2011

This is to certify that the
dissertation entitled

Towards Automated Model Revision For Fault-Tolerant Systems

presented by

FUAD ABUJARAD

has been accepted towards fulfillment
of the requirements for the

Ph.D. degree in Computer Science

[illegible]
Major Professor's Signature

8/1/10
Date

TOWARDS AUTOMATED MODEL REVISION FOR FAULT-TOLERANT SYSTEMS

By

FUAD ABUJARAD

A DISSERTATION

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

DOCTOR OF PHILOSOPHY

Computer Science

2010

ABSTRACT

TOWARDS AUTOMATED MODEL REVISION FOR FAULT-TOLERANT SYSTEMS

By

FUAD ABUJARAD

Automated model revision of distributed programs is one of the emerging and important approaches for achieving and maintaining program correctness. In this approach, an existing model is automatically revised to satisfy new properties. Such model revision is required when an existing model/program is subject to a newly identified fault, a new requirement, or a new environment. Thus, model revision is especially beneficial in the development of systems that need high assurance. To apply model revision in practice, we need to develop tools that are user friendly, comprehensive, and efficient.

However, due to their limitations, the current model revision tools and techniques are not widely used in the development of practical systems. More specifically, some of the limitation are that they suffer from a high learning curve, they require high time and space complexity, they need many details to be specified that otherwise could be automatically discovered, and they do not cover different types of revision.

Taking into consideration the aforementioned limitations, in this dissertation, we derive theories, develop algorithms, and build tools to advance the state-of-the-art of the automated model revision. Our approach comprises four main elements: First, we reduce the learning curve for the automated model revision techniques by utilizing existing design tools to perform the revision under-the-hood. Second, to permit the designer to efficiently describe the model to be synthesized and to minimize the user input, we develop algorithms and tools to automate the generation of the legitimate states of the original model, thereby reducing the burden of the designer. Third, to utilize the available computing resources and to efficiently complete the revision, we utilize both symmetry and parallelism to speedup the automated revision and to overcome its bottlenecks. Fourth, to provide comprehensive revision and to cover more types of model revision, such as nonmasking and stabilizing fault-tolerance, we develop algorithms and tools to allow for addition of new types of fault-tolerance. To validate our approach and illustrate its feasibility, we apply it to several case studies.

I dedicate this dissertation to my wonderful family. Particularly, to my parents, who believed in diligence, science, and the pursuit of academic excellence. To my beloved wife, Samah, who has been patient and supportive with these many years of research, and to our lovely kids Haya, Khaled, and Amir, who are the joy of our lives.

ACKNOWLEDGMENTS

I am extremely grateful to all who helped me complete my Ph.D. program. First and foremost, it was the unconditional support of my wife, Samah. Her support, encouragement, quiet patience, and unwavering love were undeniably the bedrock upon which the past eleven years of my life have been built. Her *tolerance* of my changing moods is a testament in itself of her unyielding devotion and love. I would like to thank our three children, Haya, Khaled, and Amir who made this all possible. My family made tremendous sacrifices so that I could spend time on my doctoral education. They encouraged and pushed me to continue in my pursuit.

I would like to gratefully and sincerely thank Dr. Sandeep S. Kulkarni for his guidance, understanding, and patience during my graduate studies at Michigan State University. His mentorship was paramount in providing a well-rounded experience consistent with my long-term career goals. He encouraged me to not only grow as an experimentalist but also as an independent thinker. For everything you have done for me, Dr. Kulkarni, I thank you.

I would like to thank the Department of Computer Science and Engineering at MSU, especially those members of my doctoral committee for their input, valuable discussions, and availablity. In particular, I would like to thank Dr. Laura Dillon and Dr. Betty H. C. Cheng, as well as Dr. Jonathan Hall from the Department of Mathematics. This dissertation would not have been nearly as complete without your help.

Additionally, I am very grateful for the friendship of all the members of the SENS lab research group, especially Ali Ebnenasir, Mahesh Arumugam, Borzoo Bonakdarpour, and Jingshu Chen, with whom I worked closely and co-authored some of my papers during my

Ph.D. program.

Finally, and most importantly, I would like to acknowledge my parents, Suleiman and Hamamah, for their unconditional love and for their faith in me. It was under their watchful eye that I gained so much self-steam and an ability to tackle challenges. Also, I would like to thank my brothers and sisters for their continuous support and unending encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The rapid growth of computer systems is increasing our reliance on them more than ever. Therefore, the burden of ensuring the correctness of reliable hardware and software systems is significantly growing. Model checking is one of the commonly used techniques to provide such assurance, especially for finite state concurrent systems [48,49,64]. Given a model of a system, a model checker verifies whether this model meets a given property. If the model does not satisfy that property, the model checker (typically) gives a counter-example. Then, the model needs to be modified to satisfy the desired property. Consequently, such modification will require another cycle of verification.

Based on this observation, in this dissertation, we focus on model revision [26,29, 61,96] where an existing model is revised so that it satisfies a given property of interest. Model revision is required in several contexts. For example, it is required to revise an existing model to fix a counter-example, i.e. a bug. It is also required if the original specification was incomplete and the model has to be revised to meet the missing specification. Furthermore, it is required to respond to faults introduced by a change in the environment. When a program is deployed in a new environment, it may be subject to faults that were not considered in the original design. Moreover, even if the faults were known in the initial design, to provide separation of concerns, it is desirable to allow the designer to focus on the functionality aspect and add fault-tolerance subsequently. In either case, it is desired that we revise the program to add fault-tolerance.

One requirement for such revision is that the existing program requirements continue to be satisfied [101]. Also, in the above contexts, it is more practical to reuse the existing program in the construction of the revised one [25]. Performing such revisions manually has the potential to incur a huge cost as well as introduce new errors. Therefore, automating such revisions is desirable for reducing cost and guaranteeing assurance. One approach to gain assurance in such program revision is by *automated model revision* (also known as automated incremental synthesis) [27,30,31,55,59,101,103], which guarantees that the revised program is correct-by-construction. The automated model revision to add fault-tolerance takes a fault-intolerant program, program specifications, and faults as an input and generates a fault-tolerant program as an output. More specifically, it reuses the original program (which is fault-intolerant) in synthesizing its fault-tolerant version [101]. Moreover, since the synthesized program is correct-by-construction there is no need to reverify its correctness.

The automated model revision (or, incremental synthesis) of fault-tolerant programs is highly desirable, as it allows the designer to focus on the normal system behavior in the absence of faults and leaves the fault-tolerance aspect to the automated techniques. Initially, Kulkarni and Arora [101,102] presented an algorithm for synthesizing fault-tolerant programs. The input to their algorithm is a fault-intolerant program that satisfies its specification in the absence of faults but provides no guarantees in the presence of faults. The output of their algorithm is a fault-tolerant program that continues to satisfy its specifications in the absence of faults and provides the desired level of fault-tolerance to tolerate the given faults. Later, in [59] Ebnenasir and Kulkarni presented an enumerative (explicit-state) implementation to the revision algorithm. This was a significantly important step, since it enabled them to verify the concepts of the revision and demonstrate the applicability of the automated revision algorithms [59]. However, similar to other enumerative implementations, it was subject to the state explosion problems and was only suitable to revise small programs. Recently, Bonakdarpour and Kulkarni presented a symbolic-based implementation for the revision algorithm [27,30]. In this implementation, the components of the revision algorithm are constructed using Boolean formulae represented by Bryants Ordered Binary Decision Diagrams [33]. This was the first time where moderate to large sized programs (a state space of 10^{50} and beyond) have been synthesized. The symbolic implementation enabled them to identify bottlenecks in the automated revision. These bottlenecks included, deadlock resolution, computation of reachable states in the presence of faults, and addition of recovery paths.

1.0.1 Motivations and Goals

In practice, applying the automated model/program revision in real life applications is difficult due to the following factors:

1. The use of the existing tools for automated model revision has a high learning curve. The designer is required to learn different aspects of modeling distributed programs, program specification, faults, and fault-tolerance [27,30,59]. To alleviate this difficulty, we focus on moving the task for adding fault-tolerance to be *under-the-hood*. In this manner, we make automated revision more accessible [3].

2. Current model revision tools require the designer to specify the fault-intolerant model, the model specifications, the model legitimate states, and the faults [27,30,31, 59,101,103]. Of those, identifying the set of legitimate states is the most demanding task. The designer needs to specify the legitimate states of the model and describe them in a logical formula. Although specifying the model, the specifications, and the faults is a must, it is an open question as to whether the explicit specification of the legitimate states is necessary. To alleviate this difficulty, we focus on designing an algorithm that provides automatic generation of the legitimate states from the model

actions and specifications [6].

3. Current model revision tools [27, 30, 59] focus on the addition of masking fault-tolerance, where both safety and liveness are preserved during recovery. However, they do not address other types of fault-tolerance, including nonmasking and stabilizing fault-tolerance. In nonmasking fault-tolerance, safety can be violated during recovery and the program should tolerate temporary perturbation. In stabilizing fault-tolerance, the program recovers to its legitimate states from any arbitrary state [57]. To provide broader domain of the problems that can be resolved by automated model revision, we develop algorithms for the automated addition of nonmasking and stabilizing fault-tolerance [4].

4. The current model revision tools utilize multiple heuristics to reduce the complexity of the revision [27,30,59,101]. However, to improve the efficiency further, we need to utilize advantages from model checking [48,49,64,93]. Hence, we develop techniques that concentrate on reducing the complexity of the revision using symmetry and/or parallelism. We show that these approaches provide a significant speedup separately as well as together [5].

1.0.2 Thesis

Thesis Statement: The automated model revision can be made more usable, comprehensive, and efficient through the use of four key elements: the use of existing design tools as front end to the automated model revision tools, the introduction of new revision algorithms that handle different classes of fault-tolerance, the use of the original model specification and actions to automatically discover other inputs to the revision algorithm, and the utilization of symmetry and parallelism.

To validate this thesis statement, we have derived theories, developed algorithms, and built tools to advance the automated model revision through a usable, comprehensive, and efficient toolset. First, to reduce the automated revision learning-curve, we utilized existing design tools (i.e. SCR toolset) such that the automated revision is done *under-the- hood* [3]. Second, to revise a broader range of programs, we developed algorithms and tools to add new types of fault-tolerance [4,5]. Next, we reduced the revision parameters by automating the discovery of the program legitimate states, thereby reducing the burden of the designer [6]. Finally, to overcome the automated revision bottlenecks and reduce its time complexity, we utilize both symmetry and parallelism to speedup the revision time [2,5].

1.0.3 Contributions

Our contributions can be grouped into four major categories:

Under-the-hood Revision

It is desirable that the designer utilizes the automated model revision tools with minimal prerequisite knowledge of the details of the automated revision techniques. We focus on performing the automated revision under-the-hood. Therefore, we utilize existing design tools, such as the SCR toolset [21,84,87], in the automated revision. The SCR toolset is a set of tools used to formally construct and verify the requirements specification document. It is widely used in constructing many mission critical systems. Our approach is to combine the SCR toolset with the tool SYCRAFT that automates the model revision. This approach is desirable, as it allows one to perform functions of the automated model revision without the need to know its details. Of course, it would be necessary to convert (1) the SCR specification into a format that can be used with SYCRAFT and (2) the revised fault-tolerant program into corresponding SCR specification.

Based on the above discussion, we combine the SCR toolset with the automated model revision tool SYCRAFT [27,30]. More specifically, we let the designer specify the program requirements through the SCR toolset interface and we handle the aspects of the automated revision of fault-tolerance using SYCRAFT.

Legitimate States Generator

One of the requirements of the model revision algorithm is identifying the set of the legitimate states of the program being synthesized. This set represents the states from where the execution of the actions of the model is correct. One approach for providing fault-tolerance is to ensure that after the occurrence of faults, the revised program eventually recovers to the legitimate states of the original model. Since the original model met its original specification from these legitimate states, we can ascertain that eventually a revised model reaches states from where subsequent computation is correct.

One of the problems in providing recovery to the legitimate states, however, is that these legitimate states are not always easy to determine. Existing model revision approaches (e.g., SYCRAFT [27,30]) have required the designer to specify these legitimate states explicitly. It is straightforward to observe that if these legitimate states could be derived automatically, then it would reduce the burden put on the designer, thereby making it easier to apply these techniques in revision of existing programs. We focus on identifying the largest set of states from where the existing model is correct.

Nonmasking and Stabilizing Fault-Tolerance.

To provide comprehensive tools for the automated model revision, we focus our attention on automated addition of *nonmasking* and *stabilizing* fault-tolerance to fault-intolerant programs. Intuitively, a nonmasking fault-tolerant program ensures that if the program is perturbed by faults to an illegitimate state, then it would eventually recover to its legitimate states. However, safety may be violated during recovery [101].

The current model revision tools [30,59] support the design of masking fault-tolerance only. However, there are several reasons that make the design of nonmasking fault-tolerance attractive. For one, the design of masking fault-tolerant programs, where both safety and liveness are preserved during recovery, is often expensive or impossible, even though the design of nonmasking fault-tolerance is easy [15]. Also, the design of nonmasking fault-tolerance can assist and simplify the design of masking fault-tolerance [105].

A special case of nonmasking fault-tolerance is the stabilizing fault-tolerance [54,56], where, starting from an arbitrary state, the program is guaranteed to reach a legitimate state. Stabilizing systems are especially useful in handling unexpected transient faults. Moreover, this property is often critical in long-lived applications where faults are difficult to predict.

We present an algorithm for adding nonmasking fault-tolerance to an existing program by performing three steps [4]. The first step is to identify the set of legitimate states of the fault-intolerant program. This set defines the constraints that should be true in the legitimate states. The second step is to identify a set of convergence actions that recover the program from illegitimate states to a legitimate state. This can be done by finding actions that satisfy one or more constraints. The last step consists of ensuring that the convergence actions do not interfere with each other. In other words, the collective effect of all recovery actions should, eventually, lead the program to legitimate states.

Expediting the Automated Revision

To reduce the time complexity of the automated model revision, we first need to identify bottleneck(s) where symmetry and parallelism features can provide the maximum impact. Based on the analysis of the experimental results from Bonakdarpour and Kulkarni [30], the performance of the revision suffers from two major complexity obstacles, namely *generation of fault-span* and *resolution of deadlock states*.

To effectively target those bottlenecks, we present two approaches for utilizing the multi-core architecture in reducing the time required to complete the automated revision. The first approach is based on the distributed nature of the program being revised. In particular, when a new transition is added (respectively, removed), since the process executing it has only a partial view of the program variables, we need to add (respectively, remove) a *group* of transitions based on the variables that cannot be read by the process. The second approach is based on partitioning deadlock states among multiple threads. We show that this provides a small performance benefit. Based on the analysis of these results, we argue that the simple approach that parallelizes the group computation is likely to provide maximum benefit in the context of deadlock resolution in the automated revision of distributed programs. To further expedite the automated model revision, we use symmetry speedup the revision algorithm.

1.0.4 Outline

The remainder of this thesis is organized as follows. Chapter 2 describes the preliminaries and presents the elements of the automated incremental model revision. In Chapter 3, we present our approach to minimize the prerequisite knowledge of the details of the automated revision techniques and provide practical approaches to perform the automated revision under-the-hood. In Chapter 4, we show how we utilize parallelism and symmetry to expedite the automated model revision. Subsequently, to revise a broader range of programs, in Chapter 5 we present our approach for the automated addition of *nonmasking* and *stabilizing* fault-tolerance to fault-intolerant programs. In Chapter 6, we show how we can reduce the designer burden by automatically discovering the legitimate states of the model being revised. Later, in Chapter 7, we analyze the effect of performing the automated model revision without explicitly specifying the legitimate states. We present the related work and literature review in Chapter 8. Finally, we present a summary of our contributions and future research direction in Chapter 9.

Chapter 2

Preliminaries

In this chapter, we formally present the elements of our automated model revision framework. Mainly, we define the notion of models, programs, specifications, faults, and fault-tolerance. The notion of distributed programs is adapted from Kulkarni and Arora [101]. Definition of faults and fault-tolerance are based on the ones given by Arora and Gouda [12], Kulkarni [100], and Bonakdarpour [25]. At the end of this chapter, we illustrate the basic constructs of this framework using a real-world example, an application in sensor networks.

2.1 Models and Programs

In this section, we present the formal definition of models and programs. A model is described by an abstract program. Intuitively, a program, p, is described using a finite set of variables $V_p = \{v_0, v_1, \ldots, v_n\}$, $n \geq 0$, and a finite set of program actions $A_p = \{a_0, a_1, \ldots, a_m\}$, $m \geq 0$. Each variable, $v_i \in V_p$, is associated with a finite domain of values, D_i. Let $a_i \in A_p$ be an action, then a_i is defined as follows: $a_i :: g_i \longrightarrow st_i$; where g_i is a Boolean formula involving program variables and st_i is a deterministic terminating statement that updates a subset of program variables.

Before we give a formal definition of programs based on this intuition, we define the notion of state space and state predicate.

Definition 2.1.1 (state) A state, s, of program p is identified by assigning each variable in V_p a value from its respective domain, D_i. ∎

Definition 2.1.2 (state space) The state space, S_p, of p is the set of all possible states of p. ∎

Definition 2.1.3 (state predicate) A state predicate of p is Boolean expression defined over the program variables V_p. Thus, a state predicate C of p identifies the subset, $S_C \subseteq S_p$, where C is *true* in a state s iff $s \in S_C$. ∎

Note that state predicate corresponds to a set of states where the Boolean value of the corresponding predicate is *true*. Thus, the intersection of two state predicates corresponds to the conjunction of corresponding functions. Likewise, disjunction corresponds to union, and so on. Hence, we use these Boolean operators for constructing different state predicates. For example, let C_1 and C_2 be state predicates that identify the state space subsets S_{C_1} and S_{C_2}, then $C_1 \wedge C_2$ (respectively $C_1 \vee C_2$) correspond to $S_{C_1} \cap S_{C_2}$ (respectively $S_{C_1} \cup S_{C_2}$).

Definition 2.1.4 (transition predicate) Intuitively, a program action consists of one or more transitions. Let $(a_i :: g_i \longrightarrow st_i;)$ be an action of the program. Then, the corresponding transitions included in this action are α_i, where $\alpha_i = \{(s_0, s_1) \mid g_i$ is true in s_0 and s_1 is obtained by executing st_i from $s_0\}$. ∎

Hence, a transition predicate correspond to an action is a subset of $S_p \times S_p$. A single transition t is specified by the tuple (s_0, s_1), where $s_0, s_1 \in S_p$ and s_0 is the *before* state and s_1 is the *after* state.

Given a program that is defined in term of V_p and A_p, we can now identify an equivalent representation in terms of its state space and transitions. In particular based on V_p and A_p, we can compute S_p, the state space of p and α_i for each action of p. Based on the above, we formally define the program as follows.

Definition 2.1.5 (program) The program p is defined as the tuple $\langle S_p, (\alpha_1, \alpha_2, \alpha_3, ...\alpha_l)\rangle$ where $\alpha_i \in S_p \times S_p$. ∎

In many instances, we do not need the details of the individual actions of p. For these cases, we utilize program transitions δ_p. For the program $p = \langle S_p, (\alpha_1, \alpha_2, \alpha_3, ...\alpha_l)\rangle$, the transitions of p is $\delta_p = (\alpha_1 \cup \alpha_2 \cup \alpha_3 \cup \ldots \cup \alpha_l)$. Whenever it is clear from the context, we use p and its transitions δ_p interchangeably.

Definition 2.1.6 (closed) Let S_c be a state predicate, then S_c is closed in a program p iff $(\forall\ (s_0, s_1) : (s_0, s_1) \in \delta_p : (s_0 \in S_c \Rightarrow s_1 \in S_c))$. ∎

Definition 2.1.7 (enabled) The action a_i is enabled in a state s_j iff the guard of $g_i = true$ in the states s_j. ∎

Definition 2.1.8 (unfair computation) A sequence of states, $\sigma = \langle s_0, s_1, ...\rangle$ is unfair computation of p iff

1. $\forall j : 0 < j < length(\sigma) : (s_{j-1}, s_j)$, is obtained by executing a program action, say $(a_i :: g_i \longrightarrow st_i)$. That is, g_i is $true$ in s_{j-1} and s_j is obtained by executing st_i, and

2. if σ is finite and terminates in s_l then all the guards of the program actions are false in s_l. ∎

Computations can also be fair. Intuitively, a fair computation allows a fair resolution for non-determinism. Next, we define weak and strong fair-computation.

Definition 2.1.9 (weak-fair computation) $\sigma = \langle s_0, s_1, ...\rangle$ is weak-fair computation of p if:

1. σ is an unfair 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 in state s_k. ∎

In weak-fair computation, if some guard , say g_i, eventually becomes continuously enabled, then the corresponding action is guaranteed to execute infinitely often.

Definition 2.1.10 (strong-fair computation) $\sigma = \langle s_0, s_1, ... \rangle$ is strong-fair computation iff:

1. σ is an unfair computation of p, and

2. there exists an action $a_i : g_i \longrightarrow st_i$ of p such that g_i is true in s and s' is obtained by executing st_i in state s, then the transitions (s, s') are included infinitely often in σ. ∎

In strong-fair computation, if some guard , say g, is continuously enabled forever then the corresponding action must execute infinitely often.

Note that, in this dissertation, we refer to weak-fair computation as a *fair* computation. Also, our definition of weak-fair computation is equivalent to *weak fairness* from [1,9,65].

2.2 Modeling Distributed Programs

Since we focus on the design of distributed programs, we specify the transitions of the program in terms of a set of processes, where every process can read and write a subset of the program variables. The transitions of a process are obtained by considering how that process updates the program variables. The transitions of the program are the union of the transitions of its processes.

Definition 2.2.1 (process) A process P_j is specified by the tuple $\langle \delta_j, R_j, W_j \rangle$ where δ_j is a transition predicate in S_p and $\delta_p = \bigcup_{j=1}^{m} \delta_j$. R_j is the set of variable that the process P_j is allowed to read, and W_j is the set of variables that the process P_j is allowed to write and $W_j \subseteq R_j \subseteq V$ (i.e., we assume that the program must, first, read the variable to be able to write it.). ∎

Notation. Let $v_a(s_0)$ denote the value of variable v_a in the state s_0.

A process in a distributed program has a partial view of the program variables, which introduces *write/read restrictions*. Therefore, when a new program transition is added/removed, we need to add/remove a *group* of transitions based on the variables that cannot be read/writen by that process. The write/read restrictions of the process are defined as follows.

2.2.1 Write Restrictions

Let $P_j = \langle \delta_j, R_j, W_j \rangle$ be a process, then the only variables that P_j can write are variables in W_j. If P_j can only write the subset of variables W_j and the value of a variable other than that in W_j is changed in the transition (s_0, s_1), then that transition cannot be used in synthesizing the transitions of P_j. In other words, being able to write the subset W_j is equivalent to providing a set of transitions $write_j(W_j)$ that P_j cannot use in the revision algorithm. Clearly, the transition predicate $write_j(W_j)$ is defined as follows.

$$write_j(W_j) = \{(s_0, s_1) : (\forall v_a :: v_a \in (V_p - W_j) : \; v_a(s_0) \neq v_a(s_1) \;)\}.$$

2.2.2 Read Restrictions

Let $P_j = \langle \delta_j, R_j, W_j \rangle$, the only variables that P_j can read are variables belonging to R_j. let $t = (s_0, s_1)$ be a transition in δ_j then we define $group_j(t)$ as the group of transitions associated with t. Such a group includes transitions of the form (s_2, s_3) where s_0 and s_2 (respectively s_1 and s_3) are undistinguishable for P_j. By undistinguishable, we mean that they differ only in terms of the variables that P_j cannot read. Thus, we formally define $group_j(t)$ as follows:

$$\begin{aligned} group_j(t) = \bigvee_{(s_2,s_3)} & \\ & (\; \textstyle\bigwedge_{v \notin R_j} (\; v(s_0) = v(s_1) \; \wedge \; v(s_2) = v(s_3) \;) \wedge \\ & \quad \textstyle\bigwedge_{v \in R_j} (\; v(s_0) = v(s_2) \; \wedge \; v(s_1) = v(s_3) \;) \;). \end{aligned}$$

2.2.3 Example (Group)

Let p be a program specied using the set of processes $P = \{P_1(= \langle \delta_1, R_1, W_1 \rangle), P_2(= \langle \delta_2, R_2, W_2 \rangle)\}$, the set of variables $V = \{v_1, v_2\}$, and the domains $D_{v_1} = \{0, 1\}$ and

$D_{v_2} = \{0,1\}$. Also, let $R_1 = \{v_1\}$ (respectively $R_2 = \{v_2\}$) and $W_1 = \{v_1\}$ (respectively $W_2 = \{v_2\}$) (i.e. each process can only to read and write its own variable). Now, consider the transition from the state $\langle v_1 = 0, v_2 = 0\rangle$ to the state $\langle v_1 = 1, v_2 = 0\rangle$. If this transition is to be included in δ_1 then it is necessary to include the transition from the state $\langle v_1 = 0, v_2 = 1\rangle$ to the state $\langle v_1 = 1, v_2 = 1\rangle$. Clearly, this should be the case since P_1 is not allowed to read the variable v_2, therefore we have to consider the case where $v_2 = 0$ as well as the case where $v_2 = 1$. The automated model revision algorithm adds/removes program transitions to complete the revision. Therefore, whenever a transition is added or removed, the revision algorithm must add or remove the corresponding group.

2.2.4 The Group Algorithm

The group algorithm (c.f. Algorithm 1) takes a transition set, *trans*, as an input and computes the transition group, $trans_g$, as an output. Specifically, it creates an array, $tPred[]$, with number of elements equal to the number of processes such that $tPred[i]$ holds the part of the group transitions associated with the process i (Line 1). Now, based on W_i (i.e. the set of variables the process i is allowed to write) the group algorithm uses the function $AllowWrite_i(W_i)$ to find the set of all transitions which process i is permitted to execute. Then, it uses this set to find which of the transitions in *trans* process i is responsible for (Line 3). Later, it uses the $tPred[i]$ and R_i in the function *FindGroup* to account for all variables that process i cannot read and compute the transitions that cannot be distinguished by, i (Line 4). Once the steps in lines 3 and 4 are completed for all processes, the algorithm collects the transitions of the group in $trans_g$ (Lines 7-9) and returns.

Observe that for the transition t, $group_j(t)$ can be executed by process P_j while respecting its read/write restrictions. Let tr_j be a set of transitions. Now, based on the notion of read/write restrictions, tr_j can be included in δ_j iff there exist transitions $t_1, t_2, \ldots t_l$ such that $tr_j = group_j(t_1) \cup group_j(t_2) \cup \ldots group_j(t_l)$. Furthermore, let p be a program whose transitions specified with the processes P_1, P_2 $\ldots P_x$. Also, let tr_p denote a set of transi-

Algorithm 1 Group

Input: transitions set *trans*.

Output: transitions group $trans_g$.

```
1: MDD* $tPred$ := MDD[ $numberOfProcesses$] ;
2: for $i := 0$ to $numberOfProcesses$ do
3:     $tPred[i] := trans \wedge AllowWrite_i(W_i)$;
4:     $tPred[i] := FindGroup(tPred[i], R_i)$;
5: end for
6: MDD $trans_g := false$;
7: for $i := 0$ to $numberOfProcesses$ do
8:     $trans_g := trans_g \vee tPred[i]$;
9: end for

10: return $trans_g$;
```

tions. Then, tr_p can be included as transitions of p iff there exists a set of transitions tr_1, $tr_{j_2}, \ldots tr_x$ such that $tr_p = \bigcup_{j=1}^{x} tr_j$ and tr_j can be included as transitions of process P_j.

The way we use this group operation is as follows: When we compute a set of transitions, say tr, that we need to either add or remove, we ensure that tr can be implemented using read/write restrictions of the synthesized program. Hence, often, we cannot add/remove tr as is. Instead, we need to revise tr so that it respects the read/write restrictions of the program being revised. One operation we utilize for this is called *Group*, where $Group_{max}(tr)$ returns a *superset*, say tr_{larg}, that can be included as transitions of the synthesized program. The intuition of $Group_{max}$ operation is as follows (c.f. Algorithm 1): Given a set of transitions, say tr, we use a loop that traverses through all the processes. While traversing process P_j, it computes subset of transitions, say tr_j, in tr such that each transition in tr_j satisfies the write restrictions of process P_j. Then, for each transition in tr_j, it applies the group operation describe above to compute other transitions that must be included. (Note that with the use of BDDs and MDDs (i.e., Binary and Multi-Valued Decision Diagrams [125]), we do not have to actually evaluate each transition in tr_j separately to compute the corresponding group.) Finally, it takes a union of all transitions obtained thus to compute $Group_{max}(tr)$.

Another operation we utilize is $Group_{min}$. This operation returns a *subset*, say tr_{small}, such that tr_{small} can be included as transitions of the revised program. The operation $Group_{min}$ is implemented in a similar fashion to that of Group by traversing through all processes.

Remark. Since $Group_{max}$ is the operation that is used most frequently in our algorithms, for simplicity of presentation, we drop the subscript and call it Group.

Remark. Note that the $group_j(t)$ is defined only if t does not violate write restrictions of process P_j. However, for brevity, we do not specify this whenever it is clear from the context.

The tasks involved in computing one such group depend on the number of processes and the number of variables in the program. As can be seen from the formula above, to compute this group the algorithm (c.f. Algorithm 1) needs to go through all the processes in the program and for each process it has to go through all the variables.

2.3 Specification

Following Alpern and Schneider [7], it can be shown that any specification can be partitioned into some "safety" specification and some "liveness" specification. Intuitively, the safety specification indicates that nothing *bad* should happen. And, a liveness specification requires that something *good* must eventually happen. Formally,

Definition 2.3.1 (safety) The safety specification, Sf_p, for program p is specified in terms of bad states, $SPEC_{bs}$, and bad transitions $SPEC_{bt}$. A sequence $\langle s_0, s_1, ...\rangle$ (denoted by σ) *satisfies* the safety specification of p iff the following two conditions are satisfied.

1. $\forall j : 0 \leq j < len(\sigma) : s_j \notin SPEC_{bs}$, and
2. $\forall j : 0 < j < len(\sigma) : (s_{j-1}, s_j) \notin SPEC_{bt}$. ∎

Definition 2.3.2 (liveness) The liveness specification, Lv_p, of program p is specified in terms of one or more leads-to properties of the form $\mathcal{F} \leadsto \mathcal{T}$. A sequence $\sigma = \langle s_0, s_1, ... \rangle$ *satisfies* $\mathcal{F} \leadsto \mathcal{T}$ iff $\forall j : (\mathcal{F}$ is true in $s_j \Rightarrow \exists k : j \leq k < len(\sigma) : \mathcal{T}$ is true in $s_k)$. We assume that $\mathcal{F} \cap \mathcal{T} = \{\}$. If not, we can replace the property by $((\mathcal{F} - \mathcal{T}) \leadsto \mathcal{T})$. ∎

Remark. Observe that if p satisfies $\mathcal{F} \leadsto \mathcal{T}$, then it cannot contain computations that start from $\mathcal{F}$ and reach a deadlock/termination state without reaching a state in $\mathcal{T}$. Likewise, it cannot contain computations that start from $\mathcal{F}$ and reach a cycle without reaching $\mathcal{T}$.

Definition 2.3.3 (specification) A specification, say *spec* is a tuple $\langle Sf_p\ ,\ Lv_p \rangle$, where Sf_p is a safety specification and Lv_p is a liveness specification. A sequence σ satisfies *spec* iff it satisfies Sf_p and Lv_p. ∎

Based on the above definition, for simplicity, given a specification, say *spec*, defined as $\langle Sf_p\ ,\ Lv_p \rangle$ we say that *spec* is an intersection of Sf_p and Lv_p.

Given a program p and its specification, say *spec*, p may not satisfy *spec* from an arbitrary state. Rather, it satisfies *spec* only from its legitimate states (also known as invariant). We use the term *legitimate state predicate I* to denote the set of legitimate states of p. In particular, we say that a program p satisfies *spec* from I iff the following two conditions are satisfied:

1. I is closed in p, and

2. every computation of p that starts from a state in I satisfies *spec*.

A program p satisfies the (safety, liveness, or a combination of both) specification from the legitimate states, I, iff every computation of p that starts from a state in I satisfies that specification.

Definition 2.3.4 (legitimate state predicate) Let I be a state predicate, and p satisfies *spec* from I, then we say that I is the legitimate state predicate of p from *spec*. Note that a program may have multiple legitimate state predicates. ∎

2.4 Faults

The faults that may perturb a program are systematically represented by transitions. Based on the classification of faults from Avižienis *et al.* [18], this representation suffices for physical faults, process faults, message faults, and improper initialization. It is not intended for program errors (e.g. buffer overflow). However, if such errors exhibit behavior, such as, a component crash, it can be modeled using this approach. Thus, a fault for $p(=\langle S_p, \delta_p\rangle)$ is a subset of $S_p \times S_p$.

We use '$p[]f$' to mean 'p in the presence of f'. The transitions of $p[]f$ are obtained by taking the union of the transitions of p and the transitions of f.

Just as we defined computations of a program in Section 2.1, we define the notion of program computations in the presence of faults. In particular, a sequence of states, $\sigma = \langle s_0, s_1, \ldots\rangle$, is a computation of $p[]f$ (i.e., a computation of $p(=\langle S_p, \delta_p\rangle)$ in the presence of f) iff the following three conditions are satisfied:

1. $\forall j : 0 < j < len(\sigma) : (s_{j-1}, s_j) \in (\delta_p \cup f)$, and
2. if $\langle s_0, s_1, \ldots\rangle$ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in \delta_p$, and
3. if σ is infinite then $\exists n : \forall j > n : (s_{j-1}, s_j) \in \delta_p)$

Thus, if σ is a computation of p in the presence of f, then in each step of σ, either a transition of p occurs or a transition of f occurs. Additionally, σ is finite only if it reaches a state from where the program has no outgoing transition. And, if σ is infinite then σ has a suffix where only program transitions execute. We note that the last requirement can be relaxed to require that σ has a sufficiently long subsequence where only program transitions execute. However, to avoid details such as the length of the subsequence, we require that σ has a suffix where only program transitions execute.

