. To overcome this limitation, our work identifies a key bottleneck that limits the scalability of model checking of verifying stabilization. Specifically, we propose a novel approach that benefits from the effectiveness of different fairness schedulers while employing the power of symbolic model checking.

We also propose a constraint-based approach to analyze stabilizing programs without introducing fairness scheduler in the program model. The key insight is to reduce the verification task into a well-studied problem of constraint solving. In turn, this problem can be solved by many existing highly optimized solutions. Specially, our approach leverages the power of off-the-shelf SMT solvers that have demonstrated the ability to solve industrial sized-satisfiability instances. To the best of our knowledge, our research is the first one that analyzes stabilization with the help of SMT solvers.

4.1.1 Organization

The rest of this chapter is organized as follows. Section 4.2 presents the formal definition of fairness constraints and stabilization. Section 4.3 recalls existing approaches that use model checking to verify stabilization. In Section 4.4, we propose a verification approach that uses the power of symbolic model checking as well as the effectiveness of fairness schedulers. In Section 4.5, we present a constraint-based verification approach that utilizes the power of the off-the-shelf SMT solver to analyze stabilization.

4.2 Background

A fair computation allows for a fair resolution of non-determinism. Next, we introduce the definition of weakly-fair computation. Intuitively, in a weakly-fair computation, if a guard of an action is continuously true then that action must be executed. Thus weakly-fair computation is defined as follows.

Definition 4.2.1. ***(Weakly-fair computation)*** *$\sigma = \langle s_0, s_1, ... \rangle$ is weakly-fair computation of p iff:*

1. *σ is a computation of p, and*
2. *if any action, say a_i, of p is enabled in all states $s_j, s_{j+1}, s_{j+2} \ldots$ then $\exists k : k \geq j :$ s_{k+1} is obtained by executing st_i from state s_k.* □

Remark 4.2.1. *Note that Definition 2.1.5 does not consider fairness. Hence, as needed, we use the term unfair computation to distinguish the computation without fairness from the one with fairness. The term unfair computation has the same meaning as computation in Definition 2.1.5.*

Definition 4.2.2. ***(Stabilization)*** *Let p be a program and let I be a state predicate of p. We say that p is self-stabilizing for I iff:*

1. *closure: if (s_0, s_1) is a transition of p and $s_0 \in I$, then $s_1 \in I$;*

2. *convergence: every computation of p reaches I, i.e., $\forall \sigma : \sigma$ is of the form $\langle s_0, s_1, s_2, \ldots \rangle$ and σ is computation of $p : (\exists j :: s_j \in I)$.* □

Note that the above definition can be instantiated with unfair computations or with weakly-fair computations. In the former case, we say that the program p is self-stabilizing for I without fairness. And, in the latter case, we say that program p is self-stabilizing for I under weak fairness. Finally, whenever I or fairness level is clear from the context, we omit it.

4.3 An Approach of Using Symbolic Model Checking to Verify Self-stabilizing Programs

In this section, we recall the existing approach of using model checking to verify self-stabilization proposed in [225]. We evaluate this approach with three classic examples in the literature on self-stabilization: Dijkstra's K-state program [59], Ghosh's mutual exclusion program [87] and Hoepman's ring-orientation program [113]. Our case studies demonstrate: (i) this approach is limited in time performance and only works for programs consisting of a small number of processes while showing feasibility of using symbolic model checking to verify stabilizing programs, (ii) the ability of using this approach to verify stabilizing programs remain essentially the same despite the improvement of hardware.

4.3.1 Modeling Self-stabilizing Program

In this section, we review the program model introduced in the approach proposed in [225], which utilize SMV [174] , a symbolic model checker to verify stabilizing programs.

In a SMV program, the behavior of each process is expressed by a module. Given a program action $g_1 \longrightarrow st_1$, we model it inside a module by using keyword ASSSIGN, as follows:

$ASSIGN$

init(v) := initial values;

next(v) := case g_1: $\mathcal{F}_{st_1}(v)$;

1:v;

esac;

where v denotes variable changed in st_1 and $\mathcal{F}_{st_1}$ denotes the assignment function used in st_1.

SMV provides each module a special variable running that is **true** iff the module is currently being executed. Hence, we can force each process to be selected to run infinitely often by adding the declaration

FAIRNESS running

to each process. Clearly this models a weakly-fair computation.

4.3.2 Case Study 1: K-State Token Ring Program

The K-state program consists of $N+1$ processes, numbered from 0 to N. The program topology is a unidirectional ring. Each process $p.i$, $0 \leq i \leq N$, has one variable $x.i$ that denotes the current state value. Each variable has the domain $[0, \ldots, K-1]$.

The program consists of two types of actions. The first type is for process 0. This action is enabled when $x.0$ equals $x.N$. When $p.0$ executes its action, it increments $x.0$ by 1 in modulo K arithmetic. The second type of action is for process $p.i$, $i \neq 0$. This action is enabled when $x.i$ is not equal to $x.(i-1)$. When $p.i$ executes its action, it copies $x.(i-1)$. Thus, the actions are as follows:

$$K_0:: \quad x.0 = x.N \quad \longrightarrow \quad x.0 = (x.0+1) \; mod \; K;$$

$$K_i:: \quad x.i \neq x.(i-1) \quad \longrightarrow \quad x.i = x.(i-1);$$

Remark 4.3.1. *This program is known to be self-stabilizing if $K > N$. In subsequent discussion, we let $K = N+1$.*

The state where x values of all processes is 0 is a legitimate state. In this state, only process 0 is enabled. After process 0 executes, $x.0$ changes to 1 and all other x values are still 0. In this state, only process 1 is enabled. Hence, it can execute and change $x.1$ to 1. Continuing this further, eventually, we reach a state where all x values are 1 where process 0 is the only enabled process and process 0 will increment $x.0$ to 2. The legitimate states of the K-state program are equal to all the states reached in such subsequent execution.

SMV provides a simple approach for modeling weak fairness. In particular, the behavior of each process can be instantiated from a specific module. As the program requires, there

are two types of actions and hence we use two modules, one for K_0 and one for K_i. The module for K_0 specifies variable $x.0$ and takes one parameter, the x value of its predecessor. In SMV, the transition $(s.0, s.1)$ for action $K.0$ is specified by the keyword ASSIGN. Within ASSIGN, '$init(x.0) := 0, 1, 2;$' specifies the value of the variable in the source state, i.e., $s.0$. Moreover, '$next(x.0) := case\ x.0 = x.N : (x.0 + 1)\ mod\ 3;\ 1 : x.0;\ esac;$ ' specifies the value in the target state, i.e., $s.1$. If the guard $(x.0 = x.N)$ is $true$ and $s.1$ is obtained by executing $x.0 = x.0 + 1\ mod\ 3$ from state $s.0$, otherwise the value of $x.0$ remains unchanged. Thus, module for action $K.0$ can be written as follows:

MODULE $type\_K_0(x.N)$

VAR $x.0 : 0, 1, 2;$

ASSIGN $init(x.0) := 0, 1, 2;$

$next(x.0) := case\ (x.0 = x.N) : (x.0 + 1)\ mod\ 3\ ;\ 1\ :\ x.0;\ esac;$

FAIRNESS $running$

Thus, action K_0 can be instantiated from module $type\_K_0$ as follows:

$K_0\ : process\ type\_K_0(x.N);$.

The module for K_i is similar to the one for K_0. It specifies variable $x.i$ and takes $x.(i-1)$, as parameter. The transition $(s.0, s.1)$ for action $K.i$ is specified by the keyword ASSIGN. Within ASSIGN statement, $init(x.i) := 0, 1, 2;$, specifies the value of the variable in the source state. And $next(x.i) := case\ !(x.i = x.j) : x.i = x.(i-1);\ 1 : x.i;\ esac;$, specifies the value in the target state. Hence Action K_i is instantiated from module $type\_K_i$ as follows:

$K_i\ : process\ type\_K_i(x.(i-1));$.

Finally, each process has the declaration FAIRNESS RUNNING to ensure that SMV only

	Execution time(s)			
	K=3	K=4	K=5	K=6
weakly-fair	0	0.03	0.63	5.33
results reported in [225]	0.1	0.4	4.6	43.5
approximate state space	10^1	10^2	10^3	10^4
	Execution time(s)			
	K=7	K=8	K=9	K=10
weakly-fair	34.30	139.10	1276.08	N/A
results reported in [225]	285.2	1836.0	N/A	N/A
approximate state space	10^5	10^7	10^8	10^{10}

Table 4.1: Verification results for the k-state program.

considers computation paths where each process executes infinitely often.

We verified the K-state program for $3 \leq K \leq 10$. Table 4.4 gives the verification time for model checking the K-state program for different values of K. N/A in this table means the result was not available within an admissible amount of time (1 hour).

4.3.3 Case Study 2: Ghosh's Binary Mutual Exclusion Protocol

In this section, we present our second case study, namely, Ghosh's binary mutual exclusion protocol [87]. Ghosh's binary mutual exclusion protocol considers a network system of $2m-1 (m \geq 2)$ nodes, numbered from 0 to $2m-1$. The neighbor relation is defined as follows:

- n_0 has one neighbor n_1;
- $n_{2i-1} (1 \leq i \leq m-1)$ has three neighbors n_{2i-2}, n_{2i}, and n_{2i+1};
- $n_{2i} (1 \leq i \leq m-1)$ has three neighbors n_{2i-2}, n_{2i-1}, and n_{2i+1};
- n_{2m-1} has one neighbor n_{2m-2}.

The state s_i of each node n_i can be either 0 or 1. Each node can read its own state and the state of its neighbor nodes. The protocol defines the four types of actions as follows:

for n_0:

$$s_0 \neq s_1 \longrightarrow s_0 = 1 - s_0;$$

for n_{2m-1}:

$$s_{2m-1} = s_{2m-2} \longrightarrow s_{2m-1} = 1 - s_{2m-1};$$

for $n_{2i-1} (1 \leq i \leq m-1)$:

$$s_{2i-2} = s_{2i-1} = s_{2i} \ \wedge \ s_{2i-1} \neq s_{2i+1} \longrightarrow s_{2i-1} = 1 - s_{2i-1};$$

for $n_{2i} (1 \leq i \leq m-1)$:

$$s_{2i-2} = s_{2i-1} = s_{2i+1} \ \wedge \ s_{2i} \neq s_{2i+1} \longrightarrow s_{2i} = 1 - s_{2i};$$

We model the program and check the self-stabilization property of the protocol using the same approach as mentioned in Section 4.4.1.2. The verification results for this case are shown in Table 4.5.

4.3.4 Case Study 3: Hoepman's Uniform Ring-orientation Program

In this section, we present our third case study, namely, Hoepman's uniform ring-orientation program [113]. Hoepman's uniform deterministic ring-orientation program considers a system of n nodes, numbered from 0 to $n-1$, which are organized as a uniform ring of odd length. Each node n_i has a *color*, $Color_i$, with domain $\{0, 1\}$. To impose a direction, each

	Execution time(s)			
	n=8	n=10	n=12	n=14
weakly-fair	0.4	2.93	22.43	138.05
results reported in [225]	3.1	22.9	182.0	1161.5
approximate state space	10^2	10^3	10^3	10^4
	Execution time(s)			
	n=16	n=18	n=20	-
weakly-fair	693.27	2819.05	N/A	-
results reported in [225]	N/A	N/A	N/A	-
approximate state space	10^4	10^5	-	-

Table 4.2: Verification results for Ghosh's mutual exclusion program.

node stores a *phase*, $Phase_i$, with domain $\{0, 1\}$. A global legitimate state is one where all the nodes are oriented in the same direction. A ring orientation program is self-stabilizing iff it reaches a legitimate state from any initial state. To achieve self-stabilization, this program defines the following four types of actions for each node n_i:

$$
\begin{aligned}
&Color_{neighbor1} = Color_{neighbor2} \\
&\qquad \longrightarrow \; Color_i = 1 - Color_{neightbor1}, \\
&\qquad\qquad Phase_i = 1;
\end{aligned}
$$

$$
\begin{aligned}
&Color_{neighbor1} = Color_i = 1 - Color_{neighbor2} \\
&\wedge \; Phase_i = Phase_{neighbor2} = 1 \\
&\wedge \; Phase_{neighbor1} = 0 \\
&\qquad \longrightarrow \; Color_i = 1 - Color_i, \\
&\qquad\qquad (Phase_i) = 0, \\
&\qquad\qquad direction_i = n_{neighbor1} \hookrightarrow n_i \hookrightarrow n_{neightbor2};
\end{aligned}
$$

$$
Color_{neighbor2} = Color_i = 1 - Color_{neighbor1}
$$

$\wedge Phase_i = Phase_{neighbor1} = 1$

$\wedge Phase_{neighbor2} = 1$

$\longrightarrow \; Color_i = 1 - Color_i,$

$Phase_i = 0,$

$direction_i{=}n_{neighbor1} \hookleftarrow n_i \hookleftarrow n_{neightbor2};$

$(Color_{neighbor1} = Color_i = 1 - Color_{neighbor2}$

$\wedge Phase_{neighbor1} = Phase_i)$

$|(Color_{neighbor2} = Color_i = 1 - Color_{neighbor1}$

$\wedge Phase_i = Phase_{neighbor2})$

$\longrightarrow \; Phase_i = 1 - Phase_i;$

In the above actions, Action 1 requires that if a node has the same color as both its neighbors, then it inverts its color. This action creates patterns such as 001 and 110 around the ring since it is of odd length. Actions 2 and 3 require that if one node has the same color and the opposite phase as one of its neighbors, then the direction is from the node with phase 0 to the node with phase 1. Action 4 requires that if one node has the same color and the same phase as one of its neighbors, then it inverts its phase.

We model the program and check the self-stabilization property of the protocol using the same approach as mentioned in Section 4.4.1.2. The verification results for this case are shown in Table 4.6.

	Execution time(s)		
	n=3	n=5	n=7
weakly-fair	0.17	18.23	1113.77
results reported in [225]	1.3	128.1	N/A
approximate state space	10^1	10^3	10^4

Table 4.3: Verification results for Hoepman's ring-orientation program.

4.3.5 Analysis

From these case studies, we observe that the approach proposed in [225] has limited scalability. In the first case study, in [225], authors have shown the feasibility of verification of k-state program for upto $K = 8$. In particular, the time reported in [225] for $K = 8$ is 1836.0s whereas the time for the corresponding verification is 139.1s. Since the underlying tool as well as the program remains the same, this change is due to improved hardware over last few years. However, what this result does show is that in spite of the improved hardware, the ability to verify under weak fairness remains essentially the same. Specifically, if we assume a reasonable time constraint permissible (e.g., one hour) for verification then the change in hardware made it possible to achieve verification for $K = 9$ as opposed to $K = 8$.

As discussed in [173], one of the reasons for this is that to model fairness, one needs to ensure that each process can execute infinitely often. Achieving this increases the OBDD size for reachable states quadratically.

4.4 Effect of Fairness in Model Checking of Self-Stabilizing Programs

In this section, we introduce our approach which utilizes the effectiveness of fairness as well as the power of symbolic model checking to verify stabilizing program. We first present our

approach of model checking of stabilizing programs under unfair computation. Then, we introduce our other two approaches for reducing verification cost purpose, including, (i) decomposing the program into sub-components and verifying each sub-components separately; and (ii) verifying a weaker version of stabilizing programs, namely weak stabilization, to trade-off the cost with precision.

4.4.1 Using Symbolic Model Checking to Verify Self-stabilizing Program Under Unfair Computation

In this section, we propose an approach of model checking self-stabilization under unfair computation for the case where weak fairness is not essential for self-stabilization. To compare the verification performance of this approach with the approach under weak fairness computation mentioned in Section 4.3, we illustrate this approach with the three same case studies. We perform the experiments in the same hardware setting. The results show that the approach of model checking self-stabilization under unfair computation is significantly more scalable.

4.4.1.1 Modeling Self-stabilizing Program under Unfair Computation

Now, we describe how to model self-stabilizing program under unfair computation in SMV. Recall that a self-stabilizing program $\mathcal{P}$ consists of a set of guarded commands of the following form:

$action_1 : \quad g_1 \longrightarrow st_1;$

$action_2 : \quad g_2 \longrightarrow st_2;$

$\ldots$

$action_i: \quad g_i \longrightarrow st_i;$

$\ldots$

$action_n: \quad g_n \longrightarrow st_n;$

To model $\mathcal{P}$ under unfair computation in SMV, we use the TRANS keyword and model $action_i$ as follows:

$$conjunction_i: \; g_i \wedge \Big(\bigwedge_{st_i \text{ updates } v_j} next(v_j) = \mathcal{F}_{st_i}(v_j) \Big) \wedge \Big(\bigwedge_{st_i \text{ does not update } v_j} next(v_j) = v_j \Big)$$

The above formula requires that g_i must be true in the initial state. Moreover, if v_j updated by $action_i$ then $next(v_j)$ corresponds to the value given by st_i. Otherwise, v_j remains unchanged.

Since the use of TRANS in SMV requires the user to explicltly ensure that the transition relation is total. The transition relation is total if every state has a successor state. Hence, we add an additional action "$\neg(\bigvee_{i=1,\ldots,n} g_i) \longrightarrow skip$;" to the program. Using the approach for modeling actions, this action is modeled as follows:

$$conjunction_{addtional}: \; \neg\Big(\bigvee_{i=1,\ldots,n} g_i\Big) \wedge \Big(\bigwedge next(v_j) = v_j\Big);$$

Based on the above modeling, we model the whole program actions as a disjunction of $conjunction_1$, $conjunction_2$, $\ldots conjunction_n$ and $conjunction_{addtional}$, as follows:

$$
\begin{aligned}
& g_1 \longrightarrow next(v_1) = \mathcal{F}_{st_1}(v_1) \ \& \bigwedge_{j=2,\ldots,n} next(v_j) = v_j \\
\vee & g_2 \longrightarrow next(v_2) = \mathcal{F}_{st_2}(v_2) \ \& \bigwedge_{j=1,\ldots,n \& j \neq 2} next(v_j) = v_j \\
& \ldots \\
\vee & g_i \longrightarrow next(v_i) = \mathcal{F}_{st_i}(v_i) \ \& \bigwedge_{j=1,\ldots,n \& j \neq i} next(v_j) = v_j \\
& \ldots \\
\vee & g_n \longrightarrow next(v_n) = \mathcal{F}_{st_n}(v_n) \ \& \bigwedge_{j=1,\ldots,n-1} next(v_j) = v_j \\
\vee & \neg \bigvee_{i=1,\ldots,n} g_i \ \& \bigwedge_{j=1,\ldots,n} next(v_j) = v_j;
\end{aligned}
$$

Hence, in the modeling approach under unfair computation, the whole program is modeled as one transition relation in SMV.

4.4.1.2 Case Study 1: K-State Token Ring Program (Cont'd)

In this section, we continue with the verification of K-state program. We first discuss how we model the K-state program in SMV under unfair computation. Then, we provide the verification results under unfair computation. We compare these results with the corresponding verification results under weak computation. The comparison results show that for K-state program, the verification performance is substantially improved under unfair computation.

Figure 4.1 gives a SMV program of K-state program with k=3 under unfair computation. As shown in Figure 4.1, $x0$, $x1$ and $x2$ denotes the states of the three processes. Lines 1-6 define $x0$, $x1$, $x2$ and initialize them. Lines 7-9 define $x0priv$, $x1priv$ and $x2priv$, that are used to describe privilege condition for each process. Lines 10-13 define $condition1$ and $condtion2$ to describe the self-stabilization. Line 15 specifies the program action, which is a

disjunction of three possible cases. These three cases include: 1)process 0 is privileged, $x0$ is assigned new value and states of other two processes $x1$ and $x2$ remains unchanged; 2) process 1 is privileged, $x1$ is assigned new value and states of other two processes $x0$ and $x2$ remains unchanged; and, 3) process 2 is privileged, $x2$ is assigned new value and states of other two processes $x1$ and $x0$ remains unchanged. Note that if multiple processes are privileged then one of them is non-deterministically chosen for execution.

MODULE main
VAR
$x0 : \{0, 1, 2\};$ (1)
$x1 : \{0, 1, 2\};$ (2)
$x2 : \{0, 1, 2\};$ (3)
INIT
$x0 = \{0, 1, 2\}\&$ (4)
$x1 = \{0, 1, 2\}\&$ (5)
$x2 = \{0, 1, 2\}$ (6)
DEFINE
$x0priv := (x0 = x2);$ (7)
$x1priv :=!(x1 = x0);$ (8)
$x2priv :=!(x2 = x1);$ (9)
$condition1 := ((x0priv\&!x1priv\&!x2priv)|$
$(!x0priv\&x1priv\&!x2priv)|$
$(!x0priv\&!x1priv\&x2priv));$
(10)
$condition2 := AFx0priv\&AFx1priv\&AFx2priv;$ (11)
$legitimate := condition1\&condition2;$ (12)

SPEC AF legitimate (13)
TRANS (14)

$((x0 = x2)\&next(x0) = (x0 + 1)mod3\&next(x1) = x1\&next(x2) = x2)$
$|(!(x1 = x0)\&next(x1) = x0\&next(x2) = x2\&next(x0) = x0)$
$|(!(x2 = x1)\&next(x2) = x1\&next(x0) = x0\&next(x1) = x1)$
$|(next(x2) = x2\&next(x0) = x0\&next(x1) = x1)$
(15)

Figure 4.1: k-state program under unfair computation.

Execution time(s)					
	K=3	K=4	K=5	K=6	K=7
unfair	0	0	0	0	0.02
weakly-fair	0	0.03	0.63	5.33	34.30
approximate state space	10^1	10^2	10^3	10^4	10^5
size of BDD nodes(under unfair)	292	786	2423	4373	8346
size of BDD nodes(under weakfair)	680	3435	11251	17880	42131
Execution time(s)					
	K=8	K=9	K=10	K=50	K=51
unfair	0.03	0.05	0.08	3466.30	N/A
weakly-fair	139.10	1276.08	N/A	N/A	N/A
approximate state space	10^7	10^8	10^{10}	10^{84}	-
size of BDD nodes(under unfair)	10067	11475	14870	1842498	-
size of BDD nodes(under weakfair)	108723	564794	N/A	N/A	-

Table 4.4: Verification results for the k-state program.

Table 4.4 gives the verification time for model checking the K-state program for different values of K, where $3 \leq K \leq 9$ *or* $K = 50$. Here, N/A means that the result is not available within an admissible amount of time (1 hour).

4.4.1.3 Case Study 2: Ghosh's Binary Mutual Exclusion Protocol (Cont'd)

In this section, we consider Ghosh's mutual protocol under unfair computation. We first describe how we model Ghosh's mutual protocol in SMV under unfair computation. Then, we provide the verification results under unfair computations and compare these results with the corresponding results under weak fair computations. The comparison results show the scalability of modeling self-stabilizing program under unfair computation for Ghosh's mutual protocol.

Figure 4.2 - 4.4 shows a SMV program of Ghosh's mutual exclusion protocol with $n = 8$ modeled under unfair computation. As shown in Figure 4.2, x_i $(i = 0 \ldots 7)$ denotes the states of the eight processes. Lines 1-4 define and initialize these state variables. Lines 5-12

	Execution time(s)				
	n=8	n=10	n=12	n=14	n=16
unfair	0	0	0	0	0.02
weakly-fair	0.4	2.93	22.43	138.05	693.27
approximate state space	10^2	10^3	10^3	10^4	10^4
size of BDD nodes(under unfair)	1099	1811	2831	4153	5813
size of BDD nodes(under weakfair)	10082	10483	11693	20786	43294
	Execution time(s)				
	n=18	n=20	n=50	n=100	-
unfair	0.02	0.03	0.35	4.77	-
weakly-fair	2819.05	N/A	N/A	N/A	-
approximate state space	10^5	10^6	10^{16}	10^{31}	-
size of BDD nodes(under unfair)	7847	10000	10134	10288	-
size of BDD nodes(under weakfair)	99088	N/A	N/A	N/A	-

Table 4.5: Verification results for Ghosh's mutual exclusion program.

define $xipriv(i = 0 \ldots 7)$, that are used to describe privilege condition for each process. Lines 13-17 define *condition*1 and *condtion*2 to describe the self-stabilization. Line 18 specifies the program action, which is a disjunction of eight possible cases. These possible cases include: 1) process 0 is privileged, x_0 is assigned new value and states of other processes remains unchanged; 2) process 1 is privileged, x_1 is assigned new value and states of other processes remains unchanged; and so on. Once again, the modeling captures non-deterministic execution one of the privileged process.

Our case study verified the Ghosh's mutual protocol for $n = 2i$ where $4 \leq i \leq 10$ or $i = 25, 50$. The verification results are shown in Table 4.5. N/A in this table means the result was not available within an admissible amount of time (1 hour).

4.4.1.4 Case Study 3: Hoepman's Uniform Ring-orientation Program (Cont'd)

In this section, we model Hoepman's uniform ring program under unfair computation in SMV. We describe the modeling with 3 processes (Figure 4.5 - 4.10).

MODULE main
VAR
$x0 : \{0,1\}; x1 : \{0,1\}; x2 : \{0,1\}; x3 : \{0,1\};$ (1)
$x4 : \{0,1\}; x5 : \{0,1\}; x6 : \{0,1\}; x7 : \{0,1\};$ (2)
INIT
$x0 = \{0,1\}\&x1 = \{0,1\}\&x2 = \{0,1\}\&x3 = \{0,1\}\&$ (3)
$x4 = \{0,1\}\&x5 = \{0,1\}\&x6 = \{0,1\}\&x7 = \{0,1\}$ (4)
DEFINE
$x0priv := (x0! = x1);$ (5)
$x1priv := ((x1 = x0)\&(x1 = x2)\&!(x1 = x3));$ (6)
$x2priv := (!(x2 = x0)\&!(x2 = x1)\&!(x2 = x3));$ (7)
$x3priv := ((x3 = x2)\&(x3 = x4)\&!(x3 = x5));$ (8)
$x4priv := (!(x4 = x2)\&!(x4 = x3)\&!(x4 = x5));$ (9)
$x5priv := ((x5 = x4)\&(x5 = x6)\&!(x5 = x7));$ (10)
$x6priv := (!(x6 = x4)\&!(x6 = x5)\&!(x6 = x7));$ (11)
$x7priv := (x7 = x6);$ (12)
$condition1 :=$ (
$(x0priv\&!x1priv\&!x2priv\&!x3priv\&!x4priv\&!x5priv\&!x6priv\&!x7priv)|$
$(!x0priv\&x1priv\&!x2priv\&!x3priv\&!x4priv\&!x5priv\&!x6priv\&!x7priv)|$
$(!x0priv\&!x1priv\&x2priv\&!x3priv\&!x4priv\&!x5priv\&!x6priv\&!x7priv)|$
$(!x0priv\&!x1priv\&!x2priv\&x3priv\&!x4priv\&!x5priv\&!x6priv\&!x7priv)|$
$(!x0priv\&!x1priv\&!x2priv\&!x3priv\&x4priv\&!x5priv\&!x6priv\&!x7priv)|$
$(!x0priv\&!x1priv\&!x2priv\&!x3priv\&!x4priv\&x5priv\&!x6priv\&!x7priv)|$
$(!x0priv\&!x1priv\&!x2priv\&!x3priv\&!x4priv\&!x5priv\&x6priv\&!x7priv)|$
$(!x0priv\&!x1priv\&!x2priv\&!x3priv\&!x4priv\&!x5priv\&!x6priv\&x7priv))$
(13)
$condition2 := AFx0priv\&AFx1priv\&AFx2priv\&AFx3priv$
$\&AFx4priv\&AFx5priv\&AFx6priv\&AFx7priv;$ (14)
$legitimate := condition1\&condition2;$ (15)

Figure 4.2: Ghosh's mutual protocol under unfair computation (1).

We verified the Hoepman's uniform ring program for $n = 2i + 1$ where $1 \leq i \leq 4$ and $i = 50, 100, 150$ and 200. The verification results for this case are shown in Table 4.6. N/A in this table means the result was not available within an admissible amount of time (1 hour).

SPEC AF legitimate (16)

TRANS

$$
\begin{aligned}
&(((x0 = !x1)\ \&\ next(x0) = x1)\ \&\ next(x1) = x1 \\
&\quad \&\ next(x2) = x2 \&\ next(x3) = x3\ \&\ next(x4) = x4 \\
&\quad \&\ next(x5) = x5\ \&\ next(x6) = x6\ \&\ next(x7) = x7) \\
&|(next(x0) = x0\ \&\ (((x1 = x0)\ \&\ (x1 = x2)\ \&\ !(x1 = x3))\ ) \\
&\quad \&\ next(x1) = 1 - x1\ \&\ next(x2) = x2 \\
&\quad \&\ next(x3) = x3\ \&\ next(x4) = x4 \\
&\quad \&\ next(x5) = x5\ \&\ next(x6) = x6 \\
&\quad \&\ next(x7) = x7) \\
&|(next(x0) = x0\ \&\ next(x1) = x1\ \&\ ((!(x2 = x0) \\
&\quad \&\ !(x2 = x1) \&\ !(x2 = x3))\ \&\ next(x2) = 1 - x2) \\
&\quad \&\ next(x3) = x3\ \&\ next(x4) = x4 \\
&\quad \&\ next(x5) = x5\ \&\ next(x6) = x6\ \&\ next(x7) = x7) \\
&|(next(x0) = x0\ \&\ next(x1) = x1\ \&\ next(x2) = x2) \\
&\quad \&\ (((x3 = x2)\ \&\ (x3 = x4) \\
&\quad\ \&\ !(x3 = x5)) \&\ next(x3) = 1 - x3) \\
&\quad \&\ next(x4) = x4\ \&\ next(x5) = x5 \\
&\quad \&\ next(x6) = x6\ \&\ next(x7) = x7) \\
&|(next(x0) = x0\ \&\ next(x1) = x1\ \&\ next(x2) = x2 \\
&\quad \&\ next(x3) = x3\ \&\ ((!(x4 = x2) \\
&\quad \&\ !(x4 = x3)\ \&\ !(x4 = x5)) \\
&\quad \&\ next(x4) = 1 - x4)\ \&\ next(x5) = x5 \\
&\quad \&\ next(x6) = x6\ \&\ next(x7) = x7) \\
&|(next(x0) = x0\ \&\ next(x1) = x1\ \&\ next(x2) = x2 \\
&\quad \&\ next(x3) = x3\ \&\ next(x4) = x4 \\
&\quad \&\ (((x5 = x4)\ \&\ (x5 = x6)) \\
&\quad \&\ !(x5 = x7))\ \&\ next(x5) = 1 - x5 \\
&\quad \&\ next(x6) = x6\ \&\ next(x7) = x7)
\end{aligned}
$$

Figure 4.3: Ghosh's mutual protocol under unfair computation (2).

$$
\begin{array}{l}
|(next(x0) = x0 \ \&\ next(x1) = x1 \ \&\ next(x2) = x2 \\
\quad \&\ next(x3) = x3 \ \&\ next(x4) = x4 \\
\quad \&\ next(x5) = x5 \& \ ((!(x6 = x4) \\
\quad \&\ !(x6 = x5) \ \&\ !(x6 = x7)) \\
\quad \&\ next(x6) = 1 - x6) \ \&\ next(x7) = x7) \\
|(next(x0) = x0 \ \&\ next(x1) = x1 \ \&\ next(x2) = x2 \\
\quad \&\ next(x3) = x3 \ \&\ next(x4) = x4 \\
\quad \&\ next(x5) = x5 \& \ next(x6) = x6 \\
\quad \&\ ((x7 = x6) \ \&\ next(x7) = 1 - x7)) \\
|(next(x0) = x0 \ \&\ next(x1) = x1 \ \&\ next(x2) = x2 \\
\quad \&\ next(x3) = x3 \ \&\ next(x4) = x4 \\
\quad \&\ next(x5) = x5 \& \ next(x6) = x6 \ \&\ next(x7) = x7) \qquad (17)
\end{array}
$$

Figure 4.4: Ghosh's mutual protocol under unfair computation (3).

MODULE main
VAR

$$
\begin{array}{ll}
p0\_s : \{0,1\}; p0\_t : \{0,1\}; p0\_dir : \{B,F\}; p0\_AP1\_pc : \{-1,1\}; & (1) \\
p1\_s : \{0,1\}; p1\_t : \{0,1\}; p1\_dir : \{B,F\}; p1\_AP1\_pc : \{-1,1\}; & (2) \\
p2\_s : \{0,1\}; p2\_t : \{0,1\}; p2\_dir : \{B,F\}; p2\_AP1\_pc : \{-1,1\}; & (3)
\end{array}
$$

INIT

$$
\begin{array}{ll}
p0\_s = \{0,1\} \& p0\_t = \{0,1\} \& p0\_dir = \{B,F\} \& p0\_AP1\_pc = \{-1,1\} \& & (4) \\
p1\_s = \{0,1\} \& p1\_t = \{0,1\} \& p1\_dir = \{B,F\} \& p1\_AP1\_pc = \{-1,1\} \& & (5) \\
p2\_s = \{0,1\} \& p2\_t = \{0,1\} \& p2\_dir = \{B,F\} \& p2\_AP1\_pc = \{-1,1\} & (6)
\end{array}
$$

Figure 4.5: Hoepman's ring program under unfair computation (1).

DEFINE

$p0\_des := ((p0\_dir = B)\&(p0\_AP1\_pc = -1))$
$\quad |((p0\_dir = F)\&(p0\_AP1\_pc = 1));$ (7)

$p1\_des := ((p1\_dir = B)\&(p1\_AP1\_pc = -1))$
$\quad |((p1\_dir = F)\&(p1\_AP1\_pc = 1));$ (8)

$p2\_des := ((p2\_dir = B)\&(p2\_AP1\_pc = -1))$
$\quad |((p2\_dir = F)\&(p2\_AP1\_pc = 1));$ (9)

$p0\_S1 := case(p0\_AP1\_pc = -1) : p2\_s;$
$\quad 1 : p1\_s; esac;$ (10)

$p0\_T1 := case(p0\_AP1\_pc = -1) : p2\_t;$
$\quad 1 : p1\_t; esac;$ (11)

$p0\_S2 := case(p0\_AP1\_pc = -1) : p1\_s;$
$\quad 1 : p2\_s; esac;$ (12)

$p0\_T2 := case(p0\_AP1\_pc = -1) : p1\_t;$
$\quad 1 : p2\_t; esac;$ (13)

$p0\_r1 := (p0\_S1 = p0\_S2);$ (14)

$p0\_r2 := (p0\_S1 = p0\_s)\&(p0\_S2 = !p0\_s)$
$\quad \&(p0\_T2 = p0\_t)\&(p0\_T1 = !p0\_t)\&(p0\_t = 1);$ (15)

$p0\_r3 := (p0\_S2 = p0\_s)\&(p0\_S1 = !p0\_s)$
$\quad \&(p0\_T1 = p0\_t)\&(p0\_T2 = !p0\_t)\&(p0\_t = 1);$ (16)

$p0\_r4 := ((p0\_S1 = p0\_s)\&(p0\_S2 = !p0\_s)$
$\quad \&(p0\_T1 = p0\_t))|((p0\_S2 = p0\_s)$
$\quad \&(p0\_S1 = !p0\_s)\&(p0\_T2 = p0\_t));$ (17)

Figure 4.6: Hoepman's ring program under unfair computation (2).

DEFINE

$p1\_S1 := case(p1\_AP1\_pc = -1) : p0\_s;$

$1 : p2\_s; esac;$ (18)

$p1\_T1 := case(p1\_AP1\_pc = -1) : p0\_t;$

$1 : p2\_t; esac;$ (19)

$p1\_S2 := case(p1\_AP1\_pc = -1) : p2\_s;$

$1 : p0\_s; esac;$ (20)

$p1\_T2 := case(p1\_AP1\_pc = -1) : p2\_t;$

$1 : p0\_t; esac;$ (21)

$p1\_r1 := (p1\_S1 = p1\_S2);$ (22)

$p1\_r2 := (p1\_S1 = p1\_s)\&(p1\_S2 = !p1\_s)$

$\&(p1\_T2 = p1\_t)\&(p1\_T1 = !p1\_t)\&(p1\_t = 1);$ (23)

$p1\_r3 := (p1\_S2 = p1\_s)\&(p1\_S1 = !p1\_s)$

$\&(p1\_T1 = p1\_t)\&(p1\_T2 = !p1\_t)\&(p1\_t = 1);$ (24)

$p1\_r4 := ((p1\_S1 = p1\_s)\&(p1\_S2 = !p1\_s)$

$\&(p1\_T1 = p1\_t))|((p1\_S2 = p1\_s)$

$\&(p1\_S1 = !p1\_s)\&(p1\_T2 = p1\_t));$ (25)

Figure 4.7: Hoepman's ring program under unfair computation (3).