We use f-span (*fault-span*) to identify the set of states from where the program satisfies its fault-tolerance requirement.

Definition 2.4.1 (*f*-**span**) Let T be a state predicate, then T is an f-span of p from I iff $I \Rightarrow T$ and $(\forall (s_0, s_1) : (s_0, s_1) \in p[]f : (s_0 \in T \Rightarrow s_1 \in T))$. ∎

Thus, at each state where I of p is true, the T of p from I is also true. Also, T, like I, is also closed in p. Moreover, if any action in f is executed in a state where T is true, the resulting state is also one where T is true. It follows that for all computations of p that start at states where I is true, T is a boundary in the state space of p, up to which (but not beyond which) the state of p may be perturbed by the occurrence of the actions in f.

2.5 Fault-Tolerance

In this section, we present a formal definition to three classical levels of fault-tolerance; namely, failsafe, masking, and nonmasking fault-tolerance.

Fault-Tolerance. In the absence of faults, a program, p, satisfies its specification and remains in its legitimate states. In the presence of faults, it may be perturbed to a state outside its legitimate states. By definition, when the program is perturbed by faults, its state will be one in the corresponding *f-span*. From such a state, it is desired that p does not result in a failure, i.e., p does not violate its safety specification. Furthermore, p recovers to its legitimate states from where p subsequently satisfies both its safety and liveness specification.

Based on this intuition, we now define what it means for a program to be (masking) fault-tolerant. Let Sf_p and Lv_p be the safety and liveness specifications for program p. We say that p is masking fault-tolerant to Sf_p and Lv_p from I iff the following two conditions hold.

1. p satisfies Sf_p and Lv_p from I.

2. $\exists T$ ::

    (a) T is f-span of p from I.

(b) $p[]f$ satisfies Sf_p from T.

(c) Every computation of $p[]f$ that starts from a state in T has a state in I.

While masking fault-tolerance is ideal, for reasons of costs and feasibility, a weaker level of fault-tolerance is often required. Two commonly considered weaker levels of fault-tolerance include *failsafe* and *nonmasking*. In particular, we say that p is *failsafe* fault-tolerant [72] if the conditions 1, 2a, and 2b are satisfied in the above definition. And, we say that p is *nonmasking* fault-tolerant [71] if the conditions 1, 2a, and 2c are satisfied in the above definition.

2.6 Example: (Data Dissemination Protocol in Sensor Networks)

In this example, we show how we model distributed programs and illustrate some of the previous definitions from the previous sections. We use the program *Infuse*, a time division multiple access (TDMA) based reliable data dissemination protocol in sensor networks [104]. In this example, a base station initiates a computation in which data are to be sent to all sensors in the network. The data message is split into fixed size packets. Each packet is given a sequence number. The base station starts transmitting the packets to its neighbor(s) in specified time slots, in the order of the packet sequence number. Subsequently, when the neighbor(s) receive a message, they, in turn, retransmit it to their neighbors and so on. The computation ends when all sensors in the network receive all the messages.

This protocol does not require explicit acknowledgments to be sent back from the receiver to the sender. For example, when a sensor sends a message to one of its neighbors it waits before sending the next message until it knows that the receiver did receive the message. In other words, it gets its acknowledgment by listening to the messages the neighboring sensors are currently transmitting. It only advances to next message if it knows that all its neighbors have attempted to transmit the last message it had sent.

To concisely describe the transitions of the program we use Dijkstra's guarded command [53] notation:

$$\langle guard \rangle \longrightarrow \langle statement \rangle;$$

where guard is a Boolean expression over program variables and the statement describes how program variables are updated and the statement always terminates. A guarded command of the form $g \longrightarrow st$ corresponds to transitions of the form $\{(s_0, s_1) |\, g$ evaluates to true in s_0 and s_1 is obtained by executing st from $s_0\}$.

The Program.

In this example, we arrange the processes in a linear topology. The base station has N packets to send to M processes. The fault-intolerant program transmits the packets in a simple pipeline. For this, each process keeps track of the messages (received/sent) using two variables $r.j$ and $s.j$, where $r.j$ is the highest message sequence number received by process j and $s.j$ is the sequence number of the message currently being transmitted by process j. Process j increments $r.j$ every time it receives a new message. It also sets $s.j$ to be the sequence number of the message it is transmitting. The base station transmits a packet if its neighbor has received the previous packet (action $IN1$). A process j, $j > 0$, receives a packet from its predecessor if its successor had received the previous packet (actions $IN2$ and $IN3$). Thus, the actions of fault-intolerant program are as follows:

Action for base station:

$(IN1)\ \ (s.0 = r.1) \longrightarrow\ \ s.0\ :=\ s.0 + 1;$

Action for process $j \in \{1..M\text{-}1\}$:

$(IN2)\ \ (\ (r.j \leq s.(j+1)) \wedge (r.j \leq s.(j-1)) \wedge (s.(j-1) = r.j+1)\ )$

$\qquad\qquad \longrightarrow\ \ r.j,\ s.j\ :=\ r.j+1,\ s.j+1;$

Action for process M:

$(IN3)$ $(\ (r.M \leq r.(M-1)\ ) \wedge (\ s.(M-1) = s.M+1\ )\ )$

$$\longrightarrow\ s.M,\ s.M\ :=\ s.M+1,\ s.M+1;$$

Faults.

The faults we consider are such that when a fault occurs a message is lost. To model such faults for the base station, we add action $(F1)$, where the base station increments $s.0$ even though its successor has not received the previous packet. To model such action for other processes, we add action $(F2)$, where a process advances $s.j$ even though the successor has not yet received the previous packet.

$(F1)$ $\mathit{true} \longrightarrow\ s.0\ :=\ s.0+1;$

$(F2)$ $(\ (r.j \leq s.(j-1)\ ) \wedge (\ s.(j-1) = s.(j+1)\ )$

$$\longrightarrow\ r.j,\ s.j\ :=\ r.j+1,\ s.j+1;$$

The Set of Legitimate States.

The constraints that define the legitimate states in the case of the data dissemination program are as follows. The first constraint states that initially the base station has all the packets $(C1)$. A process cannot receive a packet if its predecessor has not received it $(C2)$, and cannot transmit a packet that it does not have $(C3)$. A process transmits a packet that is expected by its successor ($C4$ and $C5$).

$(C1)$ $(\ s.0\ =\ N\ )$

$(C2)$ $(\ \forall j : 0 < j \leq M : (\ r.j = s.(j-1))\ )$

$(C3)$ $(\ \forall j : 0 \leq j \leq M : (s.j \leq r.j))\ )$

$(C4)$ $(\ s.0 \leq s.1+1\ )$

$(C5)$ $(\ \forall j : 0 < j \leq (M-1) : (s.j \leq s.(j-1)+1) \wedge (s.j \leq s.(j+1)+1))\ )$

And the legitimate states predicate is:

$$I = C1 \wedge C2 \wedge C3 \wedge C4 \wedge C5$$

The data dissemination program has a set of constraints imposed by the model. More specifically, these constraints identify the set of bad transitions that violate the safety specification of the program. In particular, the model requires that the reception of a packet cannot be reversed ($MT1$), packets can only be received in sequence ($MT2$), a process can only receive one packet at a time, it can only receive a packet sent by its predecessor ($MT3$ and $MT4$), a process cannot transmit a packet unless it has received it ($MT5$), and a process should not transmit a packet unless it is potentially needed by its successor ($MT6$). Thus, the set of transitions disallowed by the model are as follows:

$MT1: \ (\exists j : 0 < j \leq M : r.j' < r.j)$

$MT2: \ (\exists j : 0 < j \leq M : r.j' < (r.j) + 1)$

$MT3: \ (\exists j : 0 < j \leq M : (r.j' = (r.j) + 1) \wedge (r.j' \neq s.(j-1)) \wedge (r.j' \neq s.(j+1))$

$MT4: \ (s.M' = (s.M) + 1 \wedge s.M' \neq s.(M-1))$

$MT5: \ (\exists j : 0 \leq j \leq M : (r.j' < s.j'))$

$MT6: \ (\exists j : 0 \leq j \leq M-1 : (s.j > s.(j+1)+1) \wedge (s.j' < s.(j+1)+1)$

Fault-Tolerant program.

Using the program $IN1$-$IN3$ for each process, the faults $F1$-$F2$, the constraints $S1$-$S5$, and prohibited transitions $MT1$-$MT6$, the output was a nonmasking fault-tolerant program with the following recovery actions added to it.

$(R1) \ (r.j > s.(j+1)) \ \wedge \ (s.j > s.(j+1)+1) \ \wedge \ (r.j + 1 = s.(j-1))$
$\qquad\qquad \longrightarrow r.j := \ s.(j-1), s.j := \ s.(j+1)+1;$

$(R2) \ (r.j > s.(j+1)+1) \ \wedge \ (s.j > s.(j+1)+1) \longrightarrow s.j \ := \ s.(j+1)+1;$

Chapter 3

Under-The-Hood Revision

In this chapter, we present our contributions on performing the automated model revision while minimizing the effort and the expertise needed to perform such revision. We show how the designer can continue to utilize existing design tools while the revision is done under-the-hood. This makes automated revision more useable, as well as makes it available across different design tools. Specifically, we focus on integrating the automated revision with the SCR toolset. Part of the reasons behind our choice of SCR toolset is that the SCR descriptions are precise, unambiguous, and consistent. Also, many industrial farms use the SCR toolset to develop mission critical systems.

This chapter is organized as follows. In Section 3.1, we briefly describe the SCR formal method and we provide highlights of the SCR toolset. In Section 3.2, we present our approach for transforming the SCR specification into input for SYCRAFT. Then, in Section 3.3, we illustrate our approach using two case studies: an Altitude Switch Controller and an Automobile Cruise Controller. Finally, we summarize the chapter in Section 3.4.

3.1 Introduction to SCR

The Software Cost Reduction (SCR) formal method [22,83,84] is a set of techniques for constructing and evaluating requirements documents for high assurance systems. SCR uses tables to describe system behaviors and properties, as these tables provide a precise description of the model and capture the mathematical representation of systems. But these tables consume a considerable amount of time and resources to verify. Therefore, techniques and tools have been developed to provide a comprehensive framework that automates the validation and verification of the SCR tables. Hence, the SCR toolset [22,83–87] was created to serve this purpose. In this section, we describe the SCR formal method and show how the SCR toolset is used in the design and verification of event-driven systems.

3.1.1 SCR Formal Method

SCR is a set of formal methods for constructing and verifying requirements specification documents. The U.S. Naval Research Laboratory (NRL) developed SCR in the late 70s. Since then, it has been used in constructing many mission critical systems. SCR was used to design and model the A-7 aircraft and to document requirements. It was also used in the design of requirement specification of the Operational Flight Program (OFP) for the A-6 aircraft [114], the Bell telephone [91], submarines communication systems, nuclear plants [88], and many other systems.

The SCR formal method specifies system requirements using tabular notation. Tables provide a precise and compact way to describe requirements, making it possible for the user to automatically model and analyze those requirements to identify errors. SCR uses tables to describe both the system and its environment [85,86]. The environmental quantities whose values changes the system behavior are described using *Monitored variables*. The environmental quantities whose values are changed by the system are represented by *Controlled variables*.

To relate the variables of the system and represent constraints on those variables, the state machine model of the SCR is based on the "Four-Variable Model" that was, initially, introduced by Paranas [120]. This model describes the desired functionality of an embedded system in terms of four relations as follows.

- NAT: is the set of relations that describe the way in which the values of the variables (monitored or controlled) are restricted by the laws of the environment, whether these laws are imposed by previously deployed systems or by the physical laws.

- REQ: is the set of relations that defines the way in which the system changes the values of the *controlled variables* based on the change in the values of the *monitored variables*.

- IN: is the set of relations that maps the values of the monitored quantities to the values of the input variables.

- OUT: is the relation that maps the value of the output variable to a controlled quantity.

The IN and OUT relations describe the behavior of the input and output devices in some level of isolation. Thus, the IN and OUT relations give requirements specification the freedom of specifying the observed system behavior without going into further details.

Four more variables are also used in the constructs of the SCR. These are *modes*, *terms*, *conditions*, and *events*. The *mode* class is a state machine whose states are called modes. Changes from one mode to another are triggered by events. The *terms* are representations of a group of input variables, mode classes, or other terms in one single term. The *condition* is a predicate defined on single system state. Finally, the *event* is a predicate defined on two system states and is triggered by a change in a system entity. The following state machine formally represents a typical SCR system:

$$\Sigma = (S, S_0, E^m, T)$$

where S is the state space, $S_0 \subseteq S$ is the initial state set, E^m represent, a change in the value of the monitored events, and T is the function that identifies the transitions of the system based on monitored events (i.e. T maps $e \in E^m$ and the current state $s \in S$ to the next state $s' \in S$) [83].

In SCR, the systems are represented in the ideal state and with no time representation. The model defines the system as a before state, in terms of the system entities with guards as conditions, and an after state. The system transits from the before state to the after state by transitions triggered by a change in an input variable. These transitions are part of a transformation, T, which is defined by a set of functions that are represented by the SCR tables.

The SCR toolset [22,83–87] is a set of tools for constructing and validating requirements specifications based on the SCR formal method. It is composed of a specification editor, a user interface for creating and editing the specification in a tabular way; a dependency graph browser, which uses the directed graph representation to show the dependency of variables; and a simulator, which uses a symbolic variable representation to test if the desired system behavior is satisfied. The SCR toolset also includes different kinds of checkers: consistency checker, model checker, and property checker. This set of tools helps systems designers to check and analyze the specifications and to automatically detect errors and missed cases.

To illustrate these concepts, consider the altitude switch controller system (ASW) [21], which is responsible for turning on a device of interest when an aircraft altitude is below 2,000 feet. ASW will be disabled if it receives an *Inhibit* signal. A *Reset* signal will reset the system to its initial state. ASW has three altitude meters: two are digital and one is analog. It also has a fault indicator that is switched *On* if the DOI does not turn on in two seconds, if the system fails to initialize, or if all three altitude meters do not work for more than two seconds.

The SCR specifications for the ASW system are constructed with five monitored variables as shown in Table 3.1, one controlled variable, and a mode class. The *mAltBelow*, Boolean variable, value is *true* when the aircraft descends below 2,000 feet. The *mDOIstatus* is true when the DOI is on. The *mInitializing* indicates if the system is being initialized. The *mInhibit*, indicates whether the system can turn on the DOI or not. The *mReset* monitors the reset request. The controlled variable *cWakeupDOI* will be initialized to *false*. It will be set to *true* to wake-up the DOI.

Name	**Type**	**Init. Value**	**Description**
mAltBelow	Boolean	*true*	*true* if alt. below threshold
mDOIStatus	enum	*off*	*on* if DOI powered on; else *off*
mInitializing	Boolean	*true*	*true* iff system initializing
mInhibit	Boolean	*false*	*true* iff DOI power on inhibited
mReset	Boolean	*false*	*true* iff Reset button is pushed

Table 3.1: Monitored Variables of the altitude switch controller system (ASW).

Table 3.2 describes the mode class *mcStatus*. Each transition in the mode table describes the system transition from one mode to another as a result of change in one or more monitored variables. There are three modes for the mode class *mcStatus*: *Init*, *standby*, and *awaitDOIon*. For example, the first row of Table 3.2 states that ASW transitions from *init* mode to *standby* if it is not initializing.

Table 3.3 contains the description of the condition table for the controlled variable *cWakeupDOI*. The value of the controlled variable *cWakeupDOI* depends mainly on the current value of the mod class *mcStatus*. If the value of *mcStatus* is *awaitDOIon*, then the DOI can be powered *on*. If the value of *mcStatus* is *Init* or *Standby*, the DOI will be turned

Old Mode	**Event**	**New Mode**
init	@F(mInitializing)	Standby
standby	@T(mReset)	init
standby	@T(mAltBelow)WHEN NOT mInhibit AND mDOIStatus = off	awaitDOIon
awaitDOIon	@T(mDOIStatus = on)	standby
awaitDOIon	@T(mReset)	init

Table 3.2: Mode transition table for the mode class *mcStatus*.

off.

Mode	**cWakeupDOI**
Init, Standby	false
awaitDOIon	true

Table 3.3: Condition table for *cWakeUpDOI*.

There are two major advantages of the SCR toolset. First, all the tools interface with each other automatically. Hence, they behave as a single application [83]. Second, the toolset has been adopted by the industry and was used in the development of many real world applications [83]. Moreover, the toolset stores the specifications in an ASCII text file from which other systems can have access to those specifications. More specifically, we use this file as an interfaces channel to communicate with the tool SYCRAFT.

3.1.2 Automated Model Revision to Add Fault-Tolerance

Programs are subject to faults that may not be preventable. A program may function correctly in the absence of faults. However, it may not give the desired functionality in the presence of faults. The automated model revision to add fault-tolerance is the process of transforming a fault-intolerant program to a fault-tolerant one. This transformation guarantees that the program continues to satisfy its specification in the presence of faults. SYCRAFT, described briefly next, is a framework for automating such revisions [27,30]. In SYCRAFT, programs (input and output) and faults are represented using guarded commands. SYCRAFT takes both the program and the faults as an input and generates the fault-tolerant program version as an output. To add fault-tolerance, SYCRAFT first identifies states from where faults alone can violate safety specification. It removes such states and the transitions that reach them. Then, it adds recovery transitions to ensure that after the occurrence of faults, the program recovers to its legitimate states.

3.2 Integration of SCR toolset and SYCRAFT

In this section, we first describe how we translate the SCR program into an input for SYCRAFT. Then, we describe modeling of faults and subsequently give an outline of our tool for adding the automated model revision to the SCR toolset. Our approach, allows one to perform separation of concerns where the fault-tolerance aspect is relegated only to the tool that performs the automated addition of fault-tolerance.

3.2.1 Transforming SCR specifications into SYCRAFT input

The integration of SCR and SYCRAFT mainly focuses on the *mode* table since the mode table captures the system behavior in response to different inputs. Hence, the mode table is the most relevant in terms of the effect of the faults on system behavior. The integration focuses on translating the mode table so that it can be used as an input in SYCRAFT and then translating the SYCRAFT output so as to generate the mode table of the fault-tolerant SCR specification.

We illustrate the mode table in SCR using the simple example *mRoom* (cf. Table 3.4). As the name suggests, this table describes different modes of *mRoom* and shows how they change in response to the system events. *mRoom* has two modes: *Dark* and *Light* and one monitored variable *mSwitchOn*. This system switches the room from *Dark* mode to *Light* mode if the event *@T(mSwitchOn)* occurs, i.e. if the monitored variable *mSwitchOn* changes its value from *false* to *true*.

Old Mode	**Event**	**New Mode**
Dark	@T(mSwitchOn)	Light
Light	@F(mSwitchOn)	Dark

Table 3.4: *mRoom* Mode Table

To add fault-tolerance to the SCR specification, we need to convert the SCR tables into guarded commands. In particular, we need to translate modes, conditions, terms, and events. Next, we describe how we translate the SCR events into guarded commands for SYCRAFT. Events in SCR occur at the time when the value of their condition is switched from *false* to *true* or vice versa in a single transition. It is not only the current state of the monitored variable that initiates the transition; rather, it is the combination of both the current and the old states. The notation used to represent events is as follows:

$$(@T(c)\ \text{WHEN}\ d\,) \equiv (\,\neg\, c \wedge c' \wedge d\,)$$

where (c) represents the condition value in the before state and (c') represent the condition value in the after state [83]. For example, if we consider the SCR mode table entry in *mRoom* mode class:

From "Dark" EVENT "@T (mSwtichOn)" TO "Light"

In the "before" state, the mode value *mRoom* is *Dark* and the condition *mSwitchOn* is *false*. And, in the "after" state the mode value *mRoom* = *Light* and the condition *mSwitchOn* = *true*.

In SYCRAFT, (guarded commands) transitions are represented in the following format:

$$(\,g \;\longrightarrow\; st\,)$$

The guard, g, is a predicate whose value must be true in the before state in order for the statement, *st*, to execute. The guarded command translation for *mRoom* table entry would be:

$$\begin{aligned}&(\ (\,mRoom = Dark\,) \wedge (\,mSwtichOn = false\,)\ )\\ &\qquad\qquad \longrightarrow\ mRoom := Light;\ mSwtich := true\end{aligned}$$

Likewise, we need to convert states, terms, and modes into the corresponding input for SYCRAFT. In particular, each mode is translated into corresponding states that a program could reach. Conditions are translated into guards that determine when actions can be executed.

3.2.2 Translation from SCR Syntax to SYCRAFT Syntax

In this translation we preserve the model abstraction as well as compactness to avoid the state explosion problem. The goal of this translation is to translate the SCR table syntax into action language that the SYCRAFT can deal with. The translation rules are based on the fact that the transition relation in the SCR tables is identified using a condition on the current state and another condition on the next state. For example, the current state in SCR is defined using the "FROM mode" with a condition, and the next state is identified by the "TO mode". In the SYCRAFT syntax we translate the "FROM mode" into "mcMode== mode" and the "TO mode" into "→ mcMode:=mode". Table 3.5 shows some of the translation rules.

SCR Syntax		SYCRAFT Syntax
MODETRANS "mcMode";	⇒	process "mcMode" ;
FROM	⇒	(
"Source Mode"	⇒	(mcMode="Source Mode") &&
EVENT	⇒	)(
@F (cond1)	⇒	! Cond1
@T (cond1)	⇒	cond1
WHEN	⇒	&&
TO	⇒	) →
"Target Mode"	⇒	mcMode :="Target Mode" ;

Table 3.5: Translation rules

3.2.3 Modeling of faults

Faults in SYCRAFT are also modeled using guarded commands that change program variables. To effectively model faults for designers, we can model them using tables similar to the way the SCR specification is specified. Note that this would require changes to the SCR toolset. However, the change is minimal in that it would require adding an extra table for faults rather than putting all program/fault actions together as was done in [22]. Note that with this change, we do not expect the designer task to be more complex since faults are specified using a method similar to describing programs. For simplicity, currently, we let faults be directly represented using guarded commands so that modification to the SCR toolset is not necessary. Likewise, it would be necessary for the designer to specify requirements in the presence of faults. These specifications are also similar to that used in SCR for requirements in the absence of faults.

3.2.4 Adding fault-tolerance to SCR specifications

The scenario of adding fault-tolerance to the SCR specifications is described in Figure 3.1. The cycle begins at step 1 by creating the specifications requirement using the SCR toolset. The specifications in SCR formats are exported from the SCR toolset as in step 2. In step 3, the middle-layer imports the SCR specifications and the first translation phase generates an output file for the use in the addition of fault-tolerance by SYCRAFT. This file is imported in step 4 to SYCRAFT, which generates a fault-tolerant version of the program in step 5. In step 6, the middle-layer imports the SYCRAFT output and in step 7 translates it back to the SCR specification. Finally, in step 8, the file is imported back into the SCR toolset so that it can be visualized using the SCR toolset. Thus, the translation layer shown in Figure 3.1 allows the automated revision to add fault-tolerance where the addition is done *under-the-hood*, meaning that, it allows users of the SCR tools to add fault-tolerance to specifications without knowing the details of SYCRAFT or the theory on which SYCRAFT is based.

3.3 Case Studies

To illustrate the integration of SCR and SYCRAFT, we present two case studies: the control system for an aircraft altitude switch (ASW) [22] and the automobile cruise control system (CCS) [95]. For both systems, we briefly describe the concept and demonstrate how our 8-steps method from Section 3.2.4 works on these examples to translate the fault-intolerant

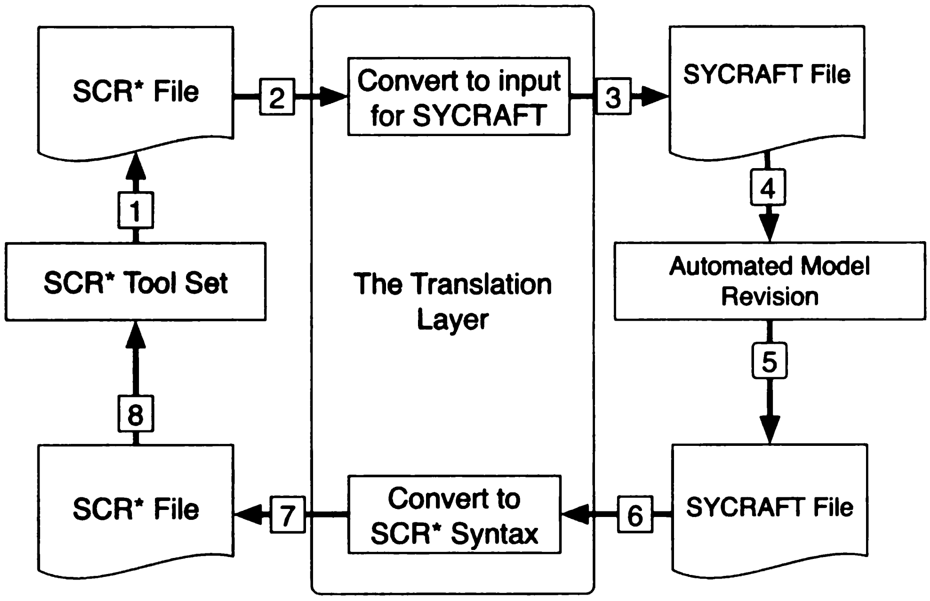

Figure 3.1: The transformation cycle between SCR toolset and SYCRAFT.

SCR specification into the corresponding fault-tolerant specification.

3.3.1 Case Study 1: Altitude Switch Controller

In Section 3.1.1, we described the ASW system and illustrated how it is modeled using the SCR formal method. In this Section, we show how to transform the SCR specification of the ASW into guarded command. Then, we use SYCRAFT to revise the specification of the ASW to add fault-tolerance. Later, we show how to transform the ASW specification from guarded command into SCR to import it back into the SCR toolset.

Step 1. As shown in Figure 3.1 at step 1, we extract the mode table of the ASW system in the SCR specification. The *mcStatus* mode table of the ASW system is illustrated in Table 3.2. It describes the mode class *mcStatus* that represents a function between the monitored variables and the current value of the *mcStatus*. The *mcStatus* class has one of the following three modes: *standby*, *init*, or *awaitDOIon*.

Steps 2 & 3. At step 2, we import the SCR specification into the middle layer. This layer generates the input in guarded command format at step 3. The result of the translation layer

$((mcStatus = init) \wedge ((mInitializing) = true))$ $\longrightarrow mcStatus := standby; (mInitializing) := false;$
$((mcStatus = standby) \wedge ((mReset) = false))$ $\longrightarrow mcStatus := init; (mReset) := true;$
$((mcStatus = standby) \wedge ((mAltBelow) = false \wedge !mInhibit \wedge mDOIStatus = off))$ $\longrightarrow mcStatus = awaitDOIon; (mAltBelow) = true \wedge !mInhibit \wedge mDOIStatus = off;$
$((mcStatus = awaitDOIon) \wedge ((mDOIStatus = on) = false))$ $\longrightarrow mcStatus := standby; mDOIStatus := true;$
$((mcStatus = awaitDOIon) \wedge ((mReset) = false))$ $\longrightarrow mcStatus := init; mReset := true;$

Table 3.6: The mcStatus mode table translated.

is as shown in Table 3.6. For example, the first entry in Table 3.6 shows that in order for this action to execute, the old value (i.e. the "before" state) of the *mcStatus* should be equal to *standby*, and *mReset* should be equal to *false*. The two statements in the right hand side represent the "after" state; both values of *mcStatus* and *mReset* should be changed.

We consider three hardware malfunctions that may alter the operation of the fault intolerant ASW controller [22]. They are an altimeter fault, an initialization fault, and DOI fault. All three faults are time-out faults, i.e., they require the system to stay in a given state for a specified amount of time. But since SYCRAFT does not include the notion of time yet, we abstract those faults to be *on/off* flags. We added a new mode, called *fault*, to the *mcStatus* class to indicate the presence of faults in the system. Table 3.7 shows how those faults are represented in the input file to SYCRAFT. Note that the fault transitions described below can be easily described using SCR tables. Therefore, the designer can specify the faults using the SCR toolset interface which they are familiar with.

Step 4. In step 4, we use the translated SCR specification and the three faults described in Table 3.7 as an input to SYCRAFT so that SYCRAFT can add fault-tolerance to ASW

$(mcStatus = init) \wedge (Init\_Duration\_Fault = true)$ $\longrightarrow Init\_Duration\_Fault := false\, ;\, mcStatus := Fault;$
$(standby = init) \wedge (Alt\_Duration\_Fault = true)$ $\longrightarrow Alt\_Duration\_Fault := false\, ;\, mcStatus := Fault;$
$(awaitDOIon = init) \wedge (AwaitDOI\_Duartion\_Fault = true)$ $\longrightarrow AwaitDOI\_Duartion\_Fault := false\, ;\, mcStatus := Fault;$

Table 3.7: The SYCRAFT fault section.

$(mcStatus = init) \wedge ((mInitializing) = true))$ $\longrightarrow mcStatus := standby\, ;\, mInitializing := false;$
$((mcStatus = standby) \wedge ((mReset) = false))$ $\longrightarrow mcStatus := init\, ;\, mReset := true;$
$((mcStatus = standby) \wedge ((mAltBelow) = false \wedge !mInhibit \wedge$ $mDOIStatus = off \wedge mAltFail = false))$ $\longrightarrow mcStatus = awaitDOIon\, ;\, (mAltBelow) = true;$
$((mcStatus = awaitDOIon) \wedge ((mDOIStatus = false))$ $\longrightarrow mcStatus := standby\, ;\, mDOIStatus := true;$
$((mcStatus = awaitDOIon) \wedge ((mReset) = false))$ $\longrightarrow mcStatus := init\, ;\, mReset := true;$
$((mcStatus = fault) \wedge ((mReset) = false))$ $\longrightarrow mcStatus := standby\, ;\, mReset := true;$

Table 3.8: The fault-tolerant mcStatus mode table.

specification that tolerates the failure of the altimeter, initialization, or DOI.

Step 5. The result of step 5 is shown in Table 3.8. The parts where SYCRAFT added the tolerance were at two places. First, the condition (*mAltFail* = *false*) was added to the guard of the third transition to prevent the *mcStatus* from activating the device when *mAltFail* is *true*. Second, the last transition in the Table 3.8 was added to provide recovery from the fault state to one of the system legitimate states.

Steps 6 & 7. We import the SYCRAFT specifications into the translation layer at step 6 to translate it to fault-tolerant SCR specifications. Table 3.9 is the result after applying the translation on the mcStatus from SYCRAFT output to SCR.

Old Mode	Event	New Mode
init	@F(mInitializing)	Standby
standby	@T(mReset)	init
standby	mDOIStatus = off AND NOT mAltFail mDOIStatus = off AND NOT mAltFail	awaitDOIon
awaitDOIon	@T(mAltBelow)WHEN NOT mInhibit AND mDOIStatus = off AND NOT mAltFail	standby
awaitDOIon	@T(mDOIStatus = on)	init
fault	@T(mReset)	init
init	@T(Init_Duration_Fault)	fault
standby	@T(Alt_Duration_Fault)	fault
awaitDOIon	@T(AwaitDOI_Duartion_Fault)	fault

Table 3.9: Fault-tolerant mode class mcStatus.

3.3.2 Case Study 2: Cruise Control System

The cruise control system (CCS) [95] manages the cruising speed of an automobile by controlling the throttle position. It depends on several monitored variables, namely, *mIgnon*, *mEngRunning*, *mSpeed*, *mLever*, and *mBrake*. The system uses monitored variables to control the automobile speed. The cruise mode is engaged by setting the *mLever* to "const", provided that other conditions like "engine running" and "ignition is on" are met. The CCS can maintain constant, decrease, or increase automobile speed depending on the current speed. Below, we show how fault-tolerant CCS is revised using the tool described in Figure 3.1.

The *mCruise* mode table is shown in Table 3.10. This table specifies the values that the *mCruise* class can take. We imported the modeTable 3.10 into the middle layer, which generated specification in SYCRAFT format. Then we translated the *mCruise* mode table to SYCRAFT.

We consider a system malfunction that may alter the operation of the fault intolerant CCS. The fault takes place when the status of the cruise becomes *unknown*. Table 3.11

Old Mode	Event	New Mode
Off	@T(mIgnOn)	Inactive
Inactive	@F(mIgnOn)	Off
Inactive	@T(mLever=const) WHEN mIgnOn AND mEngRunning AND NOT Brake	Cruise
Cruise	@F(mIgnOn)	Off
Cruise	@F(mEngRunning)	Inactive
Cruise	@T(mBrake) OR @T(mLever = off)	Override
Override	@F(mIgnOn)	Off
Override	@F(mEngRunning)	Inactive
Override	@T(mLever = resume) WHEN mIgnOn AND mEngRunning AND NOT mBrake OR @T(mLever = const) WHEN mIgnOn AND mEngRunning AND NOT mBrake	Cruise

Table 3.10: Fault intolerant mode class mcCruise.

$$(mcCruise = Override) \vee (mcCruise = Cruise) \vee (mcCruise = Inactive) \vee (mcCruise = Off) \wedge (CruiseFault = true)$$
$$\longrightarrow mcCruise := Unkown; CruiseFault := false;$$

Table 3.11: The SYCRAFT fault section.

shows how this fault is represented in the input file to SYCRAFT.

We have inputted the faults and the fault-intolerant CCS to SYCRAFT in order to add fault-tolerance to the CCS system to tolerate a recovery from an unknown state to one of the CCS safe state. SYCRAFT added two actions to recover from the unknown state to one of the system valid states depending on the value of the *IgnOn* monitored variable. The fault-tolerant specification is as shown in Table 3.12.

Old Mode	Event	New Mode
Off	@T(mIgnOn)	Inactive
Inactive	@F(mIgnOn)	Off
Inactive	@T(mLever=const) WHEN mIgnOn AND mEngRunning AND NOT mBrake	Cruise
Cruise	@F(mIgnOn)	Off
Cruise	@F(mEngRunning)	Inactive
Cruise	@T(mBrake) OR @T(mLever = off)	Override
Override	@F(mIgnOn)	Off
Override	@F(mEngRunning)	Inactive
Override	@T(mLever = resume) WHEN mIgnOn AND mEngRunning AND NOT mBrake OR @T(mLever = const) WHEN mIgnOn AND mEngRunning AND NOT mBrake	Cruise
Unknown	@T (IgnOn)	Off
Unknown	@F (IgnOn)	Inactive
Override, Cruise, Off, Inactive	@T(CruiseFault)	Unknown

Table 3.12: Fault-tolerant mode class mcCruise.

3.4 Summary

In this chapter we presented the techniques we developed to make the automated model revision more easier to use. Our goal is to make the model revision accessible to wide range of system designers. Specifically, we utilized existing design tools (e.g., SCR toolset) to be the front end of our approach and performed all the aspects related to the automated model revision behind the scene. To successfully achieve this coupling, we developed a middle layer that translated the SCR specifications into SYCRAFT specifications and from SYCRAFT back to SCR. With this middle layer, we enabled the designers to perform the tasks of the automated model revision under-the-hood.

Chapter 4

Expediting the Automated Revision Using Parallelization and Symmetry

To make the automated model revision more applicable in practice, we need to develop approaches for enhancing their performance. Specifically, we need to able to revise programs with moderate to large state space in a reasonable amount of time. Our goal in this chapter is to utilize both the properties of the programs being revised and the available infrastructure (e.g., multi-core architecture) to expedite the revision. Hence, we focus on using symmetry, inside the program being revised, and parallelism, obtained from multiple cores, to speedup the revision algorithm.

The rest of this chapter is organized as follows. We explain the bottlenecks of the automated model revision and illustrate the issues involved in the revision problem in the context of Byzantine agreement in Section 4.2. We analyze the effect of the distributed nature of the program being revised on the complexity of the revision in Section 4.2.2. We present our algorithms in Section 4.3. We analyze the results in Subsection 4.3.3 and argue that our multi-core algorithm is likely to benefit further with additional cores. We evaluate a different parallelism approach in Section 4.4. In Section 4.5, we present our approach for expediting the revision of fault-tolerant programs with the use of symmetry. Finally, we summarize in Section 4.6.

4.1 Introduction

Given the current trend in processor design where the number of transistors keeps growing as directed by Moore's law but where clock speed remains relatively flat, it is expected that multi-core computing will be the key for utilizing such computers most effectively. As argued in [90], it is expected that programs/protocols from distributed computing will be especially beneficial in exploiting such multi-core computers.

One of the difficulties in adding fault-tolerance using automated techniques, however, is its time complexity. Our focus is to evaluate the effectiveness of different approaches that utilize multi-core computing to reduce the time complexity during deadlock resolution in the revision to add fault-tolerance to distributed programs.

To evaluate the effectiveness of multi-core computing, we first need to identify bottleneck(s) where multi-core features can provide the maximum impact. To identify these bottlenecks, in [30], Bonakdarpour and Kulkarni developed a symbolic (BDD-based) algorithm for adding fault-tolerance to distributed programs with state space larger than 10^{30}. Based on the analysis of the experimental results from [30], they observed that depending upon the structure of the given distributed intolerant program, performance of the revision suffers from two major complexity obstacles: (1) generation of fault-span, the set of states reachable in the presence of faults, and (2) resolving deadlock states, from where the program has no outgoing transitions. To resolve a deadlock state, we either need to provide *recovery actions* that allow the program to continue its execution or *eliminate* the deadlock state by preventing the program execution from reaching it. Of these, generation of fault-span closely similar to program verification and, hence, techniques for efficient verification are directly applicable to it. In this chapter, we focus on expediting the resolution of deadlock states with the use of parallelism and symmetry.

In the context of dependable systems, the revised fault-tolerant program should meet its liveness requirements even in the presence of faults. Therefore, no deadlock states are permitted in the fault-tolerant program since the existence of such states can violate the liveness requirements. A program may reach a deadlock state due to the fact that faults perturb the program to a new state that was not considered in the fault-intolerant program. Or, it may reach a deadlock state due to the fact that some program actions are removed (e.g., because they violate safety in the presence of faults).

We present two approaches for parallelization. The first approach is to parallelizes the group computation. It is based on the distributed nature of the program being revised. In particular, when a new transition is added/removed, since the process executing it has only a partial view of the program variables, we need to add/remove a *group* of transitions based on the variables that cannot be read by the process. The second approach is based on partitioning deadlock states among multiple threads; each thread resolves the deadlock states that have been assigned to it. We show that this provides a small performance benefit. Based on the analysis of these results, we argue that the simple approach that parallelizes the group computation is likely to provide maximum benefit in the context of deadlock resolution for the revision of distributed programs.

To understand the use of symmetry, we observe that, often, multiple processes in a distributed program are symmetric in nature, i.e., their actions are similar (except for the renaming of variables). Thus, if we find recovery transitions for a process, then we can utilize symmetry to identify other recovery transitions that should also be included for other processes in the system. Likewise, if some transitions of a process violate safety in the presence of faults, then we can identify similar transitions of other processes that would also violate safety. If the cost of identifying these similar transitions with the knowledge of symmetry among processes is less than the cost of identifying these transitions explicitly, then the use of symmetry will reduce the overall time required for revision.

We also present an algorithm that utilizes symmetry to expedite the revision. We show that our algorithm significantly improves performance over previous implementations. For example, in the case of *Byzantine agreement* (*BA*) [107] with 25 processes, time for revision with a sequential algorithm was $1,632s$. With symmetry alone, revision time was reduced to $188s$ (8.7 times better). With parallelism (8 threads), revision time was reduced to $467s$ (3.5 times better). When we combined both symmetry and parallelism together, the total revision time was reduced to $107s$ (more than 15.2 times better).

4.2 Issues in Automated Model Revision

In this section, we use the example of *Byzantine agreement* [107] (denoted *BA*) to describe the issues in automated revision to add fault-tolerance. Towards this end, in Section 4.2.1, we describe the inputs used for revising the Byzantine agreement problem. Subsequently, in Section 4.2.2, we identify the need for explicit modeling of read-write restrictions imposed by the nature of the distributed program. Finally, in Section 4.2.3, we describe how deadlock states get created while revising the program for adding fault-tolerance and illustrate our approach for managing them.

4.2.1 Input for Byzantine Agreement Problem

The Byzantine agreement problem (*BA*) consists of a *general*, say g, and three (or more) *non-general* processes, say j, k, and l. The agreement problem requires that a process copy the decision chosen by the general (0 or 1) and finalize (output) the decision (subject to some constraints). Thus, each process of *BA* maintains a decision d; for the general, the decision can be either 0 or 1, and for the non-general processes, the decision can be 0, 1, or $\perp$, where the value $\perp$ denotes that the corresponding process has not yet received the decision from the general. Each non-general process also maintains a Boolean variable f that denotes whether that process has finalized its decision. For each process, a Boolean variable b shows whether or not the process is Byzantine; the read/write restrictions (described in Section 4.2.2) ensure that a process cannot determine if other processes are Byzantine. Thus, a state of the program is obtained by assigning each variable, listed below, a value from its domain. And, the state space of the program is the set of all possible states.

$V = \{d.g\} \cup$	(the general decision variables): $\{0, 1\}$
$\{d.j, d.k, d.l\} \cup$	(the processes decision variables): $\{0, 1, \bot\}$
$\{f.j, f.k, f.l\} \cup$	(finalized?): $\{false, true\}$
$\{b.g, b.j, b.k, b.l\}$.	(Byzantine?): $\{false, true\}$

Fault-intolerant program. To concisely describe the transitions of the (fault-intolerant) version of *BA*, we use guarded commands of the form $g \longrightarrow st$. Recall from Chapter 1 that g is a predicate involving the above program variables and st updates the above program variables. The command $g \longrightarrow st$ corresponds to the set of transitions $\{(s_0, s_1) : g$ is true in s_0 and s_1 is obtained by *executing st* in state $s_0\}$. Thus, the transitions of a non-general process, say j, is specified by the following two actions:

$$\begin{aligned} BA_{intol_j} :: BA1_j &:: (d.j = \bot) \wedge (f.j = false) \wedge (b.j = false) \longrightarrow d.j := d.g \\ BA2_j &:: (d.j \neq \bot) \wedge (f.j = false) \wedge (b.j = false) \longrightarrow f.j := true \end{aligned}$$

We include similar transitions for k and l as well. Note that the general does not need explicit actions; the action by which the general sends the decision to j is modeled by $BA1_j$.

Specification. The safety specification of the *BA* requires *validity* and *agreement*. *Validity* requires that if the general is non-Byzantine, then the final decision of a non-Byzantine, non-general must be the same as that of the general. Additionally, *agreement* requires that the final decision of any two non-Byzantine, non-generals must be equal. Finally, once a non-Byzantine process finalizes (outputs) its decision, it cannot change it.

Faults. A fault transition can cause a process to become Byzantine, if no other process is initially Byzantine. Also, a fault can change the d and f values of a Byzantine process. The fault transitions that affect a process, say j, of *BA* are as follows: (We include similar actions for k, l, and g)

$$F1 :: \neg b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l \longrightarrow b.j := true$$

$$F2 :: b.j \longrightarrow d.j, f.j := 0|1, false|true$$

where $d.j := 0|1$ means that $d.j$ could be assigned either 0 or 1. In case of the general process, the second action does not change the value of any f-variable.

Goal of automated Addition of fault-tolerance. The goal of the automated revision is to start from the intolerant program (BA_{intol_j}) and given the set of faults ($F1\&F2$) and to automatically generate the fault-tolerant program ($BA_{tolerant_j}$), given below.

$$\begin{aligned}
BA_{tolerant_j} :: BA1_j &:: (d.j = \perp) \wedge (f.j = false) \wedge (b.j = false) \longrightarrow d.j := d.g \\
BA2_j &:: (d.j \neq \perp) \wedge (f.j = false) \wedge (d.l \neq \perp \vee d.k \neq \perp) \longrightarrow f.j := true \\
BA3_j &:: (d.l = 0) \wedge (d.k = 0) \wedge (d.j = 1) \wedge (f.j = 0) \longrightarrow d.j, f.j := 0, 0|1 \\
BA4_j &:: (d.l = 1) \wedge (d.k = 1) \wedge (d.j = 0) \wedge (f.j = 0) \longrightarrow d.j, f.j := 1, 0|1
\end{aligned}$$

In the above program, the first action remains the same. The second action is restricted to execute only in the states where another process has the same d value. Actions (3&4) are for fixing the process decision.

4.2.2 The Need for Modeling Read/Write Restrictions

Since the program being revised is distributed in nature, each process can only read a subset of the program variables. It is important to articulate these restrictions precisely to ensure that the revised program is realizable under the constraints of the underlying distributed system for which it is designed. For example, in the context of the Byzantine agreement example from Section 4.2.1, non-general process j is not supposed to know whether other processes are Byzantine. It follows that process j cannot include an action of the form 'if $b.k$ is true then change $d.j$ to 0'. To permit such modeling, we need to specify read-write restrictions for a given process. For the Byzantine agreement example, process j is allowed to read $R_j = \{b.j, d.j, f.j, d.k, d.l, d.g\}$ and it is allowed to write $W_j = \{d.j, f.j\}$. Observe that this modeling prevents j from knowing whether other processes are Byzantine.

With such read/write restriction, if process j were to include an action of the form 'if $b.k$ is true then change $d.j$ to 0' then it must also include a transition of the form 'if $b.k$ is false then change $d.j$ to 0'. In general, if transition (s_0, s_1) is to be included as a transition of process j then we must also include a corresponding equivalence class of transitions (called group of transitions) that differ only in terms of variables that j cannot read. For further discussion of the group operation please refer to Section 2.2.

4.2.3 The Need for Deadlock Resolution

During revision, we analyze the effect of faults on the given fault-intolerant program and identify a fault-tolerant program that meets the constraints of Problem 2.1. This involves addition of new transitions as well as removal of existing transitions. In this section, we utilize the Byzantine agreement problem to illustrate how deadlocks get created during the execution of the revision algorithm and identify two approaches for resolving them.

- **Deadlock scenario 1 and use of recovery actions.** One legitimate state, say s_0 (c.f. Table 4.1), for the Byzantine agreement program is a state where all processes are non-Byzantine, $d.g$ is 0 and the decision of all non-generals is $\perp$. Thus, in this state, the general has chosen the value 0 and no non-general has received any value. From this state, process j (respectively k) can copy the general decision by executing the program action $BA1_j$ (respectively $BA1_k$) as in s_1 (respectively s_2) from Table 4.1. The general can become Byzantine and change its value from 0 to 1 arbitrarily as in s_3. Therefore, a non-general can receive either 0 or 1 from the general.

  Clearly, starting from s_3, in the presence of faults ($F1$ & $F2$), the program (BA_{intol}) can reach a state, say s_5, where $d.g = d.l = 1$, and $d.j = d.k = 0$. From such a state, transitions of the fault-intolerant program violate safety if they allow j (or k) and l

to finalize their decision. If we remove these safety violating transitions then there are no other transitions from state s_5. In other words, during revision, we encounter that state s_5 is a deadlock state. One can resolve this deadlock state by simply adding a *recovery* transition that changes $d.l$ to 0. (Note that based on the discussion of Section 4.2.2, adding such recovery transition requires us to add the corresponding group of transitions. It is straightforward to observe that none of the transitions in this group violate safety.)

State	Action/ Fault	b.g	b.j	b.k	b.l	d.g	d.j	d.k	d.l	f.j	f.k	f.l
s_0	—	0	0	0	0	0	$\perp$	$\perp$	$\perp$	0	0	0
s_1	$BA1_j$	0	0	0	0	0	$\underline{0}$	$\perp$	$\perp$	0	0	0
s_2	$BA1_k$	0	0	0	0	0	0	$\underline{0}$	$\perp$	0	0	0
s_3	$F1$	$\underline{1}$	0	0	0	0	0	0	$\perp$	0	0	0
s_4	$F2$	1	0	0	0	$\underline{1}$	0	0	$\perp$	0	0	0
s_5	$BA1_l$	1	0	0	0	1	0	0	$\underline{1}$	0	0	0

Table 4.1: Deadlock scenario 1 (The underlined values indicates which variable is being changed by the program action/fault. For reasons of space the *true* and *false* values are replaced by 1 and 0 respectively for the variables b and f.)

- **Deadlock scenario 2 and need for elimination.** Again, consider the execution of the program (BA_{intol}) in the presence of faults ($F1$ & $F2$). Starting from state s_0 in the previous scenario the program can also reach a state, say s_6 (c.f. Table 4.2), where $d.g = d.l = 1, d.j = d.k = 0$, and $f.j = 1$; state s_6 differs from s_5 in the previous scenario in terms of the value of $f.l$. Unlike s_5 in the previous scenario, since l has finalized its decision, we cannot resolve s_6 by adding safe recovery. Since safe recovery from s_6 cannot be added, the only choice for designing a fault-tolerant program is to ensure that state s_6 is never reached in the fault-tolerant program. This can be achieved by removing transitions that reach s_6. However, removal of such transitions can create more deadlock states that have to be eliminated. Thus, the deadlock algorithm needs to be recursive in nature.

State	Action/ Fault	b.g	b.j	b.k	b.l	d.g	d.j	d.k	d.l	f.j	f.k	f.l
s_0	-	0	0	0	0	0	$\perp$	$\perp$	$\perp$	0	0	0
s_1	$BA1_j$	0	0	0	0	0	$\underline{0}$	$\perp$	$\perp$	0	0	0
s_2	$BA1_k$	0	0	0	0	0	0	$\underline{0}$	$\perp$	0	0	0
s_3	$BA2_j$	0	0	0	0	0	0	0	$\perp$	$\underline{1}$	0	0
s_4	$F1$	$\underline{1}$	0	0	0	0	0	0	$\perp$	1	0	0
s_5	$F2$	1	0	0	0	$\underline{1}$	0	0	$\perp$	1	0	0
s_6	$BA1_l$	1	0	0	0	1	0	0	$\underline{1}$	1	0	0

Table 4.2: Deadlock scenario 2 (The underlined values indicates which variable is being changed by the program action/fault. For reasons of space the *true* and *false* values are replaced by 1 and 0 respectively for the variables b and f.)

To maximize the success of the revision algorithm, our approach to handle deadlock states is as follows: Whenever possible, we add recovery transition(s) from the deadlock states to a legitimate state. However, if no recovery transition(s) can be added from the deadlock states, we try to eliminate (i.e. make it unreachable) the deadlock states by preventing the program from reaching the deadlock states. In other words, we try to eliminate deadlock states only if adding recovery from them fails.

4.3 Approach 1: Parallelizing Group Computation

In this section, we present our approach for parallelizing the group computation to expedite the revision to add fault-tolerance. First, in Section 4.3.1, we identify the different design choices we considered and then present our algorithm. In Section 4.3.2, we describe our approach for parallelizing the group computation. Subsequently, in Section 4.3.3, we provide experimental results in the context of the Byzantine agreement example from Section 4.2.1 and the token ring [14]. Finally, in Section 4.3.4, we analyze the experimental results to evaluate the effectiveness of parallelization for group computation.

4.3.1 Design Choices

The structure of the group computation permits an efficient way to parallelize it. In particular, whenever some recovery transitions are added for dealing with a deadlock state or some states are removed for ensuring that a deadlock state is not reached, we can utilize multiple threads in a master-slave fashion to expedite the group computation. During the analysis for utilizing the multiple cores effectively, we make the following observations/design choices.

- **Multiple BDD packages vs. reentrant BDD package.** We chose to utilize different instances of BDD packages for each thread. Thus, at the time of group computation, each thread obtains a copy of the BDD corresponding to the program transitions and other BDDs from the master thread. In part, this was motivated by the fact that existing parallel BDD implementations have shown limited speedup. Also, we argue that the increased space complexity of this approach is acceptable in the context of revision, since the time complexity of the revision algorithm is high (compared with model checking) and we always run out of time before we run out of space.

- **Synchronization overhead.** Although simple to parallelize, the group computation itself is fine grained, i.e., the time to compute a group of the recovery transitions that are to be added to the program is small (100-500ms). Hence, the overhead of using multiple threads needs to be small. With this motivation, our algorithm creates the required set of threads up front and utilizes mutexes to synchronize between them. This provided a significant benefit over creating and destroying threads for each group operation.

- **Load balancing.** Load balancing among several threads is desirable so that all threads take approximately the same amount of time in performing their task. To perform a group computation for the recovery transitions being added, we need to evaluate the effect of read/write restrictions imposed by each process. A static way to parallelize this is to let each thread compute the set of transitions caused by read/write restrictions of a (given) subset of processes. A dynamic way is to consider the set of processes for which a group computation is to be performed as a *shared pool of tasks* and allow each thread to pick one task after it finishes the previous one. We find that given the small duration of each group computation, static partitioning of the group computation works better than dynamic partitioning since the overload of dynamic partitioning is high.

4.3.2 Parallel Group Algorithm Description

To better illustrate the parallel group algorithm, we first describe its sequential version. The sequential group algorithm (c.f. Algorithm 1) takes a transition set, *trans*, as an input and computes the transition group, $trans_g$, as an output. Recall from Section 2.2 that the tasks involved in computing the *group* depend on the number of processes and the number of variables in the program. The sequential group algorithm (c.f Algorithm 1) needs to go through all the processes in the program and for each process it has to go through all the variables. The revision algorithm is required to compute the *group* associated with any set of transitions added/removed from the program transitions. Based on this discussion and the design choices above, we now describe the parallel group algorithm.

Algorithm sketch. Given transition set *trans* the goal of this algorithm is to compute the *Group* of transitions associated with the set *trans*. The sequential algorithm will go through many computations for each process, one after another. However, in the parallel algorithm, we split the *Group* computation over the available number of threads. In particular, rather than having one thread find the *Group* for all the processes, we let each thread compute the *Group* for a subset of the processes. Since the tasks assigned to each thread require a very small amount of the processor time, there is considerable overhead associated with the thread creation/destruction every time the *Group* is computed. Therefore, we let the master thread create the worker threads at the initialization stage of the revision algorithm.

The worker threads stay idle until the master thread needs to compute the *Group* for a set of transitions. The Master thread activates/deactivates the worker threads through a set of mutexes. When all worker threads are done, the main thread collects the results of all worker threads in one *Group*.

The parallel group algorithm consists of three parts: the initialization of the worker threads, the assignment of tasks to worker threads, and the computation of a group with worker threads.

Initialization. In the initialization phase, the master thread creates all required threads by calling the algorithm **InitiateThreads** (c.f. Algorithm 2). These threads stay idle until a group computation is required and terminate when the revision algorithm ends. Due to the design choice for load balancing, the algorithm distributes the work among the available threads statically (Lines 3-4). Then it creates all the required worker threads (Line 7).

Algorithm 2 InitiateThreads

```
Input: $noOfprocesses, noOfThreads.$

1: for $i := 0$ to $noOfThreads - 1$ do
2:     $BDDMgr[i] = Clone(masterBDDManager)$ ;
3:     $startP[i] := \lfloor \frac{i \times noOfprocesses}{noOfThreads} \rfloor$;
4:     $endP[i] := \lfloor \frac{(i+1) \times noOfprocesses}{noOfThreads} \rfloor - 1$;
5: end for
6: for $thID := 0$ to $noOfThreads - 1$ do
7:     $SpawnThread \rightsquigarrow GroupWorkerThread(thID)$;
8: end for
```

Tasks for worker thread. Initially, the algorithm **WorkerThread** (c.f. Algorithm 3) locks the mutexes *Start* and *Stop* (Lines 1-2). Then it waits until the master thread unlocks the *Start* mutexes (Line 5). At this point, the worker starts computing the part of the *Group* associated with this thread. This section of **WorkerThread** (Lines 7-15) is similar to the *Group()* function in the sequential revision algorithm, except rather than finding the *Group* for all the processes, the **WorkerThread** algorithm finds the group for a subset of processes

(Line 8). When the computation is completed, the worker thread notifies the master thread by unlocking the mutex *Stop* (Line 17).

Algorithm 3 WorkerThread

```
Input: $thID$.

    // Initial Lock of the mutexes
 1: $mutex\_lock(thData[thID].mutexStart)$;
 2: $mutex\_lock(thData[thID].mutexStop)$;
 3: while True do
 4:     // Waiting for the signal from the master thread
 5:     $mutex\_lock(thData[thID].mutexStart)$;
 6:     $gtr[id] := false$;
 7:     BDD* $tPred$ := BDD[ endP[thID] - startP[thID]+1 ] ;
 8:     for $i := 0$ to $(endP[thID] - startP[thID]) + 1$ do
 9:         $tPred[i] := thData[thID].trans \wedge$
            $allowWrite[i + startP[thID]].Transfer(BDDMgr[thID])$;
10:         $tPred[i] := FindGroup(tPred[i], i, thID)$;
11:     end for
12:     $thData[thID].result := false$;
13:     for $i := 0$ to $(endP[thID] - startP[thID]) + 1$ do
14:         $thData[thID].result := thData[thID].result \vee tPred[i]$;
15:     end for
16:     // Triggering the master thread that this thread is done
17:     $mutex\_unlock(thData[thID].mutexStop)$;
18: end while
```

Tasks for master thread. Given transition set *trans*, the master thread copies *trans* to each instance of the BDD package used by the worker threads (cf. Algorithm 4, Lines 3-5). Then it assigns a subset of group computation to the worker threads (Lines 6-8) and unlocks them. After the worker thread completes, the master thread collects the results and returns the group associated with the input *trans*.

Algorithm 4 MasterThread

Input: transitions set *thisTr*.
Output: transition group *gAll*.

```
1: $tr := thisTr$;
2: $gAll := false$;
3: for $i := 0$ to $NoOfThreads - 1$ do
4:     $threadData[i].trans := trans.Transfer(BDDMgr[thID])$;
5: end for
   // all idle threads to start computing the group
6: for $i := 0$ to $NoOfThreads - 1$ do
7:     $mutex\_unlock(thData[i].mutexStart)$;
8: end for
   // Waiting for all threads to finish computing the group
9: for $i := 0$ to $NoOfThreads - 1$ do
10:    $mutex\_lock(thData[i].mutexStop)$;
11: end for
   // Merging the results from all threads
12: for $i := 0$ to $NoOfThreads - 1$ do
13:    $gAll := gAll + thData[i].results$;
14: end for
15: return $gAll$;
```

4.3.3 Experimental Results

In this section, we describe the respective experimental results in the context of the Byzantine agreement (described in Section 4.2.1) and the token ring [14]. In both case studies, we find that parallelizing the group computation improves the execution time substantially. Throughout this section, all experiments are run on a Sun Fire V40z with 4 dual-core Opteron processors and 16 GB RAM. The OBDD representation of the Boolean formulae has been done using the C++ interface to the CUDD package developed at University of Colorado [125]. Throughout this section, we refer to the original implementation of the revision algorithm (without parallelism) as *sequential* implementation. We use *X threads* to refer to the parallel algorithm that utilizes *X* threads.

We would like to note that the revision time duration differences between the sequential implementation in this experiment and the one in [30] is due to other unrelated improvements on the sequential implementation itself. However, the sequential, and the parallel implementations differ only in terms of the modification described in Section 4.3.2.

We note that our algorithm is deterministic and the testbed is dedicated and, hence, the only non-deterministic factor in time for revision is synchronization among threads. Based on our experience with the revision, this factor has a negligible impact and, hence, multiple runs on the same data essentially reproduce the same results.

In Figures 4.1 and 4.2, we show the results of using the sequential approach versus the parallel approach (with multiple threads) to perform the revision. All the tests have shown that we gain a significant speedup. For example, in the case of 45 non-general processes and 8 threads, we gain a speedup of 6.1. We can clearly see that the parallel 16-thread version is faster than the corresponding 8-thread version. This is surprising, given that there are only 8 cores available. However, upon closer observation, we find that the *group* computation that is parallelized using threads is fine-grained. Thus, when the master thread uses multiple slave threads for performing the *group* computation, the slave threads complete quickly and therefore cannot utilize the available resources to the full extent. Hence, creating more

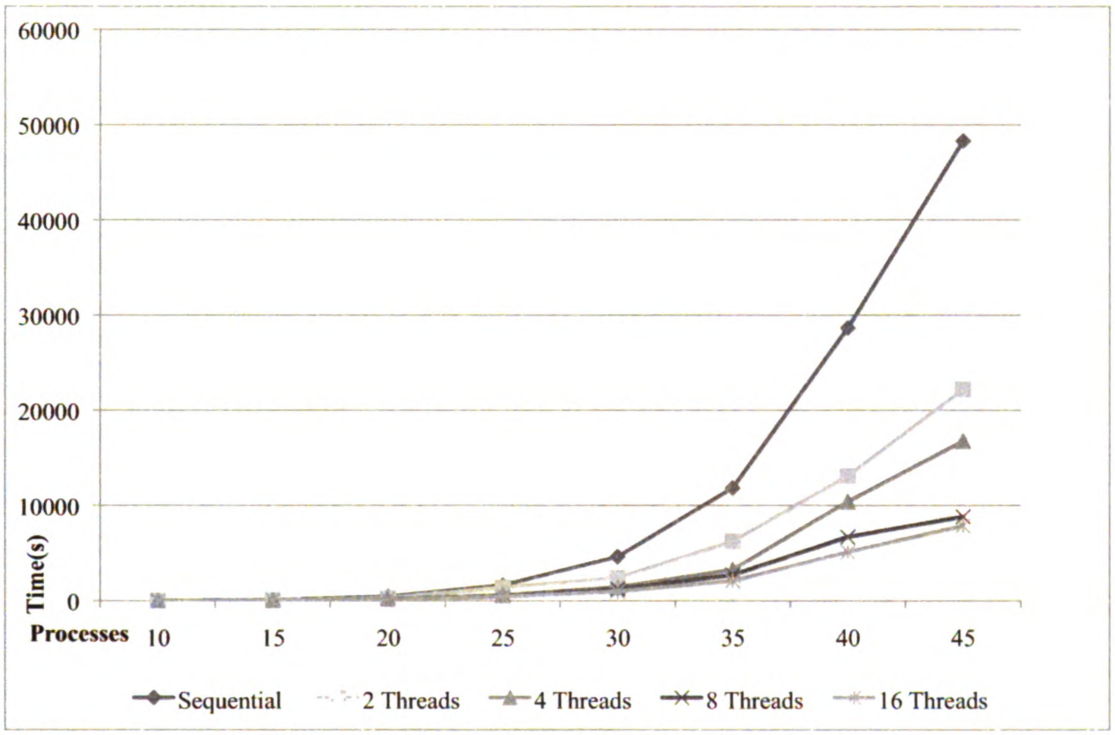

Figure 4.1: The time required to **resolve deadlock states** in the revision to add fault-tolerance for several numbers of non-general processes of **BA** in sequential and parallel algorithms.

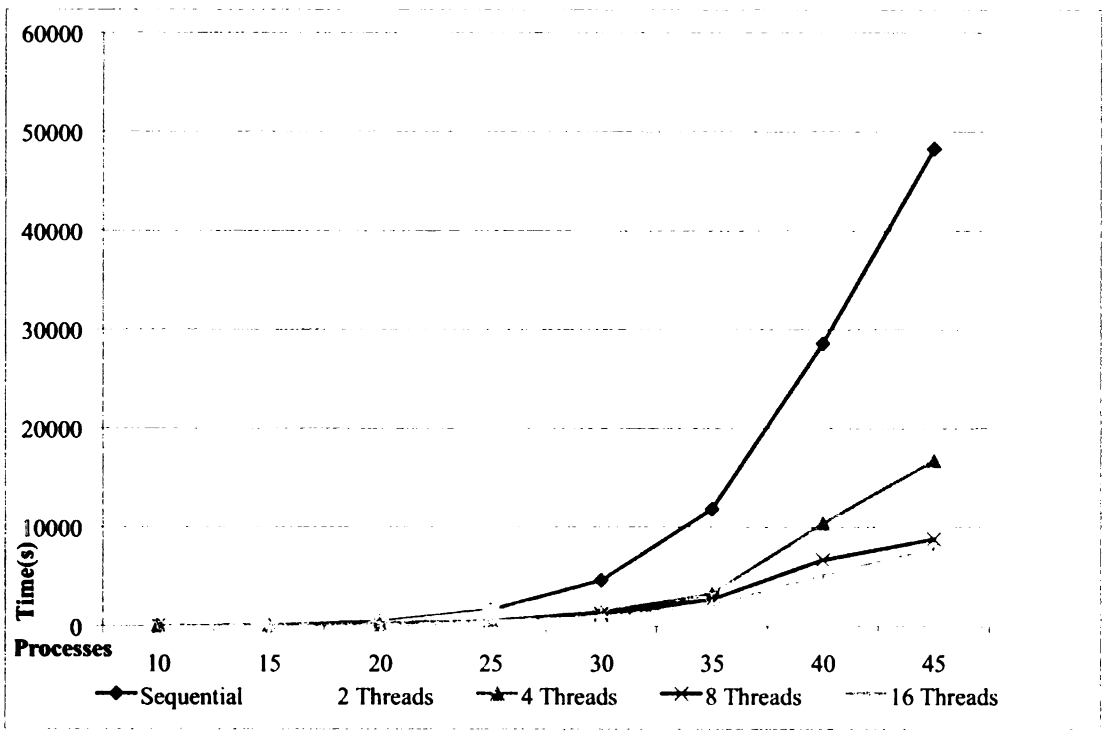

Figure 4.2: The time required for the revision to add fault-tolerance for several numbers of non-general processes of **BA** in sequential and parallel algorithms.

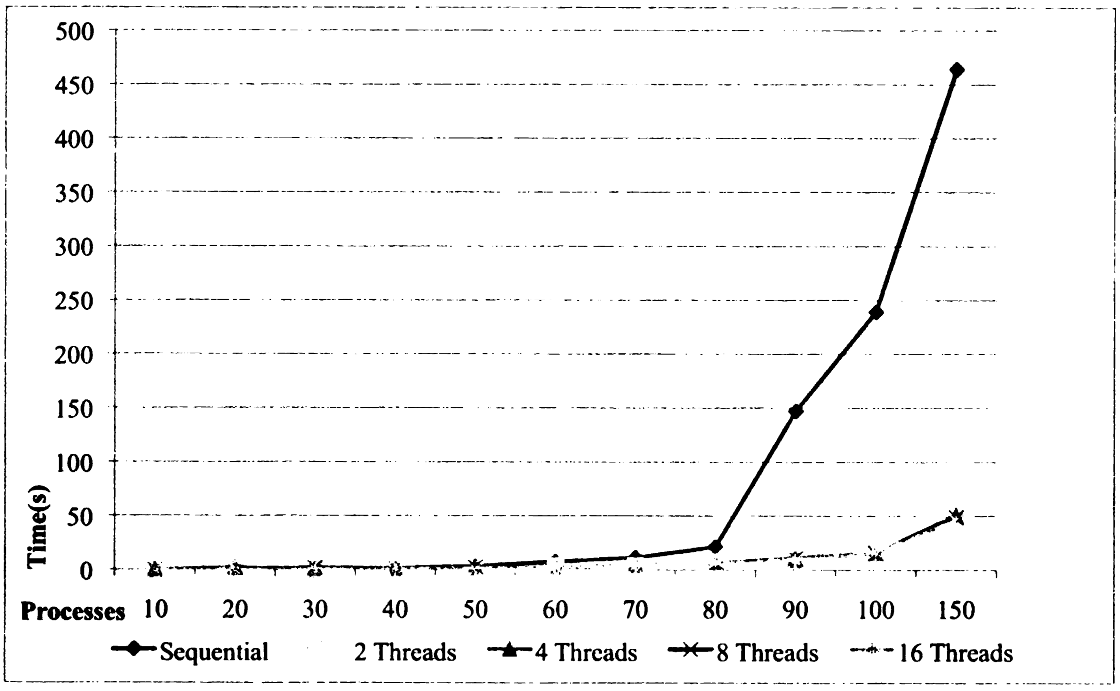

Figure 4.3: The time required to **resolve deadlock states** in the revision to add fault-tolerance for several numbers of **token ring** processes in sequential and parallel algorithms.

threads (than available processors) can improve the performance further.