	Execution time(s)				
	n=3	n=5	n=7	n=9	n=51
unfair	0	0.03	0.08	0.12	11.95
weakly-fair	0.17	18.23	1113.77	N/A	N/A
approximate state space	10^1	10^3	10^4	10^5	10^{31}
size of BDD nodes(under unfair)	4090	10047	11680	10543	63468
size of BDD nodes(under weakfair)	10425	61570	776284	N/A	N/A
	Execution time(s)				
	n=101	n=201	n=301	n=303	-
unfair	95.65	875.23	3420.98	N/A	-
weakly-fair	N/A	N/A	N/A	N/A	-
approximate state space	10^{60}	10^{121}	10^{181}	-	-
size of BDD nodes(under unfair)	125809	324420	726620	-	-
size of BDD nodes(under weakfair)	N/A	N/A	N/A	-	-

Table 4.6: Verification results for Hoepman's ring-orientation program.

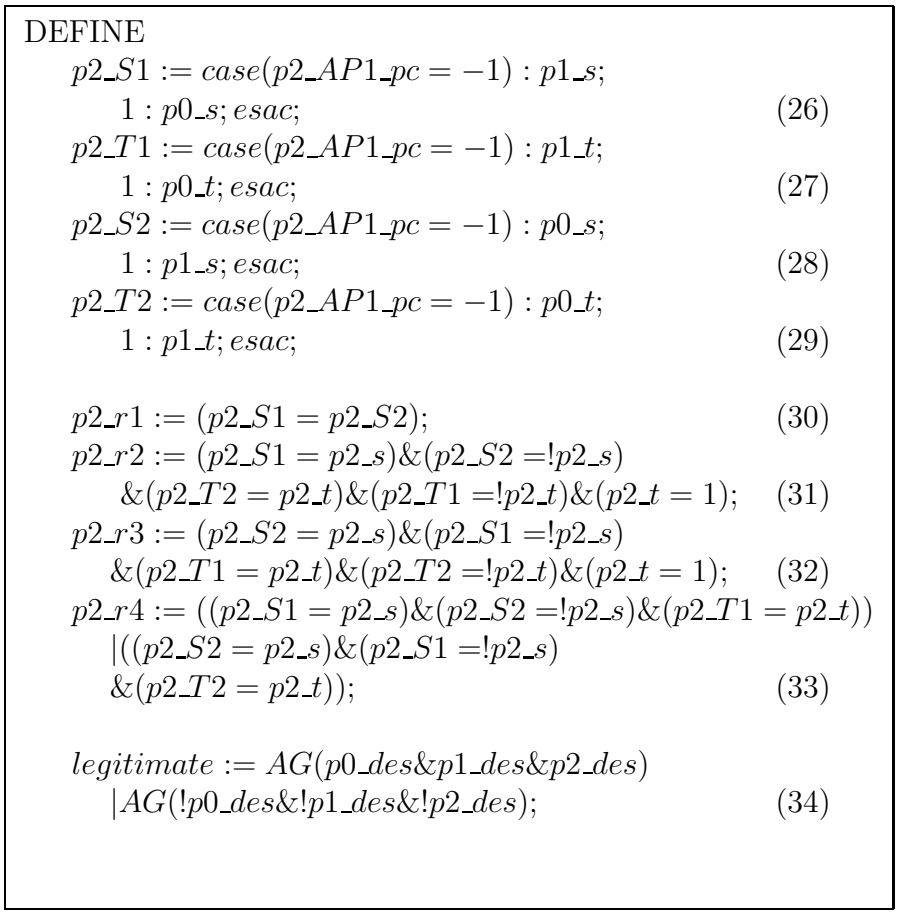

Figure 4.8: Hoepman's ring program under unfair computation (4).

SPEC AF legitimate (35)

TRANS

$(((p0\_r1 \& next(p0\_s) = !p0\_S1) | ((p0\_r2 | p0\_r3) \& next(p0\_s) = !p0\_s)$
$| next(p0\_s) = p0\_s)$
$\& ((p0\_r1 \& next(p0\_t) = 1) | ((p0\_r2 | p0\_r3 | p0\_r4)$
$\& next(p0\_t) = !p0\_t) | next(p0\_t) = p0\_t)$
$\& ((p0\_r2 \& next(p0\_dir) = F) | (p0\_r3 \& next(p0\_dir) = B)$
$| next(p0\_dir) = p0\_dir) \& (next(p0\_AP1\_pc) = p0\_AP1\_pc)$
$\& next(p1\_s) = p1\_s \& next(p1\_t) = p1\_t \& next(p1\_dir) = p1\_dir$
$\& next(p1\_AP1\_pc) = p1\_AP1\_pc \& next(p2\_s) = p2\_s$
$\& next(p2\_t) = p2\_t \& next(p2\_dir) = p2\_dir$
$\& next(p2\_AP1\_pc) = p2\_AP1\_pc) \& (p2\_T2 = p2\_t));$ (36)

$|$ (37)

$(((p1\_r1 \& next(p1\_s) = !p1\_S1) | ((p1\_r2 | p1\_r3)$
$\& next(p1\_s) = !p1\_s) | next(p1\_s) = p1\_s)$
$\& ((p1\_r1 \& next(p1\_t) = 1) | ((p1\_r2 | p1\_r3 | p1\_r4)$
$\& next(p1\_t) = !p1\_t)$
$| next(p1\_t) = p1\_t) \& ((p1\_r2 \& next(p1\_dir) = F) | (p1\_r3$
$\& next(p1\_dir) = B)$
$| next(p1\_dir) = p1\_dir) \& (next(p1\_AP1\_pc) = p1\_AP1\_pc)$
$\& next(p0\_s) = p0\_s$
$\& next(p0\_t) = p0\_t \& next(p0\_dir) = p0\_dir$
$\& next(p0\_AP1\_pc) = p0\_AP1\_pc$
$\& next(p2\_s) = p2\_s \& next(p2\_t) = p2\_t$
$\& next(p2\_dir) = p2\_dir$
$\& next(p2\_AP1\_pc) = p2\_AP1\_pc)$ (38)

Figure 4.9: Hoepman's ring program under unfair computation (5).

$$
\begin{aligned}
&| && (39)\\
&(((p2\_r1 \& next(p2\_s) = !p2\_S1) | ((p2\_r2 | p2\_r3)\\
&\& next(p2\_s) = !p2\_s)\\
&| next(p2\_s) = p2\_s)\\
&\quad \& ((p2\_r1 \& next(p2\_t) = 1) | ((p2\_r2 | p2\_r3 | p2\_r4)\\
&\quad \& next(p2\_t) = !p2\_t) | next(p2\_t) = p2\_t)\\
&\quad \& ((p2\_r2 \& next(p2\_dir) = F) | (p2\_r3 \& next(p2\_dir) = B)\\
&\quad | next(p2\_dir) = p2\_dir) \& (next(p2\_AP1\_pc) = p2\_AP1\_pc)\\
&\quad \& next(p0\_s) = p0\_s \& next(p0\_t) = p0\_t\\
&\quad \& next(p0\_dir) = p0\_dir \& next(p0\_AP1\_pc) = p0\_AP1\_pc\\
&\quad \& next(p1\_s) = p1\_s \& next(p1\_t) = p1\_t\\
&\quad \& next(p1\_dir) = p1\_dir \& next(p1\_AP1\_pc) = p1\_AP1\_pc) && (40)
\end{aligned}
$$

Figure 4.10: Hoepman's ring program under unfair computation (6).

4.4.1.5 Analysis

Based on the results in Tables 4.4, 4.5 and 4.6, verification of significantly large system is possible if we consider unfair computations. As an illustration, consider K-state program: It was possible to achieve verification for $K = 50$ in less than 1 hour. And, in this case, the corresponding state space is 10^{85}. By contrast, verification with weak fairness could not complete when state space was 10^{11}. This difference in performance can be explained by observing the size of OBDDs representing state sets in the forward search.

Compared with the approach in [225], the approach presented here has the OBDD size for the reached states running linearly. The overall time complexity for this approach is $O(n^3)$. As introduced in [173], this performance dues to three factors: a linear increase in the transition relation OBDD, a linear increase in the state set OBDD, and a linear increase in the number of iterations required for successful verification.

4.4.2 Utilizing Decomposition to Reduce the Cost of Using Symbolic Model Checking to Verify Self-stabilizing Program

The results in Section 4.3 show that verification of self-stabilization under unfair computation is substantially faster than that under weakly-fair computation. Thus, the natural question is what can a designer do if the program at hand requires weak fairness to provide self-stabilization, i.e., the program is not self-stabilizing under unfair computations. Examples of such programs include [18, 19, 116].

Generally, there are two approaches that can be used when model checking fails due to lack of sufficient time or space. The first approach is to require the designer to perform extra work (e.g., abstraction, decomposition, identifying partial order reductions, etc.) that will reduce the cost of verification. The second approach is to verify a variation of the model/property such that the variation will still provide a reasonable assurance about the goal at hand.

In this section, we focus on the first such approach where we show that decomposition of a self-stabilizing program can provide substantial benefit in reducing the cost of verification. We note that while decomposition is one of the approaches in reducing cost of verification, the effect of this approach in model checking of self-stabilizing programs is not addressed. To address this limitation, in this approach, designer needs to partition the program into components such that each component satisfies its property without fairness. Subsequently, we can use existing composition results to show that their composition is correct under fair execution. There are several such approaches to show that the composed program is self-stabilizing based on the properties of individual components. Since the focus of our work is not to identify new strategies of interference freedom, we only consider some of the

simple and commonly used approaches and describe them, next. We note, however, that the subsequent discussion also applies to other approaches [64, 88] for proving self-stabilization of a program that consists of several components.

Let $C1$ and $C2$ be two components (programs) such that variables of $C1$ and $C2$ are disjoint. Let p be the program obtained by combining the actions of $C1$ and $C2$. Let legitimate states of $C1$ and $C2$ be $I1$ and $I2$ respectively. Then, the wellknown and simple theorem about such composition is as follows:

Theorem 4.4.1. *If*

- $C1$ *is weakly-fair stabilizing for* $I1$
- $C2$ *is weakly-fair stabilizing for* $I2$

Then

- $C1[]C2$ *is weakly-fair stabilizing for* $I1 \wedge I2$*, where* $C1[]C2$ *is the program obtained by taking union of actions of* $C1$ *and* $C2$*.*

Although straightforward, this theorem can assist in reducing verification time if the fairness requirement is needed essentially to ensure that both components get a chance to execute. In other words, if the components themselves are self-stabilizing under unfair computations then the designer can verify the preconditions of this theorem easily under unfair computation. Self-stabilization under unfair computations implies self-stabilization under weak fairness. Hence, preconditions of the theorem can be proven easily. Moreover, the conclusion of the theorem allows us to ensure that the composed program is self-stabilizing under weak fairness.

There are several such theorems that provide the ability to conclude self-stabilization property of the composed program by self-stabilization property of the components. Another wellknown theorem relates to superposition where program p consists of two components $C1$ and $C2$, where $C1$ is superposed on $C2$. In other words, $C1$ can only read the variables of $C2$ and $C2$ can neither read nor write variables of $C1$. Then, the wellknown theorem about superposition is as follows:

Theorem 4.4.2. *If*

- $C1$ *is weakly-fair stabilizing for* $I1$
- $C2$ *is weakly-fair stabilizing for* $I2$
- *After a state in* $I2$ *is reached, no action in* $C2$ *is enabled*

Then

- $C1[]C2$ *is weakly-fair stabilizing for* $I1 \wedge I2$*, where* $C1[]C2$ *is the program obtained by taking union of actions of* $C1$ *and* $C2$*.*

Again, similar to the approach above, we may be able to verify each component without fairness assumption. However, fairness is required for the composed program to ensure that each component gets a chance to execute. Again, this will allow us to conclude correctness of the composed program under weak fairness by expediting the verification time for individual components.

There are several instances where such superposition or variations thereof are used. In particular, one variation is that it suffices if $C1$ ensure convergence to $I1$ by assuming that $I2$ holds already. Also, the third condition (termination of $C2$) can also be replaced by

other non-interference conditions that are less restrictive. Next, we discuss some of these examples.

4.4.2.1 Case Study 4: Huang's Mutual Exclusion in Uniform Rings

In [116], authors propose a self-stabilizing mutual exclusion program that consists of two components: (1) leader election component and (2) token circulation component. The first component consists of a leader election program on an oriented uniform ring where the number of nodes is prime. The second component consists of a token circulation component that requires a unique process (such as process 0 in Case Study 1.) Since verification of the second component is similar to that in Section 4.4.1.2, we only focus on the leader election component.

The leader election component maintains a variable $v.j$ at every process j. The actions of the processes are as follows ($left$ and $right$ denote the left and right neighbor of process j in the ring):

$$K_1:: \quad v.left = v.j = v.right \longrightarrow v.j = (v.j + 1)\ mod\ n;$$

$$K_2:: \quad gap^1(left, j) < gap(j, right) \longrightarrow v.j = (v.j + 1)\ mod\ n;$$

The above actions require that a node increments its value if either (1) its value equals that of its left and its right neighbor or (2) gap with the left node is less than the gap with the right node.

1

$$gab(a,b) = \begin{cases} n & \text{if } a = b \\ (b-a)\ mod\ n & \text{otherwise} \end{cases}$$

Execution time(s)				
	n=3	n=5	n=7	n=11
$unfair_{leader\ election}$	0	0	0.05	0.48
$unfair_{token\ circulation}$	0	0	0.02	0.15
$unfair_{total\ with\ decomposition}$	0	0	0.07	0.63
$weakly-fair_{leader\ election}$	0	4.15	N/A	N/A
$weakly-fair_{token\ circulation}$	0	0.63	-	-
$weakly-fair_{total\ with\ decomposition}$	0	4.79	N/A	N/A
$weakly-fair_{total\ without\ decomposition}$	-	-	-	-
$weakly-fair_{total\ without\ decomposition}$	0.17	N/A	N/A	N/A
approximate state space	10^4	10^{10}	10^{17}	10^{34}
Execution time(s)				
	n=23	n=29	n=31	-
$unfair_{leader\ election}$	47.12	271.05	704.48	-
$unfair_{token\ circulation}$	14.57	70.18	103.8	-
$unfair_{total\ with\ decomposition}$	61.69	341.23	808.28	-
$weakly-fair_{leader\ election}$	N/A	N/A	N/A	-
$weakly-fair_{token\ circulation}$	-	-	-	-
$weakly-fair_{total\ with\ decomposition}$	N/A	N/A	N/A	-
$weakly-fair_{total\ without\ decomposition}$	-	-	-	-
$weakly-fair_{total\ without\ decomposition}$	N/A	N/A	N/A	-
approximate state space	10^{93}	10^{127}	10^{138}	-

Table 4.7: Verification results for Huang's mutual exclusion program.

This program requires fairness for correctness. Without fairness, leader election component may not be able to execute. However, each component can be verified separately without fairness. Finally, based on Theorem 4.4.2, we can conclude that the overall program is self-stabilizing under weak fairness. Table 4.7 gives the verification performance by utilizing symbolic model checking procedure for verification of leader election component. From this table, we can see that verification of self-stabilization is significantly more scalable with decomposition and the use of unfair computation for verifying self-stabilization of each component. Moreover, the significant benefit in reduction in time is based on the use of unfair scheduler as opposed to the use of decomposition. (In Table 4.7, '-' denotes that the experiment was not performed for token circulation since the corresponding experiment for leader election could not be completed in the permissible time.)

4.4.2.2 Case Study 5: Self-stabilizing Program based on Raymond's Tree algorithm

This second example is the self-stabilizing program based on Raymond's tree algorithm for mutual exclusion [194]. In this program, the processes are arranged in a *fixed* [2] tree, called the parent tree. On this fixed tree, a dynamic *holder* tree is superposed such that the holder of a node is one of its tree neighbors (including itself). A node j has a token iff its holder $(h.j)$ equals j. There is one action that allows a node to send the token to its neighbors $(K_{passing})$. In this action, if node k has a token then it can pass it to its neighbor j by changing the holder relation of j and k. Additionally, there are three convergence actions. The first action ensures that the holder of a node is a tree neighbor. The second action

[2] By fixed, we mean that $p.j$ is fixed and hard coded in the actions themselves and, hence, cannot be corrupted.

ensures that on any edge between j and $(p.j)$, either holder of j is same as $p.j$ or the holder of $p.j$ is j. And, the third action ensures that holder relation does not have cycles. Thus, the actions of the self-stabilizing tree program are as shown next:

$$
\begin{aligned}
K_{passing}:: \quad & h.k = k \wedge h.j = k && \longrightarrow h.j = j, h.k = j; \\
K_{convergence}:: & h.j \neq NBR.j \cup j && \longrightarrow h.j = p.j; \\
& j \neq p.j \wedge h.j \neq p.j \wedge h.(p.j) \neq j && \longrightarrow h.j = p.j; \\
& j \neq p.j \wedge h.j = h.(p.j) \wedge h.(p.j) = j && \longrightarrow h.j = p.j;
\end{aligned}
$$

This program requires fairness for stabilization; without stabilization, nodes could simply execute the token passing action ($K_{passing}$) thereby preventing stabilization. However, correctness of the convergence actions and the correctness of closure actions can be independently verified without fairness. Furthermore, the results from [1] can be used to show that these two components do not interfere and, hence, the overall program is self-stabilizing. Table 4.18 gives the verification time for each component under unfair scheduler. It also gives verification time for convergence under weakly-fair scheduler. Since the token passing component changes the variables from two different nodes, we were not able to implement it under weakly-fair scheduler. However, it is straightforward to observe that the time for verification of the composed program (with token passing and convergence actions) will be more than the time for verification with convergence actions alone. Hence, the benefit of decomposition and use of unfair scheduler in reducing the cost of verification follows from the results in Table 4.18.

Execution time(s)				
	n=7	n=15	n=31	n=63
$unfair_{convergence}$	0.02	0.10	1.95	N/A
$unfair_{passing}$	0	0.10	39.97	N/A
$weakly - fair_{convergence}$	0.1	17.67	N/A	N/A
approximate state space	10^5	10^{17}	10^{46}	10^{113}

Table 4.8: Verification results for Raymond-tree based program.

4.4.2.3 Other Examples and Approaches for Identifying Components

Another example is that of distributed reset [18] where the program consists of a tree layer and a wave layer. The tree layer constructs a tree from the processes that are still *up*. Subsequently, the wave layer utilizes this tree to achieve distributed reset. Again, weak fairness ensures that each component can always execute although the component itself can be verified without fairness.

For the case where decomposition is not straightforward the proof of stabilization can assist in identifying the desired decomposition. Specifically, one common way to prove self-stabilization is to use the the approach of Gouda and Multari [93]. Specifically, in this approach, the state space itself is partitioned into concentric circles, R_0, R_1, ..., R_n, where R_0 corresponds to the entire state space, R_n corresponds to the set of legitimate states and $R_i \supset R_j$ if $0 < i < j < n$. It is required that if the program starts in any state in R_i, $0 \leq i < n$ then (1) it always stays in states in R_i, and (2) it eventually reaches a state in R_{i+1}. Again, fairness can assist in this approach in ensuring that the overall program is self-stabilizing although one or more convergence requirements can be verified without fairness thereby reducing the time for verification.

4.4.3 Utilizing Weak Stabilization to Improve Scalability of Model Checking of Self-stabilizing Program

In this section, we focus on the second approach for improving scalability for verification of self-stabilizing programs. Specifically, if the program at hand requires weakly-fair computations to provide self-stabilization and the time for such verification is prohibitive, the designer can focus on a variation of self-stabilization, namely weak self-stabilization [92], where weak stabilization has been demonstrated as a 'good approximation' of stabilization. Furthermore, in [58], Devismes et al have shown how to transform a weak-stabilizing program into a probabilistic stabilizing program. Thus, if the assurance of self-stabilization is not possible, the designer can obtain a slightly lower assurance provided by weak stabilization. Moreover, the designer can utilize the transformation in [58] to obtain probabilistic assurance regarding self-stabilization. Next, we recall the definition of weak stabilization and a relevant theorem about it from [92].

Definition 4.4.1. ***(Weak stabilization)***

Let p be a program and let I be a state predicate of p. We say that p is weakly stabilizing for I iff:

1. *closure: if (s_0, s_1) is a transition of p and $s_0 \in I$, then $s_1 \in I$;*
2. *weak convergence: for every state s, there is a computation of p that starts at s and reaches I.* □

Definition 4.4.2. ***(Strongly-fair computation)***

$\sigma = \langle s_0, s_1, ... \rangle$ is strongly-fair computation iff:

1. *σ is an unfair computation of p, and*

2. *If any state s is included infinitely often in σ and (s, s') is a transition of p then the subsequence $\langle s, s' \rangle$ must be included infinitely often in σ.* □

Theorem 4.4.3. *A weakly-stabilizing system is also a self-stabilizing system if:*

1. *The system has a finite number of states, and*

2. *Every computation is under strong fairness.*

We can utilize the above result as follows: If we cannot verify that p is self-stabilizing due to time/space limitations, we can verify that p is weakly stabilizing. By Definition 4.4.1, this does not require one to model fairness explicitly. This will allow us to obtain some assurance (although somewhat weaker) about p. Additionally, the designer can utilize the transformation from [58] to obtain program p' that is probabilistically stabilizing.

Next, we revisit case studies 1-5 to evaluate the cost of verifying weak stabilization. Tables 4.9-4.13 compare the cost of verifying self-stabilization under weak fairness with that of verifying weak stabilization. As we can see, the cost of verifying weak stabilization is substantially less (and is very close to the cost of verifying self-stabilization without fairness). Also, verification of weak stabilization is significantly more scalable than that of self-stabilization. For example, in Dijkstra's K-state program, it was possible to verify self-stabilization for only 9 processes (state space 10^8) whereas it was possible to verify weak stabilization for 50 processes (state space 10^{84}).

4.5 A Constraint-based Approach

The approach in Section 4.3 assumes fairness in the program model. And, modeling of a (weak) fairness scheduler (if essential) affects the verification performance significantly.

K-state Program			
Execution Time(s)			
Number of Processes	3	4	5
weak stabilization	0	0	0
stablization under weak fairness	0	0.03	0.63
approximate state space	10^1	10^2	10^3
Execution time(s)			
Number of Processes	6	7	8
weak stabilization	0	0.02	0.03
stablization under weak fairness	5.33	34.30	139.10
approximate state space	10^4	10^5	10^7
Execution time(s)			
Number of Processes	9	10	50
weak stabilization	0.05	0.08	3485.27
stablization under weak fairness	1276.08	N/A	N/A
approximate state space	10^8	10^{10}	10^{84}

Table 4.9: Verification cost of weak stabilization vs. stabilization (1).

Ghosh's mutual exclusion program			
Execution time(s)			
Number of Processes	8	10	12
weak stabilization	0	0	0
stablization under weak fairness	0.4	2.93	22.43
approximate state space	10^2	10^3	10^3
Execution time(s)			
Number of Processes	14	16	18
weak stabilization	0	0	0.02
stablization under weak fairness	138.05	693.27	2819.05
approximate state space	10^4	10^4	10^5
Execution time(s)			
Number of Processes	20	50	100
weak stabilization	0.03	0.35	4.9
stablization under weak fairness	N/A	N/A	N/A
approximate state space	10^6	10^{16}	10^{31}

Table 4.10: Verification cost of weak stabilization vs. stabilization (2).

Hoepman's ring-oritentation program			
Number of Processes	3	5	7
	Execution time(s)		
weak stabilization	0	0.037	0.08
stablization under weak fairness	0.17	18.23	1113.77
approximate state space	10^1	10^3	10^4
Number of Processes	9	51	101
	Execution time(s)		
weak stabilization	0.13	11.88	95.9
stablization under weak fairness	-	-	-
approximate state space	10^5	10^{31}	10^{60}
Number of Processes	201	301	-
	Execution time(s)		
weak stabilization	881.5	3442.18	-
stablization under weak fairness	-	-	-
approximate state space	10^{121}	10^{181}	10^{241}

Table 4.11: Verification cost of weak stabilization vs. stabilization (3).

Huang's ring program				
Execution time(s)				
	n=3	n=5	n=7	n=11
weak stabilization	0	0.07	0.63	N/A
stablization under weak fairness	0.17	N/A	N/A	N/A
approximate state space	10^4	10^{10}	10^{17}	10^{34}

Table 4.12: Verification cost of weak stabilization vs. stabilization (4).

Raymond-tree based mutual exclusion program				
Execution time(s)				
	n=7	n=15	n=31	n=63
weak stabilization	0	0.12	2.15	N/A
$weakly - fair_{convergence}$	0.1	17.67	N/A	N/A
approximate state space	10^{11}	10^{35}	10^{92}	10^{226}

Table 4.13: Verification cost of weak stabilization vs. stabilization (5).

Moreover, the symbolic model checking approach relies on an optimal variable ordering since this order could have a significant impact on the size of the BDD graph. Unfortunately, the problem of finding the best variable ordering is NP-hard. To address these issues, in this section, we consider a constraint-based approach to analyze stabilization. This approach does not require modeling fairness and does not assume any order information of variables. The key insight is to reduce the task of verifying whether a program is stabilizing into a well-studied problem: constraint solving, which can be solved by many existing highly optimized solutions. Specifically, our approach leverages the power of the off-the-shelf SMT solvers that have demonstrated the ability of solving industrial sized-satisfiability instances in the last decade.

4.5.1 Approach for Verifying Stabilization with SMT Solvers

In this section, we present an constraint approach of verifying self-stabilization properties. The key idea of this approach is to translate the problem into a formula and then to use SMT solvers to analyze the formula.

Verification of stabilization consists of two parts: (1) verification of *closure* and (2) verification of *convergence*. In Section 4.5.1.1, we identify the formula whose satisfiability can be used to determine whether closure property is satisfied. In Section 4.5.1.4, we identify an algorithm for verifying convergence by using the formulae developed in Sections 4.5.1.2 and 4.5.1.3.

4.5.1.1 Verification of Closure

Let $\mathcal{P}$ be the given program and let I be the legitimate state predicate used in Definition 4.2.2 to conclude that $\mathcal{P}$ is stabilizing. Let $\mathcal{T}$ be the predicate that characterizes transitions of $\mathcal{P}$. Observe that the closure property requires that if (s_0, s_1) is a transition of program $\mathcal{P}$ and state s_0 is a legitimate state then state s_1 is also a legitimate state. Thus, this can be captured by formula $\neg\Psi_l$, where

$$\Psi_l \quad = \quad (\mathcal{I}(s_0) \wedge \mathcal{T}(s_0, s_1) \wedge \neg\mathcal{I}(s_1))$$

Remark 4.5.1. *For compactness, the formula Ψ_l does not explicitly specify the program or the set of legitimate states that are inputs in deciding closure. In this dissertation, these two parameters can be determined based on the context. We use similar approach for other formulae as well.*

Based on whether Ψ_l is satisfiable or not, we have two scenarios, SC_1 and SC_2:

1. SC_1 : if Ψ_l is satisfiable then it proves that it is possible to begin in a legitimate state, execute a program transition and be in a state that is not a legitimate state. This implies that the closure property is not satisfied. Moreover, in this case, assignment to s_0 and s_1 (which in turn includes values of variables of the program in state s_0 and s_1) provides a counterexample.

2. SC_2 : if Ψ_l is unsatisfiable then this implies that the closure property is satisfied.

4.5.1.2 Verification of Convergence

To verify convergence, we use approach from bounded model checking [33]. We verify convergence by checking that starting from an arbitrary state, the program, say $\mathcal{P}$, reaches a

legitimate state (in I) in k steps, where k is a given parameter used in the verification. Observe that the convergence property requires us to consider a sequence of states, $s_0, s_1, \cdots, s_k$ such that each successive transitions are program transitions. Moreover, to verify (negation of) convergence requirement, we require that $\mathcal{I}(s_k)$ should be false. Additionally, in this verification, we can utilize the closure requirement to add additional constraints requiring that $\mathcal{I}(s_j)$, $0 \leq j \leq k$, should be false. Additionally, in bounded model checking, one typically adds constraint about what the initial state should be. However, in convergence, the initial state can be arbitrary and, hence, there is no corresponding constraint for the initial state. Thus, the formula used for verifying convergence is as follows:

$$\begin{aligned}\Psi_v = \; & \mathcal{T}(s_0, s_1) \wedge \mathcal{T}(s_1, s_2) \wedge \cdots \wedge \mathcal{T}(s_{k-1}, s_k) \\ & \neg\mathcal{I}(s_0) \wedge \neg\mathcal{I}(s_1) \wedge \cdots \wedge \neg\mathcal{I}(s_k)\end{aligned}$$

Based on whether Ψ_v is satisfiable or not, we have the following two scenarios:

1. SC_3 : if Ψ_v is satisfiable, convergence cannot be achieved in k steps. In this case, the number of steps needs to be increased. If the state space of the program is finite and k equals the number of states in the program then this implies that the convergence property is not satisfied. However, a simple cycle detection algorithm (discussed next) can be used to conclude that the program is not stabilizing for smaller values of k.

2. SC_4 : if Ψ_v is unsatisfiable, then it proves that even if we begin in an arbitrary state, it is impossible for the program to be in an illegitimate state if it executes for k steps. In other words, the convergence property is satisfied.

4.5.1.3 Resolving Ambiguity by Cycles Detection

As discussed in Section 4.5.1, when Ψ_v is satisfiable, either the given program is not stabilizing or the value of k is too small. To distinguish between these scenarios, we use an approach of resolving ambiguity by checking for an existence of a cycle outside legitimate states. The main idea of this approach is to check whether the given program can run into a cycle that is outside legitimate states.

To check whether the given program can execute a cycle that is outside legitimate states, we consider the behavior of the given program for k steps. Hence, we construct a formula similar to that of Ψ_v. Additionally, the computation created by Ψ_v creates a cycle iff some state is repeated in this path. This can be checked by adding another constraint that two of the states visited are identical. In other words, the added constraint is that there exists two states s_j and s_k, where $j < k$ and $s_j = s_k$.

Note that in case of stabilization, the initial state is arbitrary. Hence, if there exists a cycle where $s_j = s_k$ then there exists a suffix of the given computation that begins with s_j. In other words, it suffices to check whether state s_0 is repeated in the given computation. Hence, the formula used for detecting cycle is as follows:

$$\Psi_y = \ \Psi_v \wedge (s_0 \cong s_1) \vee (s_0 \cong s_2) \vee \cdots \vee (s_0 \cong s_k)$$

In the above formula, we use $(s_0 \cong s_1)$. One implementation of this is to require $s_0 = s_1$. However, in certain cases, it may be sufficient to reach a state that is *similar* to the initial state. Such similarity is application dependent. One example of this is based on symmetry of processes and values.

Based on whether Ψ_y is satisfiable or not, we have two scenarios: SC_5 and SC_6:

1. SC_5 : if Ψ_y is satisfiable then this implies that there is a computation of the given program that starts in state s_0 and revisits state s_0 without reaching a legitimate state in between. This implies that there is a possibility that the program may never reach a legitimate state. In other words, the given program is not stabilizing.

2. SC_6 : if Ψ_y is unsatisfiable then there are two possibilities: either the given program is stabilizing and, hence, such a cycle cannot exist or the number of steps is insufficient to create a cycle.

4.5.1.4 Combining Verification of Convergence and Cycle Detection

Depending upon the satisfiabilty of Ψ_v and Ψ_y, we have four possibilities. Considering these four possibilities, we can determine whether the given program is stabilizing or not. We illustrate this checking process by Algorithm presented in Figure 4.11. Line 1 first constructs formula $\Psi_v$and Ψ_y. Then, the algorithm utilizes bounded model checking techniques to check the satisfiability of the two formulas in the loop starting from Line 2 and ending at Line 12. If the condition identified in Line 3 is satisfied, the algorithm concludes that the program is stabilizing. If the condition identified in Line 6 is satisfied, the algorithm concludes that the program is not stabilizing. And if the condition identified in Line 9 is satisfied, the algorithm simply increases the value of k and repeat checking the conditions identified in Line 3, 6 and 9.

Note that the above algorithm begins with $k = 1$. However, a better approach is to begin with k to be the expected number of steps for convergence. Also, the algorithm can be tuned in terms of how k is increased. Note that for finite state program, the above program is guaranteed to terminate. For infinite state programs, however, it may not. This is expected

Algorithm 1: Stabilization Verification
Input: $\mathcal{P}$: program to be verified;
$\mathcal{I}$: set of legitimate states.

```
Construct Ψ_v and Ψ_y for P&I.                          (1)
For {k = 1 → ···}{                                      (2)
  IF {Ψ_v is unsatisfiable }                            (3)
      Print given program is stabilizing.               (4)
  ENDIF                                                 (5)
  IF {Ψ_y is satisfiable }                              (6)
      Print given program is not stabilizing.           (7)
  ENDIF                                                 (8)
   // Since Ψ_y ⇒ Ψ_l                                   (9)
   // it is impossible for Ψ_y to be satisfiable
   // and Ψ_l to be unsatisfiable.
  IF {Ψ_vis satisfiable and Ψ_y is unsatisfiable}       (10)
      increase the value of k                           (11)
  ENDIF                                                 (12)
} // endfor                                             (13)
```

Figure 4.11: The algorighm for determing stabilization.

given that the halting problem can be trivially reduced to verification of stabilization.

4.5.2 Experimental Results

In this section, we present our case studies. These case studies include K-state token ring program [59] and Ghosh's mutual exclusion program [87] and Stabilizing Tree based mutual exclusion [194]. We use the SMT solver Yices [67] to verify the stabilization property. We note that we have also used Z3 [52]. While the exact numbers associated with Z3 are different, the observations and conclusions from this section still hold. Hence, the results for Z3 are not presented in this dissertation.

Number of nodes	Number of steps for convergence	Execution time(s) for convergence	Execution time(s) for closure
3	4	0.0044	0.003928
4	14	0.01229	0.004257
5	25	0.209468	0.005399
6	39	154.1079279	0.004608

Table 4.14: Verification time for Ψ_v for token ring with unbounded variables.

4.5.2.1 K-State Token Ring Program

In this section, we illustrate the proposed constraint-based approach by studying k-state token ring program. Although the algorithm in Section 4.5.1.4 attempts different values of k to decide whether the program is stabilizing, in this section, we only focus on the value of k for which we can conclude that the program is stabilizing (respectively, not stabilizing). This is due to the fact that several heuristics (e.g., analysis of the program to evaluate expected number of steps) can be used to limit the values of k used in Section 4.5.1.4. Hence, we only focus on the value of k for which the algorithm terminates.

Remark 4.5.2. *We consider three variations of k-state token ring program: In the first variation, K is set to $N + 1$. In the second variation, value of x is unbounded and, hence, K_0 simply increments $x.0$. Finally, we consider the value of $K = 2$ in Section 4.5.3.*

We evaluate the performance of the token ring program for both bounded and unbounded setting. Tables 4.14 and 4.15 respectively illustrate the time for verifying the closure and the convergence property for the bounded and unbounded version of the token ring.

As we can observe, the verification time is significantly lower for unbounded version of the token ring. In particular, for the case where x values are unbounded, it is possible to verify the convergence property of a ring with 5 processes in less than a second. However, the corresponding time for program with bounded x value is 214 seconds.

Number of nodes	3	4	5
State space	10^1	10^2	10^3
Number of steps for convergence	4	14	25
Execution time(s) for convergence	0.008944	0.494496	214.0957
Execution time(s) for closure	0.005617	0.005979	0.013349

Table 4.15: Verification time for Ψ_v for token ring with bounded variables.

Number of nodes	3	4	5
State space	10^1	10^2	10^3
Number of steps for convergence	4	14	25
Execution time(s) for convergence	0.005855	0.090116	33.526028
Execution time(s) for closure	0.005387	0.006480	0.006716

Table 4.16: Verification time for Ψ_v for token ring with split actions for K_0.

One of the reasons for this is that the bounded version utilizes a modulo operation. One can attempt to revise the token ring program to simplify the mod operation to gain a substantial benefit. Specifically, Table 4.16 considers the case where action K_0 is split into two actions: The first action executes only if $x.0$ is not equal to $K-1$. And, it increments the value of $x.0$. The second action executes only if $x.0$ equals $K-1$. And, it resets $x.0$ to 0. With this change, the verification time for five processes reduces from 214 seconds to 33 seconds.

4.5.2.2 Ghosh's Binary Mutual Exclusion Protocol

Table 4.17 gives the performance results of Ghosh's program. The result demonstrates that the verification cost (along with number of steps necessary for convergence) increases

Number of nodes	8	10	12	14
State Space	10^2	10^3	10^3	10^4
Number of steps for convergence	9	16	25	36
Execution time(s) for convergence	0.040316	0.470958	11.166705	314.851055
Execution time(s) for closure	0.007433	0.012232	0.009348	0.015544

Table 4.17: Verification results for Ghosh's program using SMT solver.