In Figures 4.3 and 4.4, we present the results of our experiments in parallelizing the deadlock resolution of the token ring problem. After the number of processes exceeds a threshold, the execution time increases substantially. This phenomenon also occurs in the case of parallelized implementation, although it appears for larger programs. However, this effect is not as strong. Note that the spike in speedup at 80 processes is caused by the page fault behavior where the performance of the sequential algorithm is affected although the performance of the parallel algorithm is still not affected.

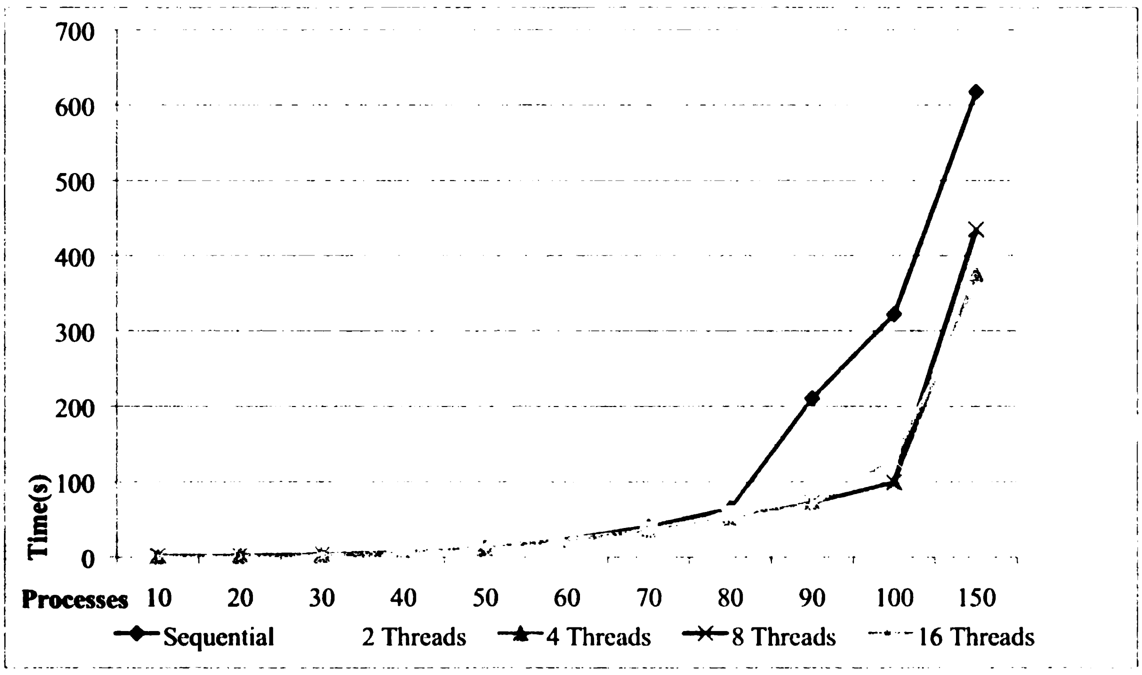

Figure 4.4: The time required for the revision to add fault-tolerance for several numbers of **token ring** processes in sequential and parallel algorithms.

4.3.4 Group Time Analysis

To understand the speedup gain provided by our algorithm in Section 4.5.2, we evaluated the experimental results closely. As an example, consider the case of 32 *BA* processes. For sequential implementation, the total revision time is 59.7 minutes of which 55 are used for group computation. Hence, the ideal completion time with 4 cores is 18.45 minutes $(55/4+4.7)$. By comparison, the actual time taken in our experiment was 19.1 minutes. Thus, the speedup using this approach is close to the ideal speedup.

In this section, we focus on the effectiveness of the parallelization of group computation by considering the time taken for it in sequential and parallel implementation. Towards this end, we analyze the group computation time for sequential and parallel implementation in the context of three examples: Byzantine agreement, agreement in the presence of failstop and Byzantine faults, and token ring [14]. The results for these examples are included in Tables 4.3-4.5.

In some cases, the speedup ratio is less than the number of threads. This is caused by the fact that each group computation takes a very small amount of time and incurs an overhead for thread synchronization. Moreover, as mentioned in Section 4.2.3, due to the overhead of load balancing, we allocate tasks of each thread statically. Thus, the load of different threads can be slightly uneven. We also observe that the speedup ratio increases with the number of processes in the program being revised. This implies that the parallel algorithm will scale to larger problem instances.

An interesting as well as surprising observation is that when the state space is large enough then the speedup ratio is more than the number of threads. This behavior is caused by the fact that with parallelization each thread is working on smaller BDDs during the group computation. To understand this behavior, we conducted experiments where we created the threads to perform the group computation and forced them to execute sequentially by adding extra synchronization. We found that such a pseudo-sequential run took less time than that used by a purely sequential run.

		Sequential	*2-threads*		*4-threads*		*8-threads*	
PR	**RS**	**GT(s)**	**GT(s)**	**SR**	**GT(s)**	**SR**	**GT(s)**	**SR**
15	10^{11}	50	29	1.72	17	2.94	11	4.55
24	10^{17}	652	346	1.88	185	3.52	122	5.34
32	10^{22}	3347	1532	2.18	848	3.95	490	6.83
48	10^{33}	33454	14421	2.32	7271	4.60	3837	8.72

Table 4.3: Group computation time for Byzantine Agreement. **PR**: Number of processes. **RS**: Size of reachable state space. **GT(s)**: Group time in seconds. **SR**: Speedup ratio.

		Sequential	*2-threads*		*4-threads*		*8-threads*	
PR	**RS**	**GT(s)**	**GT(s)**	**SR**	**GT(s)**	**SR**	**GT(s)**	**SR**
10	10^{10}	53	24	2.21	23	2.30	30	1.77
15	10^{15}	624	319	1.96	175	3.57	174	3.59
20	10^{20}	4473	2644	1.69	1275	3.51	1128	3.97
25	10^{25}	26154	11739	2.23	6527	4.01	5692	4.59

Table 4.4: Group computation time for the Agreement problem in the presence of failstop and Byzantine faults. **PR**: Number of processes. **RS**: Size of reachable state space. **GT(s)**: Group time in seconds. **SR**: Speedup ratio.

4.4 Approach 2: Alternative (Conventional) Approach

A traditional approach for parallelization in the context of resolving deadlock states, say ds, would be to partition the deadlock states into multiple threads and allow each thread to handle the partition assigned to it. Next, in Section 4.4.1, we discuss some of the design choices we considered for this approach. We give brief description of our algorithm in Section 4.4.2. Subsequently, we describe experimental results in Section 4.4.3 and analyze them to argue that for such an approach to work in revising distributed programs, group computation must itself be parallelized.

		Sequential	*2-threads*		*4-threads*		*8-threads*	
PR	**RS**	**GT(s)**	**GT(s)**	**SR**	**GT(s)**	**SR**	**GT(s)**	**SR**
30	10^{14}	0.32	0.15	2.12	0.10	3.34	0.12	2.75
40	10^{19}	0.84	0.36	2.34	0.22	3.84	0.23	3.59
50	10^{23}	1.82	0.68	2.68	0.39	4.66	0.42	4.37
60	10^{28}	3.22	1.22	2.63	0.67	4.80	0.64	5.01
70	10^{33}	5.36	1.91	2.80	1.06	5.05	0.86	6.23
80	10^{38}	7.77	2.94	2.64	1.53	5.09	1.23	6.30

Table 4.5: Group computation time for token ring. **PR**: Number of processes. **RS**: Size of reachable state space. **GT(s)**: Group time in seconds. **SR**: Speedup ratio.

4.4.1 Design Choices

To maximize the benefits from parallelism, we consider two factors when partitioning the deadlock state among available threads. First, the deadlock states should be distributed evenly among the threads. Second, the partitions should minimize the overlap between worker threads. More specifically, states considered by one thread should not be considered by an other thread. Therefore, we partition the deadlock states based on the values of the program variables. We use the size of the BDDs and the number of minterms to split the deadlock states as evenly as possible. Regarding the second factor, we chose to add limited synchronization among worker threads to reduce the overlap in the explored states by different threads. For example, we can partition ds using the partition predicates, $prt_i, 1 \leq i \leq n$, such that $\bigvee_{i=1}^{n}(prt_i \wedge ds) = ds$ and n is the number of threads. Thus, if two threads are available during the revision of the Byzantine agreement program then we can let $prt_1 = (d.j = 0)$ and $prt_2 = (d.j \neq 0)$.

After partitioning, each thread would work independently as long as it does not affect the states visited by other threads. As discussed in Section 4.2.3, to resolve a deadlock state, each thread explores a part of the state space using backward reachability. Clearly, when the states visited by two threads overlap, we have two options: (1) perform synchronization so that only one thread explores any state or (2) allow two threads to explore the states concurrently and resolve any inconsistencies that may be created due to this.

We find that following the first option by itself is very expensive/impossible due to the fact that with the use of BDDs, each thread explores a set of states specified by the BDD. And, since each thread begins with a set of deadlock states and performs backward reachability, there is a significant overlap among states explored by different threads. Hence, following the first option essentially reduced the parallel run to a sequential run. For this reason, we focused on the second approach where each thread explored the states concurrently. (We also used some heuristic-based synchronization where we maintained a set of *visited* states that each thread checked before performing backward state exploration. This provided a small performance benefit and is included in the results below.)

4.4.2 Algorithm Sketch

In this section, we focus on the descriptions of the parallel aspect of our deadlock resolution algorithm. For more details on the sequential algorithm for deadlock resolution please refer to [101].

The goal of our algorithm (c.f. Algorithm 5) is to resolve the deadlock states by adding safe recovery. However, if for some deadlock states safe recovery is not possible, the algorithm eliminates such states (i.e. makes them unreachable). To efficiently utilize the available worker threads, the master thread partitions the set of deadlock states among available threads as described in Section 4.4.1 and provides each thread with its own partition. Subsequently, the master thread activates the worker threads to add safe recovery (c.f. Algorithm 6). Once activated, in adding safe recovery mode, each worker thread works as follows. It constructs the recovery transitions that originate from the deadlock states and leads to the legitimate states of the program in a finite number of steps. Of course, the algorithm does not include any transition that reaches a state from where the safety of the program can be violated. Once all worker threads are done computing the recovery transitions, the master thread merges the recovery transitions, returned by all threads, and adds them to the program transitions.

Algorithm 5 ResolveDeadlockStates

Input: program p, faults f, legitimate state predicate I, fault span T, prohibited transitions mt, and partition predicates $prt_1..prt_n$, where n is the number of worker threads.

Output: program p' and the predicate fte of states failed to eliminate.

```
 1: $ds := T \wedge \neg g(p)$;
    // Resolving deadlock states by adding safe recovery
 2: for $i := 1$ to $n$ do
 3:     $rt_i :=$ SpawnThread $\rightsquigarrow$ AddRecovery($ds \wedge prt_i, I, mt$);
 4: end for
    // Merging results from worker threads
 5: $p := p \vee \bigvee_{i=1}^{n} rt_i$;
 6: $vds, fte := false$;
 7: $ds := T \wedge \neg g(p)$;
    // Eliminating deadlock states from where safe recovery is not possible
 8: for $i := 1$ to $n$ do
 9:     $rp_i, vds_i, fte_i$ := SpawnThread $\rightsquigarrow$ Eliminate($ds \wedge$
        $prt_i, p, I, f, T, vds, fte$);
10: end for
    // Merging results from worker threads
11: $p' := Group(\bigwedge_{i=1}^{n} rp_i)$;
12: $fte, vds := \bigvee_{i=1}^{n} fte_i, \bigvee_{i=1}^{n} vds_i$;
    // Handling inconsistencies
13: $nds := ((T \wedge \neg I) \wedge \neg g(p')) \wedge \neg((T \wedge \neg I) \wedge \neg g(p))$;
14: $p' := p' \vee Group(p \wedge nds)$;
15: $p' := p' \vee Group(g(p) \wedge \langle fte \rangle')$;
16: return $p', fte$;
```

At this point, the master thread computes the remaining deadlock states. This set identifies the deadlock states from which safe recovery is not possible. As mentioned earlier in Section 4.2.3, those states have to be eliminated (i.e., made unreachable by program

Thread 6 AddSafeRecovery

Input: deadlock states *ds*, legitimate state predicate *I*, and transition predicate *mt*.

Output: recovery transition predicate *rec*.

```
1: $lyr, rec := I, false$;
2: repeat
3:     $rt := Group(ds \wedge \langle lyr \rangle')$;
4:     $rt := rt \wedge \neg Group(rt \wedge mt)$;
5:     $rec := rec \vee rt$;
6:     $lyr := g(ds \wedge rt)$
7: until ($lyr = false$);
8: return $rec$;
```

transitions). Once again the master thread partitions the deadlock states and provides each worker thread with one such partition. Subsequently, it activates the worker threads. Once activated, in eliminating mode (c.f. Algorithm 7), the worker threads remove all program transitions that terminate at the deadlock states, thereby making them unreachable. However, if the removal of some of those transitions introduces new deadlock states, then the algorithm puts back such transitions and recursively eliminates the recently introduced deadlock states.

When threads explore states concurrently, some inconsistencies may be created. Next, we give a brief overview of the inconsistencies that may occur due to concurrent state exploration by different threads and identify how we can resolve them. Towards this end, let s_1 and s_2 be two states that are considered for deadlock elimination and (s_0, s_1) and (s_0, s_2) be two program transitions for some s_0. To eliminate s_1 and s_2, a sequential elimination algorithm removes transitions (s_0, s_1) and (s_0, s_2), which makes s_0 be a new deadlock state (cf. Figure 4.5.a).

This in turn requires that state s_0 itself must be made unreachable. If s_0 is unreachable, then including the transition (s_0, s_1) and (s_0, s_2) in the revised program is harmless. In fact, it is desirable since including this transition also causes other transitions in the corre-

Thread 7 Eliminate

Input: deadlock states ds, program p, legitimate state predicate I, fault transitions f, fault span T, visited deadlock states vds, predicate fte failed to eliminate.

Output: revised program transition predicate p, visited deadlock states vds, predicate fte failed to eliminate.

```
1: wait(mutex);
2:         $ds := ds \wedge \neg vds$;
3:         $vds := vds \vee ds$;
4: signal (mutex);
5: if ($ds = false$) then
6:         return $p$;
7: end if
8: $old := p$;
9: $tmp := (T \wedge \neg I) \wedge p \wedge \langle ds \rangle'$;
10: $p := p \wedge \neg Group(tmp)$;
11: $fs := g(T \wedge \neg I \wedge f \wedge \langle ds \rangle')$;
12: $p, vds, fte :=$ Eliminate($fs, p, I, f, T, vds, fte$);
13: $nds := g(T \wedge \neg I \wedge Group(tmp) \wedge \neg g(p))$;
14: $p := p \vee (Group(tmp) \wedge nds)$;
15: $nds := nds \wedge g(tmp)$;
    // $\langle X \rangle'' = \{(s_1, true) | (s_0, s_1) \in X\}$
16: $fte := fte \vee \neg \langle old \wedge \neg p \wedge T \wedge \langle ds \rangle' \rangle''$;
17: $p, vds, fte :=$ Eliminate($nds \wedge \neg I, p, I, f, T, vds, fte$);
18: return $p$, $vds$, $fte$;
```

sponding group to be included as well. And, these grouped transitions might be useful in providing recovery from other states. Hence, it puts back (s_0,s_1) and (s_0,s_2) (and corresponding group) and starts eliminating the state s_0. However, the concurrent execution of worker threads may create some inconsistencies. We describe some of these inconsistencies and our approach to resolve them next.

Case 1. States s_1 and s_2 are in different partitions. Therefore, th_1 eliminates s_1, which in turn removes the transition (s_0,s_1), and th_2 eliminates s_2, which removes the transition (s_0,s_2) (cf. Figure 4.5.b). Since each thread works on its own copy, neither thread tries to eliminate s_0, as they do not identify s_0 as a deadlock state. Subsequently, when the master thread merges the results returned by th_1 and th_2, s_0 becomes a new deadlock state that has to be eliminated while the group predicates of transitions (s_0,s_1) and (s_0,s_2) have been removed unnecessarily. In order to resolve this case, we replace all outgoing transitions that start from s_0 and mark s_0 as a state that has to be eliminated in subsequent iterations.

Case 2. To eliminate deadlock states, the elimination algorithm performs backward exploration starting from the deadlock state. Thus, two or more threads may consider the same state for elimination. For example, if th_1 consider s_1 for elimination and th_2 consider both s_1 and s_2 (c.f. Figure 4.5.b) then th_1 removes (s_0, s_1) and th_2 removes (s_0, s_1) and (s_0, s_2). Now, when the master thread joins the results of the two threads, the transition (s_0, s_1) is removed. However, as shown in Case 1, the removal of (s_0, s_1) is not really necessary. In fact, we would like to keep this transition in the program for the reasons mentioned above. To handle this inconsistency, we collect such transitions and add them back to the program transitions.

4.4.3 Experimental Results

We also implemented this approach for parallelization. The results for the problem of Byzantine agreement are as shown in Table 4.6. From these results, we noticed that the improvement in the performance was small. To analyze these results, we studied the effect

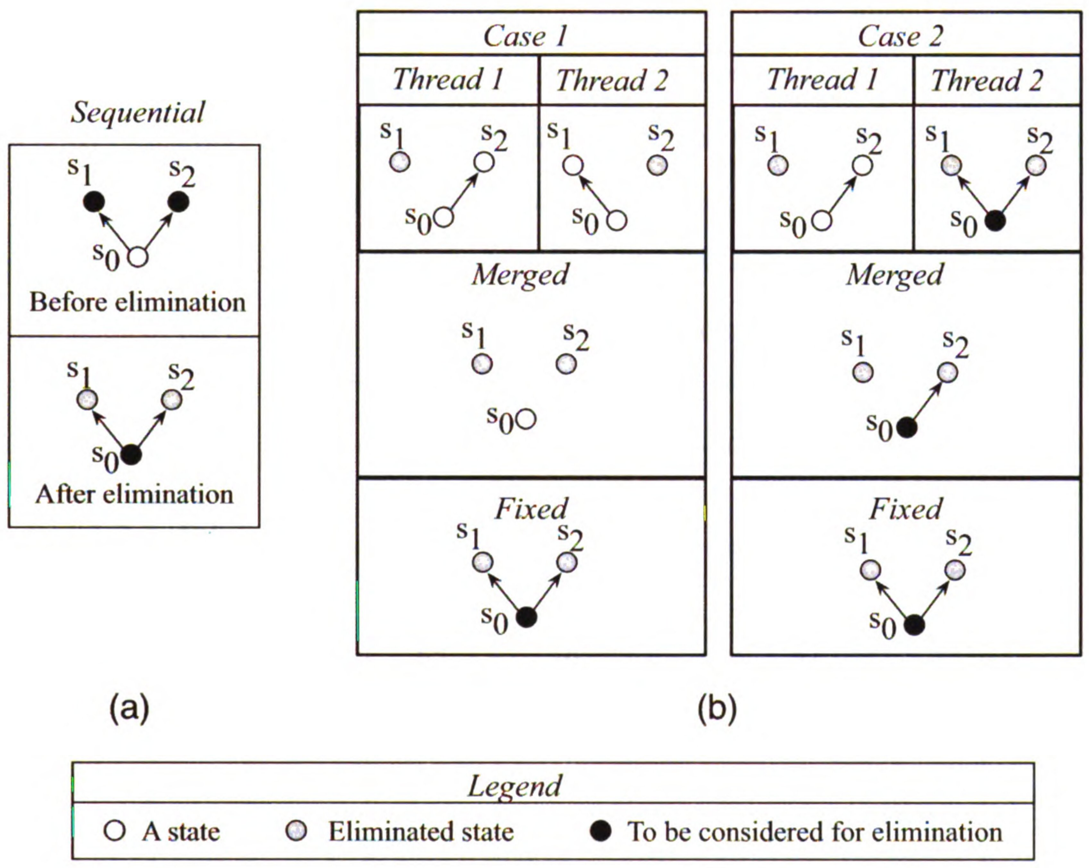

Figure 4.5: Inconsistencies raised by concurrency.

of this approach in more detail. For the case where we utilize two threads, this approach partitions the deadlock states, say ds, into two parts, $ds1$ and $ds2$. Thread 1 begins with $ds1$ and performs backward exploration to determine how states in $ds1$ can be made unreachable. In each such backward exploration, if it chooses to remove some transition, then it has to perform a group computation to remove the corresponding group. Although this thread is working with a smaller set of deadlock states, the time required for group computation is only slightly less than the sequential implementation where only one thread was working with the entire set of deadlock states ds. Moreover, the time required in such group computation is very high (more than 80%) compared to the overall time required for eliminating the deadlock states. This implies that, especially for the case where we are revising a program with a large number of processes and where the available threads are relatively small, parallelization of the group computation is going to provide us the maximum benefit.

		Sequential		*Parallel Elimination with 2-threads*	
PR	**RS**	**DR(s)**	**TST(s)**	**DRT(s)**	**TST(s)**
10	10^{7}	7	9	8	9
15	10^{12}	78	85	78	87
20	10^{14}	406	442	374	417
25	10^{18}	1,503	1,632	1,394	1,503
30	10^{21}	4,302	4,606	3,274	3,518
35	10^{25}	11,088	11,821	10,995	11,608
40	10^{28}	27,115	28,628	21,997	23,101
45	10^{32}	45,850	48,283	39,645	41,548

Table 4.6: The time required for the revision to add fault-tolerance for several numbers of non-general processes of *BA* in sequential and by partitioning deadlock states using parallelism.**PR**: Number of processes. **RS**: Size of reachable state space. **DRT(s)**: Deadlock resolution time in seconds. **TST(s)**: Total revision time in seconds.

4.5 Using Symmetry to Expedite the Automated Revision

In this section, we present our approach for expediting the revision with the use of symmetry using the input from Section 4.2.1. We utilize this approach in the task of resolving deadlock states that are encountered during the revision process. Therefore, using the example *BA* from Section 4.2.1, we describe how symmetry can help in resolving them. Then we discuss our algorithms for resolving deadlock states by utilizing symmetry to expedite the two aspects of deadlock resolution: adding recovery and eliminating deadlock states.

4.5.1 Symmetry

To describe the use of symmetry, consider the first scenario described in Section 4.2.3. In this scenario, we resolved the state s_1 by adding a recovery transition. Due to the symmetry of the non-generals, one can observe that we can also add other recovery transitions. For example, if we consider the state $d.g = d.j = d.l = 0, d.k = 1, and f.k = 0$, we can add the recovery transition by which $d.k$ changes to 0.

With this observation, if we identify recovery action(s) to be added for one process, we can add the similar actions that correspond to other processes. Therefore, to add recovery, our algorithm does the following: whenever we find recovery transition(s), we identify other recovery transitions based on symmetry. Then, we add all these recovery transitions to the program being revised (c.f. Algorithm 8).

We also apply symmetry for deadlock states elimination. To eliminate a set of deadlock states, we find the set of transitions, which if removed from one process, will prevent that process from reaching deadlock states. Then, we use this set of transitions to remove similar transitions from other processes. Therefore, to eliminate deadlock states by removing program transitions, our algorithm does the following: whenever we find a set of transition(s), if removed from one process, the algorithm prevents the program from reaching a deadlock state; we use symmetry to identify similar transitions for other processes, and we

Algorithm 8 Add_Symmetrical_Recovery

```
Input: deadlock states $ds$, legitimate state predicate $I$, and the set of unacceptable transitions including those in $spec_{bt}$  $mt$
Output: recovery transitions predicate $rec$

1: $rec := ds \wedge \langle I \rangle'$;
   // $\langle I \rangle'$ the set of states to which recovery can be added to ensure recovery to legitimate states

2: $rec := Group(rec)$;
   // Select program transition or process i while ensuring read/write restrictions

3: $rec := rec \wedge \neg Group(rec \wedge mt)$;
   // Remove transition that violate safety while ensuring distribution restrictions

   // Find similar transitions for other processes
4: for $i := 1$ to $numberOfProcesses$ do
5:     $rec := rec \vee$ SwapVariables( $rec, i$ );
       // Generate BDDs for other processes by swapping variables based on symmetry
6: end for
7: return $rec$;
```

remove these transitions from program transitions (c.f. Algorithm 9).

Algorithm 9 Group_Symmetry

```
Input: a set of transitions $trans$.
Output: a group of transitions $grp$.

1: $grp := FindGroup(trans,$ read/write restrictions on $i)$;
   // Find the group related to process i transitions while ensuring the
   read/write restrictions

   // Find similar transitions for other processes
2: for $i := 1$ to $numberOfProcesses$ do
3:   $grp := grp \vee$ SwapVariables( $grp, i$);
4: end for

5: return $grp$;
```

4.5.2 Experimental Results

In Section 4.5.1, we described the use of symmetry approaches to resolve deadlock states in the automated revision. In Sections 4.5.2-4.5.2, we describe and analyze the respective experimental results. In particular, we describe the results in the context of two classical examples in the literature of distributed computing, namely, the Byzantine agreement (described in Section 4.2.1) and the token ring [14]. In both case studies, we find that symmetry and parallelism improve the execution time substantially.

Symmetry

In this section, we present our experimental results in using symmetry for the resolution of deadlock of deadlock states in the automated revision.

Figure 4.6 shows the time spent in deadlock resolution, and Figure 4.7 shows the total revision time for different numbers of processes in the Byzantine agreement problem.

From this figure, we observe that the use of symmetry provides a remarkable improvement in the performance. More importantly, one can notice that the speedup ratio (gained using a symmetrical approach) grows with the increase in the number of processes. In particular, as shown in Figure 4.7, the speedup ratio in the case of 10 non-general processes is 4.5. However, in the case of 45 non-general processes the speedup ratio is 19. This behavior is both expected and highly valuable. Since symmetry uses transitions of one process to identify transitions of another process, it is expected that as the number of symmetric processes increases, so would the effectiveness of symmetry. Moreover, since the speedup is proportional to the number of (symmetric) processes, we argue that symmetry would be highly valuable in handling the state space explosion with an increased number of processes.

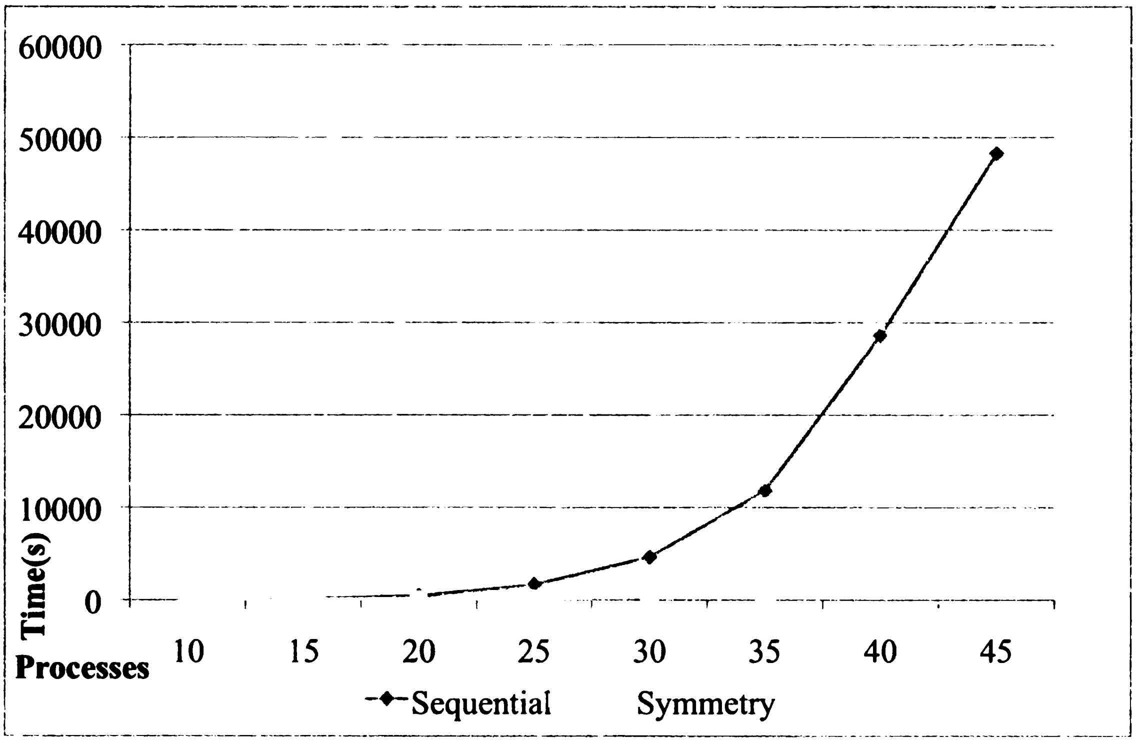

Figure 4.6: The time required to **resolve deadlock states** in the revision to add fault-tolerance for several numbers of BA non-general processes in sequential and symmetrical algorithms.

To explain this remarkable improvement, we focus on the fact that far more time is spent resolving deadlock states for each process *independently* than by simply resolving

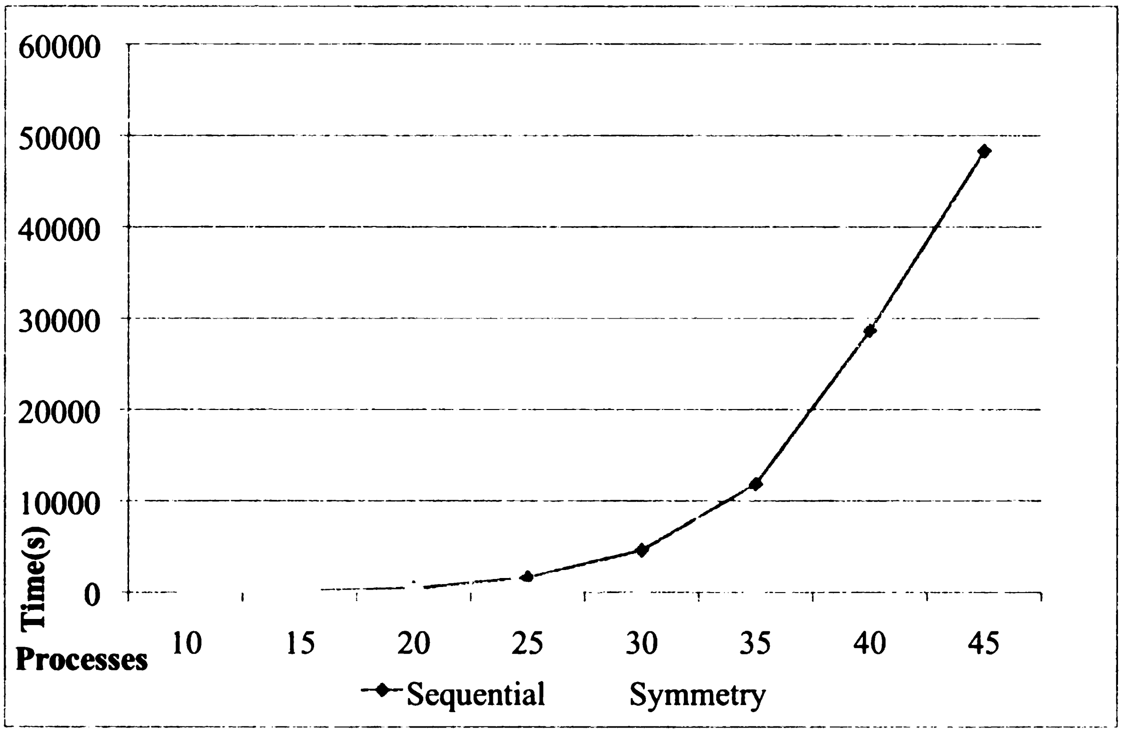

Figure 4.7: The time required for the revision to add fault-tolerance for several numbers of BA non-general processes in sequential and symmetrical algorithms.

deadlock states for single process and using symmetry to resolve deadlock states for the rest of the processes. Consequently, symmetry is expected to give better speedup ratios when the number of symmetrical processes is large.

In Figures 4.8 and 4.9, we present the results of our experiments on the token ring problem. We observe that symmetry substantially reduces the time for deadlock resolution. In fact, symmetry was able to keep this time almost a constant, i.e., independent of the problem size. One can notice a spike in the required revision time of the sequential algorithm for token ring after we hit the threshold of 90 processes. This behavior was also observed in [30] and is caused by the fact that, at this state space, we are utilizing all the available memory, causing performance to degrade due to page faults.

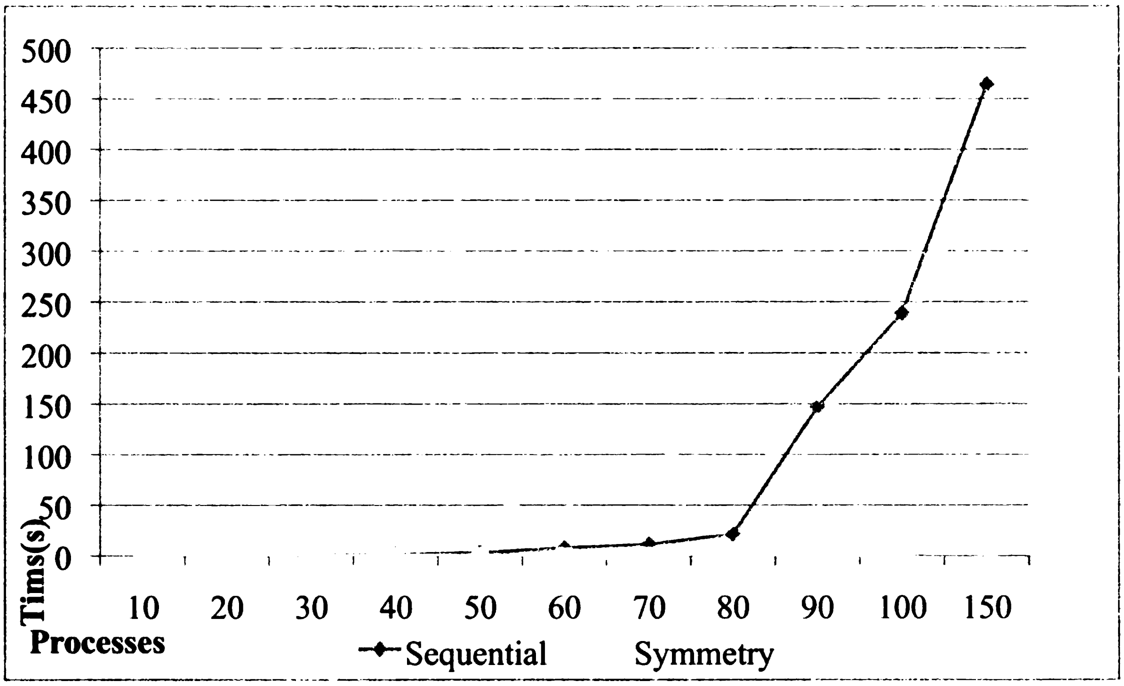

Figure 4.8: The tttttime required to **resolve deadlock states** in the revision to add fault-tolerance for several numbers of **token ring** processes in sequential and symmetrical algorithms.