Number of nodes	3	7	9
State space	10^1	10^5	10^8
Number of steps for convergence	5	8	9
Execution time(s) for convergence	0.007162	1.377709	125.633908
Execution time(s) for closure	0.005046	0.004943	0.005432

Table 4.18: Verification results for Raymond tree-based program.

substantially when the number of processes is more than 12.

4.5.2.3 Stabilizing Tolerant Version of Tree-based Mutual Exclusion Algorithm

In Table 4.18, we present the performance results for verifying the stabilizing tolerant version of tree-based mutual exclusion algorithm. The results demonstrate that the verification cost is very low when the number of processes is no more than 9. The verification cost (along with number of steps necessary for convergence) increases substantially when the number of processes is more than 9.

4.5.3 Verification of Token Ring in Synchronous Semantics

The computation model we considered in Definition 2.1.5 corresponds to interleaving semantics where in each step one of the actions is executed. Another computation model uses synchronous semantics. Here, the program actions are partitioned into groups and for every group there is a corresponding *process* responsible for executing those actions. Furthermore, in each step, every process executes one of its enabled actions (unless it has no enabled action in that state). Since the number of steps needed for convergence affects the verification time with SMT solvers significantly, in this section, we consider execution of the program in synchronous semantics and evaluate its effect on verification.

Verification of the program under synchronous semantics can assist in two scenarios: One scenario is that one can verify the stabilization property under synchronous semantics. In this case, the program is guaranteed to reach a legitimate state under synchronous semantics. One can utilize a program that is correct under synchronous semantics and compose it with the alternator in [138]. This alternator ensures that given a program that is stabilizing under synchronous semantics, it transforms into a program that is correct under read/write model. Specifically, in this case, the process in the transformed program either reads the state of its neighbor or writes its own state. In other words, the transformed program guarantees that it will reach a legitimate state even if one process executes at a time, i.e., the transformed program is stabilizing under interleaving semantics. Hence, in this scenario, we can obtain a program that is stabilizing. Another scenario is that one can identify a counterexample (illustrating the lack of stabilization) for the synchronous program. This counterexample can in turn be transformed into a counterexample for the original program.

We begin with the first scenario, where using synchronous semantics can improve verifi-

cation performance. We consider the execution of the token ring protocol under synchronous semantics. Results from Table 4.5.3 show the verification cost under synchronous semantics. These results show that the verification under synchronous semantics is substantially faster for both bounded and unbounded token ring. As discussed above, this can allow us to obtain a stabilizing program that ensures that a legitimate state is reached even if only one process executes at a time.

Numbers of Nodes	Number of Steps for Convergence	Execution time(s) for Convergence (for Bounded Variables)	Execution time(s) for Convergence (for UnBounded Variables)
3	3	0.006714	0.005945
4	5	0.049422	0.008152
5	7	0.306879	0.012003
6	9	1.150117	0.018864
7	11	11.837923	0.030548
8	13	7.741610	0.049720
9	15	18.389602	0.077605
10	17	42.7424	0.130185
11	19	789.75117	0.241754
12	21	324.921817	0.907359
17	23	N/A	22.63948

Table 4.19: Verification results for token ring under synchronous semantics.

To illustrate the second scenario, we consider the case where the given program works in read/write model where in each step, a process can read the state of its neighbor or write its own state but not both. In such a program, the variables can be partitioned into a public variables (variables that can be read by more than one process) and private variables (variables that can be read by only one process. Thus, in read/write model, the read action corresponds to the case where a process reads public variable(s) of one of its neighbors and saves a copy of it in its private variable(s). In write action, the process utilizes its own public/private variables to update them.

Observe that if $\mathcal{P}$ is a program in read/write model then for each computation of $\mathcal{P}$ in synchronous semantics there is a corresponding computation in interleaving semantics. Intuitively, in the computation in interleaving semantics, one transition of the program in synchronous semantics is split into several steps. Hence, if we can find a counterexample to show that $\mathcal{P}$ is not stabilizing under synchronous semantics then it implies that $\mathcal{P}$ is not stabilizing under interleaving semantics either.

To exploit this observation, we consider the execution of the token ring protocol under read/write model. In this case, $x.j$ is a public variable of process j. The action K_0 in Section 4.5.2.1 is not in read/write model since it reads $x.N$ and updates $x.0$. Read/write model requires that these two tasks be separated into one read action (of $x.N$) and one write action (of $x.0$). To obtain the corresponding program in read/write model, we introduce $y.j$ that maintains a copy of the x value of the predecessor. Furthermore, each action is split into a read action to read the value of the predecessor and a write action that utilizes the local copy. Thus, the actions of the token ring protocol in read/write model are as follows:

$$
\begin{array}{lllll}
K_{0_r}:: & y.0 \neq x.N & \longrightarrow & y.0 = x.N; & // \text{ read } x.N \\
K_{0_w}:: & x.0 = y.0 & \longrightarrow & x.0 = (x.0 + 1) \ mod \ K; & \\
K_{j_r}:: & y.j \neq x.(j-1) & \longrightarrow & y.j = x.(j-1); & //\text{read } x.(j-1) \\
K_{j_w}:: & x.j \neq y.j & \longrightarrow & x.j = y.j; &
\end{array}
$$

It is well-known that the above protocol can be thought of as the original token ring protocol with $2(N+1)$ processes, where the variables of these processes are $x.0$, $y.1$, $x.1$, $y.2$, $\cdots$, $y.N$, $x.N$, $y.0$.

Now, we consider the execution of the token ring protocol with N processes under inter-

	Under Interleaving Semantics		Under Synchronous Semantics	
Numbers of Nodes	Numbers of Steps	Execution time(s)	Numbers of Steps	Execution time(s)
10	20	0.253376	4	0.009663
20	40	100.605236	8	0.048594
30	N/A	N/A	4	0.040200
50	N/A	N/A	4	0.103919
100	N/A	N/A	8	0.849282
200	N/A	N/A	16	9.9811778

Table 4.20: Verification result for cycle detection.

leaving semantics. If we choose $K = 2$ this program is not stabilizing. We can identify this by checking that Ψ_y is satisfiable when it is used in the context of the token ring program with N processes using interleaving semantics. Alternatively, the lack of stabilization can also be proved by considering execution of the token ring program with $2N$ processes under synchronous semantics.

With this intuition, we evaluate the time for verifying satisfiability of Ψ_y for different processes. Table 4.20 shows the verification time with interleaving and synchronous semantics respectively. We observe that for 20 processes under interleaving semantics, it took 40 steps to detect a cycle and the time was 100.605236 seconds. However, the same property can also be verified under synchronous semantics with 30 processes in 4 steps and the time was 0.040200 seconds. Moreover, as discussed above, the latter verification suffices to conclude that the 20 process token ring program is not stabilizing under interleaving semantics if $K = 2$.

4.6 Summary

This chapter focused on automatic verification of self-stabilization. One key challenge is raised by the prohibitive cost incurred during verification. Contrary to traditional verification

that considers only a subset of reachable states starting from some initial state(s), verification of stabilization must consider all possible states, leading to the fundamental problem of 'state explosion. To address this challenge, we leveraged BDD-based symbolic model checking to overcome the scalability constraint of existing approaches. Also, a novel technique was proposed to analyze program actions against stabilization, which reduces the verification problem into a simpler problem called constraint solving, allowing the use of existing highly optimized solutions to reduce verification cost.

In particular, in Section 4.4, we focused on using symbolic model checking for verifying self-stabilizing algorithms. While a significant percentage of the literature on self-stabilization routinely assumes weak fairness, where if an action is continuously enabled, it is guaranteed to be executed, we argued that verification under such weak fairness is not scalable. Our observation was that in many cases, the assumption of weak fairness is superfluous. And, in these cases, scalable verification of self-stabilization is possible under unfair computation model. We illustrated this in the context of three case studies, Dijkstra's K-state program, Ghosh's mutual exclusion program and Hoepman's ring-orientation program. In particular, we showed that the time for verification with unfair computations is approximately $0.001\% - 0.1\%$ of that for weakly-fair computations.

For the case where program cannot preserve self-stabilization property under unfair computations, we proposed two approaches, including *decomposition* and using *weak stabilization*. The first approach is to decompose the program into parts where each part can be verified under unfair computation. Subsequently, composition of these parts can be proved to be correct using existing theorems in the literature. We showed how this approach can be used in the context of two case studies, Huang's mutual exclusion program and self-stabilizing

mutual exclusion program based on Raymond's tree algorithm. In both case studies, scalability of verification increased substantially (e.g., from 10^4 states to 10^{138} states for Huang's mutual exclusion algorithm.) Also, the approach in [93] can be used to identify layers that assist in self-stabilization. These layers, in turn, can form the components that one can verify independently. Since most of reduction in time is obtained by the use of unfair scheduler, one can obtain the savings in this manner even if the components themselves are not substantially smaller than the original program.

The second approach is to utilize weak stabilization that has been proved to be a reasonable implementation of stabilization [92]. We also showed that verification of weak stabilization is substantially more scalable. This validates the *suggestion* in [92] that weak stabilization is easier to verify than self-stabilization. However, this suggestion is only valid for the scenario where weak fairness is necessary for the correctness of stabilization. The time cost for stabilization verification under unfair computation is essentially equal to that for weak stabilization when unfair computation is possible. Furthermore, a weak stabilizing program can be transformed into a probabilistically stabilizing program thereby providing additional assurance to designer.

In Section 4.5, we proposed a constraint-based approach for verifying stabilizing programs. In particular, we investigated the effectiveness of SMT solvers in verification of stabilization. We found that the effectiveness of SMT solvers in this context is mixed. Specifically, compared with the approach that utilize symbolic model checkers to verify stabilization, the time for verification is larger with SMT solvers. However, BDD based tools require one to identify the order of program variables for constructing BDDs. An incorrect ordering of variables can increase the verification time by orders of magnitude making it

significantly worse than the corresponding verification time with SMT solvers. Also, the approach in Section 4.4 only works for verifying finite state programs. By contrast, our constraint-based approach demonstrated the feasibility of verifying infinite state program.

We also considered execution of the given program under synchronous semantics. We argued that this has a potential to reduce the cost of verification and utilize a transformation approach to achieve a program that is stabilizing under interleaving semantics and/or read/write model. We showed that execution under synchronous semantics can reduce the time for identifying a counterexample illustrating that the given program is not stabilizing.

Chapter 5

Automatic Revision for Adding Weak Multitolerance

Intuitively, a *weak* multitolerant program assumes that faults from multiple classes will not occur simultaneously (By simultaneous, we mean that a fault from one class occurs before the system has recovered from a fault from another class). Thus, while the program tolerates multiple classes of faults, it tolerates them 'one at a time'. It follows that if the program is subject to fault classes f_1, f_2, ... fn and the program is perturbed by f_i, $(0 \leq i \leq n)$ then the program provides the desired level of fault-tolerance to f_i. However, if faults from f_j $(0 \leq j \leq n, j \neq i)$ occur while the program is recovering from fault class f_i then the program may not guarantee fault-tolerance. Thus, a *weak* multitolerant program that tolerates fault classes $f_1,\ f_2,\ \ldots\ fn$ provides fault-tolerance to each f_i $(0 \leq i \leq n)$ respectively.

5.1 Problem Statement

In this section, we first present the definition of *weak* multitolerance. As mentioned in Section 2.3.1, a fault-tolerant program guarantees a desired level of fault-tolerance (i.e., failsafe/nonmasking/masking) in the presence of a specific class of faults. Now, we consider the case where the program is subject to faults from multiple fault-classes.

Definition 5.1.1. *Let* $f_\delta = \{\langle f_i, l_i\rangle \mid 0 < i \leq n, l_i \in \{failsafe,\ nonmasking,\ masking\}\}$ *where* $n \geq 0$. *Program* $\mathcal{P}$ *is* ***weak multitolerant*** *to fault set* f_δ *from* S *for spec iff the following conditions hold:*

1. *(In the* absence *of faults)* $\mathcal{P} \models_S$ spec.

2. *For each* i, $0 < i \leq n$, $\mathcal{P}$ *is* l_i f_i*-tolerant from* S *for* spec *respectively.* □

Remark 5.1.1. *Whenever the level of fault tolerance to a given fault class is clear from the context, for brevity, we omit it.* □

We note that the definition of *weak* multitolerance can also be used to define *intermediate* multitolerance. For example, consider the case where masking fault-tolerance is required for both f_1 and f_2. However, if f_2 occurs while the program is recovering from f_1 then nonmasking fault-tolerance is provided. For this case, we can model such requirements by letting $f_\delta = \{\langle f_1, masking\rangle, \langle f_2, masking\rangle, \langle f_1 \cup f_2, nonmasking\rangle\}$.

Now, using the definition of *weak* multitolerant programs, we identify the requirements of the problem of synthesizing a *weak* multitolerant program $\mathcal{P}'$ with invariant S' from its fault-intolerant version $\mathcal{P}$ with invariant S. We require that $\mathcal{P}'$ only adds *weak* multitolerance and introduces no new behaviors in the *absence* of faults. This problem statement is a natural extension of the problem statement in [135] where fault-tolerance is added to a single class

of faults. More specifically, we stipulate the following two conditions: (1) $S' \subseteq S$, i.e., the invariant S' of the *weak* multitolerant program $\mathcal{P}'$ is a subset of the invariant S of the given program $\mathcal{P}$; (2) $(s_0, s_1) \in \mathcal{P}' \wedge s_0 \in S' \Rightarrow (s_0, s_1) \in \mathcal{P}$. Thus, the problem of *weak* multitolerance synthesis is as follows:

Problem 5.1.1. ***The Weak Multitolerance Synthesis Problem.***

Given $\mathcal{P}$, S, spec and f_δ: Identify $\mathcal{P}'$ and S' such that

- *(C1)* $S' \subseteq S$,
- *(C2)* $(s_0, s_1) \in \mathcal{P}' \wedge s_0 \in S' \Rightarrow (s_0, s_1) \in \mathcal{P}$, *and*
- *(C3)* $\mathcal{P}'$ *is* weak *multitolerant to* f_δ *from* S' *for spec.* □

We state the corresponding decision problem as follows:

Problem 5.1.2. ***The Weak Multitolerance Decision Problem.***

Given $\mathcal{P}$, S, spec and f_δ: Does there exist a program $\mathcal{P}'$, with its invariant S' that satisfies the requirements of Problem 5.1.1? □

5.2 Illustrating Examples

In this section, we present examples to illustrate the scenarios where *weak* multitolerance is used and where *weak* multitolerance is essential for feasibility of solution and/or performance. We present four examples. The first example is a two-sensor agreement protocol that has the property of failsafe-failsafe *weak* multitolerance. The second example is a leader election protocol that provides masking-masking *weak* multitolerance. The third example is a vertex coloring protocol where failsafe-nonmasking *weak* multitolerance is guaranteed. Finally, the

fourth example provides masking-masking *weak* multitolerance and illustrates a case where *strong* multitolerance is impossible.

We use the non-deterministic guarded commands language to represent programs. A *guarded command* (a.k.a. *action*) is of the form $g \longrightarrow st$ where g identifies constraints under which the guarded command can be executed and st identifies the effect of such execution. Thus, guarded commands are suitable for programs that are 'event-based' where g identifies the event and st identifies the action taken by the program while responding to the event. Guarded commands are also suitable for 'time-based' programs where g corresponds to a clock tick and st denotes the action taken by the program for that clock tick. A guarded command $g \longrightarrow st$ is a compact representation for transitions $\{(s_0, s_1)|g$ is true in s_0, s_1 is obtained by atomic execution of st from $s_0\}$.

5.2.1 Failsafe-Failsafe *Weak* Multitolerance

In this section, we present a protocol that consists of two sensors. Each sensor is used to detect the status of the environment. The program outputs the detected status of the environment. To model this protocol, we introduce four variables, in, $sensor0$, $sensor1$ and out. The details are as follows:

1. in denotes the real status of the environment. The domain of in in this model is $\{0, 1\}$.

2. $sensor0$ denotes the value detected by the sensor 0. The domain of $sensor0$ is $\{0, 1\}$. The value of $sensor0$ should be the same as the value of in when no fault occurs, that is, the value detected by the sensor 0 should reflect the real status of the environment.

3. $sensor1$ denotes the value detected by the sensor 1. The domain of $sensor1$ is $\{0, 1\}$. Again, the value of $sensor1$ should be the same as the value of in when no fault occurs.

4. *out* denotes the output of the system. The domain of *out* is $\{0, 1, \perp\}$, where $\perp$ represents that the system has not made a decision yet. If the system makes a decision, the output of the system should be the same as the value of *in* when no faults occur in the sensors.

The legitimate states of the program are those where values of two sensors are equal to the value of the real status in the environment and the output is equal to the value of the real status in the environment if the output has been decided. Hence, the invariant is as follows:

$$Inv_{ff} = (sensor0 == sensor1 == in) \wedge (out == \perp \vee out == in)$$

The program includes the following action: when no output has been made by the program, i.e., $out = \perp$ and both $sensor0$ and $sensor1$ have the same values, the output will be set to the value of $sensor0$.

$$out == \perp \wedge sensor0 == sensor1 \longrightarrow out := sensor0;$$

Although this example models the values of two sensors in a similar fashion, they could be providing sensor readings based on different approaches. For example, the altitude switch controller in [145] utilizes two types of altimeters, a digital and an analog. One of the reasons for choosing different types of sensing modalities is to ensure that one root cause does not affect all sensors. Thus, although faults that affect each sensor appear similar, they are considered as two different types of faults since it is expected that both faults do not occur in the same computation. In this program, we consider two classes of faults: a transient fault

that occurs at sensor 0 and a transient fault that occurs at sensor 1. When this transient fault occurs at a sensor, it changes its value to the opposite of that of the input. Thus, the fault actions for this program are as follows:

Action of Fault Class 1: $true \longrightarrow sensor0 := 1 - in$;

Action of Fault Class 2: $true \longrightarrow sensor1 := 1 - in$;

The safety specification requires the program never reaches a state that the output of the system is not the same as the real status of the environment. Formally, the safety specification states that the program should never reach a state where the following formula is true:

$$spec_{ff} = (out == 1 \wedge in == 0) \vee (in == 1 \wedge out == 0).$$

The program provides failsafe-failsafe *weak* multitolerance, which is defined as follows.

1. When no fault occurs, the values of the two sensors $sensor0$ and $sensor1$ are equal and the output is set to the value of $sensor0$. The output of the system is equal to the value of real status in the environment.

2. When transient faults corrupt one sensor, failsafe fault-tolerance should be provided. Since the faults corrupt the value of one sensor, the two values of $sensor0$ and $sensor1$ are not equal and, hence, the program action is not executed. That is, the safety specification is not violated.

3. When both faults occur simultaneously, no guarantee is provided.

The case discussed above could be extended to be the one which has three or more sensors. The agreement is reached by the majority of sensor values. In that case, the program will have the masking-masking *weak* multitolerance property. When transient fault corrupts one of the sensors, the majority agreement will guarantee the final decision will reflect the real status of the environment.

Although the two fault classes considered here are similar, they are considered separately due to the source of these faults. The next example illustrates the case where the faults in different fault classes have different effects.

5.2.2 Masking-Masking *Weak* Multitolerance

In this section, we present a leader election program that provides masking-masking *weak* multitolerance to two types of faults f_{m1} and f_{m2}. The fault-type f_{m1} manifests itself as if the leader leaves, called the *leader leave fault*, and f_{m2} denotes the message loss fault.

The program consists of n processes $(p_1, p_2, \ldots, p_n)$ that are organized in a connected network. Each process has an unique ID, numbered from $0, \ldots, n-1$. One of these processes is selected as the leader. When no faults occur, there are no actions that are executed. However, when the current leader leaves, other processes can compete to be the leader. When a process wants to be the leader, it starts a diffusing computation [61] and declares itself to be the leader when the diffusing computation completes successfully. If multiple processes start diffusing computations, then the process with higher ID wins.

A process initiates a diffusing computation when it receives a $\langle leave \rangle$ message from the current leader and if it is not already participating in a diffusing computation. To initiate a diffusing computation, process j sends a message $\langle j \rangle$ to all its neighbors. It also sets its own

root value and its parent to be equal to j. The root value keeps track of the initiator of a diffusing computation and the parent value keeps track of the node that sent this diffusing computation to j.

When process j receives a diffusing computation message of the form $\langle ID \rangle$ from k, it does the following: If j is already participating in a diffusing computation such that the initiator of that diffusing computation (stored in $root.j$) is higher than ID then j ignores the new message. If the value of $root.j$ is equal to ID then j is receiving the same diffusing computation twice. Hence, it only replies to k. Otherwise, it forwards this diffusing computation to all its neighbors. To do so, it forwards the message $\langle ID \rangle$ and sets its own parent variable to k and its own root variable to ID.

Finally, when j receives a reply message from all neighbors except the parent, j sends a reply to its parent. Moreover, if j is the initiator ($p.j == j$) then it declares itself to be the leader. Thus, the actions in this program are as follows:

upon receiving $\langle leave \rangle$ message from departing leader

$\longrightarrow$//node j sends diffusing computation message

if $root.j == -1$

send $\langle j \rangle$ to $nbrs.j$,

$p.j := j$,

$root.j := j$;

else //ignore

upon receiving $\langle ID \rangle$ from k

$\longrightarrow$if $root == ID$ //duplicate

send reply to k with $\langle ID \rangle$;

else if $ID > root.j$

$p.j := k$,

$root.j := ID$,

send $\langle ID \rangle$ to all neighbors except k;

else //ignore;

upon receiving reply from all neighbours except $p.j$ with $\langle root.j \rangle$

$\longrightarrow root.j := -1$;

if $p.j \neq j$

send reply to $p.j$ with $\langle root.j \rangle$;

else

$leader.j := true$;

The legitimate states of the program include those states where there is a unique leader, there are no ongoing diffusing computations to elect a leader, and, the root value of every process is -1.

The program is subject to two classes of faults, f_{m1} and f_{m2}. f_{m1} is the one where the leader node leaves. We assume that when the leader node leaves the network, it notifies its neighbors. f_{m2} causes messages to be lost. Variable $channel_{i,j}$ denotes the sequence of messages in the channel between i and j. Hence, the two types of fault actions are as follows:

f_{m1} (Leader node leave):

$leader.j == true$

$\longrightarrow$send $\langle leave \rangle$ to $nbrs.j$, $leader.j := false$;

f_{m2} (Message loss):

$channel_{i,j} \neq \langle\ \rangle$ // $\langle\ \rangle$ denotes an empty channel

$\longrightarrow\ channel_{i,j} := tail(channel_{i,j})$;

The safety specification requires that in any state there is at most one leader. More precisely, the program should never reach a state where:

$spec_{mm} = (\exists j, k : j \neq k : leader.j \wedge leader.k)$

The program provides maksing-masking *weak* multitolerance to *leader node leave* and *message loss*. More specifically, it is defined as follows.

1. In the absence of faults, no action is executed. And, there is an unique leader in the network.

2. When fault f_{m1} occurs, i.e., the current leader leaves, one or more of its neighbors initiate a diffusing computation. Among the nodes that initiate the diffusing computation, the one with highest ID is elected as the leader. Hence, masking fault-tolerance is guaranteed when f_{m1} occurs.

3. When fault f_{m2} causes messages to be lost, there is no effect on the number of leaders. Thus, there is a unique leader in the network. Hence, masking fault tolerance is guaranteed when f_{m2} occurs.

4. If fault f_{m1} and f_{m2} occur simultaneously, no fault-tolerance is guaranteed. Specifically, if a message is lost during the diffusing computation, it is possible that the diffusing computation never completes, and hence, no leader is elected.

A more careful analysis of this program shows that if faults f_{m1} and f_{m2} occur together, it causes the diffusing computation to be blocked thereby resulting in states where there is no leader. In this case, failsafe fault-tolerance is provided. Thus, this is also an instance of FM *weak* multitolerance.

Moreover, as one can imagine it is possible to design a fault-tolerant program that provides masking fault-tolerance to both f_{m1} and f_{m2} simultaneously. However, providing such tolerance is expensive. For example, it requires mechanisms to detect message losses (or potentially failure of a node). Additionally, it requires an overhead in terms of message retransmission etc. Also, if such faults are considered during diffusing computation, there is a need for keeping track of multiple diffusing computations initiated by the same node, e.g., with the use of sequence numbers. Thus, this example illustrates the case where a stronger tolerance property is possible although *weak* multitolerance can provide a slightly lower assurance at a reduced cost (in terms of complexity of the code, performance, etc.). The next example illustrates the case where providing *weak* multitolerance is essential.

5.2.3 Failsafe-Nonmasking *Weak* Multitolerance

In this section, we present a vertex coloring protocol that provides failsafe-nonmasking *weak* multitolerance to Byzantine fault and transient fault, i.e., failsafe fault-tolerance when Byzantine fault occurs, nonmasking fault-tolerance when transient fault occurs and no guarantees when both faults occur simultaneously.

The vertex coloring of the program is an assignment of colors to each process of the system. The goal of the program is that every process is assigned a color and no two neighboring processes are assigned the same color. One assumption is that the degree of each process is at most d and $d+1$ colors are to be used.

The legitimate states of the program are those whose colors are assigned appropriately.

The program action for each node j is defined as follows.

$$color.j == color.k \quad \longrightarrow \quad color.j := available\_color(j) \ ;$$

In the above action, $color.j$ denotes the color assigned to process j. $available\_color(j)$ is used to denote the method that finds a new color among the available ones and assigns it to process j. If the color of any process j is the same as one of its neighbors, the above action is enabled and the color of the process will be assigned another color that does not conflict with its neighbors. No action is executed in the absence of faults.

In this program, we consider two types of faults: (1) Byzantine faults and (2) transient faults. Both these faults result in changing the color of the affected process. However, the main difference between these faults is that the former is a permanent fault, i.e., the affected process can change the color as often as it desires whereas the latter is a transient fault where there is a bound on the number of times the color of some process is affected. Another difference is that the former only affects a subset of (chosen) processes whereas the latter can affect all processes at once. To model these faults, we introduce a variable $b.j$ that denotes whether j is Byzantine, variable $count.j$ that denotes the number of times $color.j$ is affected by transient faults, and MAX that denotes the number of permitted transient faults. (Note that all these variables are auxiliary variables, i.e., variables used in the proof

but not explicitly by the program itself.) Thus, the fault actions are as follows:

f_f (Byzantine fault):

$b.j == true$

$\longrightarrow color_j$:= random(0,d); // return a random value from 0 to d;

f_n (transient fault):

$count.j < MAX$

$\longrightarrow color.j$:= random(0,d);

The above fault actions may corrupt the color of a process to be the same as that of one of its neighbors.

The safety specification requires that, any two neighboring nodes that are non-Byzantine have different colors. Thus, the program should never reach a state in the state predicate $spec_{fn}$, where

$$spec_{fn} = (\exists j, k :: (k \in nbrs.j) \wedge (b.j == false) \wedge (b.k == false) \wedge (color.j == color.k))$$

The program provides failsafe and nonmasking *weak* multitolerance to Byzantine fault and transient fault, respectively. Specifically, it is defined as follows.

1. In the absence of faults, there is no action. Thus, the program keeps a correct color assignment to all nodes.

2. When f_f corrupts color assignment of the Byzantine node, no non-Byzantine node is

affected and hence the safety specification is not violated. Thus, failsafe fault-tolerance is provided.

3. When f_n causes one node to change its color transiently, safety specification may be violated at that time. Then, the recovery action will reassign its color to be the correct one and finally the program will recover to a correct assignment to all the nodes. Thus, nonmasking fault-tolerance is provided.

4. When both faults f_f and f_n occur simultaneously, safety specification may be violated due to the occurrence of transient fault. Thus, failsafe fault-tolerance is not guaranteed. Also the program may not recover to a correct assignment to all nodes since the Byzantine node is corrupted permanently. Hence, nonmasking fault tolerance is also not guaranteed.

This example illustrates the need for *weak* multitolerance. In particular, it is not possible to provide a uniform tolerance to both types of faults. Specifically, with Byzantine faults, the faults can prevent the program to recover to legitimate state where the colors of all nodes are assigned properly, i.e., no two neighboring nodes have the same color. Hence, in this example, providing masking or nonmasking fault-tolerance to Byzantine faults is impossible. Likewise, execution of the transient fault itself can violate the safety specification. Thus, providing failsafe or masking fault-tolerance to transient faults is impossible. For this reason, the only possible solution is to provide failsafe fault-tolerance to Byzantine faults and nonmasking fault-tolerance to transient faults.

5.2.4 Masking-Masking *Weak* Multitolerance

This section presents an example scenario where MM weak multitolerance is the best possible option for designers due to the impossibility of providing MM strong multitolerance. Intuitively, strong MM multitolerance guarantees that if both faults occur in the same computation (i.e., simultaneously), then masking fault tolerance to both of them will be provided. (Please see Definition 5.8.1 in Section 5.8 for a more precise definition of strong multitolerance.)

The Agreement Program (AP) includes a *general* process and three *non-general* processes. Initially, all non-generals are undecided. The general casts a decision and each non-general copies the decision of the general and terminates; i.e., *finalizes* its decision.

The AP program [141] has to satisfy two safety properties, namely *agreement* and *validity*. Agreement requires that if the general is faulty, then all non-faulty non-generals that have finalized agree on the same decision. Validity stipulates that if the general is not faulty, then any non-faulty non-general that has finalized has the same decision as that of the general.

The AP program is subject to two classes of faults: Byzantine and fail-stop. The Byzantine faults can perturb the state of *at most* one process and make it behave arbitrarily. That is, if a process is affected by Byzantine faults (i.e., a process is Byzantine), then it can cast different decisions lying about its decision to different processes. The fail-stop faults could cause a process to crash in a detectable fashion. A fail-stopped process does not execute any actions once it crashes. We assume that fail-stop faults only affect the non-general processes.

The set of program variables is $\{d_g, d_0, f_0, up_0, d_1, f_1, up_1, d_2, f_2, up_2\}$, where (i) d_g denotes the decision of the general, which cou1d be 0 or 1, and d_i represents the decision of process i ($0 \leq i \leq 2$), where the domain of d_i is $\{0, 1, \perp\}$, and $\perp$ means that process i is

undecided; (ii) f_i is a Boolean variable representing whether or not process i has *finalized* its decision after copying a decision from the general, and (iii) up_i is also a Boolean variable that denotes whether process i has crashed in a detectable fashion; i.e., has fail-stopped. The actions of process i in the AP program are as follows ($\oplus$ denotes addition in modulo 3):

$$
\begin{aligned}
A_{i1}: \; & d_i = \bot \;\wedge\; \neg f_i \;\wedge\; up_i \\
& \qquad \longrightarrow \quad d_i := d_g \\
A_{i2}: \; & (d_i \neq \bot \;\wedge\; \neg f_i) \;\wedge\; (d_{i\oplus 1} = \bot \vee d_i = d_{i\oplus 1}) \wedge \\
& (d_{i\oplus 2} = \bot \vee d_i = d_{i\oplus 2}) \;\wedge\; (d_{i\oplus 1} \neq \bot \vee d_{i\oplus 2} \neq \bot) \;\wedge\; up_i \\
& \qquad \longrightarrow \quad f_i := true \\
A_{i3}: \; & (d_i \neq \bot \;\wedge\; \neg f_i) \;\wedge\; (d_{i\oplus 1} \neq \bot) \wedge (d_{i\oplus 2} \neq \bot) \;\wedge up_i \\
& \qquad \longrightarrow \quad d_i := majority(d_1, d_2, d_3)) \\
& \qquad\qquad\quad f_i := true
\end{aligned}
$$

If process i is undecided and not crashed, then it can copy the decision of the general (see action A_{i1}). Once decided, process i can finalize its decision if at least another non-general has made the same decision (action A_{i2}). If all non-generals have copied a decision from the general, then process i can finalize by setting d_i to the majority of decisions (action A_{i3}). That is, if d_i differs from the majority, then process i corrects d_i by setting it to the majority of decisions and finalizing. Otherwise, d_i is equal to the majority and action A_{i3} finalizes the decision of process i. Notice that a crashed process can execute none of its actions because up_i becomes *false*.

An invariant of the AP program includes states in which validity and agreement are satisfied and at most one process is faulty.

The AP program provides masking-masking weak multitolerance, including the following properties:

1. In the absence of Byzantine and fail-stop faults, the program AP satisfies both validity and agreement.

2. In the presence of Byzantine faults, if the general is Byzantine then validity vacuously holds, and agreement is guaranteed by the majority of decisions. If a non-general has become Byzantine, then validity holds for non-faulty non-generals, and agreement is vacuously satisfied. Therefore, the AP program is masking fault-tolerant to Byzantine faults.

3. In the presence of fail-stop faults, one of the non-generals stops executing (i.e., its *up* variable becomes false). Thus, the other non-generals satisfy validity. Agreement is satisfied as well since a majority of non-generals exists and the general is not faulty. Therefore, the AP program is masking fault-tolerant to fail-stop faults.

4. When both faults occur, the program may reach a state where a non-general has crashed and another non-general has become Byzantine. If the fail-stop faults have occurred before a process makes a decision, then the guard of action A_{i3} in the other two processes is false. Since another non-general has become Byzantine, its decision may not be the same as the decision of the non-faulty non-general. Thus, the guard of action A_{i2} of the non-faulty non-general is also false. Therefore, the entire program deadlocks, i.e., masking fault tolerance cannot be guaranteed for Byzantine and fail-stop faults.

This example demonstrates a case where it is impossible to design a MM strong multitol-

erant program, and designing MM weak multitolerance is the next best option. The reason behind it is that the simultaneous occurrence of both faults may get the AP program to a state where no majority of decisions exists amongst the non-faulty processes. One approach for enabling recovery from such a state is to add redundancy by including an additional non-general process in the AP program, which may not be always feasible. Nonetheless, before resorting to redundancy, we provide the following options for designers. First, one can design a MM weak multitolerant program (similar to the AP program) where masking fault tolerance is guaranteed if the faults occur one at a time. The second option is a FM weak multitolerant program, where masking fault tolerance to Byzantine faults and failsafe fault tolerance to fail-stop faults is provided (or vice versa). Notice that the AP program guarantees FM weak multitolerance as well. Third, we can lower our expectation to providing FF strong multitolerance to both faults where the program guarantees that in the presence of both faults, validity and agreement are not violated, however, agreement may never be reached (i.e., the program never recovers). Therefore, MM weak multitolerance provides the best remaining option where MM strong multitolerance is impossible using the available resources; i.e., masking fault tolerance is guaranteed to both classes of faults if they do not occur simultaneously.

5.3 Complexity Analysis of FF *Weak* Multitolerance

In this section, we investigate the synthesis problem of programs that are *weak* multitolerant to two classes of faults f_1 and f_2 for which failsafe fault-tolerance is required. That is, $f_\delta = \{\langle f_1, failsafe\rangle, \langle f_2, failsafe\ \rangle\}$ in Definition 5.1.1. We show that such a FF $(Failsafe\text{-}Failsafe)$ *weak* multitolerant program can be synthesized in polynomial time in

program state space. Towards this end, we present a sound and complete algorithm. We note that this algorithm can be easily generalized for the case where f_δ includes three or more fault classes for which failsafe fault-tolerance is desired.

Given is a program $\mathcal{P}$, with its invariant S and its specification *spec*. Let $\mathcal{P}'$ be the synthesized program with invariant S' that is *weak* multitolerant to f_1 and f_2. By definition, $\mathcal{P}'$ must maintain *spec* from every reachable state in the computations of $\mathcal{P}' \cup f_1$ (respectively, $\mathcal{P}' \cup f_2$). To this end, on Line 1 of Algorithm 2 in Figure 5.1, we first identify ms_1, a set of states from where execution of one or more f_1 transitions violates safety. Clearly $\mathcal{P}'$ cannot reach a state in ms_1 either in the absence of faults or in the presence of f_1 alone. Likewise, we compute ms_2 on Line 2. Next, we compute mt to be a set of transitions that reach $ms_1 \cup ms_2$ or those that violate *spec*. If there exist states in the invariant such that execution of one or more fault actions from those states violates *spec*, we recalculate the invariant by removing those states. In this recalculation, we ensure that all computations of $\mathcal{P} - mt$ within the new invariant, S', are infinite. By the constraints of Definition 5.1.1 and the definition of ms_1 and ms_2, S' must be a subset of $S - ms_1 - ms_2$. Likewise, $\mathcal{P}'$ cannot include transitions that begin in S' and are in mt. Hence, the only transitions $\mathcal{P}'$ can use inside S' are a subset of $\mathcal{P} - mt$. Removal of states in $ms_1 \cup ms_2$ or transitions in mt may create some deadlock states in $S - ms_1 - ms_2$, i.e., where $\mathcal{P}'$ has no outgoing transitions. Since $\mathcal{P}'$ cannot deadlock in the absence of faults, we remove such deadlock states recursively to construct S' (Line 6 and 7 of Algorithm 2). As shown in Line 8, if the invariant becomes an empty set after reconstruction, we cannot find a FF *weak* multitolerant program $\mathcal{P}'$. If the invariant is not empty, we remove transitions that start in S' and terminate outside S'; i.e., violate the closure of S'. Notice that the removal of such transitions does not introduce

any deadlock states in S'.