Symmetry and Parallelism

In this section, we present our experimental results of using parallelism in computing the symmetry. The results of parallelizing the symmetry computation with various implementations in the automated symbolic revision are presented in Figure 4.10. We have achieved the shortest revision time when we use parallelism to compute the symmetry. For example, in the case of the Byzantine agreement with 45 non-general processes using 16 threads, we achieve a speedup ratio of 1.8 times that of the symmetry alone. Since in case of the token ring, symmetry alone reduces the time of computing recovery transitions to a negligible amount, the results for this case are omitted.

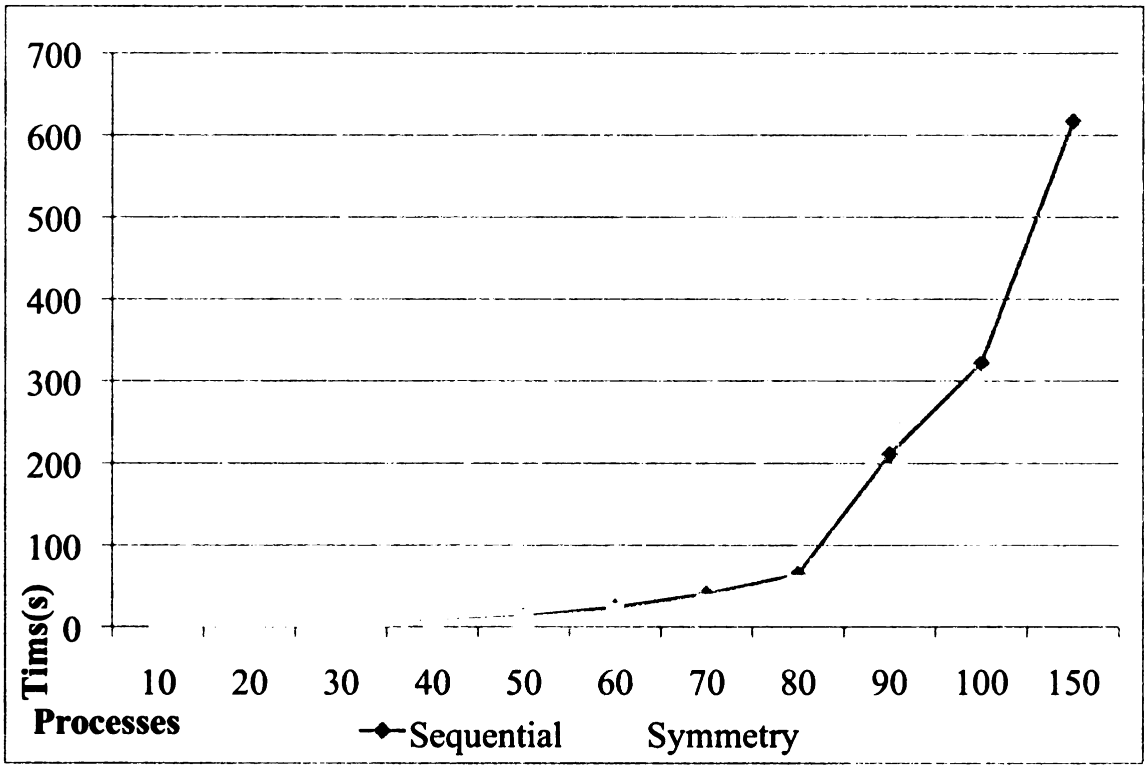

Figure 4.9: The time required for the revision to add fault-tolerance for several numbers of **token ring** processes in sequential and symmetrical algorithms.

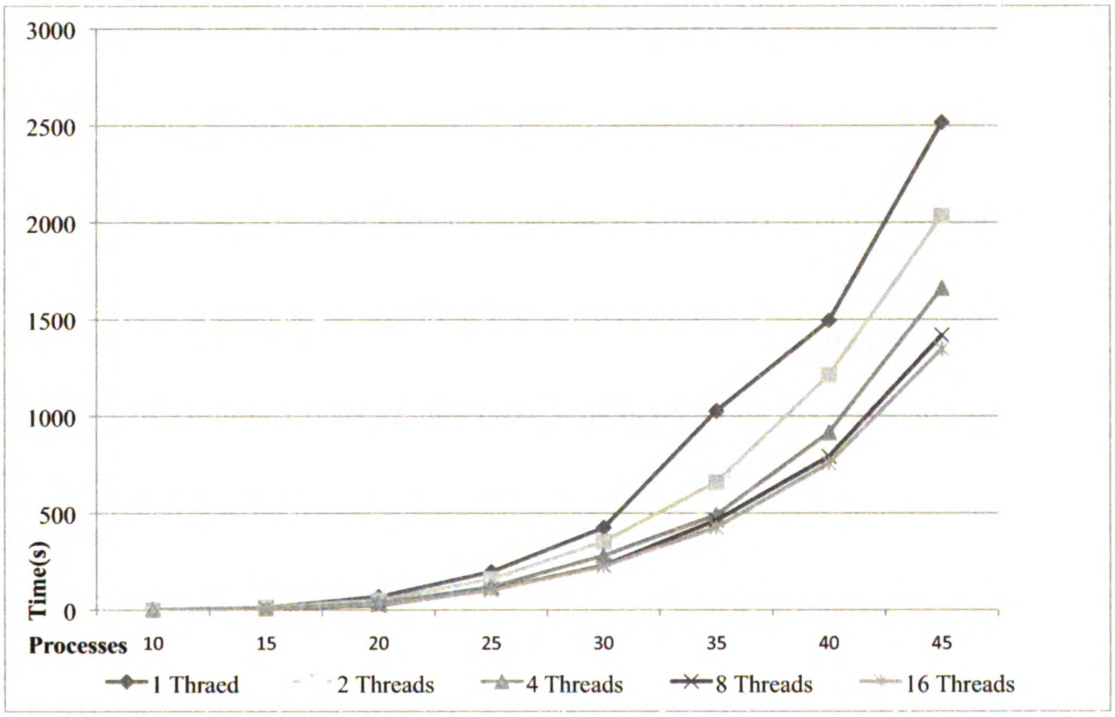

Figure 4.10: The time required for the revision to add fault-tolerance for several numbers of BA non-general processes using both symmetry and parallelism.

4.6 Summary

In this chapter, we focused on the techniques that can efficiently complete the automated model revision in a reasonable amount of time. Specifically, we used techniques that exploit symmetry and parallelism to expedite the automated model revision and to overcome its bottlenecks. For parallelism, our approach was based on parallelization with multiple threads on a multi-core architecture. We found that the performance improvement with the simple parallelization of the group computation is significantly more efficient than traditional approaches that partition the deadlock states among available threads. With group computation parallelism we achieved significant benefit that is close to the ideal. In the case of symmetry, we used the fact that multiple processes in a distributed program are symmetric in nature. We used this characteristic to efficiently expedite the automated revision. Since, the cost of identifying the transition of a given model with the knowledge of symmetry among processes is less than the cost of identifying these transitions explicitly, the use of symmetry reduces the overall time required for the revision. Moreover, the speedup increases as the number of symmetric processes increases.

Lessons Learned. The results show that a traditional approach of partitioning deadlock states provides a small improvement. However, it helped identify an alternative approach for parallelization that is based on the distribution constraints imposed on the program being revised. While parallelization reduces the time spent in eliminating deadlock states, it may also lead to some inconsistencies that have to be resolved. The time for resolving such inconsistencies is one of the bottlenecks in parallelization, as this inconsistency is resolved sequentially. We note that the synchronization on *visited states* was also added, in part, to reduce inconsistencies among threads by requiring them to coordinate with each other.

The performance improvement with the parallelizing of the group computation is significant. In fact, for most cases, the performance was close to the ideal speedup. What this suggests is that for the task of deadlock resolution, a simple approach based on parallelizing the group computation (as opposed to a reentrant BDD package or partitioning of the deadlock states, etc.) will provide the biggest benefit in performance. Moreover, the group computation itself occurs in every aspect of the revision where new transitions have to be added for recovery or existing transitions have to be removed for preventing safety violations or breaking cycles that prevent recovery to the set of the legitimate states model/program. Therefore, the approach of parallelizing the group computation will be effective in the automated model revision of distributed programs.

Impact. Automated model revision has been widely believed to be significantly more complex than automated verification. When we evaluate the complexity of automated revision to add fault-tolerance, we find that it fundamentally includes two parts: (1) analyzing the existing program and (2) revising it to ensure that it meets the fault-tolerance properties. We showed that the complexity of the second part can be significantly remedied by the use of parallelization in a *simple* and *scalable* fashion. Moreover, if we evaluate the typical inexpensive technology that is currently being used or is likely to be available in the *near future*, it is expected to be 2-16 core computers. And, the first approach used in this chapter is expected to be the most suitable one for utilizing these multi-core computers to the fullest extent. Also, since the group computation is caused by distribution constraints of the program being revised, it is guaranteed to be required even with other techniques for expediting automated revision. For example, it can be used in conjunction with the approach for parallelizing the group as well as the approach that utilizes symmetry among processes being revised. Hence, even if a large number of cores were available, this approach would be valuable together with other techniques that utilize those additional cores.

Memory Usage. Both of our approaches, symmetry and parallelism, require the use of more memory. For instance, the revision of the *BA* with 2 threads requires almost twice the amount of memory needed by the sequential algorithm for the same number of non-general processes. However, unlike model checking, in automated model revision, since we always run out of time before we run out of memory, we argue that the extra usage of memory is acceptable given the remarkable reductions we achieve in total revision time.

Chapter 5

Nonmasking and Stabilizing Fault-Tolerance

Achieving practical automated model revision requires us to derive theories and develop algorithms that provide broader domain of problems, which we can resolve by automated model revision. Towards this end, in this chapter, we focus on the constraint-based automated addition of nonmasking and stabilizing fault-tolerance to hierarchical programs. We specify legitimate states of the program in terms of constraints that should be satisfied in those states. To deal with faults that may violate these constraints, we add recovery actions while ensuring interference freedom among the recovery actions added for satisfying different constraints. Since the constraint-based *manual* design of fault-tolerance is well known to be applicable in the manual design of nonmasking fault-tolerance, we expect our approach to have a significant benefit in automation of fault-tolerant programs. We illustrate our algorithms with three case studies: stabilizing mutual exclusion, stabilizing diffusing computation, and a data dissemination problem in sensor networks. With experimental results, we show that the complexity of revision is reasonable and that it can be reduced using the *structure* of the hierarchical systems.

To our knowledge, this is the first instance where automated revision has been successfully used in revising programs that are correct under fairness assumptions. Moreover, in two of the case studies considered in this chapter, the structure of the recovery paths is too complex to permit existing heuristic-based approaches for adding recovery.

To expedite the revision, we concentrate on reducing the time complexity of such revision using parallelism. We apply these techniques in the context of constraint satisfaction. We consider two approaches to speedup the revision algorithm: first, the use of the multiple constraints that have to be satisfied during revision; second, the use of the distributed nature of the programs being revised. We show that our approaches provide significant reductions in the revision time.

The rest of the chapter is organized as follows. In Section 5.2, we define the problem statement for the automated addition of nonmasking and stabilizing fault-tolerance. We describe the algorithms for the automated addition of nonmasking and stabilizing fault-tolerance in Section 5.3. We present our multi-core algorithms in Section 5.4 and experimental results in Section 5.5. In Section 5.6, we study the ordering in which the constraints should be satisfied. We show how we can use the hierarchical structure to reduce the complexity of our algorithm in Section 5.7. Finally, we summarize the chapter in Section 5.8.

5.1 Introduction

In this chapter, we focus on automated addition of *nonmasking* and *stabilizing* fault-tolerance to fault-intolerant programs. Intuitively, a nonmasking fault-tolerant program ensures that if it is perturbed by faults to an illegitimate state, then it would eventually recover to its legitimate states. However, safety may be violated during recovery. Therefore, nonmasking fault-tolerance is useful to tolerate a temporary perturbation of the program state. After recovery is completed, a nonmasking fault-tolerant program satisfies both the safety and liveness in the subsequent computation. Nonmasking and stabilizing fault-tolerance is an ideal solution to add fault-tolerance to the programs that organize network nodes in specified topology or a predefined logical structure [13].

There are several reasons that make the design of nonmasking fault-tolerance attractive. For one, the design of masking fault-tolerant programs, where both safety and liveness are preserved during recovery, is often expensive or impossible even though the design of nonmasking fault-tolerance is easy [15]. Also, the design of nonmasking fault-tolerance can assist and simplify the design of masking fault-tolerance [105]. Moreover, in several applications nonmasking fault-tolerance is more desirable than solutions that provide fail-safe fault-tolerance (where in the presence of faults the program reaches to "safe" states from where it does not satisfy liveness requirements). This is especially true for networking related applications such as routing and tree maintenance.

A special case of nonmasking fault-tolerance is stabilization [54,56], where, starting from an arbitrary state, the program is guaranteed to reach a legitimate state. Stabilizing systems are especially useful in handling unexpected transient faults. Moreover, this property is often critical in long-lived applications where faults are difficult to predict. Furthermore, it is recognized that verifying stabilizing systems is especially hard [76]. Hence, techniques for automated revision are expected to be useful for designing stabilizing systems.

Techniques for adding nonmasking and stabilizing fault-tolerance to distributed programs can be classified in two categories. The first category includes approaches based on *distributed reset* [13], where the program utilizes approaches such as *distributed snapshot* [38] and resetting the system to a legitimate state if the current state is found to be illegitimate. Approaches from this category suffer from several drawbacks. In particular, they require the designer to know the set of all legitimate states. The cost of detecting the global state can be high. Additionally, this approach is heavy-handed since it requires a reset of the entire system, even if the fault may be localized.

The second category includes approaches based on *constraint satisfaction*, where we identify constraints that should be satisfied in the legitimate states. Typically, the constraints are local (e.g., involving one node or a node and its neighbors); therefore, detecting their violation is easy. Since the constraints are local, the recovery actions to fix them are also local.

There are several issues that complicate the design of nonmasking and stabilizing fault-tolerance [10]. One such issue is the complexity of designing and analyzing the recovery actions needed to ensure that the program recovers to legitimate states. Another issue is that to verify correctness of the nonmasking fault-tolerant program, one needs to consider all possible concurrent executions of the original program, recovery actions, and fault actions. Yet another issue is that most nonmasking algorithms assume that faults can keep happening (although they will eventually stop for a long enough time to permit recovery) even during recovery, thereby, complicating the recovery to legitimate states.

Adding nonmasking and stabilizing fault-tolerance to an existing program is achieved by performing three steps. The first step is to identify the set of legitimate states of the fault-intolerant program. This set defines the constraints that should be true in the legitimate states. The second step is to identify a set of convergence actions that recover the program from illegitimate states to legitimate states. This can be done by finding actions that satisfy one or more constraints. The last step consists of ensuring that the convergence actions do not interfere with each other. In other words, the collective effect of all recovery actions should, eventually, lead the program to legitimate states.

In this chapter, we automate the last two steps by identifying the necessary actions to ensure that the constraints are satisfied and that the recovery actions do not interfere with each other. The automation of the first step is discussed in details in Chapter 6.

However, this approach suffers from one important drawback: local actions taken to fix one constraint may violate other constraints. Consequently, these constraints need to be ordered. Furthermore, we need to ensure that satisfying one constraint does not violate constraints *earlier* in the order. Since verifying that recovery actions for satisfying one constraint do not affect other constraints is a demanding task, automated techniques that ensure correctness by construction are highly desirable. In the correct-by-construction approach, a program is automatically revised such that the output program preserves the original program specification. In addition, it satisfies new properties. However, algorithms for designing programs that are correct by construction suffer from high complexity and, hence, techniques to expedite them need to be developed. Since the time complexity of the automation algorithms can be high, we also evaluate parallelization techniques to expedite addition of nonmasking and stabilizing algorithm.

In this chapter, we present an automated model revision algorithm for constraint-based synthesis of nonmasking and stabilizing fault-tolerant programs. We illustrate our algorithm with three case studies. We note that the structure of the recovery actions in the first two case studies is too complex to permit previous approaches to achieve revision of the corresponding fault-tolerant programs [30]. We also show that the structure of the hierarchical system can be effectively used to generalize programs with a small number of processes while preserving the *correct-by-construction* property of the revised program.

Also, we present a multi-core algorithm to synthesize distributed nonmasking and stabilizing fault-tolerant programs by partitioning the satisfaction of the constraints among available threads. To further expedite the revision, we also present a multi-core algorithm that utilizes the distributed nature of programs being revised by parallelizing them.

To our knowledge, this is the first instance where programs that require fairness assumptions have been revised with automated techniques. Particularly, in our first case study, it is straightforward to observe that stabilizing fault-tolerance cannot be added without some fairness among all processes. Thus, the previous algorithms (e.g., [30]) will declare failure in adding fault-tolerance.

5.2 Programs and Specifications

In this section, we define the problem statement for adding nonmasking and stabilizing fault-tolerance. Please note that the problem statements defined in this section are instances of the original definition of the fault-tolerance from Section 2.5. Those definitions are based on the ones given by Arora and Gouda [12]. Also, we use the definitions of distributed programs, fairness, legitimate states, faults, and fault-span from Chapter 2.

The goal of an algorithm that adds nonmasking fault-tolerance is to begin with a fault-intolerant program p, its legitimate state predicate I, and faults f, and to derive the nonmasking fault-tolerant program, say p', such that in the presence of faults, p' eventually converges to I. Furthermore, computations of p' that begin in I must be the same as that of p.

Based on this discussion, we define the problem of adding nonmasking fault-tolerance as follows:

Problem statement 4.1 Given p, I, and f, identify p' such that:

- Transitions within the legitimate states remain unchanged

  $s_0 \in I \Rightarrow (\forall s_1 :: (s_0,s_1) \in p \Longleftrightarrow (s_0,s_1) \in p')$

- There exists a state predicate T (fault-span) such that

  - $I \subseteq T$,
  - $(s_0,s_1) \in (p' \vee f) \wedge (s_0 \in T) \Rightarrow s_1 \in T$,
  - $s_0 \in T \wedge \langle s_0, s_1, \ldots \rangle$ is a computation of p'
    $\Rightarrow (\exists j : j \geq 0 : s_j \in I)$.

Stabilizing fault-tolerance is a special instance of this problem statement with the requirement that $T = S_p$, i.e. the fault-span equals the set of all states. Based on this discussion, we define the problem of adding stabilizing fault-tolerance as follows:

Problem statement 4.2 Given p, I, and f, identify p' such that:

- Transitions within the legitimate states remain unchanged:
  - $s_0 \in I \Rightarrow (\forall s_1 :: (s_0, s_1) \in p \Longleftrightarrow (s_0, s_1) \in p')$
- All program transitions eventually converge to the set of legitimate states
  - $s_0 \in S_p \wedge \langle s_0, s_1, ... \rangle$ is a computation of p'

    $\Rightarrow \quad (\exists j : j \geq 0 : s_j \in I)$

Note that since each constraint is preserved by the original program p, closure property of the stabilizing program p' is satisfied from the first constraint of the problem statement. Thus, it is not explicitly specified above.

5.3 Synthesis Algorithm of the Nonmasking and Stabilizing Fault-Tolerance

Our approach for adding nonmasking and stabilizing to fault-intolerant programs, based on [13]. The goal of nonmasking and stabilizing fault-tolerance is to ensure that after faults occur, the program eventually reaches one of the legitimate states in I. We focus on the instance of the problem where $I = C_1 \wedge C_2 ... \wedge C_m$ and C_i, $1 \geq i \geq m$, is a constraint on the variables of the program. Faults perturb the program to a state in $(\neg I)$. Hence, in the presence of f, one or more of the constraints from $C_1, C_2 ... C_m$ are violated. The goal of our algorithm is to automatically synthesize the recovery actions such that when faults stop occurring, the constructed recovery actions in conjunction with the original program actions will, eventually, converge the program to a state where I holds.

5.3.1 Constraint Satisfier

Our algorithm for adding nonmasking and stabilizing fault-tolerance is shown in Algorithm 10. The input for the algorithm is the constraint array C, fault-span T, and program p. In this algorithm, the constraints from the constraint array are satisfied one after another. The algorithm starts by computing the legitimate state predicate as the intersection of all constraints in the constraint array (Lines 3).

Then, the algorithm computes the recovery transitions to satisfy $C[i]$. Let Tr denote transitions that begin in the fault-span and in a state where $C[i]$ is false and end in a state where $C[i]$ is true. Unfortunately, we cannot add Tr as is, since Tr may not be implementable using read/write constraints on processes due to the distributed nature of the program. The algorithm adds a subset of Tr, say Tr_1, such that Tr_1 can be implemented using the read/write restrictions of one or more processes. We denote this by the function $Group_{min}$ (see Line 6)[1]. This ensures that the only transitions added are those that start from a state where $C[i]$ is false and reach a state where $C[i]$ is true. These transitions are denoted by *temp* on Line 6.

Subsequently, the algorithm removes transitions from *temp* that violate the closure of the fault-span T. Thus, it computes a subset of transitions, say Tr_{fspan}, in *temp* that begin in a state in T and reach a state in $\neg T$. Again, we need to ensure that the removed transitions are consistent with read/write restrictions of processes. The algorithm achieves this by applying function $Group_{max}$ to Tr_{fspan}; this computes a superset of Tr_{fspan} such that one or more processes can execute it. Subsequently, it removes this superset from *temp* (Line 7). This ensures that all transitions that violate closure of T are removed. Therefore, it removes the group of transitions that violates T (respectively, I) (Lines 7-8).

The algorithm needs to ensure that none of the transitions used to satisfy the constraint, say $C[i]$, violates the pre-satisfied constraints $C[0]$ to $C[i-1]$. Hence, it lets V include the transitions that originate from a state where $C[i-1]$ is *true* and end in a state where

[1]$(X \wedge \langle Y \rangle')$ refers to the transitions that start in a state in X and reach Y.

$C[i-1]$ is *false* as well as similar transitions for the constraints $C[0]$ to $C[i-2]$ (Line 11). The transitions in V are used to ensure that recovery transitions do not violate other pre-satisfied constraints. The algorithm ensures that none of the transitions in *temp* interfere with earlier constraints. Therefore, it removes the transitions in V from *temp* if any are found (Line 9). At this point, the algorithm collects all recovery transitions in *rec* (Line 10). Steps 4 – 12 are repeated until all the recovery actions that satisfy all the constraints in the array C are found. Finally, it returns the recovery actions of the program p.

Algorithm 10 ConstraintSatisfier

```
Input: constraint array $C$, fault-span $T$, and program transitions $p$.
Output: recovery transitions $rec$.

1: $temp, V := false, false$;
2: $m := SizeOf(C) - 1$; // $m$ is the number of constraints
3: $I := \bigwedge_{i=0}^{m} C[i]$; //Compute $I$ (invariant) as the intersection of all constraints
4: for $i := 0$ to $m$ do
5:     //$temp$ are the transitions that start in a state in $T - C(i)$ and reach $C(i)$
6:     $temp := Group_{min}((T - C[i]) \wedge \langle C[i] \rangle')$;
       //ensure that no recovery transitions violate $T$
7:     $temp := temp - Group(temp * (T \wedge \langle \neg T \rangle'))$;
       //ensure that no recovery transitions violate $I$
8:     $temp := temp - Group(temp * (I \wedge \langle \neg I \rangle'))$;
9:     $temp := temp - V$ ;
       // Combine current recovery transitions with the new recovery transition.
10:    $rec := rec \vee temp$;

       //Compute, $V$, the set of the transitions that violating the constraints
11:    $V := V \vee Group(C[i] \wedge \langle \neg C[i] \rangle')$
12: end for
       // return the recovery transition.
13: return $rec$;
```

Theorem 5.3.1 *:*

- *Given are fault-intolerant program p, constraints $C_1, C_2 ... C_m$, and faults f.*
- *Let $I = C_1 \wedge C_2 ... \wedge C_m$.*
- *Let T= set of states reached in the execution of $p \vee f$ that start from any state in I.*
- *Let rec= $ConstraintSatisfier(C,\ T,\ p)$.*

If

$\forall s_0 : s_0 \in T - I : (\exists s_1 : s_1 \in T : (s_0, s_1) \in rec)$

Then

p' $(= p \vee rec)$ *solves the constraints in Problem statement 4.1.* ∎

Proof. To prove Theorem 3.1 we show that the p' $(= p \vee rec)$ solves the constraints of the problem statement 4.1.

- By the construction of the transitions in *rec*, it is straightforward to see that *rec* does not introduce any new transitions in I. Therefore, the transitions within the legitimate states remain unchanged.
- By the construction of T, it is clear that $I \subseteq T$ since T includes all the states in I as well as the states reachable from I by $(p \vee f)$.
- From Line 7 in the algorithm *ConstraintSatisfier*, the transitions in *rec* do not include any transition that violates T.
- Since *rec* does not include any of the transitions from V (Lines 9 and 11), none of the transitions in *rec* violate pre-satisfied constraints. Therefore, there will be no cycles between the recovery transitions themselves. Hence, the constraint $(s_0 \in T \wedge \langle s_0, s_1, ... \rangle$ is a computation of $p' \Rightarrow (\exists j : j \geq 0 : s_j \in I))$ is satisfied. ∎

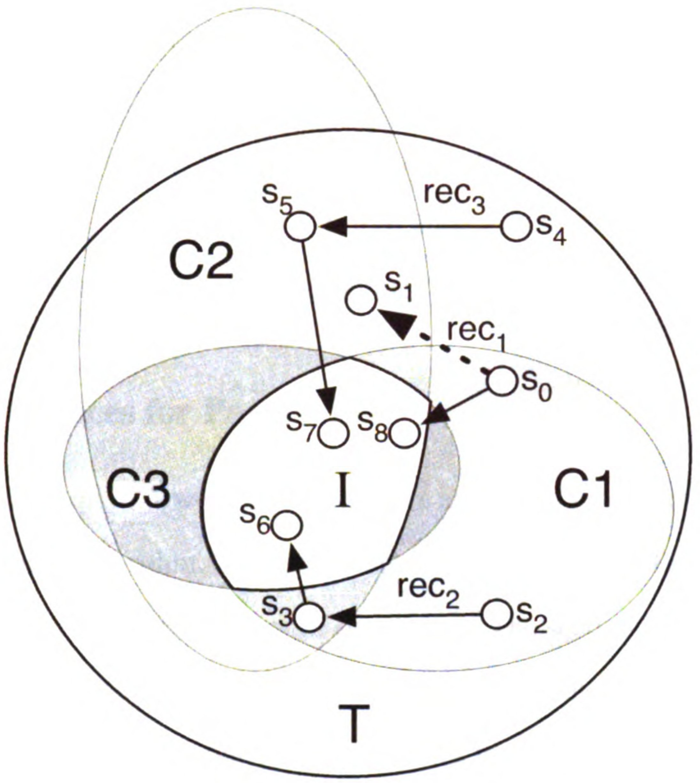

Figure 5.1: Constraints ordering and transitions selections.

5.3.2 Algorithm Illustration

To illustrate the algorithm *ConstraintSatisfier*, consider the system described in Figure 5.1. In this system, we have three ordered constraints C_1, C_2, and C_3 and $I = C_1 \wedge C_2 \wedge C_3$.

Since C_1 is the first to be satisfied, we construct all possible recovery actions that start from any state in $T - C_1$ and reach a state in $C_1 \wedge T$. We proceed to satisfy C_2 in the same manner. However, after constructing the recovery actions that satisfy C_2, we need to exclude actions that violate the constraint C_1. In particular, we exclude actions like rec_1 (c.f. Figure 5.1) since it starts from a state, s_0, where C_1 is true and ends in a state, s_1, where C_1 is false. On the other hand, we keep transitions like rec_2 and rec_3. We continue to construct the recovery actions that establish C_3 provided that they preserve T, C_1, and C_2.

5.4 Expediting the Constraints Satisfaction

In Section 5.3, we described the sequential approach (i.e., single thread) for synthesizing nonmasking and stabilizing fault-tolerant distributed programs from fault-intolerant versions. In this section, we explain our design choices and present our approaches for expediting the revision with multi-core computing (i.e., multiple threads).

5.4.1 Design Choices for Parallelism

After reviewing Algorithm 10, we can see that there are two main bottlenecks, which lower the performance of this algorithm. The first is the main loop (Lines 4-12) where the number of iterations is determined by the number of constraints. The second is the Group operation in Lines 6, 7, 8, and 11. The group operation is based on the nature of distributed programs where addition of a transition for one process requires us to add additional transitions that are computed based on what the process cannot read/write.

Choices for constraint satisfaction. One way to partition the computation of recovery transitions is to split the recovery computation among multiple threads by allowing them to work on satisfying separate constraints. However, Algorithm 10 uses the computation of V, transitions that violate preceding constraints (Line 11). Clearly, one possibility is to compute all possible values taken by V during the computation up front and utilize them appropriately for computing valid recovery transitions. Computing the possible values taken by V also requires a computation that utilizes a loop that requires $sizeOf(C)$ iterations, which can be parallelized using standard techniques from parallel computing.

After computation of V, we can partition the iterations (Lines 4-12 in Algorithm 10) among several threads. We considered several approaches for this. One approach we considered was dynamic partitioning. In particular, in this approach, a pool of uncompleted iterations is maintained. Each thread picks an iteration from this pool and computes the recovery transitions for that iteration. Subsequently, it picks another iteration from the pool and so on. We found that this dynamic partitioning approach, however, resulted in a high overhead, thereby reducing the speedup. Hence, we considered static partitioning where each thread was given fixed iterations. Even here, we tried different options. One option was to partition the iterations in an alternating manner (e.g., thread 1 gets iterations $0, 2, 4, \ldots$ and thread 2 gets iterations $1, 3, 5, \ldots$). It was expected that this would leave the size of MDDs used in each thread to be evenly balanced. However, we found that this approach and the approach of partitioning where thread 1 got iterations $0, 1, \ldots (sizeOf(C)/2) - 1$ and thread 2 got iterations $sizeOf(C)/2, \ldots sizeOf(C) - 1$ had almost identical performance in the case studies. We have used the latter in our experiments. However, we believe that the choice of partitioning could play a role in other case studies.

Choices for utilizing distributed nature. When the recovery algorithm adds new transitions (or removes transitions that violate earlier constraints), we have to add the corresponding group of transitions based on the distributed nature of the program. Moreover, with symbolic approach, we add (or remove) a set of transitions at a time. This set may include transitions that could be executed by several processes. Therefore, for a given set of transitions that are added, we need to consider read/write restrictions of each of these processes to determine the group for that set of transitions. We can utilize this feature to parallelize the group computation itself by having each thread compute the group corresponding to a subset of processes.

Again, similar to the parallelization with constraints, we considered several approaches. It turned out that even for this approach, the overhead of dynamic partitioning was more than its benefit. Thus, we utilized static approaches. Since several approaches considered for partitioning resulted in a similar speedup, we utilize the simple approach where each thread obtains a subset of processes and computes the corresponding group for those processes.

Finally, in group parallelization, the actual computation involved in the group itself is small. Hence, we found that the overhead of creating and terminating threads for each group computation was very high. For this reason, we created the threads up front and used mutexes to determine when they will be active.

Choices for parallelizing the MDD (Multi-Valued Decision Diagrams) library. Since we are using MDD-based symbolic revision [28], the constraints are characterized by Boolean formulae involving the variables in the program being revised. The MDD library [125] is not designed to be reentrant and assumes that at most one MDD package is active at any given time. Multiple threads cannot operate on the same MDD package simultaneously. Also, different threads cannot access different MDD packages simultaneously. We considered two approaches to solve this problem: (1) utilize a reentrant version of the MDD package, or (2) utilize multiple independent MDD packages. Since a reentrant MDD package is not available, we followed the second approach. We modified the MDD library so that multiple instances could be used simultaneously. We also added a *Transfer* function to transfer an MDD object from one MDD package to a different MDD package. Hence, during the parallel algorithms, a *master* thread spawns several *worker* threads, each running on a different core/processor in parallel with an instance of its own MDD package. The instance of the MDD package assigned to each worker thread is initialized using MDDs (e.g., program transitions MDD) transferred from the MDD package of the master thread.

5.4.2 Partitioning the Constraints Satisfaction

Based on the design choices from Section 5.4.1, we present a multi-core algorithm that partitions the satisfaction of such constraints among available cores/processes.

Algorithm sketch. Intuitively, our algorithm works as follows. During constraint satisfaction, a *master* thread spawns several *worker* threads, each running on a different core/processor. Each worker thread runs on its own MDD package concurrently with other threads. The instance of the MDD package assigned to each worker thread is initialized using MDDs transferred from the MDD package of the master thread. Some of those MDDs are the array of constraints to be satisfied, the program transitions, the array of constraints violating transitions, and the legitimate state predicate. The master thread partitions the constraints and provides each worker thread with one such partition. Subsequently, worker threads start resolving their assigned set of constraints in parallel by adding the required *recovery* actions. Upon completion, the master thread *merges* the results returned by the worker threads.