Theorem 5.3.1. *The algorithm* Add_FF_Weakmulti *is sound and complete.*

Proof. To show the soundness of our algorithm, we need to show that constraints $C1$, $C2$ and $C3$ of the Problem 5.1.1 are satisfied.

1. $S' \subseteq S$. By the construction of S', S' is obtained by removing zero or more states in S. Thus, $C1$ is trivially satisfied.

2. $(s_0, s_1) \in \mathcal{P}' \wedge s_0 \in S' \Rightarrow (s_0, s_1) \in \mathcal{P}$. By the construction of $\mathcal{P}'$, $\mathcal{P}'$ does not have any new transitions in the absence of faults. Therefore, $C2$ is trivially satisfied.

3. $\mathcal{P}'$ is FF *weak* multitolerant to *spec* from S'. Consider a computation c of $\mathcal{P}'$ that starts from a state in S': From 1, c starts in a state in S, and from 2, c is a computation of $\mathcal{P}$. It follows that c satisfies *spec*. Hence, every computation of $\mathcal{P}'$ that starts from a state in S' is in *spec*, i.e, $\mathcal{P}'$ refines *spec* from S'. We discuss the following two cases:

   (a) Failsafe f_1-tolerance to *spec* from S'. We let the fault-span T_1 to be the set of states reached in any computation of $\psi_{\mathcal{P}'} \cup f_1$ that starts from a state in S'. Consider a computation prefix c of $\psi_{\mathcal{P}'} \cup f_1$ that starts from a state in T_1. From the definition of T_1 there exists a computation prefix c' of $\psi_{\mathcal{P}'} \cup f_1$ such that c is a suffix of c' and c' starts from a state in S'. If c' violates the safety of *spec* then there exists a prefix of c', say $\langle s_0,\ s_1, \ldots,\ s_n \rangle$, such that $\langle s_0,\ s_1, \ldots,\ s_n \rangle$ violates the safety of *spec*. Let $\langle s_0,\ s_1, \ldots,\ s_n \rangle$ be the smallest such prefix, it follows that $(s_{n-1},\ s_n)$ violates the safety of *spec* and hence, $(s_{n-1},\ s_n)\ \in mt$. By construction, $\mathcal{P}'$ does not contain any transition in mt_1. Thus $(s_{n-1},\ s_n)$ is a transition of f_1. If $(s_{n-1},\ s_n)$ is a transition of f_1 then $s_{n-1}\ \in ms_1$

Algorithm 2: Add_FF_Weakmulti

Input: $\mathcal{P}$:transitions,
f_1, f_2:faults of two classes
that need failsafe f-tolerance,
S: state predicate, spec: safety specification

Ouput: If successful, a fault-tolerant $\mathcal{P}'$ with invariant S'
that is **weak** multitolerant to f_1 and f_2

$ms_1 := \{\int_0 \ : \ \exists \int_1, \ \int_2, \ \ldots \int_n \ :$
$(\forall j \ : \ 0 \leq j < n \ : \ (\int_j, \ \int_{j+1}) \in f_1) \wedge$
$(\int_{n-1}, \int_n)$ violates spec$\}$; (1)

$ms_2 := \{\int_0 \ : \ \exists \int_1, \ \int_2, \ \ldots \int_n \ :$
$(\forall j \ : \ 0 \leq j < n \ : \ (\int_j, \ \int_{j+1}) \in f_2) \wedge$
$(\int_{n-1}, \int_n)$ violates spec$\}$; (2)

$mt := \{(\int_0, \int_1) \ : \ ((\int_1 \ \in ms_1 \cup ms_2) \vee$
$(\int_0, \ \int_1)$ violates spec$)\}$; (3)

$S' := S - ms_1 - ms_2$; (4)

$\mathcal{P}_1 := \mathcal{P} - mt$; (5)

WHILE$(\exists \int_0 : \int_0 \in S' :$
$(\forall \int_1 : \int_1 \in S' : (\int_0, \int_1) \notin \mathcal{P}_1))$ (6)

$\{S' := S' - \{\int_0\}\}$ (7)

If$(S' = \{\})$ (8)

declare no **weak** multitolerant program $\mathcal{P}$' exists; (9)

{ Return $\emptyset, \emptyset\}$ (10)

$\mathcal{P}' := \{(\int_0, \ \int_1)|$
$(\int_0, \int_1) \in \mathcal{P}_1, \int_0 \in S', \int_1 \in S'\}$ (11)

// $\mathcal{P}'$ only specifies transitions inside invariant S';
//$\mathcal{P}'$ can be modified to include any subset of
// $\{(s_0, s_1)|s_0 \notin S' \wedge (s_0, s_1) \notin mt\}$;
//T= Reachable $(S', \mathcal{P} \cup f_1) \cup$
// Reachable $(S', \mathcal{P} \cup f_2)$, i.e., T be states
// reached by starting from S' and using transitions of
// $\mathcal{P} \cup f_1$ (respectively, $\mathcal{P} \cup f_2$);
//$\mathcal{P}'$ can include any subset of $\{(s_0, s_1)|s_0 \notin T\}$;

RETURN $\mathcal{P}', S'$; (12)

Figure 5.1: Model revision for adding FF weak multitolerance.

and $(s_{n-2},\ s_{n-1}) \in mt_1$ and hence, $(s_{n-2},\ s_{n-1})$ is a transition of f_1. By induction, if $\langle s_0,\ s_1, \ldots,\ s_n \rangle$ violates the safety of *spec*, $s_0 \in ms_1$, which is a contradiction since $s_0 \in S'$ and $S' \cap ms_1 = \emptyset$. Thus, each prefix of c' maintains *spec*. Since c is a suffix of c', each prefix of c also maintains *spec*. Thus, $\psi_{\mathcal{P}'} \cup f_1$ maintains *spec* from T_1.

(b) Failsafe f_2-tolerance to *spec* from S'. The argument is similar to part 3a.

Now we show that if a FF *weak* multitolerant program can be designed for the given fault-intolerant program then Add_FF_Weakmulti will not declare failure. Let program $\mathcal{P}''$ and predicate S'' solve Problem 5.1.1. Clearly, $S'' \cap ms_1 = \emptyset$; if $s_0 \in (S'' \cap ms_1)$ then the execution of faults alone from s_0 can violate the safety of *spec*. It follows that $S'' \subseteq (S - ms_1)$. Likewise, $S'' \subseteq (S - ms_2)$. Moreover, $\mathcal{P}''|S''$ cannot include any transitions in mt; if $\mathcal{P}''|S''$ contains a transition in mt then the execution of this transitions can violate the safety of *spec*. Thus, $\mathcal{P}''|S'' \subseteq (\mathcal{P} - mt)$. Finally, every computation of $\mathcal{P}''$ that starts in a state in S'' must be an infinite computation, if it were to be in *spec*. It follows that there exists a subset of S such that all computations of $\mathcal{P} - mt$ within that subset are infinite. Our algorithm declares that no solution for the Problem 5.1.1 exists only when there is no subset of $S - ms_1 - ms_2$ such that all the computations of $\mathcal{P} - mt$ within that subset are infinite. It follows that our algorithm declares that no *FF weak* multitolerant program exists only if the answer to Problem 5.1.2 is false. □

Remark 5.3.1. *Algorithm* Add_FF_Weakmulti *can be extended to design a multitolerant program that is subject to* 3 *or more fault classes. Towards this, we specify* ms_3 *for the third fault class and* ms_i *for the* i^{th} *fault class. Then, we calculate mt like Line* 3 *in Algorithm*

Add_FF_Weakmulti *to specify these transitions that lead to state in* $\bigcup_{i=1,...,n} ms_i$ *(n is the number of fault classes). Besides, we need to recalculate invariant* S' *by removing states in* $\bigcup_{i=1,...,n} ms_i$. *The remaining steps are similar to Algorithm* Add_FF_Weakmulti.

5.3.1 Application of Add_FF_Weakmulti

Now we present a simple example to demonstrate how the algorithm Add_FF_Weakmulti facilitates automated synthesis of FF *weak* multitolerance. We use the example of two-sensors agreement protocol introduced in Section 5.2. Specifically, we apply the Add_FF_Weakmulti algorithm in Section 5.3 to the fault-intolerant version of the input-output program.

In the absence of faults, *sensor0* produces the correct value of *in*. Hence, in this program, if no decision has been made, i.e., $out == \perp$, the output will be set to the value of *sensor0*. Hence, the program action is as follows:

$\mathcal{P}_{fi}: \ out == \perp \longrightarrow out := sensor0;$

Remark 5.3.2. *The program action could also be* $out == \perp \longrightarrow out := sensor1$ *or non-determinism execution of both actions, i.e.,* $out == \perp \longrightarrow out := sensor0|sensor1$. *For these cases, a similar analysis can be used to design* FF weak *multitolerant program.*

Next, we show the execution of Add_FF_Weakmulti with input program $\mathcal{P}_{fi}$, f_1, f_2, Inv_{ff} and $spec_{ff}$. The fault actions f_1 and f_2 and safety specification $spec_{ff}$ are defined in Section 5.2.1.

Specifically, our algorithm works as follows: first, Add_FF_Weakmulti computes ms_1 (Line 1) from where execution of f_1 violates safety. Fault f_1 changes the value of sensor0. The

safety specification is independent of the value of sensor0. Thus, if (s_0, s_1) is a transition of f_1 then (s_0, s_1) violates safety iff $spec_{ff}$ is true in state s_0. As such, $ms_1 = spec_{ff}$ where $spec_{ff} \equiv ((out == 1 \wedge in == 0) \vee (int == 0 \wedge out == 0))$. Likewise, $ms_2 = spec_{ff}$.

Next, $mt = \{(s_0, s_1) | s_1 \in spec_{ff}\}$ (Line 3). Since $Inv_{ff} \cap ms_1 = \emptyset$ and $Inv_{ff} \cap ms_2 = \emptyset$, the new invariant S' is equal to Inv_{ff} (Line 4). Then, by removing transitions in mt (Line 5), we have $\mathcal{P}_1 :=$ "$out == \bot \wedge sensor0 == in \longrightarrow out = sensor0''$, i.e., $\mathcal{P}_1$ assigns the output only when $sensor0$ is not corrupted.

On Line 9, we compute the transitions of $\mathcal{P}_1$ that start and end in S'. Thus, on Line 9, we get the program, $\mathcal{P}_1 :=$ "$out == \bot \wedge sensor0 == sensor1 == in \longrightarrow out = sensor0$".

Moreover, after computing the fault-span of this program, we observe that a state where $sensor0$ is equal to $sensor1$ but $sensor0$ is not equal to in is not reached in the presence of f_1 alone (or in the presence of f_2 alone). Hence, we can add the transition corresponding to the following action to $\mathcal{P}_1$:

$out == \bot \wedge sensor0 == sensor1 \neq in \longrightarrow out = sensor0$.

Thus, the FF *weak* multitolerant program is as follows: (Note that this is the same program from Section 5.2)

$\mathcal{P}_1 := out == \bot \wedge sensor0 == sensor1 \longrightarrow out = sensor0$.

Remark 5.3.3. *The extra transitions (added in Line 9) cannot be executed but are useful in simplifying the fault-tolerant program. Specifically, in this example, it allowed us to construct a fault-tolerant program that does not read the value of the variable in. In a more general setting, to ensure that the synthesized program reads only the variables that it is allowed to read, one can utilize the approach in [135] to synthesize distributed programs, where the program consists of multiple processes and each process can read some subset of program*

variables. Since this paper does not focus on the issue of distribution, we omit details of modeling read/write restrictions in synthesizing distributed programs.

5.4 Complexity Analysis of MM *Weak* Multitolerance

In this section, we investigate Problem 3.2 for cases where we want to add *weak* multitolerance for $f_\delta = \{\langle f_{m1}, masking\rangle, \langle f_{m2}, masking\rangle\}$. We find a surprising result that the MM ($Masking$-$Masking$) *weak* multitolerant synthesis problem is NP-complete, even though, as shown in [71], the synthesis problem of the corresponding *strong* multitolerant program is in P.

Before we present the formal proof, we give an intuition behind this complexity. Consider the case where there exists a transition $(s_1,\ s_2)$ of f_{m2} that violates the safety specification. We have the following two options: (i) ensure that s_1 is unreachable in the computations of $\mathcal{P} \cup f_{m2}$. (ii) allow s_1 to be reached only while program is 'recovering' from f_{m1}. Moreover, the choice made for this state affects other similar states. In our proof, we relate the choice made between these two options to the values of Boolean variables in the SAT formula. This allows us to reduce the SAT problem to the MM *weak* multitolerant synthesis problem.

Theorem 5.4.1. *The problem of synthesizing MM* weak *multitolerant programs from their fault-intolerant version is NP-complete.*

Proof. Given a program $\mathcal{P}$, with its invariant S, its specification *spec*, and two classes of faults f_{m1} and f_{m2}, we prove that the decision Problem 3.2 is NP-hard when $f_\delta = \{\langle f_{m1},\ masking\rangle,\ \langle f_{m2},\ masking\rangle\}$. Illustrating the NP membership of Problem 3.2 is straightforward; hence omitted.

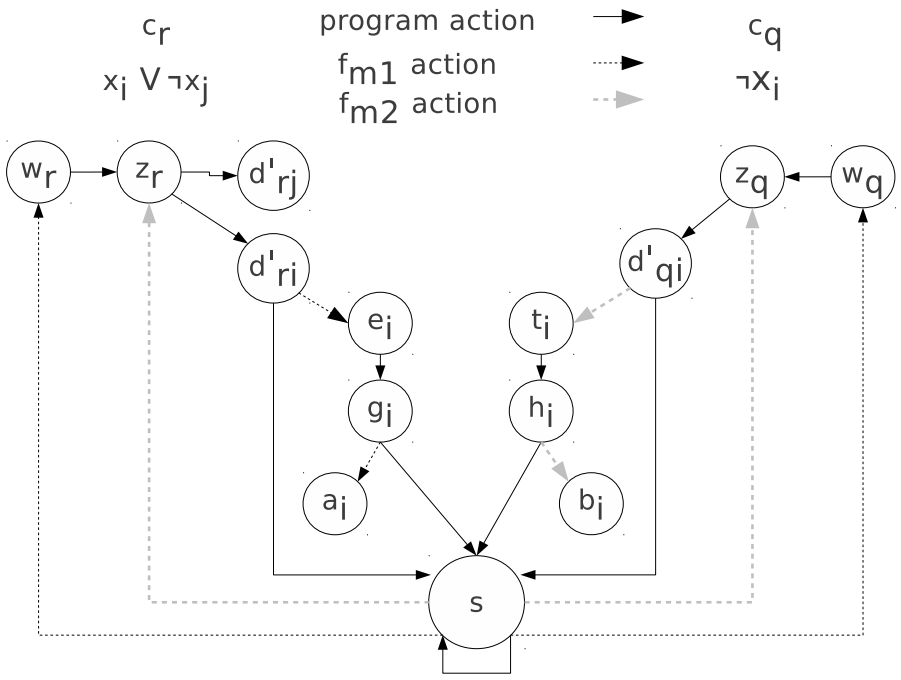

Figure 5.2: Mapping from an instance of the SAT problem.

Now, we present a polynomial-time mapping from an instance of the SAT problem to a corresponding instance $\langle \mathcal{P}, S, spec, f_{m1}, f_{m2} \rangle$ of Problem 3.2. An instance of the SAT problem is specified in terms of a set of literals $x_1, x_2, \ldots, x_n$ and $\neg x_1, \neg x_2, \ldots, \neg x_n$ where x_i and $\neg x_i$ are complements of each other. The SAT formula is of the form $\phi = C_1 \wedge C_2 \wedge C_3 \wedge \ldots \wedge C_k$, where each clause C_i is a disjunction of several literals. Then we show that the given SAT formula is satisfiable if and only if there exists a solution for the mapped instance of Problem 3.2. We construct the mapped instance as follows (see Figure 5.2):

The state space of $\mathcal{P}$ is as follows:

- We introduce a state s. This is the only state in the invariant S.

- For each propositional variables x_i, $1 \leq i \leq n$, and its complement $\neg x_i$ in the SAT instance, we introduce the following states: e_i, t_i, g_i, h_i, a_i and b_i.

- For each clause C_r, $1 \leq r \leq k$, we introduce states w_r and z_r.

- If clause C_r includes literal x_i, we introduce a state d_{ri}. If clause C_r includes literal $\neg x_i$, we introduce a state d'_{ri}.

The transitions of $\mathcal{P}|S$ include only a self-loop (s, s). The transitions of f_{m1} and f_{m2} are as follows:

- For each clause C_r, we include the fault transition (s, w_r) in f_{m1} and the fault transition (s, z_r) in f_{m2}.

- If the clause C_r includes the literal x_i, then we include the fault transition (d_{ri}, e_i) in f_{m1}.

- If the clause C_r includes the literal $\neg x_i$, then we include the fault transition (d'_{ri}, t_i) in f_{m2}.

- For each propositional variable x_i and its complement $\neg x_i$, we include the fault transition (g_i, a_i) in f_{m1} and (h_i, b_i) in f_{m2}.

The safety specification is defined as follows.

- Transitions (g_i, a_i) and (h_i, b_i) violate safety.

- Transitions (s, s), (s, w_r), (s, z_r), (d_{ri}, e_i) and (d'_{ri}, t_i) do not violate safety.

- For each clause C_r, each propositional variable x_i and its complement $\neg x_i$, the following transitions do not violate safety:

  - (w_r, z_r), (z_r, d_{ri}), (z_r, d'_{ri}), $(e_i, t_i), (t_i, e_i), (e_i, g_i), (t_i, h_i), (g_i, s)$, and (h_i, s).

- All transitions except those identified above (e.g., (z_r, w_r), (z_r, s), etc) violate safety specification.

Now, we show that the given SAT formula is satisfiable if and only if the answer to Problem 3.2 for the mapped instance is affirmative where $f_\delta = \{\langle f_{m1}, masking\rangle, \langle f_{m2}, masking\rangle\}$.

- ($\Longrightarrow$) First, we show if the given SAT formula is satisfiable, then there exists a solution that meets the requirements of the synthesis problem. Since ϕ has a satisfying truth assignment, there exists an assignment of truth values to the literals x_i, such that each C_r evaluates to true. Now, we identify the program $\mathcal{P}'$, that solves *MM weak* multitolerant problem.

  The invariant of $\mathcal{P}'$ is the same as the invariant of $\mathcal{P}$ (i.e., $\{s\}$). We derive the transitions of the *weak* multitolerant program $\mathcal{P}'$ as follows:

  - For each disjunction C_r, we include the transition (w_r, z_r).
  - If x_i is assigned $true$:
    - We introduce $(e_i, t_i), (t_i, h_i), (h_i, s)$.
    - For each disjunction C_r that includes x_i, we introduce (z_r, d_{ri}) and (d_{ri}, s).
  - If x_i is assigned $false$:
    - We include $(t_i, e_i), (e_i, g_i)$ and (g_i, s).
    - For each disjunction C_r that includes $\neg x_i$, we introduce (z_r, d'_{ri}) and (d'_{ri}, s).

  Thus, in the presence of f_{m1} alone, $\mathcal{P}'$ provides safe recovery to s through d_{ri}, e_i, t_i, h_i. In the presence of f_{m2} alone, $\mathcal{P}'$ provides safe recovery to s through d'_{ri}, t_i, e_i, g_i.

  Now, we show that $\mathcal{P}'$ is *weak* multitolerant in the presence of faults f_{m1}, f_{m2}.

– *(In the absence of faults)* $\mathcal{P}'|S = \mathcal{P}|S$. Thus, $\mathcal{P}'$ satisfies *spec* in the absence of faults.

– *Masking f_{m1}-tolerance.* If the faults from f_{m1} occur then the program can be perturbed to state w_r, $1 \leq r \leq k$. From w_r, $\mathcal{P}'$ has only one transition that reaches z_r. Since C_r evaluates to $true$, there exists i such that either x_i is a literal in C_r and x_i is assigned the truth value $true$ or $\neg x_i$ is a literal in C_r and x_i is assigned the truth value $false$. In the former case, $\mathcal{P}'$ can recover to s using the two sequences of transitions, $\langle (z_r, d_{ri}), (d_{ri}, s) \rangle$, or $\langle (z_r, d_{ri}), (d_{ri}, e_i), (e_i, t_i), (t_i, h_i), (h_i, s) \rangle$. In the latter case, $\mathcal{P}'$ can recover to s using exactly one sequence of transitions $\langle (z_r, d'_{ri}), (d'_{ri}, s) \rangle$. Note that if x_i is true then $\mathcal{P}'$ cannot reach g_i from where it can violate safety specification. Thus, any computation of $\mathcal{P}' \cup f_{m1}$ eventually reaches a state in the invariant. Moreover, from z_r, every computation of $\mathcal{P}' \cup f_{m1}$ does not violate the safety specification. Based on the above discussion, $\mathcal{P}'$ is masking tolerant to f_{m1}.

– *Masking f_{m2}-tolerance.* The argument is similar to the one showing that $\mathcal{P}'$ is masking tolerant to f_{m1}.

- ($\Longleftarrow$) Second, we show that if there exists a *weak* multitolerant program that solves the instance of the synthesis problem 3.2, then the given SAT formula is satisfiable. Let $\mathcal{P}'$ be the *weak* multitolerant program derived from the fault-intolerant program $\mathcal{P}$. The invariant of $\mathcal{P}'$, S', is not empty and $S' \subseteq S$, S' must include state s. Thus, S'=S.

  Let C_r be a clause in the given SAT formula. The corresponding states added in the instance of the synthesis problem are w_r and z_r. Note that w_r can be reached from

s by a transition in f_{m1}. Hence, $\mathcal{P}'$ must include the transition (w_r, z_r). Thus z_r is reached in the computation of $\mathcal{P}' \cup f_{m1}$. Hence, $\mathcal{P}'$ must recover to s from z_r without violating *spec*. Therefore, for some i, $\mathcal{P}'$ has to have a transition of the form (z_r, d_{ri}) or (z_r, d'_{ri}). If $\mathcal{P}'$ includes (z_r, d_{ri}), then the clause C_r contains literal x_i and we assign x_i the truth value *true*. Likewise, if $\mathcal{P}'$ includes (z_r, d'_{ri}) for some i, then the clause C_r contains literal $\neg x_i$ and we assign x_i the truth value $false$. Thus, by construction, C_r evaluates to true.

Now, to complete the proof, we have to show that the truth values assigned to all literals are consistent, i.e., it is not the case that x_i is assigned *true* in one clause and $false$ in another clause. We show this by a proof by contradiction. If x_i is assigned *true* in clause C_r and $false$ in clause C_q then $\mathcal{P}'$ includes both transitions (z_r, d_{ri}) and (z_q, d'_{qi}). Now, from d_{ri}, the program can reach e_i by the occurrence of f_{m1} alone. Hence, the program $\mathcal{P}'$ cannot include the transition (e_i, g_i), as including this transition will allow the program to reach g_i in a computation of $\mathcal{P}' \cup f_{m1}$ and violate safety by executing (g_i, a_i). Likewise, $\mathcal{P}'$ can reach t_i by the occurrence of f_{m2} alone. Hence, $\mathcal{P}'$ cannot include the transition (t_i, h_i). If both transitions (e_i, g_i) and (t_i, h_i) are not included then $\mathcal{P}'$ cannot recover from e_i to the state in the invariant. This contradicts the assumption that $\mathcal{P}'$ is masking f_{m1}-tolerant. Thus, the truth value assignment to all literals is consistent. □

5.4.1 A Heuristic for MM *Weak* Multitolerance

The previous result showed that in general, the problem of adding MM *weak* multitolerance is NP-hard. In this section, we present a sound (but incomplete) algorithm that adds MM

weak multitolerance to a given program $\mathcal{P}$ that is subject to two classes of faults $f_\delta = \{\langle f_1, masking\rangle, \langle f_2, masking\rangle\}$ in polynomial time. Our algorithm Add_MM_Weakmulti takes program actions, faults, invariant and safety specification as input and generates a MM *weak* multitolerant program. The basic idea of Add_MM_Weakmulti is to first construct the corresponding FF *weak* multitolerant program that ensures safety. Then we use the fault span of the FF *weak* multitolerant program to add recovery. Specifically, let T_1 and T_2 be the fault-spans in the presence of f_1 and f_2 respectively. We ensure that every path from T_1 reaches a state in S_1, and likewise, every path from T_2 reaches a state in S_1. Additionally, we ensure that T_1 (respectively, T_2) remains closed in transitions of f_1 (respectively, f_2) during this revision process.

Given is a program $\mathcal{P}$ with its state predicate S and its specification *spec*. Let $\mathcal{P}'$ be the synthesized program with invariant S' that is *weak* multitolerant to f_1 and f_2. By definition, $\mathcal{P}'$ is masking f_1-tolerant from S' for *spec* if only f_1 occurs, and masking f_2-tolerant from S' for *spec* if only f_2 occurs. To this end, Line 1 of Algorithm 3 identifies ms_1, a set of states from where execution of one or more f_1 transitions violates safety. On Line 2, we identify ms_2 that is a set of states from where execution of one or more f_2 transitions violates safety. Next, we compute mt, a set of transitions that reach $ms_1 \cup ms_2$ or those that violates *spec*. By calling Algorithm 1 in Line 6 of Algorithm 3, we obtain $\mathcal{P}_1$ with invariant S_1, where $\mathcal{P}_1$ is FF *weak* multitolerant to f_1 and f_2. In the loop of Lines 7-12, we reconstruct transitions to ensure that $\langle S_1, \mathcal{P}_1 \cup f_1\rangle$ maintains *spec* from T_1, $\langle S_1, \mathcal{P}_1 \cup f_2\rangle$ maintains *spec* from T_2 on Line 9. To guarantee that from each state outside S_1 there is a path that reaches a state in S_1 and there are no cycles in states outside S_1, we update $\mathcal{P}_1$ by calling the algorithm Ensure_Recovery in Lines 10 and 11. Algorithm 4 in Figure 5.4 captures the details

of Ensure_Recovery. Specifically, Ensure_Recovery is defined in such a way that from each state outside S_1 there is a path that reaches a state in S_1, and there are no cycles in states outside S_1. As shown in Line 13, if the invariant becomes an empty set after reconstruction, we cannot find a MM *weak* multitolerant program $\mathcal{P}'$. Details are as shown in Algorithm 3 in Figure 5.3.

Theorem 5.4.2. *The algorithm* Add_MM_Weakmulti *is sound.*

Proof. To show the soundness of our algorithm, we need to show that constraints $C1$, $C2$ and $C3$ of the Problem 5.1.1 are satisfied.

1. $S_1 \subseteq S$. By the correctness of Add_FF_Weakmulti, S_1 obtained at Line 6 satisfies $C1$. Since the following steps do not add any state to S_1, $C1$ is preserved by the final program $\mathcal{P}_1$.

2. $(s_0, s_1) \in \mathcal{P}_1 \wedge s_0 \in S_1 \Rightarrow (s_0, s_1) \in \mathcal{P}$. By the correctness of Add_FF_Weakmulti, $\mathcal{P}_1$ obtained at Line 6 of Algorithm 2 satisfies $C2$. Since the remaining steps do not add any transition to $\mathcal{P}_1$, $C2$ is preserved by the final program $\mathcal{P}_1$.

3. $\mathcal{P}_1$ is MM *weak* fault-tolerant to *spec* from S_1. Consider a computation c of $\mathcal{P}_1$ that starts from a state in S_1: From Part 1 of this proof, c starts in a state in S, and from Part 2, c is a computation of $\mathcal{P}$. It follows that c satisfies *spec*. Hence, every computation of $\mathcal{P}_1$ that starts from a state in S_1 is in *spec*, i.e, $\mathcal{P}_1$ refines *spec* from S_1. Next, we discuss the following two cases:

   (a) **Masking f_1-tolerance to *spec* from S_1.** To show this, we need to show the following three properties.

Algorithm 3: Add_MM_Weakmulti

Input: $\mathcal{P}$:transitions,
f_1, f_2:faults of two classes
that need masking f-tolerance,
S: state predicate, spec: safety specification

Ouput: If successful, a fault-tolerant $\mathcal{P}'$ with invariant S'
that is **weak** multitolerant to f_1 and f_2

$ms_1 := \{\int_0 \ : \ \exists \int_1, \ \int_2, \ \ldots \int_n \ :$
$(\forall j \ : \ 0 \leq j < n \ : \ (\int_j, \ \int_{j+1}) \in f_1) \wedge$
$(\int_{n-1}, \int_n)$ violates spec$\}$; (1)

$ms_2 := \{\int_0 \ : \ \exists \int_1, \ \int_2, \ \ldots \int_n \ :$
$(\forall j \ : \ 0 \leq j < n \ : \ (\int_j, \ \int_{j+1}) \in f_2) \wedge$
$(\int_{n-1}, \int_n)$ violates spec$\}$; (2)

$mt := \{(\int_0, \int_1) \ : \ ((\int_1 \ \in ms_1 \cup ms_2) \vee$
$(\int_0, \ \int_1)$ violates spec$)\}$; (3)

$T_1 := S - ms_1$ (4)

$T_2 := S - ms_2$ (5)

$\mathcal{P}_1, S_1 := Add\_FF\_Weakmulti(\mathcal{P}, f_1, f_2, S, spec)$ (6)

WHILE$(T_1' == T_1 \wedge T_2' == T_2)$ (7)

$\{T_1' := T_1, T_2' := T_2$ (8)

$\mathcal{P}_1 := \{(s_0, s_1) | (s_0 \in S_1$
$\Rightarrow (s_0, s_1) \in \mathcal{P}_1)$
$\wedge (s_0 \in T_1 \Rightarrow s_1 \in T_1)$
$\wedge (s_0 \in T_2 \Rightarrow s_1 \in T_2)\} - mt$ (9)

$T_1, S_1, \mathcal{P}_1 := Ensure\_Recovery\ (\mathcal{P}_1, f_1, T_1, S_1)$ (10)

$T_2, S_1, \mathcal{P}_1 := Ensure\_Recovery\ (\mathcal{P}_1, f_2, T_2, S_1)\}$ (11)

IF $(S_1 \neq \emptyset)$ (12)

return $\mathcal{P}_1, S_1$ (13)

ELSE (14)

declare no **weak** multitolerant program $\mathcal{P}'$ exists, (15)

return $\emptyset, \emptyset$ (16)

Figure 5.3: Model revision for adding MM weak multitolerance.

Algorithm 4: Ensure_Recovery

Input: $\mathcal{P}$:transitions,
f:fault actions, S: state predicate, T: state predicate

Goal: Find T', $\mathcal{P}'$ and S' such that $T' \subseteq T$,
$S' \subseteq S$, T' is closed in $\mathcal{P}' \cup f$,
every computation of $\mathcal{P}'$ from T' reaches a state in S'.

$S_1, S_2 := S, S;$ (1)
WHILE $(S_1 == S)$ (2)
$\{S_1 := S\}$ (3)
//Rank(s_0) = length of the shortest computation prefix of
// $\mathcal{P}$ from s_0 to some state in S.
// $Rank(s_0) = \infty$ means S is not reachable from s_0.
$T := T - \{s_0 | Rank(s_0) = \infty\}$ (4)
$T := T - \{s_0 | \exists s_1 : (s_0, s_1)$
$\in f, s_0 \in T, s_1 \notin T\}$ (5)
$S := S \wedge T$ (6)
WHILE$(\exists s_0 : s_0 \in S :$
$\forall s_1 : s_1 \in S : (s_0, s_1) \notin \mathcal{P})$ (7)
$S := S - \{s_0\}$ (8)
$\mathcal{P}_1 := removeCycles(\mathcal{P}, S, T)$ (9)
//returns $\mathcal{P}_1$ such that $\mathcal{P}_1 \subseteq \mathcal{P}$
//$\mathcal{P}_1|S == \mathcal{P}|S, \mathcal{P}_1|(T - S)$ is acyclic,
//and $\forall s_0 : s_0 \in T : S$ is reachable from s_0 in $\mathcal{P}_1$
//There are several possible implementations
//and any one of them is acceptable.
//One possible implementation is to rank each state
//based upon the shortest path from that state to a state inside S,
//and then remove these transitions that do not decrease the rank.
Return $\mathcal{P}_1, S, T$ (10)

Figure 5.4: The recovery algorithm.

- T_1 is closed in $\mathcal{P}_1 \cup f_1$. Closure of T_1 in $\mathcal{P}_1$ is by construction. Regarding closure of T_1 in f_1, observe that there is no change in the last iteration of the loop in Lines 7-12 of Algorithm 3. Thus, T_1 is closed in f_1.

- $\mathcal{P}_1 \cup f_1$ maintains *spec* from T_1. We let the fault-span T_1 to be the set of states reached in any computation of $\mathcal{P}_1 \cup f_1$ that starts from a state in S_1. Consider a computation prefix c of $\mathcal{P}_1 \cup f_1$ that starts from a state in T_1. From the definition of T_1 there exists a computation prefix c' of $\mathcal{P}_1 \cup f_1$ such that c is a suffix of c' and c' starts from a state in S_1. If c' violates the safety of *spec* then there exists a prefix of c', say $\langle s_0,\ s_1, \ldots,\ s_n \rangle$, such that $\langle s_0,\ s_1, \ldots,\ s_n \rangle$ violates the safety of spec. Let $\langle s_0,\ s_1, \ldots,\ s_n \rangle$ be the smallest such prefix, it follows that $(s_{n-1},\ s_n)$ violates the safety of *spec* and hence, $(s_{n-1},\ s_n)\ \in mt$. By construction, $\mathcal{P}_1$ does not contain any transition in mt (See Line 9 of Algorithm 2). Thus $(s_{n-1},\ s_n)$ is a transition of f_1. If $(s_{n-1},\ s_n)$ is a transition of f_1 then $s_{n-1}\ \in ms_1$ and $(s_{n-2},\ s_{n-1})\ \in mt_1$ and hence, $(s_{n-2},\ s_{n-1})$ is a transition of f_1. By induction, if $\langle s_0,\ s_1, \ldots,\ s_n \rangle$ violates the safety of *spec*, $s_0 \in ms_1$, which is a contradiction since $s_0 \in S_1$ and $S_1 \cap ms_1 = \emptyset$ (Guaranteed by Line 6 of Algorithm 2 since the following steps don't add any state in S_1). Thus, each prefix of c' maintains *spec*. Since c is a suffix of c', each prefix of c also maintains *spec*. Thus, $\mathcal{P}_1 \cup f_1$ maintains *spec* from T_1.

- Every computation of $\mathcal{P}$ that starts from a state in T_1 eventually reaches a state of S_1. The Ensure_Recovery algorithm only updates $\mathcal{P}_1$ by removing some transitions from $\mathcal{P}_1$ in Steps 4, 5 and 10 of Algorithm 3. By con-

struction, Ensure_Recovery removes all the deadlock states in Steps 7 and 8 recursively. Also the function RemoveCycles is defined in such a way that from each state outside S_1 there is a path that reaches a state in S_1, and there are no cycles in states outside S_1.

(b) Masking f_2-tolerant to *spec* from S_1. The argument is the similar as Part 3a.

□

Remark 5.4.1. *Note that* Add_MM_Weakmulti *is sound but not complete. One reason for this is that the function* removeCycles *has several possible implementations and the choice of transitions removed in* removeCycles*(Line* 10 *of Algorithm* 3*) for ensuring recovery in the presence of* f_1 *can prevent recovery in the presence of* f_2*. If one were to consider all possible choices of* removeCycles*, then the time complexity would become exponential in the state space.*

Remark 5.4.2. *Algorithm* Add_MM_Weakmulti *can be extended to design a multitolerant program that is subject to* 3 *or more fault classes. Towards this, we specify* ms_3 *for the third fault class and the corresponding* ms_i *for the* i^{th} *fault class. Then we calculate mt like Line* 3 *in Algorithm* Add_MM_Weakmulti *to speicify these transitions that lead to state in* $\bigcup_{i=1,...,n} ms_i$ *(n is the number of fault classes). Moreover, we need to calculate the corresponding* T_i *for the* i^{th} *fault class. In particular,* Ensure_Recovery *(Lines* $10-11$ *of Algorithm* 2*) will be repeated for each fault class.*

5.4.2 Application of Add_MM_Weakmulti

Now we extend the example used in Section 5.3.1 to demonstrate how the algorithm 3 facilitates automated synthesis of MM *weak* multitolerance. We extend the program in

Section 5.3.1 to be the program which consists of three sensors.

As discussed in Section 5.3.1, the action of the input program is as follows:

$\mathcal{P}_{mi} : out == \perp \longrightarrow out := sensor0;$

In this program, we consider three classes of faults: a transient fault that occurs at sensor 0, a transient fault that occurs at sensor 1 and a transient fault that occurs at sensor 2. When transient faults corrupt one sensor, the sensor changes its value to the opposite of that of the input. Thus, the fault actions for this program are as follows:

f_1: $true \longrightarrow sensor0 := 1 - in;$

f_2: $true \longrightarrow sensor1 := 1 - in;$

f_3: $true \longrightarrow sensor2 := 1 - in;$

The safety specification requires that the program never reaches a state where the output of the system is not the same as the real status of the environment. Hence, the safety specification states that the program should never reach a state where the following formula is true:

$spec_{mm} = (out == 1 \wedge in == 0) \vee (in == 1 \wedge out == 0)$

The legitimate states of the program are those where the values of the majority of sensors are equal to the value of the real status in the environment and the output is equal to the value of the real status in the environment if the output is decided. Hence, the invariant is as follows:

$Inv_{mm} =$

$((sensor0 == sensor1 == in) \vee (sensor0 == sensor2 == in)$

$\vee(sensor1 == sensor2 == in)) \wedge (out == \perp \vee out == in)$

We extend our Add_MM_Weakmulti algorithm to revise the above input-output program to be a MM *weak* multitolerant program with input $\mathcal{P}_{mi}$, f_1, f_2, f_3, Inv_{mm} and $spec_{mm}$. Specifically, our algorithm works as follows: after executing Add_FF_Weakmulti in Line 6 of Algorithm 3, the program actions include:

$out == \perp \wedge sensor0 == sensor1 \longrightarrow out := sensor0;$

$out == \perp \wedge sensor0 == sensor2 \longrightarrow out := sensor0;$

The following steps of executing Ensure_Recovery (Lines 10 and 11 of Algorithm 3) remove these deadlock states if *sensor0* is corrupted and add recovery transition:

$out == \perp \wedge sensor1 == sensor2 \longrightarrow out := sensor1.$

Hence, the final program is as follows:

$out == \perp \wedge sensor0 == sensor1 \longrightarrow out := sensor0;$

$out == \perp \wedge sensor0 == sensor2 \longrightarrow out := sensor0;$

$out == \perp \wedge sensor1 == sensor2 \longrightarrow out := sensor1;$

5.5 Complexity Analysis of FM *Weak* Multitolerance

In this section, we investigate the synthesis problem for *weak* multitolerant programs for the case where program is subject to two classes of faults f_1 and f_2 for which respectively failsafe and masking fault-tolerance is required, that is $f_\delta = \{\langle f_1, failsafe\rangle, \langle f_2, masking\rangle\}$ in Definition 5.1.1. This synthesis problem is NP-complete! This result is also surprising since the corresponding problem for *strong* multitolerance is in P.

Theorem 5.5.1. *The problem of synthesizing FM* weak *multitolerant programs from their fault-intolerant version is NP-complete.*

Proof. Given is a program $\mathcal{P}$, with its invariant S, its specification *spec*, and two classes of faults f_1 and f_2. Since demonstrating membership to NP is trivial, we only illustrate that the synthesis problem identified in Definition 5.1.1 is NP-hard when $f_\delta = \{\langle f_1,\ failsafe\rangle,\ \langle f_2,\ masking\rangle\}$.

We construct the mapping by changing that part of proof for Theorem 5.4.1 as follows:

- Replace f_{m1} fault transitions with transitions of f_2.
- Replace f_{m2} fault transitions with transitions of f_1.

Next, we show that the given SAT formula is satisfiable if and only if there exists a solution to the *FM weak* multitolerant synthesis problem.

- ($\Longrightarrow$) By the proof of Theorem 5.4.1, if the given SAT formula is satisfiable then there exists a program that is masking fault-tolerant to f_{f1} and f_{m1}. Also, by Definitions 2.3.2 and 2.3.3, a program $\mathcal{P}$ that is masking f_1-tolerant from S for *spec* is failsafe f_1-tolerant from S for *spec*. Hence, if the given SAT formula is satisfiable then there is a solution to the corresponding instance of the *FM weak* multitolerance synthesis.

- ($\Longleftarrow$) This proof is identical to the corresponding proof for Theorem 5.4.1, hence we omit it.

□

5.6 Complexity Analysis of MN *Weak* Multitolerance

In this section, we present a synthesis algorithm for *weak* multitolerant programs for the case where a program is subject to two classes of faults f_1 and f_2 for which respectively masking and nonmasking fault-tolerance is required, that is $f_\delta = \{\langle f_1, masking\rangle, \langle f_2, nonmasking\rangle\}$ in Definition 5.1.1. We show that such a MN $(Masking\text{-}Nonmasking)$ *weak* multitolerant program can be synthesized in polynomial time in the state space. This sound and complete algorithm also can be easily generalized for the case where f_δ includes one class of faults for which masking fault tolerance is desired and two or more fault classes for which nonmasking fault tolerance is desired. Note that from the results in Section 5.4, if masking fault tolerance is desired for two or more classes of faults, then the problem is NP-complete.

Given is a program $\mathcal{P}$, with its invariant S and its specification *spec*. Our objective is to synthesize a program $\mathcal{P}'$, with invariant S' that is *weak* multitolerant to f_δ. By definition, $\mathcal{P}'$ must be masking f_1-tolerant. $\mathcal{P}'$ must also be nonmasking f_2-tolerant. The algorithm for MN *weak* multitolerance utilizes the algorithm Add_Masking (from [134]) that adds masking fault-tolerance to a single class of faults. Add_Masking returns the synthesized program $\mathcal{P}'$, its invariant S' and its fault-span T' such that $\mathcal{P}'$ is masking fault-tolerant to S' and T' is the fault-span used to prove this in Definition 2.3.2. The algorithm Add_MN_Weakmulti only relies on the correctness (i.e., soundness and completeness) of Add_Masking. It does not rely on the actual implementation of Add_Masking. (For reader's

Algorithm 5: Add_MN_Weakmulti
Input: $\mathcal{P}$:transitions,
$f_{\delta 1}$: $\{\langle f_1, masking\rangle, \langle f_2, nonmasking\ \rangle\}$
S: state predicate, spec: safety specification
Ouput: If successful, a fault-tolerant $\mathcal{P}'$ with invariant S'
that is **weak** multitolerant to f_1 and f_2
$\mathcal{P}_1$, S', T_1 := Add_Masking$(\mathcal{P}, f_1, S, spec)$; (1)
if $(S' = \{\})$
declare no **weak** multitolerant program $\mathcal{P}'$ exists, (2)
return $\emptyset, \emptyset$; (3)
$\mathcal{P}' := \mathcal{P}_1|T_1 \cup \{(\int_0, \int_1) : (\int_0, \int_1)$
$\in \mathcal{P}, \int_0 \notin T_1 \wedge \int_1 \in T_1\}$; (4)
RETURN $\mathcal{P}'$, S' (5)

Figure 5.5: Model revision for adding MN weak multitolerance.

convenience, the algorithm from [134] is included in the Appendix. However, we note that the proof of Add_MN_Weakmulti only relies on the correctness of Add_Masking and not on its details.) Thus, Add_MN_Weakmulti first invokes Add_Masking on Line 1 with parameters $(\mathcal{P}, f_1, S, spec)$. As shown in Line 2, if the invariant becomes an empty set after reconstruction in Line 1, we cannot find an *MN weak* multitolerant program $\mathcal{P}'$. If the invariant is not empty, we re-compute the set of program transitions in Line 3. Algorithm 5 in Figure 5.5 describes the details.

Algorithm 5 is the same as that for adding *strong* multitolerant program [71].

Theorem 5.6.1. *The algorithm* Add_MN_Weakmulti *is sound and complete.*

Proof. By correctness of Add_Masking, $\mathcal{P}_1$ satisfies the constraints of Definition 5.1.1 for the case where $f_\delta = \{\langle f_1, masking\rangle\}$. Since Step 3 does not add or remove any state of S or a transition from $\mathcal{P}|T$, these constraints are preserved by the final program $\mathcal{P}'$. Hence, to complete the proof of this theorem, next, we show that $\mathcal{P}'$ is nonmasking f_2-tolerant. By definition of masking fault-tolerance, every computation of $\mathcal{P}_1$ that starts in a state in T_1 reaches a state in S'. If f_2 perturbs the program to a state outside T_1 then the recovery

transitions added in Step 3 will recover the program to a state in T_1 from where it can utilize the recovery paths inside $\mathcal{P}_1$ to reach S'. Thus, $\mathcal{P}'$ is nonmasking f_2-tolerant.

Our algorithm declares that an MN *weak* multitolerant program does not exist only when Add_Masking does not find a masking f_1-tolerant program. Hence, completeness of Algorithm 5 follows from the completeness of Add_Masking. □

5.7 Complexity Analysis of NN *Weak* Multitolerance

The algorithm Add_NN_Weakmulti for NN ($Nonmasking$-$Nonmasking$) *weak* multitolerant problem is identical to Algorithm 5, except that instead of invoking Add_Masking on Line 1 of Algorithm 5, we call Add_Nonmasking(from [134]).

Theorem 5.7.1. *The algorithm* Add_NN_Weakmulti *is sound and complete.*

Proof. Since the proof is similar to the proof of algorithm 5, we omit it. □

5.8 Comparison of Feasibility of *Strong* Multitolerance and *Weak* Multitolerance

In this section, we compare the relation between *strong* multitolerance [71] and *weak* multitolerance.

Specifically, we point out that if a program is *strong* multitolerant then it is also *weak* multitolerant, although the reverse is not necessarily true. We also identify circumstances where solvability for adding *weak* and *strong* multitolerance differs. In particular, we show that for high atomicity programs, (i.e., programs that can read all the variables and write all

the variables in one atomic step), if adding MN (respectively, NN or FN) *weak* multitolerance to a program is feasible then adding MN (respectively, NN or FN) *strong* multitolerance is also feasible. However, the same result does not apply for MM, MF or FF multitolerance.

Definition 5.8.1. *Let* $f_\delta = \{\langle f_i, l_i\rangle \mid 0 < i \leq n, l_i \in \{failsafe,\ nonmasking,\ masking\}\}$ *where* $n \geq 0$. *Program* $\mathcal{P}$ *is* **strong *multitolerant*** *to fault set* f_δ *from* S *for spec iff the following conditions hold:*

1. *(In the* absence *of faults)* $\mathcal{P} \models_S$ spec*;*
2. $\mathcal{P}$ *is masking* f_m*-tolerant from* S *for* spec*;*
3. $\mathcal{P}$ *is nonmasking* f_n*-tolerant from* S *for* spec*;*
4. $\mathcal{P}$ *is failsafe* f_f*-tolerant from* S *for* spec*,*

where

- $f_m = \bigcup\{f_i | \langle f_i, masking\rangle \in f_\delta\ \}$,
- $f_n = \bigcup\{f_i | \langle f_i, nonmasking\rangle \in f_\delta \vee \langle f_i, masking\rangle \in f_\delta\}$,
- $f_f = \bigcup\{f_i | \langle f_i, failsafe\rangle \in f_\delta \vee \langle f_i, masking\rangle \in f_\delta\}$. □

Observation 5.8.1. *If a program* $\mathcal{P}$ *is* strong *multitolerance to* f_δ *from* S *for spec, then* $\mathcal{P}$ *is* weak *multitolerance to* f_δ *from* S *for spec.* □

Observation 5.8.2. *It is possible that* $\mathcal{P}$ *is* weak *multitolerance to* f_δ *from* S *for spec however* $\mathcal{P}$ *is not* strong *multitolerance to* f_δ *from* S *for spec.* □

Observation 5.8.3. *Program* $\mathcal{P}$ *is* weak *multitolerant to fault set* f_δ *from* S *for spec where* $f_\delta = \{\langle f_1, nonmasking\rangle, \langle f_2, failsafe\rangle\}$ *iff* $\mathcal{P}$ *is* strong *multitolerant to fault set* f_δ *from* S *for spec .* □

Similar to Definition 5.1.1, we define the problem of *strong* multitolerance synthesis as follows:

Definition 5.8.2. ***The Strong Multitolerance Synthesis Problem.***

Given $\mathcal{P}$, S, spec and f_δ: Identify $\mathcal{P}'$ and S' such that

- *(C1)* $S' \subseteq S$,
- *(C2)* $(s_0, s_1) \in \mathcal{P}' \wedge s_0 \in S' \Rightarrow (s_0, s_1) \in \mathcal{P}$, *and*
- *(C3)* $\mathcal{P}'$ *is* strong *multitolerant to* f_δ *from* S' *for spec.* □

We state the corresponding decision problem as follows:

Definition 5.8.3. ***The Strong Multitolerance Decision Problem.***

Given $\mathcal{P}$, S, spec and f_δ: Does there exist a program $\mathcal{P}'$, with its invariant S' that satisfies the requirements of Definition 5.8.2? □

Now we consider the following question: if the *weak* multitolerance decision problem for the given program $\mathcal{P}$ is affirmative, then is the *strong* multitolerance decision problem for $\mathcal{P}$ affirmative?

To demonstrate the answer to the above question, we discuss the feasibility of stepwise design of *weak* multitolerant programs and compare it with the feasibility of stepwise design of *strong* multitolerant programs in the following combinations.

5.8.1 Feasibility Comparison of FF *Strong/Weak* Multitolerance

In this section, we show that there are instances where adding FF *weak* multitolerance is feasible although adding FF *strong* multitolerance is not feasible. We can show this with a

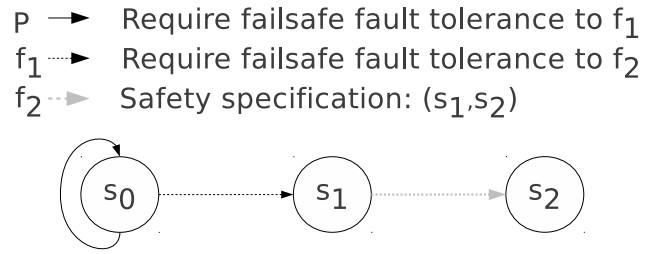

Figure 5.6: A case of FF *Weak* multitolerance (not *Strong*).

simple example illustrated in Figure 5.6. In this example, the input is as follows. The state space of the input program is $\{s_0, s_1, s_2\}$, input program consists of only one transition (s_0, s_0) and its invariant contains only one state s_0. The transition (s_1, s_2) violates safety. The class of fault f_1 includes only one transition (s_0, s_1) and fault f_2 includes only one transition (s_1, s_2).

Clearly, if faults f_1 and f_2 occur in the same computation then safety can be violated from state s_0, the only state in the invariant. Hence, adding FF *strong* multitolerance is not possible in this example, i.e., the answer to the decision problem 5.8.3 is false. However, adding FF *weak* multitolerance is feasible. In fact, the program $\mathcal{P}$ itself is FF *weak* multitolerant.

5.8.2 Feasibility Comparison of MM *Strong/ Weak* Multitolerance and MF *Strong/ Weak* Multitolerance.

In this section, we show that there are instances where adding MM (respectively, MF) *weak* multitolerance is feasible although adding MM (respectively, MF) *strong* multitolerance is not feasible. To illustrate this, we use the input obtained by mapping the SAT formula as discussed in Section 5.4. As shown in Section 5.4, if we begin with a SAT formula that is satisfiable, then the answer to the decision problem for adding MM *weak* multitolerance

(Problem 5.1.2) is affirmative. Next, we show that for this input, the answer to the decision problem for adding MM *strong* multitolerance (Problem 5.8.3) is always false. To show this observe that, in Figure 5.2, any recovery path to the invariant must go through either g_i or h_i from some i ($1 \leq i \leq n$ where n is the number of propositional variables in the instance of the SAT problem in Section 5.4). If f_1 and f_2 occur in the same computation, safety will be violated when either fault transition (g_i, a_i) or (h_i, b_i) is executed. Moreover, the agreement program in Section 5.2.4 is another example of cases where adding strong multitolerance is impossible while weak multitolerance can be designed.

We note that using a similar argument, we can show that there are instances where adding MF *weak* multitolerance is feasible although adding MF *strong* multitolerance is not feasible.

5.8.3 Feasibility Comparison of MN *Strong/ Weak* Multitolerance and NN *Strong/ Weak* Multitolerance.

Since the algorithm that is used to synthesize the MN/NN *weak* multitolerance is the same as that of MN/NN *strong* multitolerance, the synthesis problem of MN/NN *weak* multitolerance and MN/NN *strong* multitolerance have the same feasibility property.

5.9 Discussion

In this section, we discuss issues related to the way we model faults, the practical significance of the proposed work in this chapter and some limitations of our approach.

Our work models the impact of faults on programs as a set of transitions (i.e., a non-deterministic finite-state machine) that perturbs the program state. Designers can identify

the classes of faults dependent upon the domain of application and the requirements of the system users. To generate a fault-class, first we identify the faults that may perturb the program at hand. Fault forecasting methods [142] can be useful to achieve this objective. Then, we formally characterize each of these faults as state perturbations. Finally, we group the faults into fault classes based on the corresponding level of tolerance required to each fault-class. The desired level of tolerance is based on the user requirements and feasibility of providing that level of tolerance under system constraints/resources. As demonstrated in this dissertation, using this method of fault modeling, one can represent Byzantine, crash, message loss, input corruption, node leave and transient faults. Moreover, previous work [15, 71, 163, 164, 186] uses this model to capture other classes of faults such as stuck-at, omission, disk corruption and sensor failures. However, our fault model cannot capture any types of faults that cannot be represented as a finite-state machine (e.g., impact of external disturbances on aircrafts, effect of wing damage on flight control systems, mechanical systems involving several masses, springs and dampers).

Adding *weak* multitolerance enables a method for component-based design of multitolerant programs. Specifically, consider an existing program $\mathcal{P}$ that provides *weak* multitolerance to the fault classes $f_1, \cdots, f_{n-1}$. If designers detect a new fault-class f_n after the design and implementation of $\mathcal{P}$, then it is desirable to have a revised version of $\mathcal{P}$, denoted $\mathcal{P}_C$, that provides *weak* multitolerance to $f_1, \cdots, f_{n-1}$ and f_n. To design $\mathcal{P}_C$, developers have two options: redesign a new multitolerant program from scratch or simply design a component $\mathcal{C}$ and compose it with $\mathcal{P}$ such that the resulting composition preserves fault tolerance to faults f_i, for $1 \leq i \leq n-1$, and enables a specific level of fault tolerance when f_n occurs. In fact, the component $\mathcal{C}$ has to ensure that the computations of $\mathcal{P} \cup f_n$ meet the

requirements of the desired level of fault tolerance; i.e., no guarantees are provided for the computations of the composed program $\mathcal{P}_C$ in the presence of faults $f_1, \cdots, f_{n-1}$. While in our previous work [71] we present stepwise algorithms for the design of strongly multitolerant programs, there are several cases where *strong* multitolerance cannot be added. Moreover, component-based design of *weak* multitolerance is easier since no guarantees are provided for the computations in which multiple faults occur.

The proposed approach in this chapter has several practical and methodological significance regarding the development of concurrent software. First, we present a paradigm shift with respect to the traditional design and verification methods, where developers design fault-tolerant programs first and verify their correctness after the fact. Automated addition of multitolerance is especially beneficial since it is often difficult to anticipate all classes of faults in the early stages of design due to the complex and dynamic nature of today's systems. Thus, the proposed approach enables a systematic method for automated addition of fault tolerance to existing programs. Second, automated addition of *weak* multitolerance exploits computational redundancy before resorting to resource redundancy. In particular, the algorithms presented in this chapter enable the design of several intermediate levels of multitolerance that developers can consider in their risk/cost analysis so using resource redundancy becomes a last-resort option. Third, the notion of *weak* multitolerance provides an impossibility test for designers. That is, if a weakly multitolerant version of an existing program does not exist, then *strong* multitolerance cannot be added to that program too. Fourth, our algorithms can be integrated in model checkers to facilitate the detection and correction of conflicts between several levels of fault tolerance. Last but not least, while our focus in this chapter is on high atomicity programs, the proposed method can be used for the

design of highly resilient network protocols where processing nodes might have read/write restriction with respect to the variables of other nodes. Thus, this work can significantly improve the way network protocols are designed.

The approach presented in this chapter has some constraints in terms of the input to the synthesis algorithms and tool development. First, thus far we have investigated the problem of adding multitolerance for finite-state programs; i.e., the problem of adding multitolerance to infinite-state programs is still open. Second, during the addition of multitolerance our algorithms preserve only the properties that can be captured in the linear topological characterization of specifications by Alpern and Schneider [6]. For instance, if the properties of the intolerant program are specified in the Computation Tree Logic [76], then we do not guarantee that they will be preserved in the absence of faults. Third, the input program should be maximal. That is, from any state, the program should have the maximum number of non-deterministic outgoing transitions. The maximality of the intolerant programs increases the chances of success in adding multiple levels of fault tolerance. For example, consider the program synthesized in Section 5.3.1. While the guard of the action $out == \bot \wedge sensor0 == sensor1 \neq in \longrightarrow out = sensor0$ is never enabled, we included it in the multitolerant program since such transitions may be useful for adding new levels of fault tolerance if new classes of faults are detected. Fourth, our model of programs is an abstract model in that we do not add multitolerance to C/C++/Java programs. Nonetheless, we can exploit the existing model extraction techniques [50,111,227] that are used in model checking where a finite model is generated from a C/C++/Java program and then multitolerance is added to the extracted model. Fifth, we plan to reuse the tools that we have developed for the addition of a single level of fault tolerance to concurrent programs [72] for the automated

design of multitolerant programs. While the time/space complexity of synthesis is a bottleneck for tool development, we have developed distributed [68] and symbolic [37] techniques that increase the scalability of algorithms for the addition of fault tolerance significantly (e.g., for programs with 2^{100} reachable states).

Finally, in cases where the algorithms for adding *weak* multitolerance declare failure, the design of *weak* multitolerance becomes impossible under the constraints/resources of the program at hand. One solution may be to add more redundancy. It is possible to revise the algorithms in this chapter, so additional redundancy could be introduced automatically. However, we believe that addition of redundancy should be handled manually, since it requires resources and an automation algorithm cannot determine whether these resources are reasonable and available. Another possible solution may be to change the expected level of tolerance. For example, if a MM *weak* multitolerance is unfeasible, then system may provide MN *weak* multitolerance. Again, the choice of this depends upon whether the reduced level of tolerance is acceptable to *system users*. As stated in Section 6.1, this involves a tradeoff between the cost and level of fault tolerance. Automation algorithms, like the ones in our work, allow designers to identify a fault-tolerant program with the given choices in terms of redundancy and level of tolerance.

5.10 The Tool RM^2: Model Revision for Adding Multitolerance

In this section, we describe RM^2, a tool for **M**odel **R**evision for adding **M**ultitolerance. RM^2 is an extension of the tool SYCRAFT [37], which is a tool of automatically reparing

program for adding fault-tolerance. It augments SYCRAFT to address the problem of automatically revision for adding multitolerance. In particular, RM^2 considers two types of multitolerance, including *strong* multitolerance [137] and *weak* multitolerance proposed in this chapter. Specifically, given a fault-intolerant program, a set of fault actions and specifications, the tool generates a fault-tolerant program via a symbolic implementation of the respective algorithms.

RM^2 is written in C++ and the algorithms are implemented based on GLU/CUDD package [213], which provides functions to manipulate Binary Decision Diagrams (BDDs). Our tool uses Flex [146] to read the given input files, and employs Bison [56] to parse the grammar. Flex is a tool for generating scanners: programs which recognized lexical patterns in text. Bison is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar.

5.10.1 Input Program Language

Similar to SYCRAFT, every input program to RM^2 consists of eight modules, including declarations for program name, constraint, variables, processes, faults, invariant, safety specification and prohibited transiton predicates. A typical RM^2 input program structure is as follows:

program *Program name*;

const

...

var

...

process *the 1st process name*

... process actions ...

process *the 2nd process name*

... process actions ...

...

endprocess

ffault *the 1st fault name*

... fault actions ...

endffault

sfault *the 2nd fault name*

... fault actions ...

endsfault

invariant

...

specification

...

prohibited

...

Technically, the input program is defined as follows.

$$\begin{aligned}\langle prog\rangle ::= \quad & \langle progdecl\rangle\ \langle constdecl\rangle\ \langle vardecl\rangle\ \langle procdecl\rangle \\ & \langle ffaultdecl\rangle\ \langle sfaultdecl\rangle \\ & \langle invariantdecl\rangle\ \langle specdecl\rangle\ \langle prohibiteddecl\rangle\end{aligned}$$

Module ⟨*progdecl*⟩ declares the program name and is defined as follows.

⟨*progdecl*⟩ ::= **program** ⟨*identifier*⟩

An identifier can be any combination of alphanumeric characters, starting with an alphabet. And, case matters.

Module ⟨*constdecl*⟩ declares constants.The grammar is defined as follows.

⟨*constdecl*⟩ ::= **const** ⟨*constlist*⟩|⟨*empty*⟩

⟨*constdecl*⟩ ::= ⟨*concreteconst*⟩|⟨*constlist*⟩⟨*concreteconst*⟩

⟨*concreteconst*⟩ ::= **int** ⟨*identifier*⟩" = "⟨*guard*⟩";"|

boolean ⟨*identifier*⟩" = "⟨*guard*⟩";"

⟨*empty*⟩ :=

⟨*guard*⟩ can be any combination of quantified Boolean and arithmetic expressions. An example of constant declaration is as follows.

const

int N = 2;

int B = 2;

Module ⟨*vardecl*⟩ declares variables, which are of two types: integer (nonnegative) and boolean. Each interger is declared along with its domain, where the lower bound must be zero. The grammar is defined as follows.

$\langle vardecl \rangle$ ::= **var** $\langle varlist \rangle$

$\langle constdecl \rangle$ ::= $\langle concretevar \rangle | \langle varlist \rangle \langle concretevar \rangle$

$\langle concretevar \rangle$ ::= $\langle booldecl \rangle | \langle intdecl \rangle$

$\langle booldecl \rangle$ ::= **boolean** $\langle identifier \rangle$";"|

boolean $\langle identifier \rangle$"."$\langle range \rangle$";"

$\langle intdecl \rangle$ ::= **int** $\langle identifier \rangle$" : $domain$"$\langle range \rangle$";"|

int $\langle identifier \rangle$"."$\langle range \rangle$" : $domain$"$\langle range \rangle$";"

$\langle range \rangle$ ::= $\langle rangedelim \rangle$".."$\langle rangedelim \rangle$

$\langle rangdelim \rangle$ ::= $\langle guard \rangle$

An example of constant declaration is as follows.

var

boolean bg;

boolean b.0..N;

int i: domain 0..5;

int j.0..N: domain 0..2;

A program consists of a set of processes. Module $\langle processdecl \rangle$ declares process, which includes: (1) a set of actions, (2) a fault section which describes fault actions, (3) a prohibited section which defines a set of actions that is not allowed to execute, (4) a set of readable variables and (5) a set of writeable variables. The synatx of both program actions and fault actions are of form $guard \rightarrow statement$. The grammar is defined as follows.

$\langle procdecl \rangle$ ::= $\langle procstruct \rangle | \langle procdecl \rangle \langle procstruct \rangle$

$\langle procstruct \rangle$ ::= **process** $\langle identifier \rangle \langle procrange \rangle \langle actions \rangle \langle faultdecl \rangle$
$\langle prohibited \rangle \langle rwrestrict \rangle \langle endprocess \rangle$

$\langle procrange \rangle$ ::= ":" $\langle range \rangle | \langle empty \rangle$

In the above definition, $\langle rwrestrict \rangle$ defines the set of variables that are allowed to read and the set of variables that are allowed to write.

Module $\langle faultdecl \rangle$ defines faults that the process is subject to. As illustrated in Chapter 3, fault transitions are modeled using guarded commands. Thus, fault transitions have the same syntax with the program actions. The grammar is defined as follows.

$\langle faultdecl \rangle$ ::= $\langle ffaultdecl \rangle | \langle sfaultdecl \rangle | \langle ffaultdecl \rangle \langle sfaultdecl \rangle | \langle empty \rangle$

$\langle ffaultdecl \rangle$ ::= **ffaults** $\langle actions \rangle$**endffault**$| \langle empty \rangle$

$\langle sfaultdecl \rangle$ ::= **sfaults** $\langle actions \rangle$**endsfault**$| \langle empty \rangle$

Module $\langle invariantdecl \rangle$ defines invariant predicate (see Definition 2.2.4) of the given program. Module $\langle specdecl \rangle$ defines safety specification (see Definition 2.2.1). And, module $\langle prohibiteddecl \rangle$ defines transitions that are not allowed to execute. The given input program may not have prohibited transitions. Specifically, rules for these modules are defined as follows.

$\langle invariantdecl \rangle$ ::= **invariant** $\langle guard \rangle$";"

$\langle specdecl \rangle$ ::= **specification** $\langle guard \rangle$";"$| \langle empty \rangle$

$\langle prohibiteddecl \rangle$ ::= **prohibited** $\langle guard \rangle$";"$| \langle empty \rangle$

5.11 Functionality and Output Program

RM^2 implements algorithms and heuristics of model revision for adding weak multitolerance proposed in this chapter, as well as algorithms and heuristics for adding strong multitolerance in [137].

The output of RM^2 is a fault-tolerant program in terms of unchanged actions, revised actions and recovery actions, as defined in Section 2.4.

5.11.1 Example 1: Two-Sensors Program

The details of this example is in Section 5.3.1. The input file is shown in Figure 5.7. After running RM^2 with paramater *wff*, where *w* denotes weak and *ff* denotes failsafe-failsafe, the output is a fault-tolerant version shown in Figure 5.8.

5.11.2 Example 2: Byzantine Agreement

We now use the Byzantine Agreement problem to describe the functionality of RM^2. Figure 5.9 specifies program actions. Figure 5.10 specifies fault actions. After running RM^2 with paramater *smn*, where *s* denotes weak and *mn* denotes masking-nonmasking, the output is a fault-tolerant version shown in Figure 5.11.

5.12 Summary

In this chapter, we addressed the problem of automatic revision of multitolerant program. The problem of model revision for adding multitolerance is motivated by the observation that a program is often subject to multiple classes of faults and the level of tolerance pro-

```
program SensorFF;
const N = 1;
var
    int inp.0..N: domain 0..1;
    int out: domain 0..2;
process prog
    (out == 2) -> out := inp.0;
    []
    (out == 2) -> out := inp.1;
    read: out, inp.0..N;
    write: out;

endprocess

ffault Corruption
    true -> (inp.0 := 1) [] (inp.0 := 0);
endffault

sfault Corruption1
   true -> (inp.1 := 1) [] (inp.1 := 0);
endsfault

invariant
   ((out == 2) & (forall p, q in 0..N:: (inp.p == inp.q))) |
   (exists a, b in 0..N: (a != b):: ((out == inp.a) & (inp.a == inp.b)));
specification
   (exists a, b in 0..N: (a != b):: ((inp.(a)' == inp.(b)')
      & (inp.(a)' != out') & (out' != 2)));
prohibited
   (out != 2) & (out != out');
```

Figure 5.7: Input file of two-sensors program.

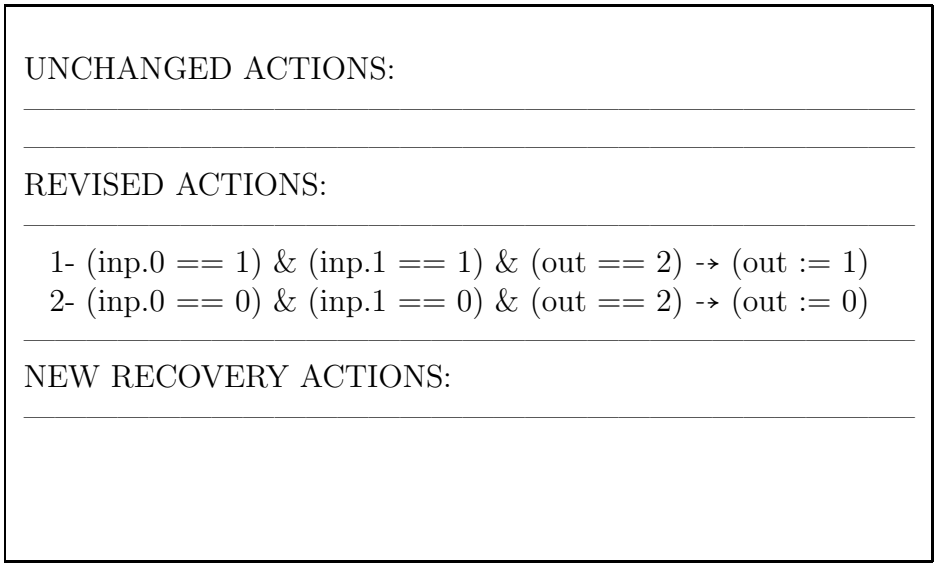

Figure 5.8: Output of two-sensors Program.

vided to them is often different. The problem of adding *weak* multitolerance considers the special case where the faults are independent, i.e., occurrence of faults from multiple classes are unlikely to happen simultaneously and, hence, it suffices to ensure that the program provides the required tolerance to each fault-class. By contrast, the *strong* multitolerance considered in [71] focuses on scenarios where faults from several fault classes can happen simultaneously. For this reason, we illustrated that there are several instances where adding *weak* multitolerance is feasible although adding strong multitolerance is not feasible.

We investigated the complexity of adding *weak* multitolerance, To this end, we considered five possible combinations MM, FM, FF, MN and NN. In each combination, the first letter indicates the fault-tolerance level for the first class of faults, denoted f_1, and the second letter indicates the fault-tolerance level for the second class of faults f_2. We found a surprising result that if masking fault-tolerance is desired for f_1 and masking (or failsafe) fault-tolerance is desired for f_2, then adding *weak* multitolerance is NP-hard (in program state space). This result is counterintuitive since the corresponding problem for adding *strong* multitolerance

```
program Byzantine_Agreement;


const N = 2;

var

boolean bg;
boolean b.0..N;
boolean f.0..N;

int dg: domain 0..1;
int d.0..N: domain 0..2;


process j: 0..N
      {non-Byzantine non-general process copies value from the general }
      ((d.j == 2) & !f.j & !b.j) -→ d.j := dg;
[ ]
       non-Byzantine non-general finalizes decision
      ((d.j != 2) & !f.j & !b.j) -→ f.j := true;
   ffault Byzantine_NonGeneral
      (!bg) & (forall p in 0..N:: (!b.p)) -→ b.j := true;
[ ]
      (b.j) -→ (d.j := 1) [] (d.j := 0) [] (f.j := false) [] (f.j := true);
   endffault
   sfault Byzantine_NonGeneralAnother
      true -→ (d.j := 1) [] (d.j := 0);
   endsfault
   prohibited !b.j & !b.(j)' & f.j & ((d.j != d.(j)') | !f.(j)');
   read: d.0..N, dg, f.j, b.j;
   write: d.j, f.j;
endprocess
```

Figure 5.9: Program actions of the Byzantine agreement program.

```
ffault Byzantine_General

   !bg & (forall p in 0..N:: (!b.p)) -> bg := true;
   [ ]
   bg -> (dg := 1) [] (dg := 0);
endffault

sfault Byzantine_GeneralAnother

   true -> (dg := 1) [] (dg := 0);
endsfault
invariant

   (!bg & (forall p, q in 0..N:(p != q):: ((!b.p) | (!b.q))) &
   (forall r in 0..N:: (!b.r ⇒ ((d.r == 2) — (d.r == dg)))) &
   (forall s in 0..N:: ((!b.s & f.s) ⇒ (d.s != 2))))
   |
   (bg & (forall t in 0..N:: (!b.t))
   & (forall a, b in 0..N:(a != b)::((d.a == d.b)
   & (d.a != 2))));

specification

   (exists p, q in 0..N: (p != q):: (!b.(p)'
   & !b.(q)' & (d.(p)' != 2) & (d.(q)' != 2)
   & (d.(p)' != d.(q)') & f.(p)' & f.(q)')) |
   (exists r in 0..N:: (!bg' & !b.(r)'
   & f.(r)' & (d.(r)' != 2) & (d.(r)' != dg')));
```

Figure 5.10: Fault actions of the Byzantine agreement program.