Algorithm 11 ParallelConstraintsSatisfaction [Master Thread]

```
Input: constraint array $C$, program transitions $p$, fault-span $T$, and number
    of threads $n$.
Output: recovery transitions $recAll$.
 1: $gAll := false$;
 2: $I := \bigwedge_{i=0}^{m} C[i]$;
    // Notation: $C[i] \wedge \langle \neg C[i] \rangle'$ refers to transitions that start in $\neg C[i]$ and
    ends in $C[i]$
 3: for $i := 1$ to $n-1$ do
 4:     SpawnThread $\rightsquigarrow$ ComputeViolate($i$);
 5: end for
 6: for $i := 1$ to $SizeOf(C) - 1$ do
 7:     $V[i] := V[i-1] \vee V[i]$;
 8: end for
 9: for $i := 0$ to $n-1$ do
10:     $C_p[i] = Split(i, C)$;
11:     $V_p[i] = Split(i, V)$;
12: end for
13: for $i := 1$ to $n-1$ do
14:     $rec[i]$ := SpawnThread $\rightsquigarrow$ PConstraintSatisfier($C_p[i], p$, fault-span
        $T, V_p[i], I$);
15: end for
16: ThreadJoin(0..$n-1$);
17: $recAll := \bigvee_{i=0}^{n-1} rec[i]$; // Merging the results from all threads
18: return $recAll$;
```

Parallel Constraints Satisfaction. Our algorithm for satisfying the constraints in parallel is as shown in Algorithm 11. This algorithm begins with the array of constraints to be satisfied C, fault-intolerant program p, fault-span T, and the number of worker threads to be spawned n. The goal of this algorithm is to discover the set of recovery transitions *recAll* such that all the constraints in C are satisfied in a way that enables the fault-tolerant program to recover to its legitimate states. Initially, the algorithm starts by computing the legitimate state predicate I as the intersection of all constraints (Lines 2). Now, the algorithm constructs the array V such that $V[i]$ includes the transitions that start from a state where $C[i]$ is true and end in a state where $C[i]$ is false as well as the similar transitions for the constraints $C[j]$, where $0 \leq j \leq i-1$ (Lines 3-8). A more efficient way to do this computation is by letting the master thread use the worker threads such that each worker thread computes its share of V elements such that $V[i]$ contains the transitions that starts from $C[i]$ and end in $\neg C[i]$. Once all threads are done, the master thread updates the array V such that $V[i] = V[i-1] \vee V[i]$. In other words, $V[i]$ contains all transitions that violate the constraint $C[0]$ to $C[i]$.

After constructing the array V, the algorithm proceeds to evenly distribute the elements of the arrays C and V among the worker threads (Lines 9-12). Specifically, $C_p[i]$ includes the array of constraints assigned to the thread i, and $V_p[i]$ includes the array of corresponding constraints violating transitions. Note that the availability of the array V_p enables each worker thread to work independently without interfering with the other threads. To compute the respective recovery transitions, each worker thread (Lines 13-15) calls the algorithm *PConstraintSatisfier*, which is similar to Algorithm 10 except that in addition to C_p and p it also takes V_p and I as an input. Once all worker threads complete their jobs (Line 16), the master thread collects all the recovery transitions returned by worker threads in *recAll* (Lines 17-19) and returns the overall recovery transitions.

5.5 Case Studies

In Section 5.3, we presented our approach for constraint-based automated addition of nonmasking and stabilizing fault-tolerance. In Section 5.4, we presented different approaches to exploit parallelism. In Subsections 5.5.1-5.5.3, we describe and analyze three case studies, namely the Stabilizing Mutual Exclusion [124], the stabilization of Data Dissemination Problem in Sensor Networks [104], and the Stabilizing Diffusing Computation [13]. Of these, the first and the third case study are classic problems from distributed computing and illustrate the feasibility of algorithms that add stabilizing fault-tolerance. In the second case, study we demonstrate the applicability of our approach on a real world problem, particularly, in the field of sensor networks. In all of these case studies, we find that our approach for constraint-based automated addition of nonmasking and stabilizing fault-tolerance was successful in synthesizing the nonmasking fault-intolerant programs. Furthermore, we find that parallelism significantly reduces the total revision time.

Throughout this section, all experiments are run on, sun x4275 with 4 x Quad-core Intel Xeon E5520 (2.27GHz w/ 8MG cache each) processors with 24 GB RAM. The MDD representation of the Boolean formulae has been done using a modified version of the MDD/BDD Glu 2.1 package [125] developed at the University of Colorado.

5.5.1 Case Study 1: Stabilizing Mutual Exclusion Program

Mutual exclusion is one of the fundamental problems in distributed/concurrent programs. One of the classical solutions to this problem is the token-based solution due to Raymond [124]. In this solution, the processes form a directed rooted tree, a *holder tree*, in which there is a unique token held at the tree root. If a process wants to access the critical section, it must first acquire the token. Our goal in this case study is to add stabilization to the fault-intolerant program in [15]. When faults occur and perturb the holder tree, the new program will stabilize and reconstruct a correct holder tree within a finite number of steps under weak fairness assumption.

Fault-Intolerant Program. In Raymond's algorithm, the processes are organized in a logical tree, denoted as a parent. The holder tree is superimposed on top of the parent tree such that the root of the holder tree is the process that has the token. For example, Figure 5.2.a represents the undirected parent tree and Figure 5.2.b shows the holder tree when c has the token. In the fault-intolerant program, each process j has a variable $h.j$. If $h.j = j$ then j has the token. Otherwise, $h.j$ contains the process number of one of j's neighbors. The holder variable forms a directed path from any process in the tree to the process currently holding the token.

In this program, a process can send the token to one of its neighbors. For example, Figure 5.2.c shows the case where process c sends the token to e. In particular, if j and k are adjacent (in the parent tree), then the action by which k sends the token to j is as follows:

$$A1 :: (h.k = k \wedge j \in Adj.k) \wedge (h.j = k) \longrightarrow h.k,\ h.j := j,\ j;$$

Constraints. Recall from Section 5.2 that we define the legitimate states to be a set of constraints on the program state space. In this case study, this set is the conjunction of the constraints $S1, S2$, and $S3$, described next. Moreover, each of these constraints is specified for each process separately. Therefore, if n is the number of processes then we have $3n$ constraints to satisfy. Constraint $S1$ requires that j's holder can either be j's parent, j itself, or one of j's children. $S2$ requires that the holder tree conforms to the parent tree. Finally, S3 requires that there are no cycles in the holder relation. Thus, predicates $S1, S2$, and $S3$ are as follows:

$$(S1)\ \ \forall j : (h.j = P.j) \vee (h.j = j) \vee (\exists k : (P.k = j) \wedge (h.j = k))$$

$$(S2)\ \ \forall j : (P.j \neq j) \Rightarrow (h.j = P.j) \vee (h.(P.j) = j)$$

$$(S3)\ \ \forall j : (P.j \neq j) \Rightarrow \neg((h.j = P.j) \wedge (h.(P.j) = j))$$

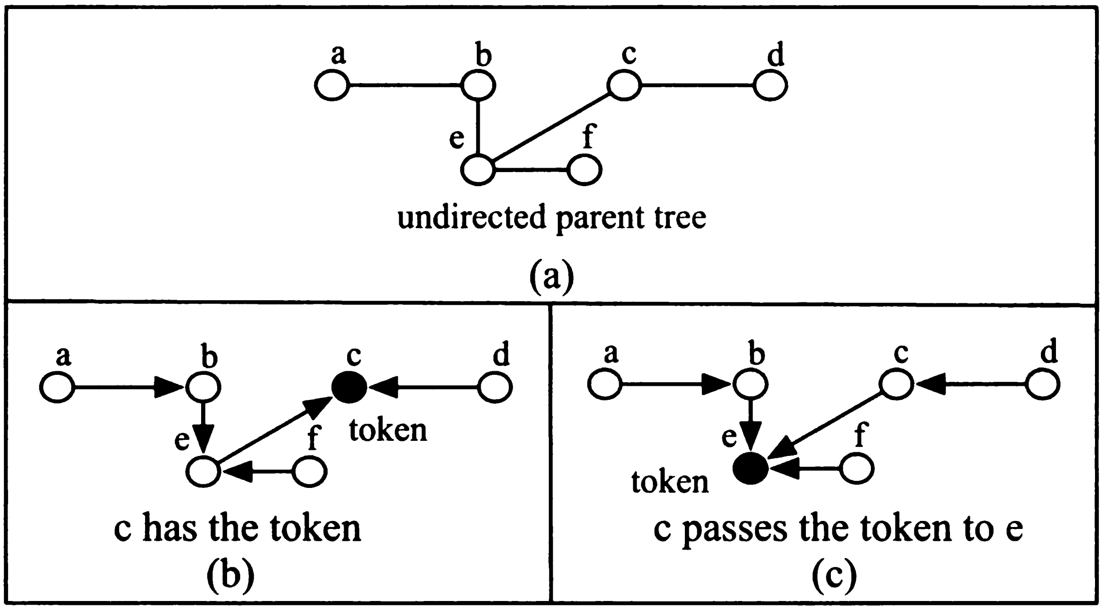

Figure 5.2: The holder tree

Faults. Since we focus on stabilizing fault-tolerance, we consider faults that perturb the holder relation of all processes to an arbitrary value. Thus the fault action is as follows:

$(F1)$ $\mathit{true} \longrightarrow \{h.j :=$ any arbitrary value from its domain$\}$;

Fault-Tolerant Program. To add stabilizing fault-tolerance to the above program, we used the revision algorithm as follows. The fault intolerant program for each process is specified by actions $A1$; the faults are specified by the fault action $F1$; and the constraints are from $S1$, $S2$, and $S3$. We specified these constraints in the following order: first, we specified constraints $S1$ for the root, then its children, then its grandchildren, and so on. Subsequently, we specified constraint $S2$ likewise. Finally, we specified constraint $S3$ in the reverse order. The recovery actions computed by the revision algorithm are as follows:

$$R1 :: \neg(\ (h.j = P.j) \lor (h.j = j) \lor (\exists k : (P.k = j) \land (h.j = k))\ )$$
$$\longrightarrow h.j := j \mid h.j := P.j \mid h.j := \{\text{child of } j\};$$

$$R2 :: \neg(\ (P.j \neq j) \Rightarrow (h.j = P.j) \vee (h.(P.j) = j)\ )$$
$$\longrightarrow h.j := P.j \mid h.(P.j) := j;$$
$$R3 :: \neg(\ (P.j \neq j) \Rightarrow \neg((h.j = P.j) \wedge (h.(P.j) = j))\ )$$
$$\longrightarrow h.j := j \mid h.(P.j) := P.j \mid h.(P.j) := P.(P.j);$$

Analysis of experimental results. Table 5.1 shows the results of synthesizing the Stabilizing Mutual Exclusion program with various numbers of processes organized in *linear* topology. It shows the time needed, in seconds, to add recovery, validate the recovery transitions (against pre-satisfied constraints), and the total revision time in terms of the number of processes being revised. Table 5.2 shows the result of a similar case study where the processes are arranged in a *binarytree* topology.

No. of Processes	*Time(s)*		
	constraint satisfaction		**total**
	Recovery	Validation	
30	19	21	40
40	78	74	153
50	217	238	457
60	505	509	1020
70	1110	1103	2238

Table 5.1: Stabilizing Mutual Exclusion, linear topology.

Table 5.2 illustrates that given the same state space, the complexity is higher in the tree topology than the linear topology. This is due to the following reason: the constraints of a process compare its variables with that of its neighbors. To model this effectively, the process variables and the variables of its neighbors need to be close to each other in the MDD variable ordering. This can be achieved easily on a linear topology. However, for a tree topology, this is not possible for all the processes. Hence, computing recovery transitions for those cases is more expensive.

No. of Processes	*Time(s)*		
	constraint satisfaction		**total**
	Recovery	Validation	
7	< 1	< 1	< 1
15	2	< 1	< 3
17	3	< 1	< 4
21	3	5	10
31	30	19	49

Table 5.2: Stabilizing Mutual Exclusion, binary tree topology.

Table 5.3 shows the results of using parallelism during constraints satisfaction in synthesizing the stabilizing Mutual Exclusion program. The table illustrates the results for various numbers of processes organized in linear topology using different numbers of processors/cores. It shows the time needed, in seconds, to satisfy the constraints, and the total revision time. It also shows the amount of memory in megabytes. As we can see from this table, using parallelism has substantially reduced the time needed for the revision. As a concrete example, observe that the time required to synthesize a stable mutual exclusion program with 50 processes dropped from 457 seconds, using the sequential algorithm, to 374 seconds when two cores were used, and to 178 seconds when four cores were used.

Table 5.4 shows the results of exploiting the distributed nature of the program being revised (i.e., *Group* parallelism) in synthesizing the stabilizing Mutual Exclusion program. It shows the time needed, in seconds, to compute the *group*, and the total revision time. It also shows the amount of memory in megabytes needed by our algorithm.

We can clearly see the feasibility of adding stabilizing fault-tolerance using automated revision. Both time and space complexity are reasonable and proportional to the reachable state space. Furthermore, as specified in Section 5.7, the complexity for a larger number of processes can be reduced by utilizing the hierarchal structure.

		Sequential			*2 threads*			*4 threads*			*8 threads*		
No. of Processes	**reachable states**	**Cnst t(s)**	**Syn t(s)**	**Mem (MB)**	**Cnst t(s)**	**Syn t(s)**	**Mem (MB)**	**Cnst t(s)**	**Syn t(s)**	**Mem (MB)**	**Cnst t(s)**	**Syn t(s)**	**Mem (MB)**
20	10^{26}	6	7	6	5	5	23	3	3	33	3	3	41
30	10^{44}	40	40	13	36	36	42	19	20	66	19	19	91
40	10^{64}	152	153	14	100	101	42	72	73	71	74	75	120
50	10^{84}	455	457	15	374	374	46	186	187	77	198	199	132
60	10^{106}	1014	1020	16	752	753	51	515	515	82	418	419	137
70	10^{129}	2213	2238	17	1673	1674	74	1060	1062	114	893	896	190

Table 5.3: Stabilizing Mutual Exclusion using *Constraints* partitioning. **Cnst t(s)** : Total time spent in constraints satisfaction in seconds. **Syn t(s)**: Total revision time in seconds. **Mem (MB)**: Memory usage in MB.

No. of	**reachable**	*Sequential*			*2 threads*			*4 threads*			*8 threads*		
Processes	**states**	**Grp t(s)**	**Syn t(s)**	**Mem (MB)**	**Grp t(s)**	**Syn t(s)**	**Mem (MB)**	**Grp t(s)**	**Syn t(s)**	**Mem (MB)**	**Grp t(s)**	**Syn t(s)**	**Mem (MB)**
20	10^{26}	6	7	6	4	4	16	3	3	25	3	3	42
30	10^{44}	40	40	13	36	37	32	20	20	50	16	16	83
40	10^{64}	152	153	14	105	106	40	77	78	68	54	56	122
50	10^{84}	455	457	15	320	322	44	224	226	74	150	154	133
60	10^{106}	1014	1020	16	679	686	45	547	553	71	370	376	125
70	10^{129}	2213	2238	17	2110	2135	47	1090	1116	77	700	725	136

Table 5.4: Stabilizing Mutual Exclusion using *Group* threading. **Grp t(s)** : Total time spent in *Group* computation in seconds. **Syn t(s)**: Total revision time in seconds. **Mem (MB)**: Memory usage in MB.

5.5.2 Case Study 2: Data Dissemination in Sensor Networks

In this problem, a base station initiates a computation in which a block of data is to be sent to all sensors in the network. The data message is split into fixed size packets. Each packet is given a sequence number. The base station starts transmitting the packets to its neighbor(s) in specified time slots, in the order of the packet sequence number. Subsequently, when the neighbor(s) receive a message, they, in turn, retransmit it to their neighbors and so on. The computation ends when all sensors in the network receive all the messages.

Our goal in this case study is to synthesize a nonmasking fault-tolerant version of the data dissemination program that can tolerate a finite number of lost packets. The revised program is the same as Infuse [104] that is designed manually.

Fault-Intolerant Program. In this case study, we arrange the processes in a linear topology. The base station has N packets to send to M processes. (We note that similar revision is possible for any other fixed topology.) The Fault-intolerant program transmits the packets in a simple pipeline. For this, each process keeps track of the messages (received/sent) using two variables $u.j$ and $l.j$, where $u.j$ is the highest message sequence number received by process j, and $l.j$ is the sequence number of the message currently being transmitted by process j. Process j increments $u.j$ every time it receives a new message. It also sets $l.j$ to be the sequence number of the message it is transmitting. The base station transmits a packet if its neighbor has received the previous packet (action $IN1$). A process j, $j > 0$, receives a packet from its predecessor if its successor had received the previous packet (actions $IN2$ and $IN3$). Thus, the actions of the fault-intolerant program are as follows:

Action for base station:

$(IN1)$ $(L0 = U1) \longrightarrow L0 := L0 + 1;$

Action for process $j \in \{1..M\text{-}1\}$:

$(IN2)$ $(U.j \leq U.(j+1)) \wedge (U.j \leq U.(j-1)) \wedge (L.(j-1) = U.j+1)$

$\longrightarrow U.j, L.j := U.j+1, L.j+1;$

Action for process M (the last process):

$(IN3)$ $U.M \leq U.(M-1) \wedge L.(M-1) = U.M+1 \longrightarrow U.M, L.M := U.M+1, L.M+1;$

Faults. In this section, we consider faults that lose a message. To model such faults for the base station, we add action $(F1)$, where the base station increments $L.0$, even though its successor has not received the previous packet. To model such action for other processes, we add action $(F2)$, where a process advances $L.j$, even though the successor has not yet received the previous packet.

$(F1)$ $true \longrightarrow L0 := L0+1;$

$(F2)$ $(U.j \leq U.(j-1)) \wedge (L.(j-1) = U.(j+1)) \longrightarrow U.j, L.j := U.j+1, L.j+1;$

Constraints. The constraints that define the legitimate states in the case of the data dissemination program are as follows. The first constraint states that initially the base station has all the packets $(S1)$. A process cannot receive a packet if its predecessor has not received it $(S2)$, and cannot transmit a packet that it does not have $(S3)$. A process transmits a packet that is expected by its successor ($S4$ and $S5$).

$(S1)$ $(U.0 = N)$

$(S2)$ $(\forall j : 0 < j \leq M : (U.j = U.(j-1)))$

$(S3)$ $(\forall j : 0 \leq j \leq M : (L.j \leq U.j)))$

$(S4)$ $(L.0 \leq U.1+1)$

$(S5)$ $(\forall j : 0 < j \leq (M-1) : (L.j \leq U.(j-1)+1) \wedge (L.j \leq U.(j+1)+1)))$

The data dissemination program has a set of constraints imposed by the model. More specifically, these constraints identify the transitions that the revised algorithm is not allowed to use as recovery transitions. Notice that Algorithm 10 is slightly modified to consider such transitions; these transitions are removed from *temp* right before Step 4. This set is specified by predicates imposed on the current and the next state. In particular, the model requires that the reception of a packet cannot be reversed ($MT1$), packets can only be received in sequence ($MT2$), a process can only receive one packet at a time, it can only receive a packet sent by its predecessor ($MT3$ and $MT4$), a process cannot transmit a packet unless it has received it ($MT5$), and a process should not transmit a packet unless it is potentially needed by its successor ($MT6$). Thus, the set of transitions disallowed by the model are as follows:

$MT1: \ (\exists j : 0 < j \leq M : U.j' < U.j)$

$MT2: \ (\exists j : 0 < j \leq M : U.j' < (U.j) + 1)$

$MT3: \ (\exists j : 0 < j \leq M : (U.j' = (U.j) + 1) \wedge (U.j' \neq L.(j-1)) \wedge (U.j' \neq L.(j+1))$

$MT4: \ (U.M' = (U.M) + 1 \wedge U.M' \neq L.(M-1))$

$MT5: \ (\exists j : 0 \leq j \leq M : (U.j' < L.j'))$

$MT6: \ (\exists j : 0 \leq j \leq M-1 : (L.j > U.(j+1) + 1) \wedge (L.j' < U.(j+1) + 1)$

Fault-Tolerant program. Using the program actions ($IN1$-$IN3$) for each process, the faults ($F1$-$F2$), the constraints ($S1$-$S5$), and prohibited transitions ($MT1$-$MT6$), the output was a nonmasking fault-tolerant program with the following recovery actions added to it.

$(R1) \ (U.j > U.(j+1)) \ \wedge \ (L.j > U.(j+1) + 1) \ \wedge \ (U.j + 1 = L.(j-1))$

$\quad \longrightarrow U.j := \ L.(j-1), L.j := \ U.(j+1) + 1;$

$(R2) \ (U.j > U.(j+1) + 1) \ \wedge \ (L.j > U.(j+1) + 1)$

$\quad \longrightarrow L.j \ := \ U.(j+1) + 1;$

Table 5.5 shows the results of synthesizing the data dissemination protocol with a various numbers of processes. One can notice that most of the total revision time was spent on adding recovery, while a smaller amount of time was spent in validating the recovery transitions.

The main reason for this behavior is that the structure of the fault-span in this case study is simpler: if a message is lost on one link, then until it is recovered, that message cannot be sent again (it is possibly lost on subsequent links).

No. of Processes	*Space*		*Time(s)*		
	reachable states	**memory (MB)**	**constraint satisfaction**		**total**
			Recovery	Validation	
50	10^{25}	11	4	2	6
100	10^{59}	12	32	14	48
150	10^{70}	15	153	47	207
200	10^{93}	16	452	162	633

Table 5.5: Nonmasking with linear topology data dissemination program.

Table 5.6 shows the results of synthesizing the data dissemination protocol with various numbers of processes by partitioning the constraints among available threads. Note that, in the case of the data dissemination problem, there were only 5 constraints to satisfy. Hence, when the revision is launched with 8 threads, we are only utilizing 5 of them. As can be seen from Table 5.6 if the number of constraints is not large enough then the speedup gained from portioning the constraints is limited.

Table 5.7 shows the results of synthesizing the data dissemination protocol with various numbers of processes by exploiting the distributed nature of this program.

5.5.3 Case Study 3: Stabilizing Diffusing Computation

In distributed systems, diffusing computation is used to inquire about (e.g., termination detection) or establish (e.g., distributed reset) a system global state. We consider a diffusing computation on a system where processes are arranged in a logical tree. The root initiates a diffusing computation and propagates it to its children and the children forward it to their children and so on until it reaches all processes. Once the computation reaches a leaf, it marks the leaf as *completed* and reflects back to the parent. When all children of a process are marked *completed*, that process marks itself *completed* and reflects the computation to its parent. The diffusing computation ends when it marks the root as *completed*.

No. of	**reachable**	*Sequential*			*2 threads*			*4 threads*			*8 threads*		
Processes	**states**	**Cnst t(s)**	**Syn t(s)**	**Mem (MB)**	**Cnst t(s)**	**Syn t(s)**	**Mem (MB)**	**Cnst t(s)**	**Syn t(s)**	**Mem (MB)**	**Cnst t(s)**	**Syn t(s)**	**Mem (MB)**
50	10^{47}	6	6	11	4	4	28	5	5	44	5	5	63
100	10^{95}	46	48	12	31	32	40	31	31	65	39	39	110
150	10^{143}	200	207	15	118	119	41	121	121	68	130	130	115
200	10^{190}	614	633	16	322	324	46	285	287	72	329	330	116

Table 5.6: Data Dissemination program using *Constraints* partitioning. **Grp t(s)** : Total time spent in *Group* computation in seconds. **Syn t(s)**: Total revision time in seconds. **Mem (MB)**: Memory usage in MB.

No. of	**reachable**	*Sequential*			*2 threads*			*4 threads*			*8 threads*		
Processes	**states**	**Grp t(s)**	**Syn t(s)**	**Mem (MB)**	**Grp t(s)**	**Syn t(s)**	**Mem (MB)**	**Grp t(s)**	**Syn t(s)**	**Mem (MB)**	**Grp t(s)**	**Syn t(s)**	**Mem (MB)**
50	10^{47}	68	6	11	4	4	27	2	3	44	2	2	68
100	10^{95}	149	48	12	27	30	39	14	16	66	9	11	119
150	10^{143}	245	207	15	87	93	42	64	70	68	32	39	121
200	10^{190}	446	633	16	233	253	46	173	192	72	87	106	127

Table 5.7: Data Dissemination program using *Group* threading. **Grp t(s)** : Total time spent in *Group* computation in seconds. **Syn t(s)**: Total revision time in seconds. **Mem (MB)**: Memory usage in MB.

Fault-Intolerant Program. The fault-intolerant program in this case study is the diffusing computation program from [13]. Each process j has two Boolean variables $c.j$ (color) and $sn.j$ (session number) and an integer variable P (the parent of j). A new diffusing computation can start if the root is colored *green* ($c.root = green$) and the session number of the root is the same as its children. To start a new diffusing computation, the root sets $c.root = red$ and flips $sn.root$. When a *green* process finds that its parent is *red*, it copies its parent color and session number. Moreover, if a process has no children or all its children switched colors from *red* to *green*, the process then switches its color to *green*. The program for the diffusing computation consists of three actions. The first action starts the diffusing computation at the root ($DC1$). The second action propagates the diffusing computation to the children ($DC2$). The third action completes the diffusing computation when all the children complete computation ($DC3$). The program actions are described below:

$DC1 :: (c.root = green) \longrightarrow c.root := red, sn.root := \neg sn.root;$

$DC2 :: c.j = green \wedge c.(P.j) = red \wedge sn.j \neq sn.(P.j) \longrightarrow c.j, sn.j = c.(P.j), sn.(P.j);$

$DC3 :: (c.j = red) \wedge (\forall k : P.k = j \Rightarrow (c.k = green \wedge sn.j = sn.k)) \longrightarrow c.j := green;$

Constraints. The first disjunction of (S1) states that j's parent has participated in a diffusing computation while j did not participate yet. The second disjunction of ($S1$) states that j and its parent are participating in a computation or they both have completed a computation.

$(S1)\ \ \forall j : (c.j = green \wedge c.(P.j) = red) \vee (c.j = c.(P.j) \wedge sn.j = sn.(P.j))$

Faults. We now consider the faults that change the values of $c.j$ and $sn.j$ to an arbitrary value. The fault actions are as follows:

$(F1)\ \ true \longrightarrow c.j := red \mid green;$

$(F2)\ \ true \longrightarrow sn.j := true \mid false;$

Fault-Tolerant Program. To construct the nonmasking fault-tolerant program of the fault-intolerant program of Diffusing Computation, we used our algorithm with program actions ($DC1 - DC3$), and the constraint ($S1$) with the fault actions ($F1, F2$) as an input. The revised program has the actions ($DC1 - DC3$) in addition to the following recovery actions:

$(R1)\ \ (c.j = red) \wedge (sn.j \neq sn.(P.j) \longrightarrow c.j := green,\ sn.j := sn.(P.j);$

$(R2)\ \ (c.(P.j) = green) \wedge (c.j = red) \longrightarrow c.j := green;$

$(R3)\ \ (c.(P.j) = c.j) \wedge (sn.j \neq sn.(P.j)) \longrightarrow sn.j := sn.(P.j);$

$(R4)\ \ (c.(P.j) = red) \wedge (c.j = red) \wedge (sn.j \neq sn.(P.j)) \longrightarrow c.j := green;$

No. of Processes	*Time(s)*		
	constraint satisfaction		**total**
	Recovery	Validation	
50	1	3	4
100	12	19	32
150	57	53	113
200	151	124	282

Table 5.8: Stabilizing Diffusing Computation, linear topology.

No. of Processes	*Time(s)*		
	constraint satisfaction		**total**
	Recovery	Validation	
15	< 1	< 1	< 1
17	1	1	2
21	1	3	25
23	2	4	6

Table 5.9: Stabilizing Diffusing Computation, binary tree topology.

Table 5.8 shows the results for synthesizing a stabilizing diffusing computation program with a various numbers of processes organized in a linear topology. Table 5.9 shows the result where the processes are arranged in a binary tree.

Table 5.10 shows the results of synthesizing the diffusing computation program with a various numbers of processes by exploiting the distributed nature of this program.

Table 5.11 shows the results of synthesizing the diffusing computation program with a various numbers of processes by partitioning the constraints among available threads.

Memory Usage. Notice that the amount of memory needed during revision is proportional to the number of threads being used. It is approximately the amount of memory used by the sequential algorithm multiplied by the number of cores being used. Clearly, this is expected since for every thread used, we create a new MDD package. We argue that using extra memory to gain a speedup is acceptable, since in the automated revision, time complexity is a far more serious barrier than space complexity.

5.6 Choosing Ordering Among Constraints

To apply Theorem 3.1, we need to identify an order among the constraints. In our case studies, we attempted several orderings and most were successful in synthesizing the non-masking and stabilizing fault-tolerant program. Hence, choosing the "right" order does not appear to be very crucial. Also, [13] identifies several heuristics that can assist in identifying the right order among constraints.

One possible approach is to consider different combinations as part of the revision algorithm. With such an approach, $O(n^2)$ combinations suffice for most examples. In particular, to identify an ordering, we can utilize an algorithm similar to insert-sort as follows: first consider only constraints C_1 and C_2 and attempt both orderings between them. If both orderings fail, then adding nonmasking fault-tolerance cannot be achieved using the constraint based approach that uses constraints $C1$ and $C2$. If both succeed, then we can choose any order. Without loss of generality, let the order be C_1 and C_2. Then, we consider constraint C_3 in conjunction with C_1 and C_2. There are three possible combinations to insert C_3 without affecting the order between C_1 and C_2. We can evaluate all three options and

No. of	**reachable**	*Sequential*			*2 threads*			*4 threads*			*8 threads*		
Processes	**states**	**Grp t(s)**	**Syn t(s)**	**Mem (MB)**	**Grp t(s)**	**Syn t(s)**	**Mem (MB)**	**Grp t(s)**	**Syn t(s)**	**Mem (MB)**	**Grp t(s)**	**Syn t(s)**	**Mem (MB)**
50	10^{30}	4	4	6	3	3	15	2	2	22	1	2	40
100	10^{60}	31	32	14	24	24	37	19	20	58	13	14	95
150	10^{90}	110	113	15	74	76	39	63	67	65	46	49	118
200	10^{120}	275	282	15	177	184	40	138	144	66	117	124	119

Table 5.10: Stabilizing Diffusing Computation program using *Group* threading. **Grp t(s)** : Total time spent in *Group* computation in seconds. **Syn t(s)**: Total revision time in seconds. **Mem (MB)**: Memory usage in MB.

No. of	**reachable**	*Sequential*			*2 threads*			*4 threads*			*8 threads*		
Processes	**states**	**Cnst t(s)**	**Syn t(s)**	**Mem (MB)**	**Cnst t(s)**	**Syn t(s)**	**Mem (MB)**	**Cnst t(s)**	**Syn t(s)**	**Mem (MB)**	**Cnst t(s)**	**Syn t(s)**	**Mem (MB)**
50	10^{30}	4	4	6	3	3	17	2	2	27	2	2	34
100	10^{60}	31	32	14	22	23	40	15	15	62	15	15	98
150	10^{90}	110	113	15	78	78	40	49	49	66	47	47	113
200	10^{120}	275	282	15	186	186	41	116	116	67	97	98	114

Table 5.11: Stabilizing Diffusing Computation using *Constraints* partitioning. **Cnst t(s)** : Total time spent in constraints satisfaction in seconds. **Syn t(s)**: Total revision time in seconds. **Mem (MB)**: Memory usage in MB.

then consider C_4 and so on. It follows that the number of such runs will be $O(n^2)$. In all the case studies in this chapter as well as several other algorithms in the literature, the above approach would succeed in identifying the right order of constraints. It follows that one does not need to consider all possible $(n!)$ orderings among the constraints.

Another approach is to allow the revision algorithm to chooses a random ordering for satisfying the constraints. If the revision algorithm fails to find a solution using a given constraints ordering, then it choses a different random order. The revision algorithm keeps trying different random ordering for the constraints until it finds a solution or it exhausts all possible combinations.

We implemented this approach. We found that depending on the program being revised the time required to complete the revision may vary significantly. More specifically, in the case of the Stabilizing Mutual Exclusion from Section 5.5.1, the order of the constraints is almost always irrelevant and the revision algorithm found a solution using any order it tried. Table 5.12 shows the results of 10 experiments. In each experiment, the revision algorithm randomly chose an order for the constraints and tried to synthesize using that order. In all cases the revision completed successfully for any order and from the first try. The time needed to complete the revision was almost identical to that of the case where the constraints were manually ordered (c.f. Table 5.1). However, this was not always the case. For example, Table 5.13 shows the results of synthesizing the Stabilizing Diffusing Computation from Section 5.5.3. In this case, the order in which the revision algorithm satisfies the constraints is significantly important. More specifically, the revision algorithm has to try different orderings (on average 3-4 times) before it successfully synthesizes the stabilizing fault-tolerant program. Moreover, the time required to complete the revision, in this case, was much higher than that when the constraints were manually ordered (c.f. Table 5.8).

	Total Revision Time(s)										
No. Of Processes	**Exp. 1**	**Exp. 2**	**Exp. 3**	**Exp. 4**	**Exp. 5**	**Exp. 6**	**Exp. 7**	**Exp. 8**	**Exp. 9**	**Exp. 10**	**Average**
30	51	54	51	58	58	52	51	51	53	51	51
40	213	197	198	184	184	188	184	226	189	190	181
50	552	557	574	638	571	574	556	537	541	536	517
60	1230	1250	1410	1233	1249	1231	1235	1262	1232	1246	1149
70	2877	2767	2726	2758	3062	2754	2751	2781	2711	2905	2560

Table 5.12: Stabilizing Mutual Exclusion with linear topology using *random* constraints satisfaction.

	Total Revision Time(s)										
No. Of Processes	**Exp. 1**	**Exp. 2**	**Exp. 3**	**Exp. 4**	**Exp. 5**	**Exp. 6**	**Exp. 7**	**Exp. 8**	**Exp. 9**	**Exp. 10**	**Average**
50	20	26	26	14	20	31	34	24	15	31	26
100	289	244	218	164	203	218	243	174	208	283	213
150	859	997	569	716	853	725	567	764	611	853	697
200	3280	1457	2181	2187	2218	2217	2196	2209	1451	1826	1947

Table 5.13: Stabilizing Diffusing Computation with linear topology using *random* constraints satisfaction.

5.7 Reducing the Complexity with Hierarchical Structure

Based on the case studies, we can observe that as the number of nodes in the hierarchy increases, the time complexity can increase substantially. For example, in the first case study, when we increased the height of the binary tree from 3 to 4 (i.e., from 7 to 15 processes), the revision time increased from 5 to 72 seconds. This is expected since the state space increases from 10^5 to 10^{16} states. Thus, a natural question in this area is whether the structure of the hierarchical system can assist in reducing the complexity. We show that the answer to this question is affirmative. For simplicity, we illustrate this in the context of the linear topology and binary tree topology.

Linear topology. Consider the case where the system is as shown in Figure 5.3.a. Let the constraints used during revision be $\forall j :: C_j$, where the quantification is over the set of all processes in the system. Let C_j be a constraint that depends on the variables of process j, $j-1$ (if it exists) and $j+1$ (if it exists). Furthermore, assume that constraints for intermediate processes are identical except for the renaming variables. Let the order of predicates added for system in Figure 5.3.a be C_A, C_B, C_D. Furthermore, let the added recovery actions be rec_A, rec_B, rec_D.

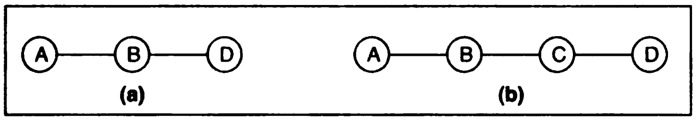

Figure 5.3: Complexity and hierarchy for linear topology

Theorem 5.7.1 *If $(rec_A \vee rec_B \vee rec_D)$ form the recovery actions for the program in Figure 5.3.a then $(rec_A \vee rec_B \vee rec'_C \vee rec'_D)$ form the recovery actions for the program in Figure 5.3.b where rec'_C is obtained by replacing B by C and (then) replacing A by B from rec_B and rec'_D is obtained by replacing B by C in rec_D.* ∎

Proof. Based on the order of constraints and the rules used in constructing recovery actions, constraints C_A and C_B will be satisfied even for the network in Figure 5.3.b. Since recovery actions do not execute after the corresponding constraint is satisfied, eventually, the recovery actions in rec'_C and rec'_D (and the fault-intolerant program) will execute. Since C_D only depends on the variables of D and its predecessor and they correct a predicate involving D and its predecessor, if actions in rec'_D execute then they will correct C_D. Moreover, if actions in rec'_D execute then they terminate (after satisfying C_D). Hence, given the fairness assumption, actions in rec'_C will execute. Observe that rec'_C is obtained from rec_B by replacing B by C and A by B. Furthermore, based on the definitions of the constraints, C_C is obtained from C_B by replacing B by C and A by B. Thus, rec'_C will correct C_C. Note that rec'_C can violate C'_D. However, it will be corrected again by rec'_D. ∎

Binary tree topology. Consider the case where the system is as shown in Figure 5.4.a. Let the constraints used during revision be $\forall j :: C_j$, where the quantification is over the set of all processes in the system. Let C_j be a constraint that depends on the variables of process j, j's parent (if it exists) and j' children (if they exist). Furthermore, assume that constraints for intermediate processes (respectively the leaves) are identical except for the renaming variables. Let the order of predicates added for system in Figure 5.4.a be $C_A, C_B, C_C, C_D, C_E, C_F, andC_G$. Furthermore, let the added recovery actions be $rec_A, rec_B, rec_C, rec_D, rec_E, rec_F, andrec_G$.

Theorem 5.7.2 *If* $(rec_A \vee rec_B \vee rec_C \vee rec_D \vee rec_E \vee rec_F \vee rec_G)$ *form the recovery actions for the program in Figure 5.4.a then* $(rec_A \vee rec_B \vee rec_C \vee rec'_D \vee rec'_E \vee rec'_F \vee rec'_G \vee rec_H \vee rec_I \vee rec_J \vee rec_K \vee rec_L \vee rec_M \vee rec_N \vee rec_O)$ *form the recovery actions for the program in Figure 5.4.b where:*

1. *rec_B is used generate rec'_D by:*

   (a) replacing D by H and E by I,

   (b) replacing B by D, and (then)

   (c) replacing A by B,

2. *rec_H is obtained by replacing D by H and (then) by replacing B by D in rec_D,*

3. rec_I is obtained by replacing D by I and (then) by replacing B by D in rec_D;

rec'_E, rec'_F, rec'_G, rec_J, rec_K, rec_L, rec_M, rec_N, and rec_O are generated by using steps similar to the steps 1-3. ∎

Proof.

The proof of Theorem 5.7.2 is similar to that of Theorem 5.7.1∎

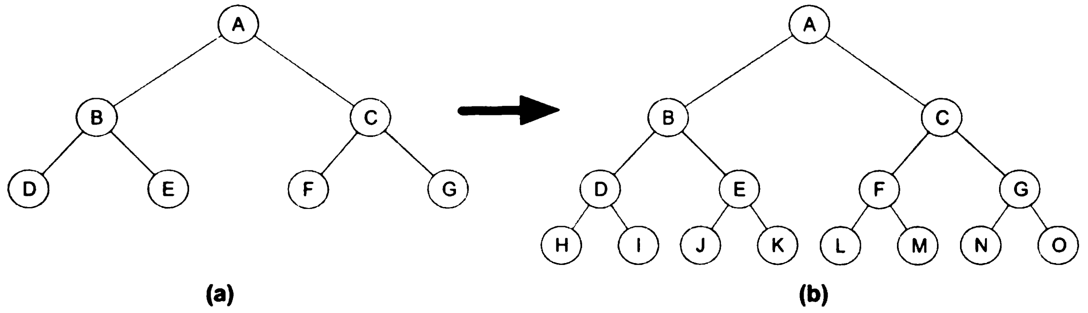

Figure 5.4: Complexity and hierarchy for the binary tree topology

While the above result is straightforward and widely understood, it is especially useful for managing complexity of hierarchical systems. While results of this form have been presented in the literature, the pre-conditions that must be satisfied to apply it are often difficult to evaluate during automated revision. However, the conditions of the above theorem are easy to evaluate and this theorem can reduce the complexity of synthesizing systems with a larger number of nodes. Clearly, constructing and verifying the recovery action which satisfy the conditions of Theorem 5.7.1 and Theorem 5.7.2 is *syntactical* and requires a minimal amount of time to complete.

5.8 Summary

In this chapter, we focused on making the automated model revision more comprehensive and covering more levels of fault-tolerance. In particular, we derived theories, developed algorithms, and built tools to automate the addition of nonmasking and stabilizing fault-tolerance. Our algorithm ensures that it adds recovery actions that enable the program to recover to its legitimate states from any arbitrary state. This algorithm is based on describing the legitimate states using a set of constraints. Then, it finds recovery actions that satisfy each constraint. Finally, it makes sure that the recovery actions do not interfere with each others and work collectively to reach the legitimate states.

Also, we used the multi-core technology to parallelize our algorithm to substantially reduce the revision time. We illustrated our approach with three case studies. Furthermore, we demonstrated that automated revision in these case studies was feasible and achieved in a reasonable time.

Chapter 6

Legitimate States Automated Discovery

Existing algorithms for the automated model revision require that the designers have to identify the legitimate states of original model. Experience suggests that of the inputs required for model revision, identifying such legitimate states is the most difficult and creates a burden on the use of these methods. To reduce this burden, we develop an algorithm wLspGenerator (i.e., weakest legitimate state predicate generator) for identifying the largest set of states from where the program satisfies its specification. Furthermore, we show how this algorithm can be integrated with existing algorithms for the addition of fault-tolerance. With an example, we show that a straightforward approach of using reachability analysis from initial states to compute legitimate states is not relatively complete.

The rest of the chapter is organized as follows: In Section 6.2, we present our algorithm, wLspGenerator, for computing the weakest legitimate state predicate for the given program. In Section 6.3, we demonstrate the application of this algorithm with four case studies to show that it computes the largest set of legitimate states required for model revision. Finally, we present the summary in Section 6.4.

6.1 Introduction

In automated model revision to add fault-tolerance, it is required that after the occurrence of faults, the revised program eventually recovers to the legitimate states of the original program. Since the original program met its original specification from these states, we can ascertain that eventually a revised program reaches states from where subsequent computations are correct. One of the problems in providing recovery to legitimate states, however, is that these legitimate states are not always easy to determine.

Current approaches for automated model revision for revising an existing model to add fault-tolerance include [27,30,101,111] as well as the approaches presented in Chapters (4 - 3). These approaches describe the model as an abstract program. They require the designer to specify (1) the existing abstract program that is correct in the absence of faults, (2) the program specification, (3) the faults that have to be tolerated, and (4) the program legitimate states, from where the existing program satisfies its specification. Of these four inputs, the first three are easy to identify and are unavoidable. For example, one is expected to utilize model revision only if they have an existing model that fails to satisfy a required property. Thus, if model revision is applied in the context of newly identified faults, original model and faults are already available. Likewise, specification identifies what the model was supposed to do. Clearly, requiring it is unavoidable. Identifying the legitimate states from where the fault-intolerant program satisfies its specification is, however a difficult task. Our experience in this context shows that while identifying the other three arguments is often straightforward, identifying precise legitimate states requires significant effort. It is straightforward to observe that if these legitimate states could be derived automatically, then it would reduce the burden put on the designer, thereby making it easier to apply these techniques in revision of existing programs.

One approach for identifying legitimate states is to use initial states as legitimate states. While identifying these initial states is typically easy for the designer, this approach is very limiting. A variation of this approach is to define the legitimate states to be those states that are reachable from the initial states. While less limiting, this approach fails to identify states from where the existing program is correct, although such states are not reached in fault-free execution. While the knowledge of these states is irrelevant for fault-free execution, it is potentially useful in adding fault-tolerance. In particular, if faults perturb the program to one of these states, no recovery may be needed. Furthermore, recovery could be added to these states so that subsequent computation is correct.

In this chapter, we focus on automated model revision where we begin with the specification of the original program and discover the legitimate states automatically. In particular, we focus on identifying the largest set of legitimate states from where the original fault-intolerant program satisfies its specification. Subsequently, we utilize this set of legitimate states in obtaining the fault-tolerant program that is correct by construction. (If we view a set of states as a predicate that is true only in those states, then this corresponds to the weakest state predicate.) Of course, an enumerative approach, where we consider each state as a potential initial state, is impractical. Our goal in this chapter is to identify efficient techniques for identifying the largest set of legitimate states for a given program.

Our algorithm for computing the largest set of legitimate states takes two inputs: the program (specified in terms of its transitions) and its specification. The program specifications consists of: (1) a safety specification, which is specified in terms of (bad) states that the program should not reach and (bad) transitions that the program should not execute, and (2) zero or more liveness specifications of the form $\mathcal{F}$ *leads to* $\mathcal{T}$ (written as $\mathcal{F} \leadsto \mathcal{T}$), which states that if the program ever reaches a state where $\mathcal{F}$ is true then in its subsequent computation it reaches a state where $\mathcal{T}$ is true.

In this chapter, we present the algorithm wLspGenerator for identifying the set of legitimate states with respect to the given program and specification. We show that our algorithm for finding the largest set of legitimate states is sound. With a BDD based implementation, we show that our algorithm manages the state explosion problem. We illustrate our algorithm in the context of four case studies: the Byzantine agreement program [108], the token ring program [30], the Stabilizing Tree Based Mutual Exclusion problem based on the fault-intolerant version by Raymond [124], and the Stabilizing Diffusing Computation [13]. The set of legitimate states computed in these examples are identical to those in Chapters (3 -5) and in [30,102]. In particular, the sets of legitimate states computed in this paper for mutual exclusion is used in [15] for adding nonmasking fault-tolerance. It follows that by combining our algorithm with that in [102] for adding fault-tolerance, it would be possible to permit the revision to add fault-tolerance without requiring the designer to specify the legitimate states explicitly.

6.2 The "Weakest Legitimate State Predicate Generator (wLspGenerator)" Algorithm

In this section, we present our algorithm to automatically generate the largest set of legitimate states using the program transitions and its specification. The goal of our algorithm is to generate the largest set of legitimate states (i.e., weakest legitimate state predicate) from where the program satisfies its safety and liveness specification. Our algorithm consists of three main parts: the legitimate states generator, the safety checker, and the liveness checker. We will describe each of the three algorithms in subsections 6.2.1-6.2.3.

We use a symbolic representation in terms of Boolean formulas since we implemented this algorithm using Ordered Binary Decision Diagrams (OBDD) [34].

Algorithm sketch. Intuitively, our algorithm consists of two main steps. The first step is to generate the initial set of legitimate states from the program transitions and safety specifications. In this step, we identify the initial set of legitimate states, say I, to be all the states in the state space excluding the set of bad states, $SPEC_{bs}$ (the states that should not be reached). Then we proceed to ensure that I does not include any state that violates safety. The second step is to ensure that I suffices the liveness properties. To verify a specific liveness property, say $X \leadsto Y$, the algorithm needs to ensure that all program transitions paths from all states in X reach to Y. Furthermore, all paths should be cycles-free. If such cycles exist, then all Y states in X that leads to the cycles are removed from I.

We now describe our algorithm in detail: First, we describe the algorithm wLspGenerator, which computes the largest set of legitimate states, say I, that satisfy the program specifications. Then, we proceed to describe the algorithm SafetyChecker that computes the set of state in which the program does not violate the safety property. Finally, we describe the algorithm LivenessChecker that removes any state that may violate the liveness of the program from the set of legitimate states, I.

6.2.1 Weakest Legitimate State Predicate Generator

The input to wLspGenerator consists of the program transitions, $SPEC_{bs}$ (the states that should not be reached), $SPEC_{bt}$ (the transitions that should not be executed), and the liveness properties. The algorithm returns the largest set of legitimate states from where the program satisfies its specification. First, it initializes the legitimate states I to be the whole state space (Line 1). Then, the algorithm computes the largest set of legitimate states by calling the function SafetyChecker (Line 4). At this point, I includes the set of states from where the program satisfies the given safety specification. Later, the algorithm satisfies the liveness properties one after another by calling the function LivenessChecker that removes states that violate the given liveness property (Lines 5-7). Removal of states due to liveness properties may require re-computation of I. Hence, this computation is in a loop and terminates when a fixpoint is reached.

6.2.2 Safety Checker

The input of the SafetyChecker algorithm consists of the initial set of legitimate states, the program transitions, the $SPEC_{bs}$, and the $SPEC_{bt}$. The output is the computed largest set of legitimate states, I_{sf}, for the given safety specification.

First, the algorithm initializes the set of legitimate states I_{sf} to be I_{inp} excluding the

Algorithm 12 WeakestLegitimateStatePredicateGenerator (wLspGenerator)

```
Input: program transitions $p$, $SPEC_{bs}$ (states that should not be reached),
    $SPEC_{bt}$ (transitions that should not be executed), $\mathcal{F}$ [], and $\mathcal{T}$ [] state
    predicates describing leads-to properties .
Output: weakest legitimate state predicate $I_w$.

   // Initially $I_w$ equals $S_p$, the program states space.
1: $I_w = S_p$
2: repeat
3:     $tmp = I_w$
4:     $I_w$:= SafetyChecker($I_w, p, SPEC_{bs}, SPEC_{bt}$);
       //check the $i^{th}$ liveness properties
5:     for $i := 0$ to $NoOfLivenessProperties$ do
6:         $I_w$:= LivenessChecker($I_w, p, \mathcal{F}[i], \mathcal{T}[i]$);
7:     end for
8: until $tmp = I_w$
   // return the largest set of legitimate states.
9: return $I_w$;
```

states in $SPEC_{bs}$ (Line 1). Then, the algorithm starts a fixpoint computation that removes undesired states from I_{inp}. If I_{sf} contains a state s_0 such that the program can execute the transition (s_0, s_1), which violates safety, then s_0 cannot be in I_{sf}. Hence, we remove s_0 from I_{sf} (Line 4). Note that a state is removed from I_{sf} only if the given program violates safety from that state. If I_{sf} contains a state s_0, then p contains a transition (s_0, s_1), and s_1 has been removed from I_{sf}, then s_0 must also be removed from I_{sf} (Line 5). This process continues until a fixpoint is reached. At this point, it exits the loop and returns the desired set of legitimate states I_{sf}.

6.2.3 Liveness Checker

The input of the LivenessChecker algorithm consists of the initial set of legitimate states, I_{inp}, the program transitions, the $\mathcal{F}$ and $\mathcal{T}$ where $\mathcal{F} \leadsto \mathcal{T}$ is a given state predicate describing leads-to properties. The output is the largest set of states that is a subset of I_{inp} from

Algorithm 13 SafetyChecker

Input: initial legitimate states I_{inp}, program transitions p, $SPEC_{bs}$ (states that should not be reached), $SPEC_{bt}$ (transitions that should not be executed).

Output: weakest legitimate state predicate I_{sf}.

```
    // $S_P$ is the state space of $p$
1: $I_{sf} := I_{inp} - SPEC_{bs}$;

2: repeat
3:     $tmpI := I_{sf}$;
4:     $I_{sf} := I_{sf} - \{s_0 : (s_0, s_1) \in p \cap SPEC_{bt} \}$;
5:     $I_{sf} := I_{sf} - \{s_0 : (s_0, s_1) \in p \wedge s_0 \in I_{sf} \wedge s_1 \notin I_{sf}\}$;
6: until $tmpI = I_{sf}$
    // return the set of states from where the program satisfies safety properties.
7: return $I_{sf}$;
```

where the given program satisfies $\mathcal{F} \rightsquigarrow \mathcal{T}$.

First, the algorithm creates a program *tmpP* where we add a self-loop to all the deadlock states where the program p has no outgoing transitions from s_0 and $s_0 \notin \mathcal{T}$ (Line 1). All computations of *tmpP* are infinite or terminate in a state in $\mathcal{T}$. Now we remove all transitions in *tmpP* that reach $\mathcal{T}$ (Line 2). If p satisfies ($\mathcal{F} \rightsquigarrow \mathcal{T}$), then it follows that *tmpP* cannot include any infinite computation that includes a state in $\mathcal{F}$. Hence, the algorithm iteratively removes deadlock states in *tmpP* (Lines 5-7). If some states in $\mathcal{F}$ still remain, then it implies that there are infinite computations of *tmpP* that begin in a state in $\mathcal{F}$ but do not reach a state in $\mathcal{T}$. We remove such states from I_{inp} and iteratively compute I_{inp}.

Extension. In some cases, the program actions are partitioned in terms of *system actions* and *environment actions*. It is expected that the environment actions will eventually stop (for a long enough time) so that the system actions can make progress (and satisfy liveness property). In such cases, we can apply the above algorithm as follows: The program

Algorithm 14 LivenessChecker

Input: initial legitimate states I_{inp}, program transitions p, $\mathcal{F}$, and $\mathcal{T}$ state predicates describing leads-to properties.

Output: weakest legitimate state predicate I_{inp}.

```
    // ASSUMPTION: $\mathcal{F} \cap \mathcal{T} = \{\}$. If not, change $\mathcal{F}$ to $(\mathcal{F} - \mathcal{T})$.

    // let $ds(p) = \{s_0 : \forall s_1, (s_0, s_1) \notin p\}$ be the set of deadlock states.
    // add self-loop to the states in $ds(p)$.
 1: $tmpP := p \cup \{(s_0, s_0) : s_0 \notin \mathcal{T} \wedge s_0 \in ds(p)\}$;
 2: $tmpP := \{(s_0, s_1) : (s_0, s_1) \in tmpP \wedge s_1 \notin \mathcal{T}\}$;

 3: repeat
 4:   $invF := I_{inp}$ ;
 5:   while $(invF \cap ds(tmpP)) \neq \{\}$ do
 6:     $invF := invF - ds(tmpP)$;
 7:   end while
 8:   if $\mathcal{F} \cap invF \neq \{\}$ then
 9:     $I_{inp} := I_{inp} - (\mathcal{F} \cap invF)$;
10:   end if
11: until $\mathcal{F} \cap invF = \{\}$
    // return the set of states from where the program satisfies liveness properties.
12: return $I_{inp}$;
```

actions used in SafetyChecker will consist of both the system actions and the environment actions. The program actions used for LivenessChecker will consist of only the system actions.

Theorem 6.2.1 *The Algorithm* wLspGenerator *is sound (i.e., the generated set of legitimate states is the largest set of legitimate states).*

Proof. The proof consists of two parts: (1) if state, say s_0, is not included in the output of wLspGenerator, then the program does not satisfy its specification from s_0, and (2) if a state, say s_0, is included in the output of wLspGenerator, then the program satisfies its specification from s_0.

We now prove the first part by considering all parts of the code where some state is removed from the output.

- Line 1 of SafetyChecker: Clearly, states in $SPEC_{bs}$ cannot be included in the final set of legitimate states.
- Line 4 of SafetyChecker: If (s_0, s_1) is a transition of the program that violates safety then there is a computation of the program that starts from s_0 and violates the specification.
- Line 5 of SafetyChecker: If s_1 is a state already removed from the final set of legitimate states, i.e., there is a program computation that starts from s_1 and violates the specification, and (s_0, s_1) is a program transition, then there exists a computation that starts from s_0 and violates the specification.
- Line 9 of LivenessChecker: Observe that in *tmpP*, transitions that reach $\mathcal{T}$ are removed. Now, the loop on Lines 5-7 removes all deadlock states in *invF*. If any state, say s_0, in $\mathcal{F}$ is not removed, then that implies that there are infinite computations of *tmpP* that start from s_0. For instance, this happens if a cycle is reachable from s_0. By construction, this computation cannot reach $\mathcal{T}$. Thus, if a state s_0 is removed

on Line 9 of LivenessChecker, then there is a computation from s_0 that violates the specification.

We use proof by contradiction for the second part. Suppose s_0 is included in the output of wLspGenerator and there is a computation, say $\langle s_0, s_1, \ldots \rangle$ that violates the specification from s_0. We consider two cases depending upon whether this computation violates the safety specification or the liveness specification.

- **Safety specification.** Consider the first state where safety violation is detected, e.g., because a state, say s_j, in $SPEC_{bs}$ is reached or a transition, say (s_{j-1}, s_j) in $SPEC_{bt}$ is executed.

  - Case 1: $s_j \in SPEC_{bs}$. By Line 1 of SafetyChecker, $j \neq 0$. Also, from Line 5 of SafetyChecker, s_{j-1} would be removed from the final set of legitimate states. Likewise, s_{j-2} would be removed and so on. Thus, s_0 cannot be in the output of wLspGenerator. This is a contradiction.

  - Case 2: $(s_{j-1}, s_j) \in SPEC_{bt}$. By the same argument as in Case 1, we can show that s_0 cannot be in the output of wLspGenerator. This is a contradiction.

- **Liveness specification.** If this computation does not satisfy the liveness specification then this implies that it has a suffix where $\mathcal{F}$ is true in some state, say s_j, but $\mathcal{T}$ is false in all states. Now, we define a computation σ that starts from s_j. If the computation $\langle s_0, s_1, \ldots \rangle$ is infinite then σ is the suffix that starts from s_j. If the computation $\langle s_0, s_1, \ldots \rangle$ is not infinite, it ends in a state say, s_l, where p has no outgoing transitions, then σ is obtained by concatenating the suffix starting from s_j and an infinite stuttering of state s_l. By construction, σ is also a computation of $tmpP$ (Line 2 from LivenessChecker). Thus, s_j is removed from the output of wLspGenerator. Again, by an argument similar to the case of safety specification, we can conclude that s_0 cannot be in the output of wLspGenerator. This is a contradiction. ∎

6.3 Application of wLspGenerator in Automated Model Revision

In this section, we describe and analyze our approach for generating the legitimate states of the four case studies: the Byzantine agreement program [108], the token ring program [30], the Stabilizing Tree Based Mutual Exclusion problem based on the fault-intolerant version by Raymond [124], and the Stabilizing Diffusing Computation [13]. We chose these classical examples from the literature of distributed computing to illustrate the feasibility and applicability of our algorithm in generating the weakest legitimate state predicate. Furthermore, these case studies illustrate that the overhead of computing the legitimate states using wLspGenerator is very small compared to the overall time required for the addition of fault-tolerance. Thus, reducing the burden of the designer in terms of requiring the explicit legitimate states increases the complexity by a very small factor.

Throughout this section, all case studies are run on a MacBook Pro with 2.6 Ghz Intel Core 2 Duo processor and 4 GB RAM. The OBDD representation of the Boolean formula has been done using the C++ interface to the CUDD package developed at the University of Colorado [125].

6.3.1 Case Study 1: Byzantine agreement program

We illustrate our algorithm in the context of the Byzantine agreement program from Section 4.3.3. We start by specifying the fault-intolerant program. Then, we provide the program specification. Finally, we describe the weakest legitimate state predicate generated by our algorithm.

Program. The Byzantine agreement program consists of a "general" and three or more non-general processes. Each process copies the decision of the general and finalizes (outputs) that decision.

Recall from Section 4.3.3, the actions of the Byzantine agreement program are as shown in action below. The only difference is in the third and fourth actions that allow a Byzantine process to change its decision and finalized status. The last two actions are environment actions.

$1 :: (d.j = \bot) \wedge (f.j = \mathit{false}) \longrightarrow d.j := d.g;$

$2 :: (d.j \neq \bot) \wedge (f.j = \mathit{false}) \longrightarrow f.j := \mathit{true};$

$3 :: (b.j) \longrightarrow d.j := 1|0,\ f.j := \mathit{false}|\mathit{true};$

$4 :: (b.g) \longrightarrow d.g := 1|0;$

Where $j \in \{1,n\}$ and n is the number of non-general processes.

Specification. The safety specification of the Byzantine agreement requires *validity* and *agreement*:

- *Validity* requires that if the general is non-Byzantine, then the final decision of a non-Byzantine process must be the same as that of the general. Thus, $validity(j)$ is defined as follows.

  $validity(j) = ((\neg b.j \wedge \neg b.g \wedge f.j) \Rightarrow (d.j = d.g))$

- *Agreement* means that the final decision of any two non-Byzantine processes must be equal. Thus, $agreement(j,k)$ is defined as follows.

  $$agreement(j,k) = ((\neg b.j \wedge \neg b.k \wedge f.j \wedge f.k) \\ \Rightarrow (d.j = d.k))$$

- The final decision of a process must be either 0 or 1. Thus, $final(j)$ is defined as follows.

  $final(j) = f.j \Rightarrow (d.j = 0 \vee d.j = 1)$

We formally identify safety specification of the Byzantine agreement in the following set of bad states:

$$SPEC_{BA_{bs}} = (\ \exists j,k \in \{1..n\} :: \\ (\ \neg(validity(j) \wedge agreement(j,k) \wedge final(j))\ )$$

Observe that $SPEC_{BA_{bs}}$ can be easily derived based on the specification of the Byzantine Agreement problem.

The liveness specification of the Byzantine agreement requires that eventually every non-Byzantine process finalizes a decision. The requirement that process j eventually finalizes a decision can be specified as follows:

$$\neg b.j \leadsto (f.j)$$

Application of our algorithm. The weakest predicate computed (for 3 non-general processes) is as follows. If the general is non-Byzantine, then it is necessary that $d.j$, where j is also a non-Byzantine, be either $d.g$ or $\perp$. Furthermore, a non-Byzantine process cannot finalize its decision if its decision equals $\perp$. Now, we consider the set of states where the general is Byzantine. In this case, the general can change its decision arbitrarily. Also, the predicate includes states where other processes are non-Byzantine and have the same value that is different from $\perp$. Thus, the generated weakest legitimate state predicate is as follows:

$$\begin{array}{l} I_{BA} = \\ (\ \neg b.g \wedge (\ \forall p \in \{1..n\} :: ((\neg b.p \wedge f.p) \Rightarrow d.p \neq \perp) \\ \wedge\ (\neg b.p \Rightarrow (d.p = \perp \vee d.p = d.g)))\ )\ \vee \\ (\ b.g \wedge (\ \forall\ j,k \in \{1..n\} : j \neq k :: (d.j = d.k) \\ \wedge\ (d.j \neq \perp)\ )\ ) \end{array}$$

Observe that I_{BA} cannot be easily derived based on the specification of the Byzantine Agreement problem. More specifically, the set of states where the general is Byzantine, are not reachable from the initial states of the program.

We used the exact same predicate in the case study from Section 4.3.3 to add fault-tolerance to Byzantine faults. (In [30], where we reported the results for addition of fault-tolerance with symbolic techniques, the set of legitimate states used was a conjunction of the above predicate and a formula that states that at most one process is Byzantine. However, this extra formula does not affect the revised program or the time complexity.)

The amount of time required for computing this set of legitimate states for a different number of processes is as shown in Table 7.2. We would like to note that the set of legitimate states computed in these case studies is the same as that used in the addition of fault-tolerance.

No. of Process	**Reachable States**	**Legitimate States Generation Time(Sec)**
10	10^9	0.57
20	10^{15}	1.34
30	10^{22}	4.38
40	10^{30}	9.25
50	10^{36}	26.34
100	10^{71}	267.30

Table 6.1: The time required to generate the weakest legitimate state predicate (Byzantine Agreement).

We note that the time required to compute the set of legitimate states is very small as compared with the total time needed to complete the revision. For example, to synthesize a fault-tolerant Byzantine agreement program with 40 processes, it takes more than 9,000 seconds as shown in Section 4.3.3. By contrast, the time to compute the legitimate states is only 9.25 seconds. Thus, the overhead of synthesizing with the specification without explicit legitimate states is negligible.

We use this case study to illustrate that computing the set of legitimate states to be those that are reachable from initial states is not relatively complete. In particular, for the Byzantine agreement example, the initial state is one where all processes are non-Byzantine and the decision of all non-general processes is equal to $\perp$. Clearly, all processes are non-Byzantine in all states reached by the program from these initial states. It follows that recovery to these reachable states is not always feasible in the presence of faults. Hence, these reachable states are insufficient to obtain the fault-tolerant program. By contrast, the weakest legitimate state predicate can be utilized to find the fault-tolerant program.

6.3.2 Case Study 2: Token Ring

In this section, we illustrate our algorithm in the context of the token ring program. First, we specify the fault-intolerant program. Then, we provide its specification. Finally, we identify the largest set of legitimate states generated by the algorithm from Section 6.2.

Program. The token ring program consists of n processes organized in a ring. A token is circulated among the processes in a fixed direction. When a process gets the token it can access the critical section. Each process j, where $j \in \{0..n\}$, has a variable $x.j$ with the domain $\{0, 1, \perp\}$, where $\perp$ denotes that the process is in an illegitimate state. A process 0 has the token iff $x.n$ is equal to $x.0$ and a process j, where $1 \leq j \leq n$, has the token iff $x.j \neq x.(j-1)$.

The actions of the token ring program are as follows:

$$\begin{array}{lll} 1 :: x.j \neq x.(j-1) & \longrightarrow & x.j := x.(j-1); \\ 2 :: x.0 = x.n & \longrightarrow & x.0 := x.n +_2 1; \end{array}$$

where $+_2$ denotes modulo 2 addition.

Specification. The safety specification of the token ring requires that the value of x at any process is either 0 or 1 and that no two processes have a token simultaneously. Thus, the safety specifications of the token ring program can be identified using the following set of bad states (i.e. states that should not be reached by normal program execution).

$$\begin{array}{l} SPEC_{TR_{bs}} = \\ \quad (\exists j,k : j \neq k \wedge j,k \in \{1..n\} :: ((x.(j-1) \neq x.j) \wedge (x.(k-1) \neq x.k))) \vee \\ \quad (\exists j : j \in \{1..n\} :: ((x.(j-1) \neq x.j) \wedge (x.0 = x.n))) \vee \\ \quad (\exists j : j \in \{0..n\} :: (x.j = \perp)) \end{array}$$

The liveness specification of the token ring requires that eventually every process gets the token. The requirement that process 0 eventually gets the token can be specified as:

$$true \leadsto (x.0 = x.n).$$

Application of our algorithm. After applying our algorithm with the above inputs, the generated largest set of legitimate states can be represented using the following regular expression:

$$\langle x.0, x.1, x.2 \ldots x.n \rangle \in (0^l 1^{(n+1-l)} \cup 1^l 0^{(n+1-l)}), \text{ where } 0 \le l \le n+1.$$

Thus, the above predicate states that the sequence of $\langle x.0, x.1, x.2 \ldots x.n \rangle$ is a sequence of zeros followed by ones or ones followed by zeros. The value of $l+1$ in the above sequence identifies the process with the token.

We note that this is the exact same set of legitimate states used in Section 4.3.3 for adding fault-tolerance to the fault where up to n processes are detectably corrupted. Furthermore, the time for computing this set of legitimate states for different values of n is as shown in Table 6.2. As we can see, its very small.

No. of Process	**Reachable States**	**Legitimate States Generation Time(Sec)**
10	10^4	0.1
20	10^9	0.2
30	10^{14}	0.3
40	10^{19}	0.4
50	10^{23}	0.6
100	10^{47}	0.19

Table 6.2: The time required to generate the weakest legitimate state predicate (token ring).

6.3.3 Case Study 3: Mutual Exclusion

In this section, we illustrate our algorithm in the context of the Raymond's tree-based mutual exclusion program from Section 6.3.3. Our goal in this case study is to automatically generate the weakest legitimate state predicate for the program in [15].

We start by specifying the fault-intolerant program. Then, we provide the program specification. Finally, we identify the weakest legitimate state predicate generated by our algorithm.

Program. Recall that the action by which k sends the token to j is as follows:

$$1 :: (h.k = k \ \wedge \ j \in Adj.k) \ \wedge \ (h.j = k) \longrightarrow h.k := j, \ \ h.j := j;$$

Where $Adj.k$ denotes one of the neighbors of k.

Specification. Since the goal of Raymond's mutual exclusion algorithm is to maintain a tree rooted at the token, it requires that the holder of any process is one of its tree neighbors. It also requires that there should be no cycles in the holder relation.