UNCHANGED ACTIONS:

1- (d0 == 2) & !(f0 == 1) & !(b0 == 1) -→ (d0 := dg)

REVISED ACTIONS:

2- (b0==0) &(d0==0)& (d1==1) & (f0==0) -→ (f0 := 1)
3- (b0==0) &(d0==0)& (d2==0) & (f0==0) -→ (f0 := 1)
4- (b0==0) &(d0==0)& (d1==0) & (f0==0) -→ (f0 := 1)
5- (b0==0) &(d0==0)& (d2==1) & (f0==0) -→ (f0 := 1)

NEW RECOVERY ACTIONS:

6- (b0==0) &(d0==0)& (d1==1) & (d2==1) & (f0==0) -→ (d0 := 1)
7- (b0==0) &(d0==0)& (d2==0) & (d2==0) &(f0==0) -→ (d0 := 0)
8- (b0==0) &(d0==0)& (d1==0) & (d2==1) &(f0==0) -→ (d0 := 1), (f0 := 1)
9- (b0==0) &(d0==0)& (d2==1) & (d2==0) &(f0==0) -→ (d0 := 0),(f0 := 1)

Figure 5.11: Output of two-sensors program.

can be solved in polynomial time. We also presented a sound heuristic for designing MM weakly multitolerant programs. For other combinations FF, MN and NN, we illustrated that the problem of synthesizing *weak* multitolerance is in P. To demonstrate this, we presented a sound and complete algorithm for each combination.

We also investigated the relation between *strong* multitolerance and *weak* multitolerance. Specifically, we argue that if a program is strongly multitolerant then it is also weakly multitolerant, although the reverse is not necessarily true. Moreover, we identify circumstances where solvability of adding *weak* and *strong* multitolerance differs. We show that (1) there are situations where adding FF *weak* multitolerance is feasible although adding FF *strong* multitolerance is not feasible; (2) there are instances where adding MF *weak* multitolerance is feasible although adding MF *strong* multitolerance is not feasible, and (3) the synthesis

problem of MN/NN *weak* multitolerance and MN/NN *strong* multitolerance have the same feasibility property. Besides, we implemented the algorithms for automatically repairing programs for adding multitolerance in the tool RM^2. Our tool augments the exsiting tools for the problem of model revision for adding multitolerance.

Chapter 6

Automatic Revision of UML State Diagrams

6.1 Introduction

The utility of formal methods in the development of high assurance systems has gained widespread use in some segments of industry. In general, there are two main approaches to utilizing formal methods in providing assurance: *correct-by-verification* and *correct-by-construction*. The first approach, *correct-by verification*, is the most commonly used approach. In this approach, one begins with an existing model (or program) and a set of properties (specification), and verifies that the given program meets the given properties of interest. An embodiment of this approach is *model-checking* [26,42,110,143]. *Model checking* has been widely studied in the literature and proved effective in identifying bugs in system design. However, a pitfall of this approach is that if the manually designed model does not satisfy the requirements then it is often unclear how one can proceed further. Hence, for

scenarios where an existing model needs to be revised to deal with the new environment, the new identified faults, or the new requirements, one needs to manually develop the new model if one needs to obtain assurance via modeling checking.

The second approach, *correct-by-construction*, utilizes the specification of the desired system and constructs a model that is correct. Examples of this approach include [10, 16, 22, 77, 139, 169]. These works differ in terms of the expressiveness of the specifications they permit and in terms of their complexity. This approach has proven effective in constructing a model that meets the requirements of specification. However, a pitfall of this approach is the loss of reuse (of the original model) and a potential for significant increase in complexity.

To obtain the benefits of these two approaches while minimizing their pitfalls, one can focus on an intermediate approach, *model revision*. Model revision deals with the problem where one is given a model/program and a property. And, the goal is to revise the model to meet the requirement of the given property. Applications of model revision include scenarios where model checking concludes that the model violates the requirements specified by the given property. Other applications include scenarios where an existing model needs to be revised due to changes in requirements and/or changes in the environment. For this reason, model revision has been studied in contexts where an existing model needs to be revised to add new fault-tolerance properties, safety properties, liveness properties and timing constraints [35].

Since model revision generates a model through correct-by-construction, it provides assurance comparable to *correct-by-construction* approaches. Also, since it begins with an existing model, it has the potential to reuse the existing model during the revision process. Moreover, there are several instances where complexity of model revision is comparable to

that of model checking [35].

To permit wider utilization of model revision in practice, it is desirable to reduce the learning curve faced by the designer in applying it. In particular, it is especially valuable if the details of the model revision are hidden from the designer so that the designers continue to work with a framework that they are familiar with.

One challenge of exploring such an approach of applying the model revision on the program design is that the current model-based design is often modeled in a non-computational way, such as UML [199]. UML is a well-known modeling language utilized by industry, with focus on system architecture as a means to organize computation, communication and constraints. Of UML diagram sets, state diagram is especially helpful when designers discuss the logic architecture and workflow of the whole system with the need of independence from a particular programming language. Since the UML state diagram is able to illustrate the high-level overview of the whole system, it is widely used to model program design. With this motivation, we propose a framework, namely MR4UM, for applying model revision on program design modeled in the UML state diagram. To overcome the challenge of applying model revision on UML state diagram, MR4UM proposes an automatic transformation mechanism from UML state diagram to the corresponding model in the underlying computational model (UCM). Subsequently, MR4UM applies model revision on the program modeled in the UCM and generates a revised program modeled in the UCM. Finally, MR4UM converts the revised program modeled in UCM into the corresponding program design modeled in UML state diagram.

6.2 Motivating Scenario

In this section, we present a motivating scenario from automotive industry. An adaptive cruise control program (ACC) in automotive system is designed to control the distance between the vehicle and the front vehicle (called leader car) automatically. Figure 6.1 describes the logic design of the ACC program.

As shown in Figure 6.1, when ACC program is on after initialization, the system enters one of the three modes: *active*, *ACC_active* or *inactive*. In the *active* mode, the sensor keeps checking whether a leader car exists within a predefined safe distance and the ACC program keeps reading the sensor result. In the *ACC_active* mode, the ACC program is working to control the distance between the leader car and the current car because a leader car is detected within the predefined safe distance. In the *inactive* mode, the driver is pressing the brakes and the adaptive cruise control relinquishes control to driver for manual control without considering the effect of cruise. By switching among these three modes, the ACC system targets to ensure that the leader car is at a desired distance away and the relative speed of the current car is zero with respect to the leader car. Moreover, if the leader car does not exist then the system resumes the previous speed chosen by the driver.

Initially, when no leader car is detected by the sensor, the program stays in *active* mode. If a leader car is detected within the predefined safe distance, the ACC program enters into *ACC_active* mode. ACC program stays in *ACC_active* mode until the leader car goes away or the drive takes the brake. If the leader car goes away from the detectable distance, the ACC system goes back to the *active* mode. Under *ACC_active* mode or *active* mode, the ACC system enters into *inactive* mode when driver taps the brake. When driver stops tapping the brake and presses resume button, the ACC system enters into *active* mode if no leader car is

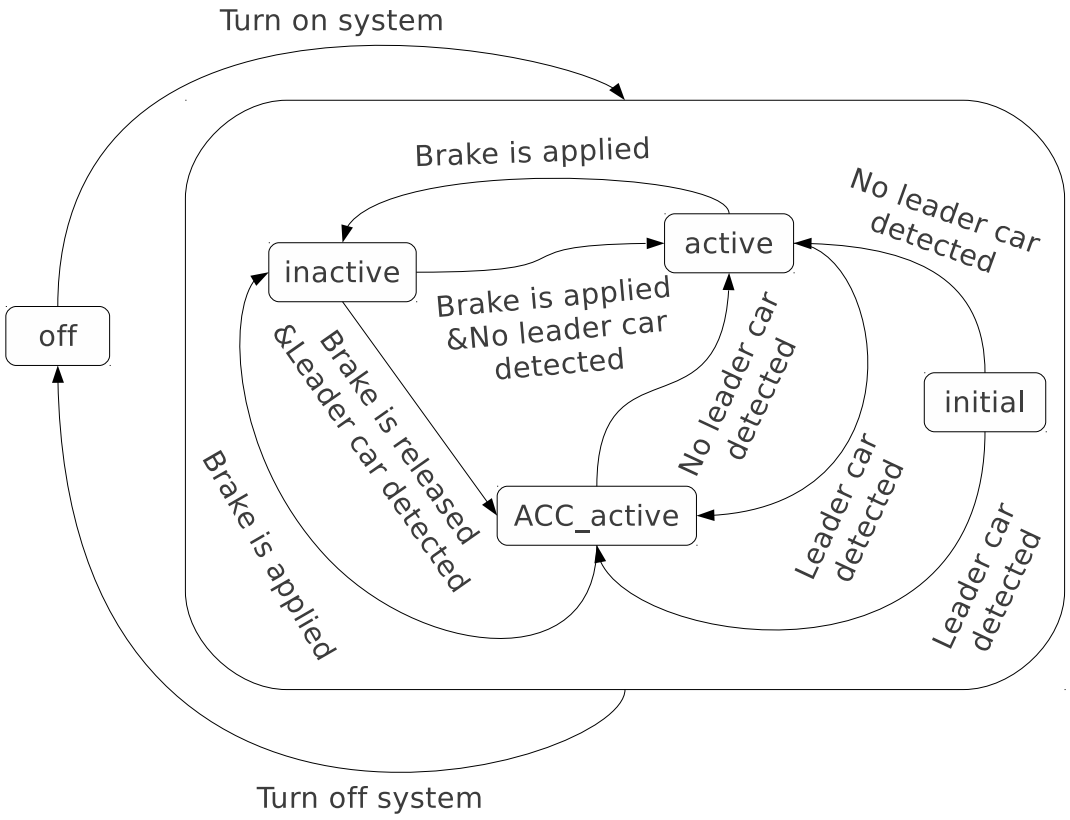

Figure 6.1: Logic design of the ACC program.

detected and *ACC_active* mode if a leader car is detected. (For simplicity, we do not model the *resume* action. We simply assume that releasing the break is equivalent to resume.)

6.2.1 Need for Model Revision for Tolerating Sensor Failure

While the ACC program in Figure 6.1 works correctly in the absence of faults, it may result in undesired behavior if some faults affect the sensor. Specifically, a sensor failure may cause two problems: *false positive* and *false negative*. A *false positive* sensor may cause the sensor to detect a non-existing leader vehicle causing the system to change the state from *active* to *ACC_active*. This would potentially cause the car to slow down unnecessarily to prevent collision with a fictitious car.

A more serious error can result in a *false negative* sensor that fails to detect a leader

vehicle. In this scenario, the car would stay in *active* mode; thereby potentially cause a collision with the leader car.

For the above reasons, the model in Figure 6.1 needs to be revised to deal with such false alarms (*false positive* & *false negative*). To tolerate the false alarm (*false positive* and *false negative*) caused by the faulty sensor, one typical fault tolerance policy is to provide redundancy. Thus, if the redundancy policy is chosen to tolerate the sensor failure, the problem of revising program design to resist false alarm is to modify the previous system design to utilize the sensor redundancy. After revision, the new system design should get correct information about whether the leader car exists, even in the presence of the false alarm of one sensor.

The goal of MR4UM is to facilitate such a revision with minimal overhead to the designer. Clearly, during such revision process, the designer would need to specify the original (fault-intolerant) model. Moreover, this model should be in the format they utilize already. This does not introduce a new overhead since the problem of model revision is applicable in scenarios where the designer already has a model. The designer also needs to specify the specific nature of faults. Towards this end, MR4UM provides a mechanism to describe commonly encountered faults. Subsequently, the designer utilizes MR4UM to generate the corresponding fault-tolerant model for the ACC program.

Thus, MR4UM is proposed to start with the (fault-intolerant version of) UML state diagram. Then, MR4UM converts the fault-intolerant UML state diagram into the corresponding underlying computational model (UCM) automatically. This conversion is annotated to facilitate generation of revised UML model in the last step. Additionally, MR4UM facilitates the description of faults to minimize the designer overhead. Next, MR4UM applies a

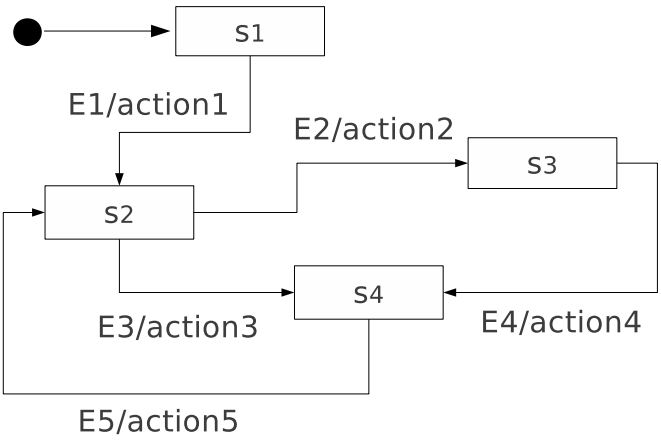

Figure 6.2: A case to illustrate modeling program in UML state diagram.

Model Revision algorithm on the program in UCM to add fault-tolerance. Finally, MR4UM converts the fault-tolerant program in UCM into the corresponding UML state diagram with the use of annotations from the first steps.

6.3 An Overview of UML State Diagram

UML is a standardized general-purpose modeling language in the field of object-oriented software engineering. Specifically, UML state diagrams enable us to visualize the program design. Moreover, UML state diagrams enable us to capture any form of fault-tolerance that can be expressed in a state machine-based formalism [70]. In this section, we introduces the background knowledges of UML state diagram involved in our framework.

A UML state diagram is a directed graph. The nodes in this graph denote states, and the edges in this graph represent transitions. The graph could represent potential execution scenario. Specifically, the modeling of individual states and transitions include:

1. **State.** The state is represented as rounded rectangle. The initial state (if any) is denoted by a filled circle. The final state (if any) is denoted as a hollow circle containing

a smaller filled circle. The states may also be hierarchical in nature. In particular, a hierarchical state can be decomposed into several states. Currently, our framework only considers the states at the lowest level. However, it can be easily extended to utilize the hierarchical model and obtain the corresponding states in lower level.

2. **Transition.** The transition between states is represented with an arrow. Each transition is associated with a triggering event followed by the list of executed actions. A special transition, denoted as the initial transition originates from the solid circle (initial state). It specifies the default state when the model first begins.

As an illustration, the UML state diagram in Figure 6.2 consists of five states: *initial state*, $s1$, $s2$, $s3$ and $s4$, and six transitions, as follows:

1. The transition from initial state to s1;
2. The transition from s1 to s2 (the triggering condition is E1 and action1 happens concurrently with this transition);
3. The transition from s2 to s3 (the triggering condition is E2 and action2 happens concurrently with this transition);
4. The transition from s2 to s4 (the triggering condition is E3 and action3 happens concurrently with this transition);
5. The transition from s3 to s4 (the triggering condition is E4 and action4 happens concurrently with this transition);
6. The transition from s4 to s2 (the triggering condition is E5 and action5 happens concurrently with this transition);

Remark 6.3.1. *The UML state diagram is capable of specifying finite state machines that are important for event driven programming. And, the UML state diagram has the potential to cut down on the number of execution paths through the code and simplify switching between different modes of execution [200].*

6.4 Framework Description

The workflow of MR4UM consists of four steps, including: 1) step A, converting the UML state diagram into the underlying computational model (UCM); 2) step B, identifying the effect of faults; 3) step C, utilizing model revision to add fault tolerance to program and 4) step D, converting the revised program in UCM into UML state diagram while utilizing the annotations generated in Step A.

6.4.1 Step A: Translating from UML State Diagram to UCM.

In order to utilize the underlying model revision machine, we translate the given UML state diagram into the corresponding UCM model. This step utilizes syntactic transformation of the UML state diagram. In particular, for every syntactic feature of the UML state diagram, we convert it into the corresponding UCM. This step is automated with the help of following rules:

- **Rule 1: Translation of states in UML state model.** For all the states in the UML state diagram, we introduce variable STATE with integer domain $[0, n-1]$ where n is the number of states in the UML. All the states in the UML state diagram are numbered from 0 to n-1. For each state, it is mapped to a concrete value assignment of variable STATE. If UML model consists of hierarchical states then this rule is applied

to the states at the lowest level. For example, for state 0 in the UML state diagram, it is mapping into STATE==0 .

- **Rule 2: Translation of trigger conditions.** For each trigger condition c mentioned in the UML state diagram, we introduce one variable X_C with domain $\{0, 1\}$. $X_C = 0$ denotes the negative of this trigger condition is satisfied. $X_C = 1$ denotes this trigger condition is satisfied. Note that the state space is now defined in terms of the STATE variable declared in the previous rule and the declaration of variables from this step.

- **Rule 3: Translation of transitions.** For each transition in the UML state diagram, we introduce one corresponding program transition P. The guard of P is the conjunction of the corresponding STATE assignment of source state and the corresponding variables of each trigger conditions. The action changes the assignment of STATE according to the target state of the original transition in the UML.

Observe that the application of these rules will facilitate the translation of the UML model into the underlying computational model. Since this model is obtained by the above rules, it is straightforward to observe that it has the same behavior as the original UML model.

6.4.2 Step B: Generating Fault Actions, Specification and Invariants from Parameters specified by Designer

After Step A, we have program actions modeled in UCM. To revise the program design to satisfy the new specification, we need (1) fault actions modeled in UCM, (2) specification,

that is, requirements in the presence of faults, and (3) states where program should recover after faults occurs. Moreover, these parameters are closer in spirit to the underlying computational model. For this reason, asking the designer to generate these in UCM is not desired. Unfortunately, they cannot be derived automatically either. For this reason, our framework facilitates modeling of typical faults that one may encounter in the revision process. Next, we describe how we obtain these inputs in our framework.

1. **Fault Actions Modeled in UCM.** In our framework, *fault actions* are automatically generated from parameters which are specified by designer from GUI. From GUI, designer needs to specify the following parameters:

   (a) *What type of faults?* Currently, there are three types of faults modeled in our framework: (1) Byzantine, (2) transient and (3) crash (failstop). The Byzantine fault allows the fault to change the affected component (variables of the component) in an arbitrary manner. Moreover, this fault can perturb the program several times. A transient fault perturbs the component (variables of the component) once (respectively, finitely many times where the bnound is known upfront. And, crash disables certain variables from being updated and, hence, captures the notion that a component (containing those variables) has crashed. The default setting of our framework is *transient*.

   (b) *Effect of faults on program.* Designer needs to specify the variables the fault affects after specifying types of faults that may occur during the execution: In particular,

      i. *Byzantine.* For this type of fault, designer needs to specify which variable(s) may be corrupted by the Byzantine component and the possible value(s). For

example, in the motivating scenario (see Section 6.2), the variable representing the leader is affected by faults. And, this fault can perturb the program to an arbitrary state. Hence, the default action for this fault is that the variable can be corrupted to any value in its domain. However, if the likely fault is only a false positive, then, the fault action could perturb leader value to 1.

ii. *Transient.* For this type of fault, designer needs to specify which variable is perturbed to the random value. The default for this fault is that the variable can be corrupted to any value in its domain. The difference between the transient and Byzantine fault is that the former perturbs the program once whereas the latter could perturb the program a finite number of times.

iii. *Crash or Failstop.* For this type of fault, designer needs to specify which variables are prevented from update due to the fault. In our framework, we use the *crash* faults to denote the fault that is not detectable. By contrast, *failstop* is detectable.

(c) *Number of occurrences of faults.* Designer also needs to specify the occurrences of the specified faults. In case of Byzantine/crash faults, the number denotes the number of Byzantine/crashed components. In turn, this determines the required level of redundancy. Regarding transient faults, the number denotes the occurrences of transient faults that may occur during the computation. The default setting value is 1.

2. **Specification, that is, Requirements in the Presence of Faults.** In our framework, *specification* is automatically generated from parameters which are specified by designer from GUI. Designer needs to specify each state with variables and corre-

sponding values. The union of the specified states from GUI are used to generate the specification automatically.

3. **States Where Program should Recover after Fault Occurs.** The states where program should recover after faults occurs are generated automatically from initial states (by performing reachability analysis) specified in UML state diagram.

6.4.3 Step C: Model Revision for Adding Fault-tolerance

In this step, we utilize the program generated in Step A and the faults, specification and invariant generated in Step B to obtain a fault-tolerant program in UCM. Since UCM was chosen due to its compatibility with the synthesis tool [36], we utilize it to obtain the fault-tolerant program.

Next, we review the model revision algorithm. This algorithm begins with four inputs, p (original program in UCM). The details of algorithm are shown in Figure 6.3. This algorithm consists of five steps, as follows:

1. **Initialization (Lines 1-3).** In this step, we identify state and transition predicates from where execution of faults alone may violate safety specification. Specifically, if (s_1, s_2) is a fault transition that violates safety then the program should never reach s_1. This is due to the fact that if the program reaches s_1 then execution of fault action can violate safety. Furthermore, if (s_0, s_1) is also a fault transition then s_0 should also not be reached. Otherwise, execution of two fault transitions would violate safety. Continuing this process (with backward reachability in fault transitions), we obtain the set ms such that the fault-tolerant program should not reach a state in ms. For the same reason, the program should not include a transition that reaches ms. Likewise,

it should not include transitions that violate safety. Hence, mt denotes the set of these transitions that the program should not include.

2. **Identification of Fault-span (Lines 9-11).** In this step, we identify the fault-span, that is, the reachable states by the program in the presence of faults starting from the program invariant.

3. **Identifying and removing unsafe transition (Line 12-13).** In this step, we identify and remove transitions in mt. It turns out that in several scenarios, the transitions of the fault-tolerant program need to be executed with partial information. Consider the example in Section 6.2, the program cannot read the variable that denotes whether the leader car actually exists. However, such a variable is needed during modeling so we can verify that the transitions of the adaptive cruise control system are consistent with each other. Hence, any time a transition is removed/added, we need to add the corresponding *Group* of transitions; specifically, this group is obtained by changing the values of variables that the program cannot read.

   Unfortunately, this restriction of adding and removing groups causes the complexity of model revision. Hence, we utilize the following heuristics.

   Heuristic H_1: $\mathcal{P}'$ may include $(s_0, s_1), s_0 \in ms$.

   Reasoning behind H_1. H_1 is based on the premise that the algorithm will ensure that the program never reaches a state in ms. Hence, even if we include such transitions (e.g., because they are grouped with otherwise useful transitions), it will not cause any problems. Moreover, such inclusion is helpful to increase the success of model revision when (s_0, s_1) is grouped with some other transitions that is desirable in the $\mathcal{P}'$.

Heuristic H_2: $\mathcal{P}'$ may include (s_0, s_1) where $(s_0, s_1) \in mt$ and s_0 is not reachable by transitions in $\mathcal{P}'$ starting from $\mathcal{I}'$ in the presence of faults.

Reasoning behind H_2. H_2 is based on the premise that if the current version of $\mathcal{P}$ does not reach s_0 then it is likely to be true in the final program as well. Hence, including transition (s_0, s_1) is expected to be acceptable. If at some later point, state s_0 is reachable then this transition may be removed. The fault-span computed in this step is used to determine whether state s_0 is reachable in the presence of faults.

4. **Resolving deadlock states (Line 15-18).** To ensure that no new finite computations are introduced to the input fault-intolerant program, we resolve deadlock states in this step by either adding recovery path or eliminating states. First, we attempt to add recovery to the invariant. This recovery could be achieved in a single step or in multiple steps. Only if the recovery is not feasible then we remove transitions so that the deadlock state is not reached. In this step, we use the following heuristic.

   Heuristic H_3. Given a deadlock state $s, s \notin \mathcal{I}'$, $\mathcal{P}'$ either includes a recovery action $(s, s_I), s_I \in \mathcal{I}'$, or makes s unreachable from $\mathcal{I}'$ without eliminating any invariant states.

   Reasoning behind H_3. H_3 is based on the principle that $\mathcal{P}'$ would not eliminate any states and/or transitions unless absolutely required to do so. This is due to the fact that if we remove several states from the invariant then it may be impossible to satisfy specification in the absence of faults. Hence, invariant states are removed only as a last resort.

5. **Re-computing the invariant (Line 20).** In this step, we recomputed the program

invariant due to identifying offending states during state elimination.

The algorithm keeps repeating steps until the three fixpoints in Line 14, 19 and 20 are reached. The algorithm terminates when no progress is possible in all the steps.

Algorithm 6: Model Revision for Adding Fault-tolerance

Input: $\mathcal{P}$:transitions,f: fault transitions
spec: safety specification , $\mathcal{I}_{\mathcal{P}}$: invariant predicate

Ouput: If successful, a fault-tolerant $\mathcal{P}'$ with invariant S' that is fault-tolerant to f

ms:=$\{s_0 : \exists s_1, s_2, \ldots s_n : (\forall j : 0 \leq jn : (s_j, s_j + 1 \in f)$
$\wedge(s_n - 1, s_n)$ violates $spec\}$; (1)
mt:=$\{(s_0, s_1) : ((s_1 \in$ ms$) \vee (s_0, s_1)$ violates $spec)\}$; (2)
$\mathcal{I}_1$,fte:= $\mathcal{I}_{\mathcal{P}}$ - ms, false ; (3)
REPEAT (4)
$\mathcal{I}_2 := \mathcal{I}_1$; (5)
REPEAT (6)
$S_1, \mathcal{P}_2 := \mathcal{I}_1, \mathcal{P}_1$; (7)
REPEAT (8)
$S_2 := S_1$; (9)
$S_1 :=$ FWReachStates($\mathcal{I}_1$, $\mathcal{P}_1 \vee f$); (10)
$S_1 := S_1$ - fte; (11)
mt:= mt $\wedge S_1$; (12)
$\mathcal{P}_1$:= $\mathcal{P}_1$ - Group($\mathcal{P}_1 \wedge mt$; (13)
UNTIL $S_1 = S_2$; (14)
$ds := \{s_0 | s_0 \in S_1 \cup (\forall s_1 : s_1 \in S : (s_0, s_1) \notin \mathcal{P}_1\}$; (15)
$\mathcal{P}_1 :=$ $\mathcal{P}_1 \vee$ AddRecovery $(ds, \mathcal{I}_1, S_1, mt)$; (16)
$ds := \{s_0 | s_0 \in S_1 \cup (\forall s_1 : s_1 \in S : (s_0, s_1) \notin \mathcal{P}_1\}$; (17)
$\mathcal{P}_1$, fte:= Eliminate$(ds, \mathcal{P}_1, \mathcal{I}_1, S_1, f, false, false)$; (18)
UNTIL $\mathcal{P}_1 = \mathcal{P}_2$; (19)
$\mathcal{P}_1, \mathcal{I}_1 :=$ ConstructInvariant$(\mathcal{P}_1, \mathcal{I}_1, fte)$; (20)
UNTIL $\mathcal{I}_1 = \mathcal{I}_2$; (21)
$\mathcal{I}_{\mathcal{P}} = \mathcal{I}_1$, $\mathcal{P}' = \mathcal{P}_1$; (22)
RETURN $\mathcal{I}_{\mathcal{P}'} = \mathcal{P}'$; (23)

Figure 6.3: Model revision for adding fault-tolerance.

Thus, at the end of this step, we have a fault-tolerant program in UCM that is obtained by the corresponding input from Steps A and B. Moreover, as mentioned earlier, the output in this step consists of: (1) original transitions that are preserved as is, (2) original transitions

that are strengthened and (3) recovery transitions.

6.4.4 Step D: Translating the Revised Program in UCM to UML state diagram.

Till now, we have obtained the revised program modeled in UCM, including (1) *original* program actions, (2) *revised* program actions and (3) *recovery* program actions. In step D, the fault-tolerant UML state diagram is generated as follows:

1. First, we utilize the *revised* program actions and *recovery* program actions to identify the *changed* transitions in the original UML state diagram.

2. Second, we re-annotate these transitions in the UML state diagram by the guard conditions of these *revised* program actions and *recovery* program actions.

Thus, after the completion of all four steps, we obtain the fault-tolerant UML state diagram.

6.5 Case Study 1: The Adaptive cruise control system

In this section, we present the stepwise application of MR4UM on the case of the adaptive cruise control (ACC) system introduced in Section 6.2. In particular, we begin with the fault-intolerant UML model for this case study and apply our framework to generate a fault-tolerant UML model that satisfies Problem 2.4.1. This case study is organized as follows: First, in Section 6.5.1, we describe UML state diagram for the fault-intolerant ACC program. In Section 6.5.2, we describe how MR4UM transforms this UML state diagram into program actions modeled in UCM. In Section 6.5.3, we describe the user inputs for generating the fault

model, specification and invariant. Subsequently, in Section 6.5.4, we describe how MR4UM utilizes model revision algorithm to add fault tolerance to the input program. Finally, in Section 6.5.5, we show how MR4UM transforms the fault-tolerant program obtained by model revision into the corresponding fault-tolerant UML state diagram.

6.5.1 Fault-intolerant UML model for ACC

The model of the ACC system design in the UML state diagram includes five states, namely active, ACC_active, inactive, initial and off. The *active* state captures the status of ACC system in "active" mode. The *ACC_active* state captures the status of the ACC system in "ACC_active" mode. The *inactive* state captures the status of the ACC system in "inactive" mode. The *initial* state captures the status of the ACC system in the initializing process. The *off* state captures the status that the ACC system is turned off.

The state diagram in Figure 6.4 visualizes model design of the ACC system. For better understanding, Figure 6.5 labels out formal expression of the corresponding annotation. As shown in Figure 6.4, when the ACC system is turned on, the system will enter into the *active* state after initialization if there is no leader car detected. The system will enter into the *ACC_active* state if the leader car exists according to the information from sensor system. In other words, existence (or nonexistence) of leader car is the trigger condition to change system state between *active* and *ACC_active*. When the brake is applied, irrespective whether it were in *active* or *ACC_active*, the ACC system enters into *inactive* state. When the brake is released, the system will change from *inactive* state into *active* state or *ACC_active* state depending upon the existence of leader car. The whole ACC system will continue to stay in one of these three states until the system is turned off.

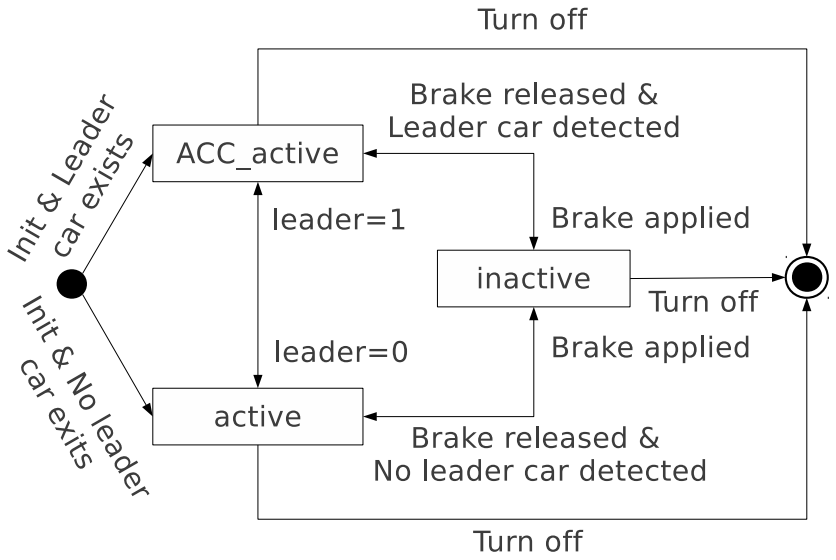

Figure 6.4: ACC system modeled in UML state diagram.

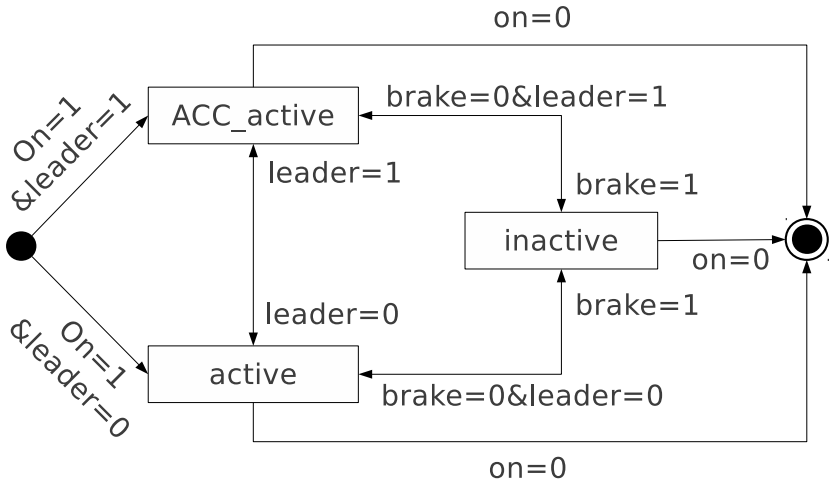

Figure 6.5: Annotation in formal expression.

As one can observe in the absence of faults, this program works correctly. Anytime a leader car exists, the system enters into *ACC_active* to ensure that the current car maintains a safe constant distance to the leader car. And, when the leader moves away, the system enters into *active* to ensure that the original speed is resumed.

6.5.2 Application of Step A: Generating UCM of the ACC System

The UML model for the ACC system is as shown in Figure 6.5. Based on the transformation rules, the corresponding UCM needs four variables, namely *state*, *on*, *brake* and *leader*. The details of these variables are as follows:

1. *state*. Based on the number of state in the UML state diagram in Figure 6.5, the domain of *state* is $\{0, 1, 2, 3, 4\}$. This variable models the five states of the ACC system. $state = 0$ denotes the initial status of the system when it is turned on. $state = 1$ denotes the system is in *active* status. $state = 2$ denotes the system is in ACC_active status. $state = 3$ denotes the system is in *inactive* status. And, $state = 4$ denotes the status that the system is turned off.

2. *on*. The domain of variable *on* is $\{0, 1\}$. It is used to denote whether the ACC system is turned on. When the ACC system is turned on, the variable *on* is assigned with 1, otherwise 0. $on = 0$ models the trigger condition that causes the ACC system enters into the stop status.

3. *leader*. The domain of variable *leader* is $\{0, 1\}$. It is used to model whether a leader car is detected by the sensor system. $leader = 1$ denotes that a leader car is detected. $leader = 0$ denotes that no leader car is detected.

4. *brake*. The domain of variable *brake* is $\{0, 1\}$. It is used to denote whether the brake is applied by the driver. $brake = 1$ models the event that the brake is applied during the execution of ACC system. $brake = 0$ models the event that the brake is released during the execution of ACC system.

Program actions of the ACC system are generated as follows:

1. $state = 0$ & $on = 1$ & $leader = 0 \longrightarrow state := 1$;
2. $state = 0$ & $on = 1$ & $leader = 1 \longrightarrow state := 2$;
3. $state = 1$ & $leader = 1 \longrightarrow state := 2$;
4. $state = 2$ & $leader = 0 \longrightarrow state := 1$;
5. $state = 1$ & $brake = 1 \longrightarrow state := 3$;
6. $state = 2$ & $brake = 1 \longrightarrow state := 3$;
7. $state = 3$ & $brake = 0$ & $leader = 0 \longrightarrow state := 1$;
8. $state = 3$ & $brake = 0$ & $leader = 1 \longrightarrow state := 2$;
9. $state = 1$ & $on = 0 \longrightarrow state := 4$;
10. $state = 2$ & $on = 0 \longrightarrow state := 4$;
11. $state = 3$ & $on = 0 \longrightarrow state := 4$;

In the above program actions, Action 1 models the transition from *initial* state ($state = 0$) to *active* state ($state = 1$). The triggering condition of this transition includes: 1) no leader car exists within the predefined safety distance and 2) the ACC system is on. This condition is modeled as $on = 1 \& leader = 0$. Action 2 models the transition from *initial* state to $ACC\_active(state = 2)$. The triggering condition is that a leader car is detected within the predefined safety distance when the system is on, that is, $on = 1 \& leader = 1$. Action 3 models the transition from *active* ($state = 1$) to ACC_active ($state = 2$). The triggering condition for this action is a leader car is detected within the predefine safety distance. Action 4 models a reverse transition of action 3, that is, from ACC_active ($state = 2$) to

active ($state = 1$). The triggering condition for this action is no leader car is detected within the predefined safety distance. Actions 5 and 6 models the transition from *active* (or *ACC_active*) to *inactive*. The triggering condition for both actions is that driver is pressing brakes, that is, $brake = 1$. Actions 7 and 8 model the transitions from state *inactive* to *active* (or *ACC_active*). The triggering condition for these two actions is brakes are released and no leader car is detected within the predefined safety distance. Actions $9-11$ model the transitions from state *active*, *ACC_active* or *inactive* to state *off*($state = 4$). The triggering condition for these three actions is that the system is turned off.

6.5.3 Application of Step B: Generating Remaining Inputs for Model Revision

In this framework, the faults cause the sensor to provide an incorrect value. This can be modeled with Byzantine faults. Moreover, the number of occurrences of this fault is at most 1. And, the fault affects the leader variable from Figure 6.5.

Since the Byzantine fault affects at most one leader variable, we need a redundancy of three, i.e., we need variables, $leader1$, $leader2$ and $leader3$. Observe that since the redundancy is added for the leader variable, in the absence of faults, all leader values are equal. Hence, $leader1$ can be perturbed from such a state. Moreover, since at most one fault can occur, $leader1$ cannot be perturbed further. If two occurrences of faults were permitted, this would need to be changed so that $leader1$ could be perturbed even if some other (and only 1) leader variable were corrupted. Observe that by performing this analysis, it is possible to generate the guard that identifies when $leader1$ (respectively, $leader2$ and $leader3$) are corrupted. Based on this the fault actions can be modeled as follows:

1. $leader1 == leader2 == leader3 \longrightarrow\ leader1\ :=\ 0[]leader1\ :=\ 1;$

2. $leader1 == leader2 == leader3 \longrightarrow\ leader2\ :=\ 0[]leader2\ :=\ 1;$

3. $leader1 == leader2 == leader3 \longrightarrow\ leader3\ :=\ 0[]leader3\ :=\ 1;$

where [] denotes the non-deterministic execution of statement.

We use an auxiliary variable *car* to denote whether there is a car in front of the current car. The value of the variable *car* is only included for modeling purpose. If the value of the sensors are not corrupted by fault, the value will be equal to the variable *car*. Based on the requirement of the ACC, transitions between *Active* and *ACC_active* must take the status of *car* into account. Moreover, this has to be done without utilizing the variable *car* in the revised program. Thus, the set of states the program should not reach are as follows:

$((car == 1)\&(state'! = state)\&(on == 1)\&(brake == 0)\&(state'! = 2))|$
$((car == 0)\&(state'! = state)\&(on == 1)\&(brake == 0)\&(state'! = 1));$

The invariant, that is, states where program should recover after fault occurs, is as follows. Note that the above predicate is generated automatically from the initial state of the UML model.

$(((car == 1)\&(on == 1)\&(brake == 0)\&(state == 2)$
$(((leader1 == 1)\&(leader2 == 1)\&(leader3 == 0))|;$
$((leader1 == 1)\&(leader3 == 1)\&(leader2 == 0))|;$
$((leader2 == 1)\&(leader3 == 1)\&(leader1 == 0))|;$
$((leader2 == 1)\&(leader3 == 1)\&(leader1 == 1))));$

$(((car == 0)\&(on == 1)\&(brake == 0)\&(state == 1)$

$(((leader1 == 0)\&(leader2 == 0)\&(leader3 == 1))|;$

$((leader1 == 0)\&(leader3 == 0)\&(leader2 == 1))|;$

$((leader2 == 0)\&(leader3 == 0)\&(leader1 == 1))|;$

$((leader2 == 0)\&(leader3 == 0)\&(leader1 == 0))));$

6.5.4 Application of Step C: Generation of Fault-Tolerant UCM

In Step C, we utilize the inputs from Sections 6.5.2 and 6.5.3. Since this step utilizes the tool SYCRAFT [36], we only provide the output of the synthesized program as follows: (Note that this output is only intended for use in Step D and not meant for designer to analyze it.)

$\mathcal{T}_O$:

$state = 1$ & $on = 0 \longrightarrow state := 4;$

$state = 2$ & $on = 0 \longrightarrow state := 4;$

$state = 3$ & $on = 0 \longrightarrow state := 4;$

$\mathcal{T}_S$:

$state = 0$ & $on = 1$ & $leader1 = 0$ & $leader2 = 0 \longrightarrow state := 1;$

$state = 0$ & $on = 1$ & $leader1 = 0$ & $leader3 = 0 \longrightarrow state := 1;$

$state = 0$ & $on = 1$ & $leader2 = 0$ & $leader3 = 0 \longrightarrow state := 1;$

$state = 0$ & $on = 1$ & $leader1 = 1$ & $leader2 = 1 \longrightarrow state := 2;$

$state = 0$ & $on = 1$ & $leader1 = 1$ & $leader3 = 1 \longrightarrow state := 2;$

$state = 0$ & $on = 1$ & $leader2 = 1$ & $leader3 = 1 \longrightarrow state := 2;$

$state = 1$ & $leader1 = 1$ & $leader2 = 1 \longrightarrow state := 2;$

$state = 1$ & $leader1 = 1$ & $leader3 = 1 \quad \longrightarrow state := 2;$

$state = 1$ & $leader2 = 1$ & $leader3 = 1 \quad \longrightarrow state := 2;$

$state = 2$ & $leader1 = 0$ & $leader2 = 0 \quad \longrightarrow state := 1;$

$state = 2$ & $leader1 = 0$ & $leader3 = 0 \quad \longrightarrow state := 1;$

$state = 2$ & $leader2 = 0$ & $leader3 = 0 \quad \longrightarrow state := 1;$

$state = 3$ & $brake = 0$ & $leader1 = 0$ & $leader2 = 0 \quad \longrightarrow state := 1;$

$state = 3$ & $brake = 0$ & $leader1 = 0$ & $leader3 = 0 \quad \longrightarrow state := 1;$

$state = 3$ & $brake = 0$ & $leader2 = 0$ & $leader3 = 0 \quad \longrightarrow state := 1;$

$state = 3$ & $brake = 0$ & $leader1 = 1$ & $leader2 = 1 \quad \longrightarrow state := 2;$

$state = 3$ & $brake = 0$ & $leader1 = 1$ & $leader3 = 1 \quad \longrightarrow state := 2;$

$state = 3$ & $brake = 0$ & $leader2 = 1$ & $leader3 = 1 \quad \longrightarrow state := 2;$

The action set $\mathcal{T}_O$ denotes the original actions which are from the original program and unchanged in the fault-tolerant program. The actions in $\mathcal{T}_S$ utilize the redundancy of sensors to tolerate the false alarm (false positive and false negative) caused by an unreliable sensor. Hence the system can get correct information about whether the leader car exists, even in the presence of the false alarm of one sensor.

6.5.5 Application of Step D: Generating Fault-tolerant UML model for ACC System

In this step, we utilize the fault-tolerant UCM into the corresponding UML state diagram. Observe that some parts of the UML state diagram remain unchanged. For those, we utilize the corresponding part from the UML state diagram. As an example, this results in that

three transitions in the UML state diagram remain unchanged in the fault-tolerant UML state diagram, that is, the transition that is from state 1 to state 4, the transition that is from state 2 to state 4 and the transition that is from state 3 to state 4.

Moreover, for revised actions, we strengthen the conditions under which the actions can be executed. For example, the triggering condition of transition from state 0 to state 1 is revised from $on = 1 \& leader = 0$ to $on = 1 \& ((leader1 = 0 \& leader2 = 0) | (leader1 = 3 \& leader2 = 0) | (leader1 = 0 \& leader3 = 0))$. Thus, the UML state diagram of fault-tolerant ACC system is shown in Figure 6.6.

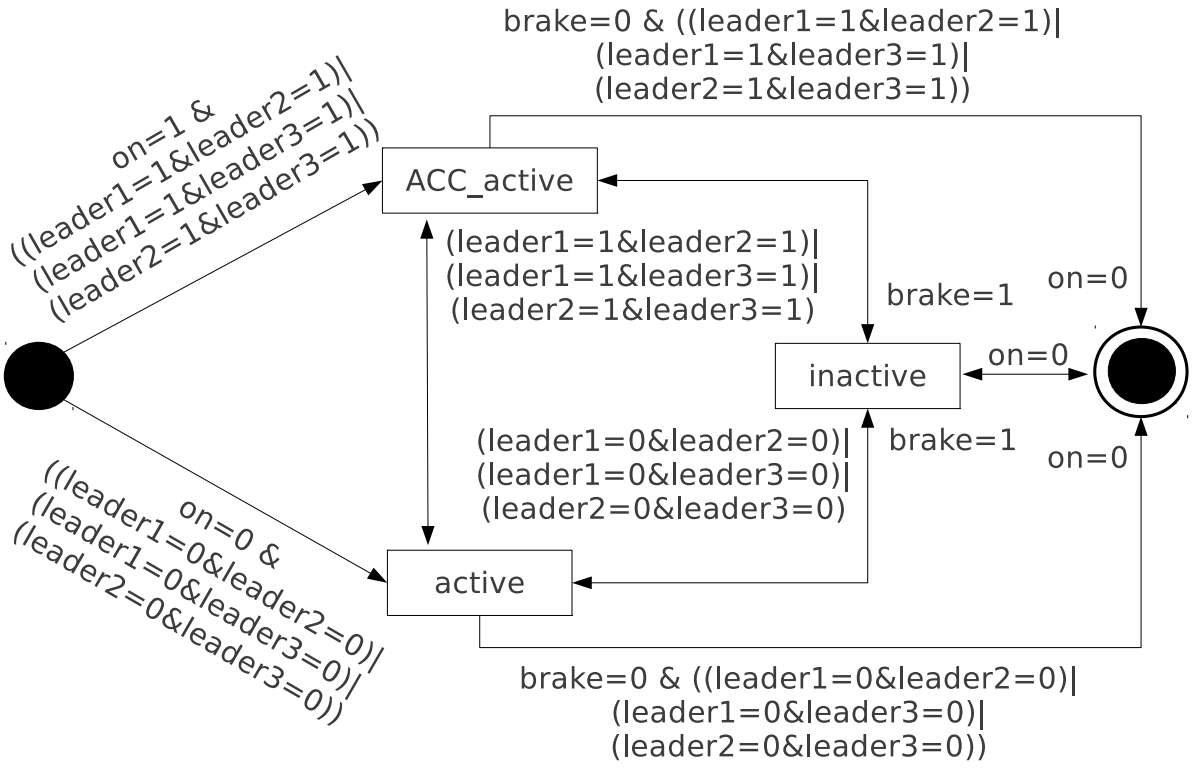

Figure 6.6: The revised ACC program in UML state diagram.

6.6 Case Study 2: The Altitude Switch Controller

In this section, we show the stepwise application of MR4UM on a simplified version of an altitude switch (ASW) program from the aircraft altitude controller system. This case is adapted from [31]. The ASW program monitors a set of input variables coming from two analog altitude sensors and a digital altitude sensor. And, it generates an output and activates an actuator when the altitude is less than a pre-determined threshold.

In particular, we begin with UML state diagram of the ASW program that is fault-intolerant, and, we apply MR4UM to generate UML state diagram of the fault-tolerant ASW program that satisfies Problem 2.4.1. Specifically, in Section 6.6.1, we describe the fault-intolerant UML model for the ASW program. In Section 6.6.2, we describe how MR4UM transforms UML state diagram of the ASW program into program actions modeled in UCM. In Section 6.6.3, we describe the user inputs for generating the fault model, specification and invariant. Next, Section 6.6.4 describes how MR4UM utilizes model revision algorithm to synthesize fault tolerance to the input program. Finally, Section 6.6.5 show how MR4UM transforms the fault-tolerant program obtained by model revision into the corresponding fault-tolerant UML state diagram.

6.6.1 Fault-intolerant UML model for ASW

The UML state diagram of fault-intolerant *ASW* program includes three state, namely, *initialization*, *awaitActuator* and *standby*. The *initialization* state captures the status when the *ASW* program is initializing. The *awaitActuator* state captures the status when the *ASW* program is waiting for the actuator to power on. The *standby* state captures the status when the *ASW* program is in *standby* mode.

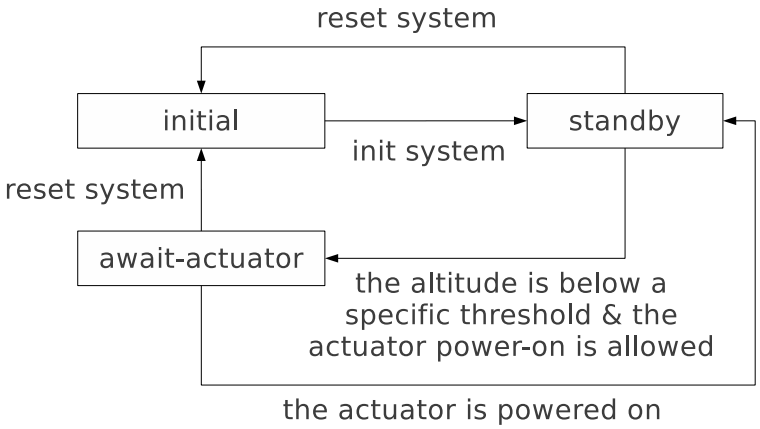

Figure 6.7: UML state diagram of the ASW program.

The state diagram in Figure 6.7 visualizes model design of *ASW* program. Figure 6.8 provides formal expression of the corresponding annotation in Figure 6.7. As shown in Figure 6.7, the *ASW* program will enter into the *standby* state after initialization from *initialization* state, and, the flag variable *init* is set to be 1 to denote the initialization process is done. The *ASW* program will reset into the *initialization* state from *standby* state if the reset process is triggered, and, the flag variable *reset* is assigned 1 to denote the reset process is done. When the actuator is powered off and actuator power-on is allowed, the *ASW* program will change state from *standby* to *awaitActuator*, and, the flag variable *altBelow* will be assigned 1 to denote that the altitude is below a specific threshold. The *ASW* program will change from *awaitActuator* state to *standby* state if the actuator is powered on, and, the flag variable *actuatorStatus* will be assigned 1 to denote the actuator is powered on. The *ASW* program will change from *awaitActuator* state to *initialization* state if the program is reset, and, the flag variable *reset* will be assigned 1 to denote that the program is reset. This program works correctly in the absence of faults.

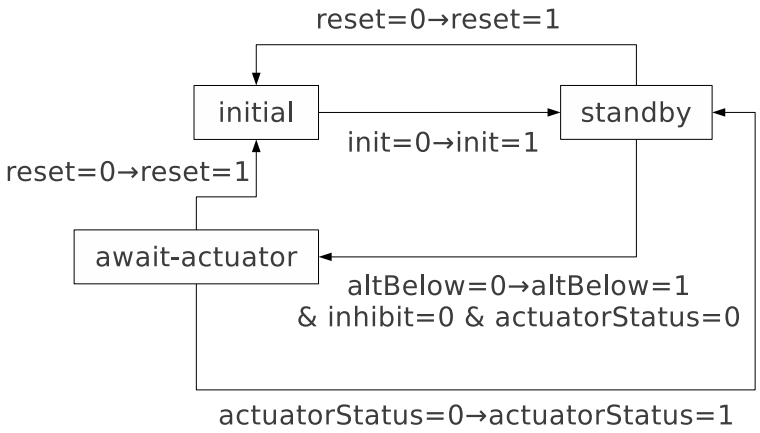

Figure 6.8: the ASW program with formal annotation.

6.6.2 Application of Step A: Generating UCM of the ASW Program

The UML state diagram of the ASW program is as shown in Figure 6.8. Based on the transformation rules, the corresponding program actions in UCM needs seven variables, namely, *state*, *init*,*reset*, *inhibit*, *altBelow*, *actuatorStatus* and *altFail*. These variables are defined as follows:

1. *state*. The domain of variable *state* is $\{0, 1, 2, 3\}$. It is used to denote the four states of the ASW program. $state = 0$ denotes the initial status of the system when it is turned on. $state = 1$ denotes the system is in *Await-Actuator* status. $state = 2$ denotes the system is in *standby* status. Plus, $state = 3$ denotes the system is in *faulty* status.

2. *init*. The domain of variable *init* is $\{0, 1\}$. It is used to denote whether the altitude controller program is initialized. When the program is initialized, the variable *init* is assigned 1, otherwise 0.

3. *reset*. The domain of variable *reset* is $\{0, 1\}$. It is used to model whether the program is being reset. When the program is reset, the variable *reset* is assigned 1, otherwise 0.

4. *inhibit*. The domain of variable *inhibit* is $\{0, 1\}$. It is used to model whether the actuator power-on is inhibited. $inhibit = 1$ denotes that the actuator power-on is inhibited. And, $inhibit = 0$ denotes the actuator power-on is allowed.

5. *altBelow*. The domain of variable *brake* is $\{0, 1\}$. It is used to denote whether the altitude is less than a pre-determined threshold. $altBelow = 1$ models the event that the altitude is below a specific threshold. $altBelow = 0$ models the event that the altitude is not below the threshold.

6. *actuatorStatus*. The domain of variable *actuatorStatus* is $\{0, 1\}$. It is used to model whether the actuator is powered on. $actuatorStatus = 1$ denotes that the actuator is powered on. And, $inhibit = 0$ denotes the actuator is not powered on.

7. *altFail*. The domain of variable *brake* is $\{0, 1\}$. *altFail* is equal to 1 when analog and digital altitude meters are failed, otherwise 0.

Hence, the program actions of ASW program in UCM are as follows:

1. $state = 0$ & $init = 1 \longrightarrow state := 2; init := 0;$

2. $state = 2$ & $reset = 0 \longrightarrow state := 0; reset := 1;$

3. $state = 2$ & $altBelow = 0$ & $inhibit = 0$ & $actuatorStatus = 0 \longrightarrow state := 1; altBelow := 1;$

4. $state = 1$ & $actuatorStatus = 0 \longrightarrow state := 2; actuatorStatus := 1;$

5. $state = 1$ & $reset = 0 \longrightarrow state := 0; reset := 1;$

In the above program actions, action 1 models the program transition from *initialization* mode ($state = 0$) to *standby* mode ($state = 2$). This transition triggers the reassignment of the variable *init*, that is $init = 0 \rightarrow init = 1$ (which denotes the program is done with initialization process). Action 2 models the transition from *standby* mode to *initialization* mode ($state = 0$). This transition triggers the reassignment of the variable *reset*, that is, $reset = 0 \rightarrow reset = 1$ (which denotes the program is done with the resetting process). Action 3 denotes the transition from *standby* mode ($state = 2$) to *await-Actuator* mode ($state = 1$). The triggering condition of this action is the actuator power-on is not inhibited and the actuator is not powered on. This action triggers the reassignment of the variable *altBelow*, that is $altBelow = 0 \rightarrow altBelow = 1$ (which denotes that the altitude is below a specific threshold). Action 4 denotes that transition from *await-Actuator* ($state = 1$) mode to *standby* mode ($state = 2$). This action triggers the reassignment of the variable *actuatorStatus*, that is $actuatorStatus = 0 \rightarrow actuatorStatus = 1$ (which denotes that the actuator is powered on). Action 5 denotes the transition from *await-Actuator* ($state = 1$) to *initialization* mode ($state = 0$). This action triggers the reassignment of the variable *reset*, that is $reset = 0 \rightarrow reset = 1$ (which denotes the system is reset).

6.6.3 Application of Step B: Generating Remaining Inputs for Model Revision

The targeted fault-tolerant program is required to tolerate such a faulty status: the altitude sensors incur malfunction. This type of fault is recognized as *transient* fault. To model this fault action in UCM, the designer needs to specify which variable is perturbed and the possible value of the corrupted variable. In this case, the designer specifies the fault may

corrupt variable *state* into 3, that is, $state = 3$ denotes the faulty status of the program. Besides, the designer also needs to specify the triggering condition of the faults. In this case, the designer specifies three triggering conditions for three different transitions that corrupt the program into faulty status.

- *initFailed = 1.* This condition represent the situation where the program stays in the initialization mode for more than 0.6 second.
- *altFailOver = 1.* This condition represents the situation where the condition $altFail = 1$ remains true more than 2 seconds.
- *awaitOver=1.* This condition represents the situation where the program stays in the *await-Actuator* mode for more than 2 seconds.

Hence, MR4UM generates fault actions as following:

1. $initFailed == 1 \longrightarrow initFailed := 0, state := 3;$
2. $altFailOver == 1 \longrightarrow altFailOver := 0, state := 3;$
3. $awaitOver == 1 \longrightarrow awaitOver := 0, state := 3;$

In this case, the designer requires the safety specification, as follows:

- If the altitude sensor are failed, the program does not transfer from *standby* mode to *await-Actuator* mode;
- The program can only go to the initialization mode from the faulty mode;
- The program can recover from the faulty mode if the program is not reset.

Hence, the designer specifies the specification from GUI of the framework as follows:

$((altFails == 1)\&(state == 2)\&(state' == 0))|$

$((state == 3)\&(state == 2)\&(state' == 1))|$

$((state == 3)\&(reset == 1));$

The invariant of the program consists of the states where the program is not in the faulty states, i.e., $state! = 3$. Hence, the invariant of this program is as follows:

$state! = 3$

6.6.4 Application of Step C: Generation of Fault-Tolerant UCM

In Step C, MR4UM utilizes the inputs from Section 6.6.2 and 6.6.3. After Step C applies the symbolic model revision on the input program, the MR4UM generates the revised program modeled in UCM. Note that this output is only intended for use in Step D and not meant for designer to analyze it. The obtained fault-tolerant program is as follows:

$\mathcal{T}_O$:

1). $state = 0$ & $init = 1 \longrightarrow state := 2; init := 0;$

2). $state = 2$ & $reset = 0 \longrightarrow state := 0; reset := 1;$

3). $state = 1$ & $actuatorStatus = 0 \longrightarrow state := 2; actuatorStatus := 1;$

4). $state = 1$ & $reset = 0 \longrightarrow state := 0; reset := 1;$

$\mathcal{T}_S$:

5). $state = 2$ & $altBelow = 0$ & $inhibit = 0$ & $actuatorStatus = 0$ & $altFail = 0$

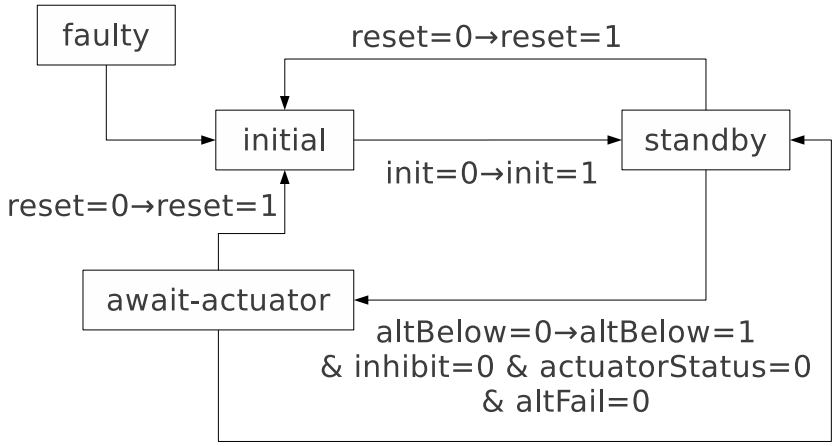

Figure 6.9: The revised ASW program in UML state diagram.

$\longrightarrow state := 1; altBelow := 1;$

$\mathcal{T}_r$:

6). $state = 3$ & $reset = 0$ $\longrightarrow state := 0; reset := 1;$

Note that Action 5 has been added a new constraint $altFail == 0$ which denotes that the program is allowed to change its state to the *await-Actuator* mode only when the input sensors are not corrupted. Action 6 is newly added to the program which denotes that the program recovers from faulty mode ($state = 3$) to the initialization mode ($state = 0$). The other actions remains the same as the actions of the input program.

6.6.5 Application of Step D: Generating Fault-tolerant UML model for ASW Program

In this step, MR4UM converts the fault-tolerant program in UCM into the corresponding model in UML state diagram. The UML state diagram of fault-tolerant ASW program is as shown in Figure 6.9.

6.7 Discussion and Lessons Learnt

In this section, we discuss several questions that are raised by our work as well as lessons learnt from the case study.

One question is the scalability of the approach of automatic revising the existing program design proposed in our work. Our framework benefits from the fact that the underlying synthesis engine utilizes OBDDs to mitigate the state explosion problem as well as a heuristic based algorithm [36] to mitigate the complexity (NP-complete) of model revision. Specifically, this approach has been used to permit model revision of programs with state space exceeding 10^{100}. Hence, we expect the framework to be able to handle moderate sized problems.

Another question is about the choice of UML as the front end for our framework. We chose UML because it is one of the commonly used platforms to specify requirements. And, although there is an existing work on formalization of UML models, the problem of model revision has not been addressed in this context. Our approach is also feasible for revising program design modeling in other approaches (e.g. AADL [80]) by modifying the mapping mechanism between the model of program design and underlying computational model. It has also been demonstrated in revising SCR specifications [107].

One of the difficulties in developing this framework lies in the fact that the revised fault-tolerant model in UCM is BDD based. Although converting the UCM model into UML state diagram involves some challenges, they can be overcome by understanding (1) the part of the UML model that will remain intact in the final model, (2) the part of the UML model where the structure of the original model will remain intact in the final model although some of the details (e.g., conditions on the arrows) will change, and (3) the part of the UML

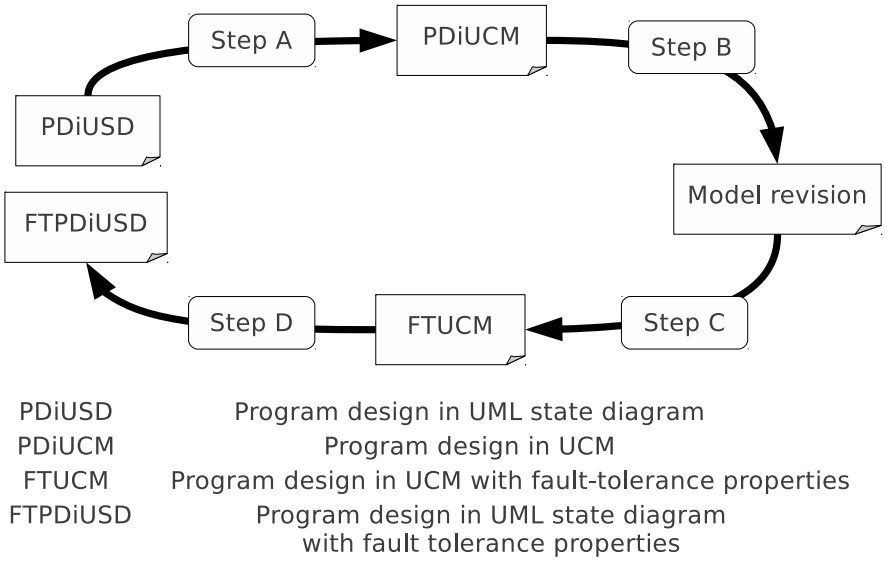

Figure 6.10: The stepwise procedure of MR4UM.

model that is completely new and added for dealing with recovery from faults. Since BDD based approaches permit us to check conditions (1) and (2) effectively with negligible cost, obtaining the revised UML state diagram is feasible. One of the future work in this area is to optimize the third part that identifies the actions that provide recovery.

6.8 Summary

Our work focuses on reducing the learning curve required for application of formal methods, specifically model revision, by keeping the formal methods under-the-hood to a large extent. We chose to apply model revision for UML models since it is one of the popular modeling techniques. Specifically, in this chapter, we proposed a framework, namely MR4UM, which allows designers to apply model revision to existing UML models for adding fault-tolerance. The stepwise-procedure of MR4UM is shown in Figure 6.10. Since the model revision process of MR4UM is based on BDDs, it has the potential to deal with large state space, e.g., the underlying synthesis engine has been used to revise models with state space exceeding 10^{100}.

We illustrated the stepwise procedure of MR4UM with two case studies. One is the adaptive cruise controller system (ACC) from automotive systems. The other one is the altitude switch (ASW) used in the aircraft altitude controller.

Chapter 7

Related Work

This chapter is dedicated to illustrate the related work on this dissertation. In particular, Section 7.0.1 presents the related work on automatic verification of stabilizing programs. In Section 7.0.2, we present the related work on automatic revision for multitolerant programs. Section 7.0.3 introduces the related work on model repair of UML state diagrams for adding fault-tolerance.

7.0.1 Automatic Verification of Stabilizing Programs

Formal verification of stabilizing programs has been studied mainly in two directions, including *mechanism theorem proving* and *model checking*. Mechanism theorem proving based approaches [132, 188, 189] have been demonstrated as a very powerful verification technique, especially for these infinite systems. Unfortunately, this approach usually requires considerable experience in logic reasoning.

Model checking is one of the most successful approaches to verification. However, using model checking for verification purpose suffers from the 'state explosion' problem. To address

this issue, researchers have proposed the technique of symbolic model checking, using OBDDs to obtain a compact representation of programs. Also, Tsuchiya et al [225] have proposed an approach to verify stabilizing programs based on symbolic model checking. Although this work has demonstrated the feasibility of using symbolic model checking to verify stabilizing programs, it only works for programs consisting of a small number of processes. To solve this bottleneck, we proposed an approach to verify stabilizing programs, that is using effectiveness of fairness while using symbolic model checking.

With the advancement of model checking techniques, the SAT/SMT based model checking has emerged as an viable alternative to symbolic model checking. In particular, symbolic model checking requires one to identify the order of program variables when constructing OBDD representations. An incorrect ordering of variables can increase the verification time. That motivates our research on investigating the effectiveness of SMT-based model checking in verifying stabilizing programs.

7.0.2 Automatic Revision for Multitolerant Programs

Automated program revision is studied from different perspectives. One approach (e.g., [23]) focuses on synthesizing fault-tolerant programs from their specification in a temporal logic (e.g., Linear Temporal Logic(LTL) [76], etc.). The word synthesis is also used in the context (e.g., [53] [101] [158] [114]) of transforming an abstract (such as UML) program into a concrete (such as C++) program while ensuring that the location of concrete program in memory, its data flow etc. meet the constraints of an embedded system. By contrast, our approach focuses on transformation of one abstract program into another that meets additional properties of interest. Our approach will advance the applicability of this existing

work by allowing designers to add properties of interest in the abstract model and then using existing work to generate concrete program. Hence our approach is desirable when one needs to extend an existing system by adding fault-tolerance.

Our work is closely related to the work on controller synthesis [20] [21] [38] [66] and game theory [51] [79] [123]. In control-theoretic approaches, the supervisory control of real-time systems has been studied under the assumption that the existing program (called a plant) and/or the given specification is deterministic. Moreover, in both game theory and controller synthesis, since highly expressive specifications are often considered, the complexity of the proposed synthesis methods is very high. For example, the synthesis problems presented in [20] [21] [51] [79] are EXPTIME-complete. Furthermore, deciding the existence of a controller in [38] [66] is 2EXPTIME-complete. In addition, these approaches do not address some of the crucial concerns of fault-tolerance (e.g., providing recovery in the presence of faults) that are considered in the our work. In addition, the high complexity of these methods is a serious barrier to actually synthesize moderate sized programs. By contrast, our approach focus only on specifications needed to express properties of interest. Hence, the complexity of our algorithm is substantially lower.

The algorithms in [134] [136] have addressed the problem of adding fault-tolerance to only one class of fault. Also the algorithm in [71] is targeted toward the synthesis of programs that simultaneously tolerate multiple classes of faults whereas we address the synthesis of programs that provide the appropriate level of fault-tolerance and not to provide any tolerance if faults from one class occurs while the program is 'recovering' from faults from another class. The 'weak' multitolerance way in this dissertation is necessary especially when it is impossible to guarantee any level of tolerance in a computation where faults from two classes

occur.

7.0.3 Model Revision of UML State Diagrams for Adding Fault-tolerance

Previous work in [125,130,156] addresses the problem of formalization of UML state diagram. Specifically, these approaches define operational semantics of the UML state diagram and then utilize it for simulation, verification and/or code generation. The first step in our framework is inspired by these approaches. However, unlike the previous work, in our work the translations of UML model needs to be annotated so that we can subsequently obtain a revised UML model after adding fault-tolerance. Another important difference between our work and these works is that our work focuses on the problem of model revision whereas they focus on the problem of model checking. Thus, our work is complementary to previous work in that our framework can be applied in scenarios where the given UML model fails to satisfy the given property.

Our work is orthogonal to related work (e.g., [54, 102, 115, 159, 171]) that focuses on transforming an abstract UML model into a concrete (such as C++) program while ensuring that the location of concrete program in memory, its data flow etc. meet the constraints of the underlying system. In particular, our work focuses on revising the given model into another UML model that satisfies the fault-tolerance property. Thus, our work will advance the applicability of this existing work by allowing designers to add properties of interest in the abstract model and then using existing work to generate concrete program.

Approaches in [69,70] develop corrector pattern for specifying nonmasking fault-tolerance and failsafe fault-tolerance respectively. These proposed analysis methods are validated in

terms of UML diagrams. While these works simplify and modularize fault-tolerance concerns and facilitate to analyze the functional and fault-tolerance concerns and their mutual impact, application of a synthesis tool in automatically adding fault-tolerance is an on-going direction of these works. Our work utilizes the synthesis tool [37] to automate the revision process for adding fault-tolerance.

The work in [2] proposes an approach of automating and formalizing the translation from high level design models, specifically, Software Cost Reduction (SCR) [31], to a format that can be used by the automated revision/synthesis tools of example. SCR is a set of formal methods for constructing and verifying requirements specification document. By contrast, our work focuses on issues of automatic revision of UML state diagram for adding fault-tolerance.

Chapter 8

Conclusion and Future Work

This dissertation focused on the problem of automatic verification and revision of multitolerant programs. In particular, our goal was to build a model-based framework that helps to build multitolerant programs in an automatic way. To this end, we have developed new theory, algorithms as well as tools to advance the state-of-the-art research. First, we have investigated the effectiveness of our fault modeling approach, that is, using transition systems to model faults. Then, leveraging model repair techniques, we proposed approaches to efficiently verify the fault-tolerance properties (stabilization) provided by programs. The proposed approaches mitigate the fundamental challenge of 'state explosion', the key problem that limits the scalability of previous approaches to verify stabilization. We also investigated the complexity of automatic program revision for adding multitolerance. And, we have developed novel model repair algorithms for revising programs to add multitolerance. We have implemented these algorithms as a tool. We also have demonstrated the applicability of model repair techniques in a lightweight framework, where the given UML state diagram is automatically revised to add fault-tolerance.

In what follows, we summarizes the contributions of this dissertation in Section 8.1. Then, in Section 8.2, we discuss possible directions of future work.

8.1 Contributions

The main results of this dissertation are as follows.

1. We investigated the effectiveness of transition systems to model faults. Starting from a taxonomy of faults that is based on practitioner's point of view [24], we showed that: (i) faults from 20 categories of all 31 categories can be modeled using transition systems. These faults include Byzantine actions in a networked system, physical deterioration of brake in a braking control system and so on. And, (ii) faults from 11 categories cannot (or should not be) represented using transition systems. These include faults such as buffer overflow, hardware errata and so on. We also investigated the feasibility and practicability of using transition systems to model faults. We showed that (i) the modeling of faults from 18 categories as transition systems is practical and feasible and (ii) the modeling of faults from 2 categories as transition systems is not practical although feasible. Besides, We identified the relative completeness of the proposed approach with recent literature.

2. We identified one bottleneck involved in existing automatic verification of self-stabilization. In particular, we investigated the issues of fairness and its effects on the performance of verifying self-stabilizing programs. We showed that if self-stabilization is possible with unfair computation, then the cost of stabilization verification is substantially reduced. In particular, we showed that the time for verification with unfair computation

is approximately 0.001%-0.1% of that with weakly fair computation.

3. For the cases where weak fairness is essential for the correctness of stabilization, we proposed two approaches to improve the scalability of verifying stabilizing programs, including *decomposition* and *weak stabilization*. We showed that using decomposition would result in a substantial increase for the scalability of verifying stabilization (e.g., from 10^4 states to 10^{138} states fro Huang's mutual exclusion algorithm). We also showed that verification of weak stabilization is substantially more scalable. This result validates the suggestion in [92] that weak stabilization is easier to verify than stabilization.

4. We proposed a constraint-based approach to analyze stabilizing programs without introducing fairness scheduler in the program model, and without assuming any order information of variables. The key insight is to reduce the problem of stabilization verification into a well-studied problem, named constraint solving, which can be solved by many existing highly optimized solutions.

5. We investigated the complexity issues in automatic revision of programs for adding (weak) multitolerance.

6. We presented a counterintuitive result where we showed that the problem of automatic addition of Masking-Masking (respectively, Masking-Failsafe) weak multitolerance is NP-complete. This result is especially surprising given that the corresponding problem can be solved in polynomial time for strong multitolerance.

7. We presented a sound and complete algorithm for automatic program revision to add Failsafe-Failsafe (respectively, Masking-Nonmasking) weak multitolerance, Also, we

presented a polynomial time heuristic for designing Masking-masking weak multitolerant program.

8. We implemented the proposed algorithms for automatic program revision to add multitolerance as a tool.

9. We compared our proposed multitolerance approach, named weak multitolerance with previous techniques, named strong multitolerance. We also identified circumstances where solvability for adding weak and strong multitolerance differs.

10. We presented a lightweight framework for automatic revision of UML state diagrams for adding fault-tolerance.

8.2 Future Work

This section presents some of the problems that we believe are both important and interesting.

Scalability of Static Analysis and Model Revision Techniques. Emerging hardwares and novel software architectures pose new challenges to the analysis and/or repair techniques for system. There is a great need for developing novel and effective verification and revision techniques to analyzing parallel programs and large data space-based software systems. One such open problem is to exploit the advantages of highly parallel data architectures to improve the performance of existing DPLL based static analysis techniques.

Analyzing and Improving the Reliability of Software Systems in the Presence of Faults such as Configure Errors or Hardware Faults. Recent research on system failure shows that human mistakes (e.g., wrong configurations) and hardware faults, rather than software bugs,

are becoming the dominating causes for system's down time. We are currently conducting a comprehensive study to understand these problems in real world. Leveraging the techniques of program analysis, data mining and software engineering, we plan to develop novel configuration patterns/suggestions for a software application to eliminate or alleviate the negative effects of human mistakes, especially misconfiguration. Also, based on the understanding of real world data, we expect to identify the taxonomy of hardware faults and investigate corresponding tolerance solutions.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] F. Abujarad and S. Kulkarni. Constraint based automated synthesis of nonmasking and stabilizing fault-tolerance. In *Proceedings of the 2009 28th IEEE International symposium on Reliable Distributed Systems*, pages 119–128, 2009.

[2] F. Abujarad and S. S. Kulkarni. Automated addition of fault-tolerance to scr toolset: A case study. In *The Seventh International Workshop on Assurance in Distributed Systems and Networks (ADSN), in ICDCSW '08: Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems Workshops*, pages 539–544, 2008.

[3] M. K. Aguilera, K. Keeton, A. Merchant, K.-K. Muniswamy-Reddy, and M. Uysal. Improving recoverability in multi-tier storage systems. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[4] I. E. Akkus and A. Goel. Data recovery for web applications. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, 2010.

[5] S. Al-Kiswany, M. Ripeanu, S. S. Vazhkudai, and A. Gharaibeh. stdchk: A checkpoint storage system for desktop grid computing. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[6] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[7] B. Alpern and F. B. Schneider. Proving boolean combinations of deterministic properties. *Proceedings of the Second Symposium on Logic in Computer Science*, pages 131–137, 1987.

[8] R. Alur, C. Courcoubetis, T. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1993.

[9] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science 126:183-235, 1994 (preliminary versions appeared in Proc. 17th ICALP, LNCS 443, 1990, and Real Time: Theory in Practice*, 1991.

[10] R. Alur, T. Feder, and T. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, 1996.

[11] K. H. amd Shouhuai Xu. Protecting cryptographic keys from memory disclosure attacks. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[12] Y. Amir, B. A. Coan, J. Kirsch, and J. Lane. Byzantine replication under attack. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '08, 2008.

[13] H. Ando, R. Kan, Y. Tosaka, K. Takahisa, and K. Hatanaka. Validation of hardware error recovery mechanisms for the sparc64 v microprocessor. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '08, 2008.

[14] B. Archive, E. Anderson, X. Li, A. Merchant, M. A. Shah, K. Smathers, J. Tucek, M. Uysal, and J. J. Wylie. Efficient eventual consistency in pahoehoe–an erasure-coded key. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, 2010.

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

[16] A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC)*, 1998.

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

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

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

[20] E. Asarin and O. Maler. As soon as possible: Time optimal control for timed automata. In *In Proc. of the 2nd International Workshop on Hybrid Systems: Computation and Control*, pages 19–30, London, UK, 1999. Springer-Verlag.

[21] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *In IFAC symposium on System Structure and Controller*, pages 469–474, Nantes,France, 1998. Elsevier.

[22] P. Attie and A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM TOPLAS (a preliminary version of this paper appeared in PODC96)*, 23(2), March 2001.

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

[24] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 2004.

[25] L. N. Bairavasundaram, M. Rungta, N. Agrawal, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift. Analyzing the effects of disk-pointer corruption. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '08, 2008.

[26] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *In: IFM. (2004*, pages 1–20. Springer, 2004.

[27] N. Banu, T. Izumi, and K. Wada. Doubly-expedited one-step byzantine consensus. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, 2010.

[28] P. Bergheaud, D. Subhraveti, and M. Vertesi. Fault tolerance in multiprocessor systems via application cloning. In *27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007)*, 2007.

[29] M. Bertier, A.-M. Kermarrec, and G. Tan. Message-efficient byzantine fault-tolerant broadcast in a multi-hop wireless sensor network. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, ICDCS '10, 2010.

[30] A. N. Bessani, V. V. Cogo, M. Correia, P. Costa, M. Pasin, F. Silva, L. Arantes, O. Marin, P. Sens, and J. Sopena. Making hadoop mapreduce byzantine fault-tolerant. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, 2010.

[31] R. Bharadwaj and C. Heitmeyer. Developing high assurance avionics systems with the scr requirements method. In *Proceedings of the 19th Digital Avionics Systems Conference*, pages 1D1/1 – 1D1/8, Philadelphia, PA, 2000.

[32] M. Bhide, K. Ramamritham, and M. Agrawal. Efficient execution of continuous incoherency bounded queries over multi-source streaming data. In *Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems*, pages 681–688, 2007.

[33] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *In Proc. Of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, 1999.

[34] W. J. Bolosky, J. R. Douceur, and J. Howell. The farsite project: a retrospective. In *in ACM SIGOPS Operating Systems Review*, 2007.

[35] B. Bonakdarpour. *Automated Revision of Distributed and Real-Time Programs*. PhD thesis, Michigan State University, 2008.

[36] B. Bonakdarpour and S. Kulkarni. Exploiting symbolic techniques in automated synthesis of distributed programs. In *Proceedings of In IEEE International Conference on Distributed Computing Systems(ICDCS)*, ICDCS '07, pages 3–10, Toronto, Canada, 2007.

[37] B. Bonakdarpour and S. Kulkarni. Sycraft: A tool for automated synthesis of fault-tolerant distributed programs. In *Proceedings of International Conference on Concurrency Theory (CONCUR)*, pages 167–171, Toronto, Canada, 2008.

[38] P. Bouyer, D. D'Souza, P. Madhusudan, and A. Petit. Timed control with partial observability. *Lecture Notes in Computer Science*, 2725:180–192, 2003.

[39] A. Brito, C. Fetzer, and P. Felber. Minimizing latency in fault-tolerant distributed stream processing systems. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, 2009.

[40] J. C. Brustoloni and D. Kyle. Updates and asynchronous communication in trusted computing systems. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

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

[42] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[43] L. E. Buzato, G. M. D. Vieira, and W. Zwaenepoel. Dynamic content web applications: Crash, failover, and recovery analysis. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '09, 2009.

[44] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '09, 2009.

[45] L. Chen and J. Leneutre. Selfishness, not always a nightmare: Modeling selfish mac behaviors in wireless mobile ad hoc networks. In *ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems*, 2007.

[46] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr. Surviving sensor network software faults. In *SOSP'09*, New York, NY, USA, 2009. ACM.

[47] V. Claesson, H. Lonn, and N. Suri. An efficient tdma start-up and restart synchronization approach for distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 15(8):725–739, 2004.

[48] C. Constantinescu, I. Parulkar, R. Harper, and S. Michalak. Silent data corruption - myth or reality? In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '08, 2008.

[49] R. Curtmola, O. Khan, R. C. Burns, and G. Ateniese. Mr-pdp: Multiple-replica provable data possession. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[50] D. Dams, W. Hesse, and G. J. Holzmann. Abstracting C with abC. In *14th International Conference on Computer Aided Verification (CAV)*, pages 515–520, London, UK, 2002. Springer-Verlag.

[51] L. de Alfaro, M. Faella, T. A. Henzinger, R. Majumdar, and M. Stoelinga. The element of surprise in timed games. In *Internatinal Conference on Concurrency Theory (CONCUR)*, Marseille, France, 2003. Springer.

[52] L. M. de Moura and N. Bjorner. Z3: An efficient smt solver. In *TACAS 2008*, 2008.

[53] D. de Niz and R. Rajkumar. Glue code generation: Closing the loophole in model-based development. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004). Workshop on Model-Driven Embedded Systems*, Toronto, Canada, 2004. IEEE Computer Society.

[54] D. de Niz and R. Rajkumar. Glue code generation: Closing the loophole in model-based development. *2nd RTAS Workshop on Model-Driven Embedded Systems*, 2004.

[55] C. Delporte-Gallet, H. Fauconnier, and A. Tielmann. Fault-tolerant consensus in unknown and anonymous networks. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, 2009.

[56] A. Demaille, J. E. Denny, and P. Eggert. Bison - gnu parser generator. Technical report, http://www.gnu.org/software/bison/, 2012.

[57] M. Demirbas, A. Arora, V. Mittal, and V. Kulathumani. A fault local self-stabilizing clustering service for wireless ad hoc networks. *IEEE Transactions on Parallel and Distributed Systems*, 17, 2006.

[58] S. Devismes, S. Tixeuil, and M. Yamashita. Weak vs. self vs. probabilistic stabilization. In *ICDCS '08: Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, pages 681–688, 2008.

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

[60] E. Dijkstra. *A Discipline of Programming*. New Jersey, USA, 1990.

[61] E. Dijkstra and C. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11:1–4, 1980.

[62] X. Ding and H. Jin. Efficient and progressive algorithms for distributed skyline queries over uncertain data. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, ICDCS '10, 2010.

[63] S. Dolev. Self-stabilizing routing and related protocols. *Journal of Parallel and Distributed Computing*, 42(2), 1997.

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

[65] Q. Dong and D. Liu. Adaptive jamming-resistant broadcast systems with partial channel sharing. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, ICDCS '10, 2010.

[66] D. D'Souza and P. Madhusudan. Timed control synthesis for external specifications. In *STACS '02: Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science*, pages 571–582, London, UK, UK, 2002. Springer-Verlag.

[67] B. Dutertre and L. D. Moura. The yices smt solver. Technical report, Computer Science Laboratory, SRI International, 2006.

[68] A. Ebnenasir. Diconic addition of failsafe fault-tolerance. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 44–53, 2007.

[69] A. Ebnenasir and B. H. C. Cheng. A pattern-based approach for modeling and analyzing error recovery. In *Workshops on Software Achitectures for Dependable systems (WADS)*, pages 115–141, 2006.

[70] A. Ebnenasir and B. H. C. Cheng. Pattern-based modeling and analysis of failsafe fault-tolerance in uml. In *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium*, HASE '07, pages 275–282, Washington, DC, USA, 2007. IEEE Computer Society.

[71] A. Ebnenasir and S. Kulkarni. Feasibility of stepwise design of multitolerant programs. *ACM Transactions on Software Engineering and Methodology*, 21, 2011. In Press.

[72] A. Ebnenasir, S. S. Kulkarni, and A. Arora. FTSyn: A framework for automatic synthesis of fault-tolerance. *International Journal on Software Tools for Technology Transfer*, 10(5):455–471, 2008.

[73] J. G. Elerath. A simple equation for estimating reliability of an n+1 redundant array of independent disks (raid). In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '09, 2009.

[74] J. G. Elerath and M. Pecht. Enhanced reliability modeling of raid storage systems. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[75] M. Elhaddad, H. Iqbal, T. Znati, and R. Melhem. Scheduling to minimize the worst-case loss rate. In *27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007)*, 2007.

[76] E. Emerson. *Handbook of Theoretical Computer Science: Chapter 16, Temporal and Modal Logic*. Elsevier Science Publishers B. V., 1990.

[77] E. Emerson and E. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[78] P. D. Ezhilchelvan, F. V. Brasileiro, and N. A. Speirs. A timeout-based message ordering protocol for a lightweight software implementation of tmr systems. *IEEE Trans. Parallel Distrib. Syst.*, 15(1):53–65, 2004.

[79] M. Faella, S. Torre, and A. Murano. Dense real-time games. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 167–176, Copenhagen, Denmark, 2002. IEEE Computer Society.

[80] P. Feiler, B. Lewis, and S. Vestal. The sae architecture analysis and design language (aadl) standard: A basis for model-based architecture-driven embedded systems engineering. In *In Proceedings of the RTAS Workshop on Model-driven Embedded Systems*, pages 1–10, 2009.

[81] J. Fonseca and M. Vieira. Mapping software faults with web security vulnerabilities. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '08, 2008.

[82] F. Freitas, E. Marques, R. Rodrigues, C. Ribeiro, P. Ferreira, and L. Rodrigues. Verme: Worm containment in overlay networks. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '09, 2009.

[83] V. K. Garg and V. Ogale. Fusible data structures for fault-tolerance. In *27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007)*, 2007.

[84] M. Garofalakis, R. Rastogi, and K. Sabnani. Streaming algorithms for robust, real-time detection of ddos attacks. In *Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems*, pages 681–688, 2007.

[85] F. Gartner. Automating the addition of fault-tolerance: Beyond fusion-closed specifications. Personal Communication.

[86] N. J. George, C. R. Elks, B. W. Johnson, and J. Lach. Transient fault models and avf estimation revisited. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, 2010.

[87] S. Ghosh. Binary self-stabilization in distributed systems. *Information Processing Letter*, 40(3), 1991.

[88] S. Ghosh. *Distributed Systems: An Algorithmic Approach*. CRC Press, 2006.

[89] S. Ghosh and a. Gupta. An exercise in fault-containment: Self-stabilizing leader election. *Information Processing Letters*, 1996.

[90] S. Ghosh, A. Gupta, T. Herman, and S. Pemmaraju. Fault-containing self-stabilizing distributed protocols. *Distributed Computing*, 20, 2007.

[91] J. Gong, X. Zhong, and C.-Z. Xu. Energy and timing constrained system reward maximization on wireless networks. In *Proceedings of the 28th International Conference on Distributed Computing Systems*, ICDCS '08, 2008.

[92] M. G. Gouda. The theory of weak stabilization. In *Proceedings of the 5th International Workshop on Self-Stabilizing Systems*, pages 114–123, 2001.

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

[94] J. Gracia-Moran, D. Gil-Tomas, L. J. Saiz-Adalid, J. C. Baraza, and P. J. Gil-Vicente. Experimental validation of a fault tolerant microcomputer system against intermittent faults. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, 2010.

[95] D. Graham, S. R. Per Strid, and F. Rodriguez. A low-tech solution to avoid the severe impact of transient errors on the ip interconnect. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '09, 2009.

[96] J. Gray. Functionality, availability, agility, manageability, scalability – the new priorities of application design. In *Proc. Int'l Workshop High Performance Trans. Systems*, 2001.

[97] F. Greve and S. Tixeuil. Tixeuil: Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In *Proceedings of the 37th Annual*

IEEE/IFIP International Conference on Dependable Systems and Networks, DSN '07, 2007.

[98] M. Grottke, A. P. Nikora, and K. S. Trivedi. An empirical investigation of fault types in space mission system software. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, 2010.

[99] W. Gu, Z. Yang, C. Que, D. Xuan, and W. Jia. On security vulnerabilities of null data frames in ieee 802.11 based wlans. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[100] Y. Gu and T. He. Bounding communication delay in energy harvesting sensor networks. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, ICDCS '10, 2010.

[101] Z. Gu and K. G. Shin. Synthesis of real-time implementations from component-based software models. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 167–176, Washington, DC, USA, 2005. IEEE Computer Society.

[102] Z. Gu, S. Wang, and K. G. Shin. Synthesis of real-time implementation from uml-rt models. *2nd RTAS Workshop on Model-Driven Embedded Systems*, 2004.

[103] R. Guerraoui, D. Kostic, R. Levy, and V. Quema. A high throughput atomic storage algorithm. In *27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007)*, 2007.

[104] W. He, X. Liu, L. Zheng, and H. Yang. Reliability calculus: A theoretical framework to analyze communication reliability. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, ICDCS '10, 2010.

[105] P. E. Heegaard and K. S. Trivedi. Survivability quantification of communication services. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '08, 2008.

[106] J. Heinzmann and A. Zelinsky. A safe-control paradigm for human–robot interaction. *J. Intell. Robotics Syst.*, 25(4):295–310, 1999.

[107] C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. Tools for constructing requirements specifications: The scr toolset at the age of ten. In *International Journal of Computer Systems Science and Engineering*, pages 19–35, 2005.

[108] J. Hendricks, S. Sinnamohideen, G. R. Ganger, and M. K. Reiter. Zzyzx: Scalable fault tolerance through byzantine locking. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, 2010.

[109] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Fault isolation for device drivers. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '09, 2009.

[110] G. Holzmann. The spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[111] G. Holzmann. Logic verification of ansi-c code with spin. In *The Sixth SPIN Workshop*, pages 131–147, 2000.

[112] M. Honda, J. Nakazawa, Y. Nishida, M. Kozuka, and H. Tokuda. A connectivity-driven retransmission scheme based on transport layer readdressing. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[113] J. H. Hopeman. Uniform deterministic self-stabilizing ring-orientation on odd-length rings. *Workshop on Distributed Algorithms*, 1994.

[114] P.-A. Hsiung and S.-W. Lin. Automatic synthesis and verification of real-time embedded software for mobile and ubiquitous systems. *Computer Languages, Systems and Structures*, 34(4):153–169, 2008.

[115] P.-A. Hsiung and S.-W. Lin. Automatic synthesis and verification of real-time embedded software for mobile and ubiquitous systems. *Comput. Lang. Syst. Struct.*, 34(4):153–169, 2008.

[116] S.-T. Huang. Leader election in uniform rings. *ACM Transactions on Programming Languages and Systems*, 15(6):435–450, Jul 1993.

[117] M. Hutle and A. Schiper. Communication predicates: A high-level abstraction for coping with transient and dynamic faults. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[118] S. Y. J. Duato and L. Ni. Interconnection network. *IEEE Computer Society Press*, 1997.

[119] M. L. James, A. A. Shapiro, P. L. Springer, and H. P. Zima. Adaptive fault tolerance for scalable cluster computing in space. *Int. J. High Perform. Comput. Appl.*, 23(3):227–241, 2009.

[120] M. Jayaram and G. Varghese. Crash failures can drive protocols to arbitrary states. *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, 1996.

[121] K. Jayaraman, W. Du, B. Rajagopalan, and S. J. Chapin. Escudo: A fine-grained protection model for web browsers. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, ICDCS '10, 2010.

[122] H. Jin. Checkpointing orchestration for performance improvement. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, 2010.

[123] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Computer Aided Verification (CAV)*, pages 226–238, Scotland, UK, 2005. Springer.

[124] T. Johnson, S. Mitra, and K. Manamcheri. Safe and stabilizing distributed cellular flows. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, ICDCS '10, 2010.

[125] T. A. J. Jori Dubrovin. Symbolic model checking of hierarchical uml state machines. In *ACSD: 8th International Conference on Application of Concurrency to System Design*, pages 108 – 117, 2008.

[126] G. Jung, M. A. Hiltunen, K. R. Joshi, R. D. Schlichting, and C. Pu. Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, ICDCS '10, 2010.

[127] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '09, 2009.

[128] L. Keller, P. Upadhyaya, and G. Candea. Conferr: A tool for assessing resilience to human configuration errors. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '08, 2008.

[129] K. S. Killourhy and R. A. Maxion. Comparing anomaly-detection algorithms for keystroke dynamics. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '09, 2009.

[130] A. Knapp and S. Merz. Model checking and code generation for uml state machines and collaborations. In *In Dominik Haneberg, Gerhard Schellhorn, and Wolfgang Reif, editors, Proc. 5th Wsh. Tools for System Design and Verification*, pages 59–64, 2002.

[131] K. Kourai and S. Chiba. A fast rejuvenation technique for server consolidation with virtual machines. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[132] S. Kulkarni, J. M. Rushby, and S. Natarajan. A case-study in component-based mechanical verification of fault-tolerant programs. In *Workshop on Self-stabilizing System*, pages 33–40, 1999.

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

[134] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Proceedings of the 6th International Symposium of Formal Techniques in Real-Time and Fault-Tolerant Systems*, page 82, Pune, India, 2000. Springer.

[135] S. S. Kulkarni, A. Arora, and A. Ebnenasir. *Adding Fault-Tolerance to State Machine-Based Designs*, volume 19 of *Series on Software Engineering and Knowledge Engineering*, pages 62–90. Springer Verlag, 2007.

[136] S. S. Kulkarni and A. Ebnenasir. The complexity of adding failsafe fault-tolerance. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, page 337, Vienna, Austria., 2002. IEEE Computer Society.

[137] S. S. Kulkarni and A. Ebnenasir. Automated synthesis of multitolerance. *The International Conference on Dependable Systems and Networks*, 2004.

[138] S. S. Kulkarni, C. B. J. Oleszkiewicz, and A. Robinson. Alternators in read/write atomicity. *Information Processing Letters*, 2005.

[139] O. Kupferman and M. Vardi. Synthesizing distributed systems. In *Proc. 16th IEEE Symp. on Logic in Computer Science*, July 2001.

[140] S. Kutten and D. Peleg. Fault-local mending. *Journal of Algorithms*, 30(1).

[141] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, July 1982.

[142] J.-C. Laprie and B. Randell. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004. Avizienis, Algirdas and Landwehr, Carl.

[143] K. Larsen, P.Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[144] S. B. Lee and V. D. Gligor. Floc : Dependable link access for legitimate traffic in flooding attacks. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, ICDCS '10, 2010.

[145] N. Leveson. Completeness in formal specification language design for process-control systems. In *IN: PROCEEDINGS OF THE THIRD WORKSHOP ON FORMAL METHODS IN SOFTWARE PRACTICE*, pages 75–87, Portland, Oregon, 2000. ACM.

[146] J. R. Levine, T. Mason, and D. Brown. A freely available version of lex is flex. Technical report, http://flex.sourceforge.net/, 1992.

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

[148] D. Li, Y. Zhu, L. Cui, and L. M. Ni. Hotness-aware sensor networks. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[149] D. Li, Y. Zhu, L. Cui, and L. M. Ni. Hotness-aware sensor networks. In *Proceedings of the 28th International Conference on Distributed Computing Systems*, ICDCS '08, 2008.

[150] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Trace-based microarchitecture-level diagnosis of permanent hardware faults. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '08, 2008.

[151] Q. Li and D. Rus. Global clock synchronization in sensor networks. *IEEE Trans. Comput.*, 55(2):214–226, 2006.

[152] W. Li, E. Chan, D. Chen, and S. Lu. Maintaining probabilistic consistency for frequently offline devices in mobile ad hoc networks. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, 2009.

[153] Y. Li, C. Ai, and Y. W. Wiwek P. Deshmukh. Data estimation in sensor networks using physical and statistical methodologies. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[154] Y. Li and Z. Lan. A fast restart mechanism for checkpoint/recovery protocols in networked environments. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '08, 2008.

[155] Q. Lian, Z. Zhang, M. Yang, B. Y. Zhao, Y. Dai, and X. Li. An empirical study of collusion behavior in the maze p2p file-sharing system. In *ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems*, 2007.

[156] J. Lilius and I. P. Paltor. Formalising uml state machines for model checking. In *UML'99 Proceedings of the 2nd international conference on The unified modeling language: beyond the standard*, pages 430–444, 1999.

[157] M. Y. Lim, F. L. R. III, T. K. Bletsch, and V. W. Freeh. Padd: Power aware domain distribution. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, 2009.

[158] S. Lin, C. Tseng, T. Lee, and J. Fu. Vertaf: An application framework for the design and verification of embedded real-time software. *IEEE Transactions on Software Engineering*, 30(10):656–674, 2004. Member-Hsiung, Pao-Ann and Member-See, Win-Bin.

[159] S.-W. Lin, S.-W. Lin, C.-H. Tseng, T.-Y. Lee, and J.-M. Fu. Vertaf: An application framework for the design and verification of embedded real-time software. *IEEE Trans. Softw. Eng.*, 30(10):656–674, 2004. Member-Pao-Ann Hsiung and Member-Win-Bin See.

[160] Y. Lin, B. Li, and B. Liang. Differentiated data persistence with priority random linear codes. In *27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007)*, 2007.

[161] D. Liu. Resilient cluster formation for sensor networks. In *ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems*, 2007.

[162] D. Liu. Protecting neighbor discovery against node compromises in sensor networks. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, 2009.

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

[164] Z. Liu and M. Joseph. Specification and verification of fault-tolerance, timing, and scheduling. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(1):46–89, 1999.

[165] L. Lu, P. Sarkar, D. Subhraveti, S. Sarkar, M. Seaman, R. Jain, and A. Bashir. Carp: Handling silent data errors and site failures in an integrated program and storage replication mechanism. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, 2009.

[166] M. Lubaszewski and B. Courtois. A reliable fail-safe system. *IEEE Transactions on Computers*, 47(2):236–241, 1998.

[167] J. Luo, C. Huang, and L. Xu. Decoding star code for tolerating simultaneous disk failure and silent errors. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, 2010.

[168] T. Ma, J. Hillston, and S. Anderson. On the quality of service of crash-recovery failure detectors. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[169] 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.

[170] P. K. Manna, S. Ranka, and S. Chen. Analysis of maximum executable length for detecting text-based malware. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[171] A. F. Martinez and K. Kuchcinski. Graph matching constraints for synthesis with complex components. In *DSD '07: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pages 288–295, Washington, DC, USA, 2007. IEEE Computer Society.

[172] M. Marwah, S. Mishra, and C. Fetzer. Enhanced server fault-tolerance for improved user experience. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '08, 2008.

[173] K. McMillan. The smv system for smv version 2.5.4. Technical report, Carnegie Mellon University, 2000.

[174] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[175] A. Meixner and D. J. Sorin. Detouring: Translating software to circumvent hard faults in simple cores. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '08, 2008.

[176] J. W. Mickens and B. D. Noble. Concilium: Collaborative diagnosis of broken overlay routes. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[177] H. Moniz, N. F. Neves, and M. Correia. Turquois: Byzantine consensus in wireless ad hoc networks. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, 2010.

[178] R. Morales and I. Gupta. Avmon: Optimal and scalable discovery of consistent availability monitoring overlays for distributed systems. In *ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems*, 2007.

[179] M. Muthuprasanna and G. Manimaran. Distributed divide-and-conquer techniques for effective ddos attack defenses. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[180] A. R. Ningfang Mi, E. Smirni, and E. Riedel. Enhancing data availability in disk drives through background activities. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '08, 2008.

[181] O. of Inspector Genera. Audit report: Advance automation system. Technical Report Report Av-1998-113, USA Department of Transportation, April 1998.

[182] S. L. Pallemulle, H. D. Thorvaldsson, and K. J. Goldman. Byzantine fault-tolerant web services for n-tier and service oriented architectures. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[183] J. C. Park and J. R. Crandall. Empirical study of a national-scale distributed intrusion detection system: Backbone-level filtering of html responses in china. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, ICDCS '10, 2010.

[184] U. Paul, S. R. Das, and R. Maheshwari. Detecting selfish carrier-sense behavior in wifi networks by passive monitoring. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, 2010.

[185] R. Perdisci, M. Antonakakis, X. Luo, and W. Lee. Wsec dns: Protecting recursive dns resolvers from poisoning attacks. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '09, 2009.

[186] L. Pike, J. Maddalon, P. S. Miner, and A. Geser. Abstractions for fault-tolerant distributed system verification. In *17th International Conference Theorem Proving in Higher Order Logics (TPHOLs)*, pages 257–270, 2004.

[187] T. Pionteck and W. Brockmann. A concept of a trust management architecture to increase the robustness of nano age devices. In *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*, 2010.

[188] I. Prasetya. Mechanically verified self-stabilizing hierarchical algorithms. In *Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 399–415, 1997.

[189] S. Qadeer and N. Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In *IFIP International Conference on Programming Concepts and Methods (PROCOMET' 98)*, 1998.

[190] A. Rahmati, L. Zhong, M. A. Hiltunen, and R. Jana. Reliability techniques for rfid-based object tracking applications. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[191] P. Ramanathan, K. G. Shin, and R. W. Butler. Fault-tolerant clock synchronization in distributed systems. *Computer*, 23(10):33–42, 1990.

[192] D. Ramsbrock, R. Berthier, and M. Cukier. Profiling attacker behavior following ssh compromises. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[193] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan. Towards understanding the effects of intermittent hardware faults on programs. In *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*, 2010.

[194] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems (TOCS)*, 7:61–77, 1989.

[195] V. K. Reddy and E. Rotenberg. Coverage of a microarchitecture-level fault check regimen in a superscalar processor. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '08, 2008.

[196] R. Riley, X. Jiang, and D. Xu. An architectural approach to preventing code injection attacks. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[197] E. Rozier, W. Belluomini, V. Deenadhayalan, J. Hafner, K. K. Rao, and P. Zhou. Evaluating the impact of undetected disk errors in raid systems. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '09, 2009.

[198] H. Ruess, N. Shankar, and M. K. Srivas. Modular verification of srt division. *Form. Methods Syst*, 14, 1999.

[199] J. Rumbaugh, I. Jacobson, and B. G. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.

[200] J. Rumbaugh, I. Jacobson, and B. G. *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*. Newnes, 2008.

[201] O. Rutti, Z. Milosevic, and A. Schiper. Generic construction of consensus algorithms for benign and byzantine faults. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, 2010.

[202] T. Sakata, T. Hirotsu, H. Yamada, and T. Kataoka. A cost-effective dependable microcontroller architecture with instruction-level rollback for soft error recovery. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[203] N. Salatge and J.-C. Fabre. Fault tolerance connectors for unreliable web services. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[204] H. Schiöberg, R. Merz, and C. Sengul. A failsafe architecture for mesh testbeds with real users. In *MobiHoc S3 '09: Proceedings of the 2009 MobiHoc S3 workshop on MobiHoc S3*, pages 29–32, New York, NY, USA, 2009. ACM.

[205] N. Schiper and S. Toueg. A robust and lightweight stable leader election service for dynamic systems. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '08, 2008.

[206] R. D. Schlichting and F. B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst*, 1983.

[207] C. Sengul and R. Kravets. Heuristic approaches to energy-efficient network design problem. In *ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems*, 2007.

[208] M. Serafini, P. Bokor, D. Dobre, M. Majuntke, and N. Suri. Scrooge: Reducing the costs of fast byzantine replication in presence of unresponsive replicas. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, 2010.

[209] M. Serafini, N. Suri, J. Vinter, A. Ademaj, W. Brandstatter, F. Tagliabo, and J. Koch. A tunable add-on diagnostic protocol for time-triggered systems. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[210] Y. Shi and G. Lee. Augmenting branch predictor to secure program execution. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[211] A. Shye, T. M. andVijay Janapa Reddi, J. Blomstedt, and D. A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[212] R. Sion. Strong worm. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[213] F. Somenzi. Cudd: Cu decision diagram package. Technical report, University of Colorado at Boulder, 2012.

[214] P. Sommer and R. Wattenhofer. Gradient clock synchronization in wireless sensor networks. In *IPSN '09: Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pages 37–48, Washington, DC, USA, 2009. IEEE Computer Society.

[215] H. J. Song and A. A. Chien. Feedback-based synchronization in system area networks for cluster computing. *IEEE Trans. Parallel Distrib. Syst.*, 16(10):908–920, 2005.

[216] W. Sootkaneung and K. K. Saluja. Gate input reconfiguration for combating soft errors in combinational circuits. In *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*, 2010.

[217] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, 2002.

[218] P. Sousa, A. N. Bessani, W. S. Dantas, F. Souto, M. Correia, and N. F. Neves. Intrusion-tolerant self-healing devices for critical infrastructure protection. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '09, 2009.

[219] A. C. Squicciarini, A. Trombetta, and E. Bertino. Robust and secure interactions in open distributed systems through recovery of trust negotiations. In *ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems*, 2007.

[220] C. T. and T. S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 1996.

[221] G. Tan, S. A. Jarvis, and A.-M. Kermarrec. Connectivity-guaranteed and obstacle-adaptive deployment schemes for mobile sensor networks. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[222] C. Temple. Avoiding the babbling-idiot failure in a time-triggered communication system. In *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, page 218, Washington, DC, USA, 1998. IEEE Computer Society.

[223] M. T. Thai, Y. Xuan, I. Shin, and T. Znati. On detection of malicious users using group testing techniques. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[224] B. Tong, Z. Li, G. Wang, and W. Zhang. How wireless power charging technology affects sensor network deployment and routing. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, ICDCS '10, 2010.

[225] T. Tsuchiya, S. Nagano, R. B. Paidi, and T. Kikuno. Symbolic model checking for self-stabilizing algorithms. *IEEE Trans. Parallel Distrib. Syst*, 12:81–95, 2001.

[226] N. Tuck, B. Calder, and G. Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, 2004.

[227] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Journal of Automated Software Engineering*, 10(2):203–232, 2003.

[228] S. Wan, Q. Cao, C. Xie, B. Eckart, and X. He. Code-m: A non-mds erasure code scheme to support fast recovery from up to two-disk failures in storage systems. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, 2010.

[229] W. Wang and C. Amza. On optimal concurrency control for optimistic replication. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, 2009.

[230] W. Wang, D. Pu, and A. Wyglinski. Detecting sybil nodes in wireless networks with physical layer network coding. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, 2010.

[231] Y. Wang, H. Wu, F. Li, and N.-F. Tzeng. Protocol design and optimization for delay/fault-tolerant mobile sensor. In *27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007)*, 2007.

[232] Y. Wang and J. Wu. A nonblocking approach for reaching an agreement on request total orders. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[233] S. Weiss, P. Urso, and P. Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, 2009.

[234] J. Widder, G. Gridling, B. Weiss, and J.-P. Blanquart. Synchronous consensus with mortal byzantines. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[235] S.-H. Wu, C.-M. Chen, and M.-S. Chen. An asymmetric quorum-base power saving protocol for clustered ad hoc networks. In *ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems*, 2007.

[236] S.-H. Wu, M.-S. Chen, and C.-M. Chen. Fully adaptive power saving protocols for ad hoc networks using the hyper quorum system. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, ICDCS '08, 2008.

[237] W. Xiao and Q. Yang. Can we really recover data if storage subsystem fails? In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[238] T. Xie and A. Sharma. Collaboration-oriented data recovery for mobile disk arrays. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, 2009.

[239] K. Xing, F. Liu, X. Cheng, and D. H.-C. Du. Real-time detection of clone attacks in wireless sensor networks. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[240] Y. Yamauchi, T. Masuzawa, and D. Bein. Preserving the fault-containment of ring protocols executed on trees. *British Computer Journal*, 52(4), July 2009.

[241] G. Yan, L. Cuellar, S. Eidenbenz, and N. W. Hengartner. Blue-watchdog: Detecting bluetooth worm propagation in public areas. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '09, 2009.

[242] G. Yan and S. Eidenbenz. Modeling propagation dynamics of bluetooth worms. In *ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems*, 2007.

[243] X. Yang, P. Wang, H. Fu, Y. Du, Z. Wang, and J. Jia. Compiler-assisted application-level checkpointing for mpi programs. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[244] Q. Ye and L. Cheng. Dtp: Double-pairwise time protocol for disruption tolerant networks. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[245] M. Young, A. Kate, I. Goldberg, and M. Karsten. Practical robust communication in dhts tolerating a byzantine adversary. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, ICDCS '10, 2010.

[246] S. Yu and Y. Zhang. R-sentry: Providing continuous sensor services against random node failures. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[247] W. Yu, W. Hongyi, L. Feng, and T. Nian-Feng. Protocol design and optimization for delay/fault-tolerant mobile sensor networks. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, ICDCS '07, 2007.

[248] W. Yu, N. Zhang, X. Fu, R. Bettati, and W. Zhao. On localization attacks to internet threat monitors: An information-theoretic framework. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '08, 2008.

[249] Y. Yue, L. Tian, H. Jiang, F. Wang, D. Feng, Q. Zhang, and P. Zeng. Rolo: A rotated logging storage architecture for enterprise data centers. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, ICDCS '10, 2010.

[250] D. Zhang and D. Liu. Dataguard: Dynamic data attestation in wireless sensor networks. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, 2010.

[251] L. Zhang and Y. Guan. Detecting click fraud in pay-per-click streams of online advertising networks. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[252] Y. Zhang, Z. M. Mao, and J. Wang. A firewall for routers: Protecting against routing misbehavior. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[253] Y. Zhang, Z. Zhang, Z. M. Mao, and Y. C. Hu. Hc-bgp: A light-weight and flexible scheme for securing prefix ownership. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '09, 2009.

[254] Z. Zhang, W. Wu, and S. Shekhar. Optimal placements in ring network for data replicas in distributed database with majorityvoting protocol. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, 2008.

[255] Y. Zhao, S. Vemuri, J. Chen, Y. Chen, H. Zhou, and Z. Fu. Exception triggered dos attacks on wireless networks. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '09, 2009.

[256] Y. Zhou, X. Chen, M. R. Lyu, and J. Liu. Sentomist: Unveiling transient sensor network bugs via symptom mining. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, ICDCS '10, 2010.

[257] Q. Zhu and C. Yuan. A reinforcement learning approach to automatic error recovery. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.

[258] Y. Zhuang, L. Chen, Xiaoyang, X. S. Wang, and J. Lian. A weighted moving average-based approach for cleaning sensor data. In *27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007)*, 2007.

[259] P. Zielinski. Automatic verification and discovery of byzantine consensus protocols. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, 2007.