We formally describe the safety specifications in the following predicate:

$$\begin{aligned} SPEC_{ME_{bs}} &= (\ \exists j \in \{0..n\} :: ((h.j \neq j) \vee (h.j \neq p.j) \vee (h.j \neq ch.j))\ ) \vee \\ &\quad (\ \exists j,k \in \{0..n\} : j \neq k :: ((h.j = k) \wedge (h.k = j)\ ))\ \vee \\ &\quad (\ \exists j,k \in \{0..n\} : j \neq k :: ((h.j = j) \wedge (h.k = k)\ )) \end{aligned}$$

Where $ch.j$ denotes one of the children of j.

Application of our algorithm. The generated weakest legitimate state predicate of the mutual exclusion program computed by our algorithm is as follows. The legitimate states predicate require that j's holder can either be j's parent, j itself, or one of j's children. It also requires that the holder tree conforms to the parent tree and there are no cycles in the holder relation.

$$\begin{aligned} I_{ME} &= (\ \forall j \in \{0..n\} :: (h.j = P.j) \vee (h.j = j) \vee (\exists k : (P.k = j) \wedge (h.j = k))\ ) \ \wedge \\ &\quad (\ \forall j \in \{0..n\} :: (P.j \neq j) \Rightarrow (h.j = P.j) \vee (h.(P.j) = j)\ ) \ \wedge \\ &\quad (\ \forall j \in \{0..n\} :: (P.j \neq j) \Rightarrow \neg((h.j = P.j) \wedge (h.(P.j) = j))\ ) \end{aligned}$$

Where $P.j$ denote the parent of j.

Recall that I_{ME} is equivalent to the conjunction of the constraints (S_1, S_2, and S_3), from Section 6.3.3, used in deriving the the non-masking fault-tolerant version of the mutual exclusion program.

The amount of time required for computing this set of legitimate states for a different number of processes is as shown in Table 6.3.

No. of Process	**Reachable States**	**Legitimate States Generation Time(Sec)**
10	10^9	0.01
20	10^{26}	0.1
30	10^{44}	0.2
40	10^{64}	0.5
50	10^{84}	0.9
100	10^{200}	0.43

Table 6.3: The time required to generate the weakest legitimate state predicate (Mutual Exclusion).

6.3.4 Case Study 4: Diffusing Computation

In this case study, we consider a diffusing computation on a system where processes are arranged in a logical tree. The root initiates a diffusing computation and propagates it to its children and the children forward it to their children and so on until it reaches all processes. Once the computation reaches a leaf, it marks the leaf as *completed* and reflects back to the parent. When all children of a process are marked *completed*, that process marks itself *completed* and reflects the computation to its parent. The diffusing computation ends when it marks the root as *completed.*

Program. The fault-intolerant program in this case study is the diffusing computation program from [13]. Each process j has two Boolean variables $c.j$ (color) and $sn.j$ (session number) and an integer variable P (the parent of j). A new diffusing computation can start if the root is colored *green* ($c.root = green$) and the session number of the root is the same as its children. To start a new diffusing computation, the root sets *c.root* = *red* and flips *sn.root*. When a *green* process finds that its parent is *red*, it copies its parent color and session number. Moreover, if a process has no children or all its children switched colors from *red* to *green*, the process then switches its color to *green*. The program for the diffusing computation consists of three actions. The first action starts the diffusing computation at the root (1). The second action propagates the diffusing computation to the children (2). The third action completes the diffusing computation when all the children complete computation (3). The program actions are described below:

$1 :: (c.root = green) \longrightarrow c.root := red, sn.root := \neg sn.root;$

$2 :: c.j = green \wedge c.(P.j) = red \wedge sn.j \neq sn.(P.j) \longrightarrow c.j, sn.j = c.(P.j), sn.(P.j);$

$3 :: (c.j = red) \wedge (\forall k : P.k = j \Rightarrow (c.k = green \wedge sn.j = sn.k)) \longrightarrow c.j := green;$

Specification. The safety specifications for the diffusing computation program requires that all processes must have the same color and the same session number. We formally define the safety specifications in the following predicate:

$$SPEC_{DC} = ((\exists j,k \in \{0..n\} : j \neq k :: (sn.j \neq sn.k \vee c.j \neq c.k))$$

Application of our algorithm. The generated weakest legitimate state predicate of the diffusing computation is as follows : The set of legitimate states requires that all processes should have the same colors and session numbers.

$$I_{DF} = \forall j : (c.j = green \wedge c.(P.j) = red) \vee (c.j = c.(P.j) \wedge sn.j = sn.(P.j))$$

6.4 Summary

In this chapter, we provided techniques that permit the designer to efficiently describe the model to be revised. Specifically, we derived theories, developed algorithms, and built tools to automate the discovery of the legitimate states of the model. Our techniques relieve the designer from performing unnecessary steps, thereby simplifying the application of the automated model revision. Our algorithm uses the program actions and specification to automatically generate the weakest legitimate state predicate. First, it initializes weakest legitimate state predicate to be the set of the states from where the given program does not violate the safety specification. Second, it ensures that the generated weakest legitimate state predicate satisfies the liveness properties by removing any state that violates liveness properties. Also, we considered four case studies. We used our algorithm to automatically discover the set of legitimate state for each case. In each of these examples, the generated set of legitimate states was the same as the one specified explicitly in automated addition of fault-tolerance an the time to generate the legitimate states was very small when compared with that for performing the corresponding model revision.

Chapter 7

Automated Model Revision Without Explicit Legitimate States

In Chapter 6 we introduced our algorithm for the automated discovery of the legitimate state. We also showed how such automation reduces the burden put on the designer, making it easier to apply these techniques in the revision of existing programs. However, one question that we need to answer is regarding the completeness of this approach. In other words, if it were possible to perform model revision with explicit legitimate states, then is it possible to do so without the explicit identification of the legitimate states.

In this chapter, we consider the problem of automated model revision without explicit legitimate states. We show that this formulation is relatively complete, i.e., if it were possible to perform model revision with explicit legitimate states, then it is possible to do so without the explicit identification of the legitimate states.

We also identify instances where the complexity class of model revision without explicit legitimate states is the same as that with explicit legitimate states. In turn, this identifies heuristics for performing model revision without explicit legitimate states. Finally, we show that with these heuristics, the increased cost for model revision without explicit legitimate states is small.

The rest of this chapter is organized as follows: In Section 7.1, we present an alternative approach for performing model revision. In Section 7.2, we state the automated model revision problem statement. In Sections, 7.3, 7.4, and 7.5, we answer three questions related to the completeness, complexity, and coast of our approach. Finally, we summarize the chapter in Section 7.6.

7.1 Introduction

In this chapter, we focus on the problem of model revision where the legitimate states are computed using automation techniques. In particular, when the algorithm wLspGenerator from Chapter 6 is used to generate the set of legitimate states. Recall from Chapter 6 that the current approaches for automated model revision describe the model as an abstract program. They require the designer to specify (1) the existing abstract program that is correct in the absence of faults, (2) the program specification, (3) the faults that have to be tolerated, and (4) the program legitimate states, from where the existing program satisfies its specification (c.f. Figure 7.1). We call this problem as *the problem of model revision with explicit legitimate states*.

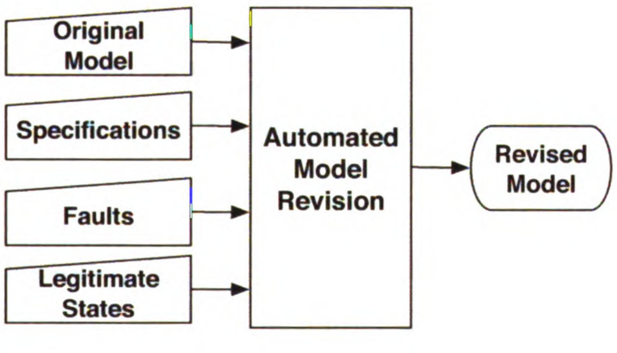

Figure 7.1: Model Revision with Explicit Legitimate States.

We focus on the problem of model revision where the input only consists of the fault-intolerant program, faults and the specification, i.e., it does not include the legitimate states.

We call this problem as *the problem of model revision without explicit legitimate states* (cf. Figure 7.2).

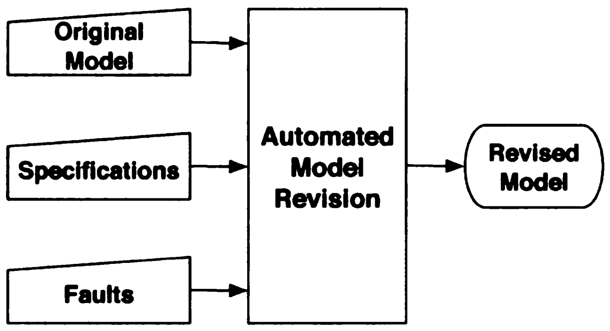

Figure 7.2: Model Revision *without* Explicit Legitimate States.

There are several important questions that have to be addressed for such a new formulation.

Q. 1 Is the new formulation relatively complete? (i.e., if it is possible to perform model revision using the problem formulation in Figure 7.1, is it guaranteed that it could be solved using the formulation in Figure 7.2?)

An affirmative answer to this question will indicate that reduction of designers' burden does not affect the solvability of the corresponding problem.

Q. 2 Is the complexity of both formulation in the same class? (By same class, we mean polynomial time reducibility, where complexity is computed in the size of state space.)

An affirmative answer to this question will indicate that the reduction in the designers' burden does not significantly affect the complexity.

Q. 3 Is the increased time cost, if any, small comparable to the overall cost of program revision?

While Question 2 focuses on qualitative complexity, assuming that the answer is affirmative, Question 3 will address the quantitative change in complexity.

In this chapter, we show that the answer to Q. 1 is affirmative (cf. Theorem 7.3.1). Furthermore, we show that the answer to Q. 2 is partially affirmative. Specifically, we identify two versions of problem revision: *partial revision* and *total revision*. We show that the answer is affirmative for total revision (cf. Theorem 7.4.3). We point out that the answer is negative for partial revision. In other words, for partial revision, complexity of solving the problem in Figure 7.2 can be larger (cf. Section 7.4.5). Even though the answer to Q. 2 is negative for partial revision, we show that there is a subclass of this problem where the complexity of the approach in Figure 7.2 is the same as that in Figure 7.1. In particular, we show that for all instances where the answer to the problem in Figure 7.1 is affirmative, it is possible to solve the corresponding problem in Figure 7.2 under the same complexity class. However, it is possible that the answer to the problem in Figure 7.1 is negative, i.e., the corresponding algorithm declares failure to generate the fault-tolerant program, although the answer to the corresponding problem in Figure 7.2 is affirmative. For these cases, complexity of solving the problem in Figure 7.2 can be high. Regarding Q. 3, we show that for instances where the answer to the question in Figure 7.1 is affirmative, the extra computation cost of solving the problem using an approach in Figure 7.2 is small.

7.2 Problem Statement

In this section, we formally define the problem of model revision with and without explicit legitimate states.

Model Revision *with* Explicit legitimate states (Approach in Figure 7.1). Recall that in Section 2.5 we defined what it means for a program to be (masking) fault-tolerant. Using a similar definition we now formally specify the problem of deriving a fault-tolerant program from a fault-intolerant program with explicit legitimate states I, safety specification Sf_p, and liveness specification Lv_p. The goal of the model revision is to modify p to p' by *only adding fault-tolerance*, i.e., without adding new behaviors in the absence of faults.

Since the correctness of p is known only from its legitimate states, I, it is required that the legitimate states of p', say I', cannot include any states that are not in I. Additionally, inside the legitimate states, it cannot include transitions that were not transitions of p. Also, by Assumption II.1, p cannot include new terminating states that were not terminating states of p. Finally, p' must be fault-tolerant. Thus, the problem statement (from [101]) for the case where the legitimate states are specified explicitly is as follows.

Problem Statement 7.1

Revision for Fault-Tolerance *with* Explicit Legitimate States.

Given p, I, Sf_p, Lv_p and f such that p satisfies

Sf_p and Lv_p from I

Identify p' and I' such that

(Respectively, does there exist p' and I' such that)

A1: $I' \Rightarrow I$.

A2: $s_0 \in I' \;\Rightarrow\; \forall s_1 : s_1 \in I' : ((s_0, s_1) \in p' \Rightarrow (s_0, s_1) \in p)$.

A3: p' is f-tolerant to Sf_p and Lv_p from I'.

Note that this definition can be instantiated for each level of fault-tolerance (i.e., masking, failsafe, and nonmsaking). Also, the above problem statement can be used as a revision problem or a decision problem (with the comments inside parenthesis).

We call the above problem as the problem of 'partial revision' because the transitions of p' that begins in I' are a subset of the transitions of p that begins in I'. An alternative formulation is that of total revision where the transitions of p' that begins in I' is *equal to* the transitions of p that begins in I'. In other words, the problem of *total revision* is identical to the problem statement 7.1 except that A2 is changed to A2$'$ described next:

A2$'$: $s_0 \in I' \Rightarrow \forall s_1 : s_1 \in I' : ((s_0, s_1) \in p' \iff (s_0, s_1) \in p)$

Modeling Revision *without* Explicit legitimate states (Approach in Figure 7.2) Now, we formally define the new problem of model revision without explicit legitimate states.

The goal in this problem is to find a fault-tolerant program, say p_r. It is, also, required that there is some set of legitimate states for p, say I, such that p_r does not introduce new behaviors in I. Thus, the problem statement for partial revision for the case where the legitimate states are not specified explicitly is as follows.

Problem Statement 7.2

Revision for Fault-Tolerance *without* Explicit Legitimate States.

Given p, Sf_p and Lv_p, and f

Identify p_r such that

(Respectively, does there exist p_r such that)

($\exists I$ ::

B1: $s_0 \in I \Rightarrow \forall s_1 : s_1 \in I : ((s_0, s_1) \in p_r \Rightarrow (s_0, s_1) \in p)$

B2: p_r is a f-tolerant to Sf_p and Lv_p from I

)

Just like problem statement 7.1, the problem of total revision is obtained from problem statement 7.2 by changing B1 with B1$'$ described next:

B1$'$: $s_0 \in I \Rightarrow \forall s_1 : s_1 \in I : ((s_0, s_1) \in p_r \iff (s_0, s_1) \in p)$

Existing algorithms for model revision [27,30,101,111] are based on Problem Statement 7.1. Also, the tool *SYCRAFT* [27] utilizes Problem Statement 7.1 for the addition of fault-tolerance. However, as stated in Section 7.1, this requires the users of *SYCRAFT* to identify the legitimate states explicitly. Our goal is to evaluate the effect of simplifying the task of the designers by permitting them to omit explicit identification of legitimate states.

7.3 Relative Completeness (Q. 1)

In this section, we show that if the problem of model revision can be solved with explicit legitimate states (Problem Statement 7.1), then it can also be solved without explicit legitimate states (Problem Statement 7.2). Since each problem statement can be instantiated with partial or total revision, this requires us to consider four combinations. We prove this result in Theorem 7.3.1.

Theorem 7.3.1 -

If

- *the answer to the decision problem 7.1 is affirmative with input p (fault-intolerant program), Sf_p (safety specification), Lv_p (liveness specification), f (faults) and I (legitimate states).*

Then

- *the answer to the decision problem 7.2 is affirmative with input p (fault-intolerant program), Sf_p (safety specification), Lv_p (liveness specification) and f (faults).*

Proof. Intuitively, a slightly revised version of the program that satisfies Problem 7.1 can be used to show that Problem 7.2 can be solved. Specifically, let the transitions of p_r to be $\{(s_0,s_1)|\,(s_0 \in I' \wedge s_1 \in I' \wedge (s_0,s_1) \in p) \vee (s_0 \notin I' \wedge (s_0,s_1) \in p')\}$.

Formally, since the answer to the decision Problem 7.1 is affirmative, there exists program p' and I' that satisfy constraints in Problem Statement 7.1. To show that the answer to the decision problem 7.2 is affirmative, we need to find p_r such that constraints of Problem Statement 7.2 are satisfied. We let transitions of p_r be

$$\{(s_0,s_1)|\quad (s_0 \in I' \wedge s_1 \in I' \wedge (s_0,s_1) \in p) \vee (s_0 \notin I' \wedge (s_0,s_1) \in p')\}.$$

Next, we show that p_r satisfies the constraints of Problem Statement 7.2. Towards this end, we instantiate I to be I' and show that constraints $B1$ and $B2$ are satisfied.

- **Constraint B1**: By construction of transitions of p_r, this constraint is satisfied for the case where we consider partial revision and for the case where we consider total revision.

- **Constraint B2**: By construction, I' is closed in p_r. Also, since $I' \Rightarrow I$ and p satisfies Sf_p and Lv_p from I, it is straightforward to observe that p_r satisfies Sf_p and Lv_p from I'.

  Also, transitions of p_r that begin outside I' are identical to that of p'. The second constraint "$(\exists T :: \ldots)$" from the definition of fault-tolerance is also satisfied. Thus, p_r is fault-tolerant to Sf_p and Lv_p from I'. ∎

Implication of Theorem 7.3.1 for Q. 1: From Theorem 7.3.1, it follows that answer to Q. 1 from Introduction is affirmative for both partial and total revision. Hence, the new formulation (c.f. Figure 7.2) is relatively complete.

7.4 Complexity Analysis (Q. 2)

In this section, we focus on the second question and compare the complexity class for Problem 7.1 with that of Problem 7.2. In particular, in Section 7.4.1, we show that the complexity of the model revision can increase substantially for partial revision if legitimate states are not specified explicitly. Then in Section 7.4.2, we show that for total revision Problem 7.2 can be reduced to Problem 7.1 in polynomial time. In Section 7.4.3, we give a heuristic-based approach for partial revision. Furthermore, we show that the heuristic is guaranteed to work when the answer to the corresponding Problem in Figure 3.1 is affirmative. In section 7.4.4, we show how one can obtain an algorithm for model revision without explicit legitimate states by utilizing an algorithm that requires explicit legitimate states. Finally, we mention other complexity results in Section 7.4.5.

7.4.1 Complexity Comparison for Partial Revision

In this section, we show that solving Problem 7.2 for partial revision is NP-complete. Since the complexity of the revision Problem 7.1 is in P [101], it follows that the complexity of partial revision increases substantially when the legitimate states are not specified explicitly. We show this by a reduction from the well-known 3-SAT problem. The 3-SAT instance is specified as follows:

3-SAT Instance. Let $x_1, x_2, ..., x_n$ be propositional variables. Given is a Boolean formula $y = y_1 \wedge y_2 \cdots \wedge y_M$, where each y_j $(1 \leq j \leq M)$ is a disjunction of exactly three literals. Does there exist an assignment of truth values to $x_1, x_2, ..., x_n$ such that y is satisfiable?

Since the membership of Problem 7.2 in NP is straightforward, we focus on showing that it is NP-complete. Hence, we first present the mapping from the 3-SAT instance to the problem of partial revision without explicit legitimate states. Then, we show that the given 3-SAT instance is satisfiable iff the answer to the corresponding instance of partial revision is affirmative.

Mapping 3-SAT to Partial Revision without Explicit Legitimate States

We now present the mapping of an instance of the 3-SAT problem to an instance of the partial revision problem without explicit legitimate states. Recall that this instance consists of the program (specified in terms of its state space and transitions), safety, and liveness specification and faults. We begin with identifying the input program. Then, we identify faults and finally we identify safety and liveness specification.

The state space of the input program. Corresponding to each variable x_i of the given 3-SAT instance, we introduce eight states $P_i, Q_i, R_i, T_i, a_i, b_i, c_i$, and d_i where $1 \leq i \leq n$ (cf. Figure 7.3). For each disjunction y_j, we introduce states Z_j and e_j, where $1 \leq j \leq M$, in the state space. Thus, state space of the input program is $S_p = \{P_i, Q_i, R_i, T_i, a_i, b_i, c_i, d_i \mid 1 \leq i \leq n\} \cup \{Z_j, e_j \mid 1 \leq j \leq M\}$.

Transitions of the input program. Corresponding to each variable x_i, we include the following transitions in δ_p: $(P_i, a_i), (a_i, c_i), (c_i, b_i), (b_i, Q_i)$, $(R_i, b_i), (b_i, d_i), (d_i, a_i)$, (a_i, T_i), (Q_i, e_j) and (T_i, e_j) where $1 \leq j \leq M$. Moreover, corresponding to each disjunction y_j, we include the following transitions:

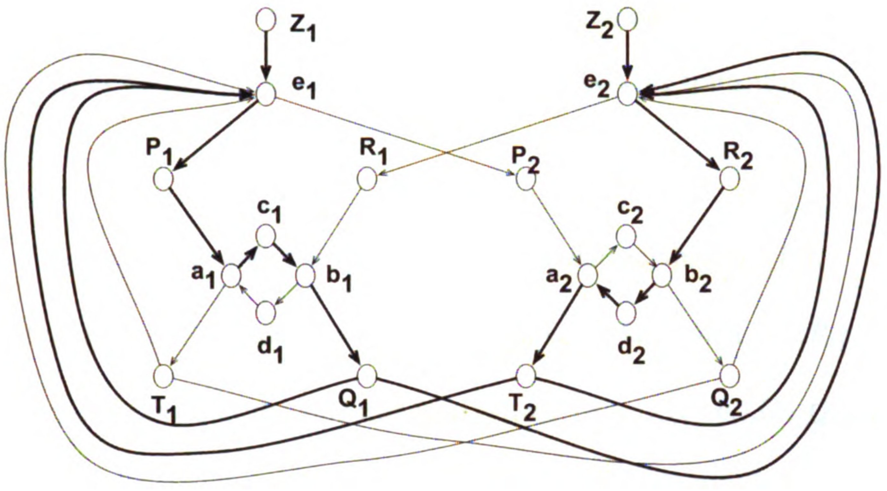

Figure 7.3: Mapping of $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ into corresponding program transitions. The transitions in bold show the revised program where $x_1 = true$ and $x_2 = false$.

- (Z_j, e_j),
- If x_i is a literal in y_j, then we include the transition (e_j, P_i) in δ_p, and
- If $\neg x_i$ is a literal in y_j, then we include the transition (e_j, R_i) in δ_p.

Fault transitions. The fault transitions $f = \{(T_i, Z_j), (Q_i, Z_j) | 1 \leq i \leq n,\ 1 \leq j \leq M\}$.

Safety specification Sf_p. All transitions except those in $\delta_p \cup f$ that violate safety.

Liveness specification Lv_p. The liveness specification is $P_i \leadsto c_i$, $c_i \leadsto Q_i$, $R_i \leadsto d_i$ and $d_i \leadsto T_i$, where $1 \leq i \leq n$.

Reduction from the 3-SAT Problem.

Theorem 7.4.1 *The given instance of the 3-SAT problem is satisfiable iff the corresponding instance of the partial revision problem has an affirmative answer for masking fault-tolerance.*

Proof.

First, we prove the $\Rightarrow$ part, then we prove $\Leftarrow$ part.

- $\Rightarrow$ If the given instance of the 3-SAT problem is satisfiable, then we construct the transitions of revised program by including the following transitions:
  - (Z_j, e_j), $1 \leq j \leq M$,
  - If y_j contains x_i and x_i is assigned truth value true, then (e_j, P_i),
  - If y_j contains $\neg x_i$ and x_i is assigned truth value false then, (e_j, R_i),
  - If x_i is assigned truth value true then $(P_i, a_i), (a_i, c_i), (c_i, b_i)$,$(b_i, Q_i)$, and (Q_i, e_j), $1 \leq i \leq n$,
  - If x_i is assigned truth value false then $(R_i, b_i), (b_i, d_i), (d_i, a_i)$, (a_i, T_i), and (R_i, e_j), $1 \leq i \leq n$.

  The predicate, I', used to show that this program satisfies *SPEC* includes all reachable states except $\{ Z_j | 1 \leq j \leq M \}$. It is straightforward to show that the constraints $B1$ and $B2$ are satisfied.

- $\Leftarrow$ The legitimate state predicate of the revised program contains at least one state. Our first step is to show that for some i, Q_i or T_i is included in the legitimate state predicate of the revised program. To show this, we observe that if Z_j, $1 \leq j \leq M$, is included in the legitimate state predicate for some j, then the corresponding state e_j must also be included in the legitimate state predicate. Hence, the revised program must include at least one transition that begins in e_j. It follows that either P_i or R_i, $1 \leq i \leq n$, must also be included the legitimate state predicate. If P_i (respectively, R_i) is included in the legitimate state predicate, then c_i and Q_i (respectively, d_i and T_i) must also be included so that Lv_p is satisfied. Also, if a_i (respectively, b_i) is included in the legitimate state predicate, then T_i or Q_i must also be included in the legitimate state predicate. From the above discussion, it follows that for some i, Q_i or T_i is included in the legitimate state predicate of the revised program. Now, based on the definition of faults, all states in $\{Z_j | 1 \leq j \leq M \}$ are reachable in the presence of faults. Hence, transition (Z_j, e_j) must be included for $1 \leq j \leq M$ in the revised program.

Furthermore, some transition originating from e_j must also be included. Transitions from e_j correspond to literals in disjunction y_j. If a transition of the form (e_j, P_i) is included, then we set x_i to true. If a transition of the form (e_j, R_l) is included, then we set x_l to false.

Observe that if P_i is reachable in the revised program, then it must also include (P_i, a_i), (a_i, c_i), (c_i, b_i), and (b_i, Q_i) so that Lv_p is satisfied. And, if R_i is reachable in the revised program, then it must also include (R_i, b_i), (b_i, d_i), (d_i, a_i), and (a_i, T_i). However, if all these transitions are included, then Lv_p will not be satisfied. Therefore, for any i, revised program cannot reach both P_i and R_i. This implies that the truth value assigned to x_i by any disjunction is the same. Moreover, based on the construction of the instance of the program of partial revision, the truth assignments to literals make each clause to be satisfied, i.e., the assignment of truth values to literals causes the given 3-SAT formula to be satisfiable. ∎

From the above theorem, it follows that the problem of partial revision without explicit legitimate states is NP-hard. Moreover, in [101], it is shown that the problem of partial revision can be solved in polynomial time if legitimate states are specified explicitly. Thus, it follows that the complexity of partial revision increases substantially when explicit legitimate states are not available.

Intuition behind the increased complexity of partial revision. We analyze the NP-completeness proof to determine why the complexity of partial revision increased substantially. Towards this end, we carefully look at the instance of partial revision generated from the SAT formula. Observe that the fault-intolerant program does not satisfy Lv_p from P_i or R_i, as the program can be stuck in the loop $(a_i, c_i), (c_i, b_i), (b_i, d_i), (d_i, a_i)$. However, removal of some transitions allows P_i (or, R_i) to be included as a legitimate state. The increased complexity of partial revision is caused by the need to remove the "right" transitions so that the additional states can be included in the set of legitimate states. Choosing these "right" transitions increases the complexity substantially.

7.4.2 Complexity Comparison for Total Revision

Although the complexity of partial revision increases substantially when legitimate states are not available explicitly, we find that complexity of total revision effectively remains unchanged. We note that this is the first instance where complexity difference between partial and total revision has been identified. To show this result, we show that in the context of total revision Problem 7.2 is polynomial time reducible to Problem 7.1 Since the results in this section require the notion of weakest legitimate state predicate, we define it next. Recall that we use the term legitimate state predicate and the corresponding set of legitimate states interchangeably. Hence, weakest legitimate state predicate corresponds to the largest set of legitimate states.

Definition. Let $I_w = wLsp(p, Sf_p, Lv_p))$ be the *weakest legitimate state predicate* of p for $SPEC(=\langle Sf_p\ ,\ Lv_p\rangle)$ iff:

1: p satisfies $SPEC$ from I_w, and

2: $\forall I ::\ (\,p$ satisfies $SPEC$ from $I) \Rightarrow I_w$. ∎

Recall from Chapter 6 that, we identified the algorithm wLspGenerator(p, Sf_p, Lv_p) that computes weakest legitimate state predicate in polynomial time in the state space of p.

Theorem 7.4.2 *If the answer to the decision problem 7.2 (***with total revision***) is affirmative (i.e., $\exists$ p_r that satisfies the constraints of the Problem 7.2) with input p, Sf_p, Lv_p, and f, then the answer to the decision problem 7.1 (with total or partial revision) is affirmative (i.e., $\exists$ p' and I' that satisfy the constraints of the Problem 7.1) with input p, Sf_p, Lv_p, f, and $wLsp(p, Sf_p, Lv_p)$.*

Proof. Intuitively, the program p_r obtained for solving problem statement 7.2 can be used to show that problem 7.1 is satisfied. Specifically, let I_2 be the predicate used to show that

p_r satisfies constraints of Problem 7.2. Then, let $p' = p_r$ and $I' = I_2$.

Formally, since the answer to the decision problem 7.2 is affirmative, there exists program p_r that satisfies constraints in Problem Statement 7.2 (with total revision). Let I_2 denote the predicate used to show that constraints $B1$ and $B2$ are satisfied. Let $I_w = wLsp(p, Sf_p, Lv_p)$. To show that the answer to the decision problem 7.1 is affirmative, we need to find p' and I' such that constraints of Problem Statement 7.1 are satisfied. We let $p' = p_r$ and $I' = I_2$. Based on constraint $B2$, p_r satisfies Sf_p and Lv_p from I_2. Also, from constraint $B1$, (for total revision), p satisfies Sf_p and Lv_p from I_2. Now, we show that constraints $A1, A2$, and $A3$ are satisfied.

- **Constraint A1**: Based on definition of weakest legitimate state predicate, $I_2 \Rightarrow I_w$. Thus, constraint $A1$ is satisfied.
- **Constraint A2**: Based on constraint $B1$, constraint $A2$ is satisfied for both total and partial revision.
- **Constraint A3**: Based on constraint $B2$, p_r is $fault-tolerant$ to Sf_p and Lv_p from I_2. Thus, constraint $A3$ is satisfied. ∎

Remark: Note that if the phrase 'with total revision' shown in bold in Theorem 7.4.2 is replaced by 'with partial revision', then the corresponding theorem is not valid.

Theorem 7.4.3 *For total revision, the revision problem 7.2 is polynomial time reducible to the revision problem 7.1.*

Proof. Given an instance, say X, of the decision problem 7.2 that consists of p, Sf_p, Lv_p, and f, the corresponding instance, say Y, for the decision problem 7.1 is p, Sf_p, Lv_p, f, and $wLsp(p, Sf_p, Lv_p)$. From Theorems 7.3.1 and 7.4.2 it follows that answer to X is affirmative iff answer to Y is affirmative. ∎

7.4.3 Heuristic for Polynomial Time Solution for Partial Revision

Theorem 7.4.2 utilizes the weakest legitimate state predicate to solve the problem of total revision without explicit legitimate states. In this section, we show that a similar approach can be utilized to develop a heuristic for solving the problem of partial revision in polynomial time. Moreover, if there is an affirmative answer to the revision problem with explicit legitimate states, then this heuristic is guaranteed to find a revised program that satisfies constraints of Problem 7.2. Towards this end, we present Theorem 7.4.4.

Theorem 7.4.4 *For partial revision, the revision problem 7.2 consisting of* (p, Sf_p, Lv_p, f) *is polynomial time reducible to the revision problem 7.1 provided there exists a legitimate states predicate* I *such that the answer to the decision problem 7.1 for instance* (p, I, Sf_p, Lv_p, f) *is affirmative.*

Proof.

Clearly, if an instance of Problem 7.1 has an affirmative answer, then from Theorem 7.3.1, the corresponding instance of Problem 7.2 has an affirmative answer. Similar to the proof of Theorem 7.4.3, we map the instance of Problem 7.2 to an instance of Problem 7.1 where we use the weakest legitimate state predicate. Now, from Theorem 7.3.1 it follows that the answer to this revised instance of Problem 7.1 is also affirmative. ∎

What the above theorem shows is that even for partial revision, if it were possible to obtain a fault-tolerant program with explicit legitimate states, then it is possible to do so in the same complexity class without explicit legitimate states. However, there may be instances where answer to the decision problem 7.1 may be negative and the answer to the corresponding decision problem 7.2 is affirmative. For these instances, for partial revision, the complexity can be high.

7.4.4 Algorithm for Model Revision Without Explicit Legitimate States

In this section, we utilize the results in Section 7.4.2 to obtain an algorithm for model revision without explicit legitimate states. In particular, we present algorithm Add_fs_fr_spec that adds failsafe fault-tolerance (where safety is satisfied in the presence of faults although liveness may not be) to high atomicity programs (where a program transition can read any number of variables as well as write any number of variables in one atomic step). This algorithm is obtained by combining the algorithm wLspGenerator from Chapter 6 to compute the weakest legitimate state predicate as well as the algorithm Add_failsafe from [101].

Given the program transitions p, the fault transitions f, and the program specification $\langle Sf_p\ , Lv_p\rangle$, the goal of this algorithm is to compute the failsafe fault-tolerant program p_r that satisfies the constraints of problem statement 7.2 (with total revision). It first identifies the weakest legitimate state predicate I_w. If p has any state in I_w where it has no outgoing transitions, we add self-loops at those states. These self-loops help us distinguish between a state where p has no outgoing transitions and states that become a deadlock state because we removed some transitions of p. Then it identifies ms as the states that violate safety or the states from where execution of one or more fault transitions violates safety (Lines 4-7). Then, the algorithm finds the transitions, mt, of p that reach states in ms as well as transitions of p that violate the safety specification $SPEC_{bt}$ (Line 8).

If there exist states in I'_w such that execution of one or more fault actions from those states violates the safety of the specification, then it recalculates I'_w by removing those states (Lines 10-13). In this recalculation, it ensures that all computations of $p-mt$ within, I'_w, are infinite. In other words, the final value of I'_w is the largest subset of I_w-ms such that all computations of $p-mt$ when restricted to that subset are infinite. At this point, if I'_w is empty the algorithm declares that no failsafe fault tolerant program can be found. Otherwise, the algorithm removes mt from p to compute p_r where no program transitions violate the program specification (Line 18). Now, it ensures that all the transitions of p_r

Algorithm 15 Add_fs_fr_spec: Addition of Failsafe Fault-Tolerance

```
Input: program transitions $p$, fault transitions $f$, safety specification $Sf_p$
    (consisting of $SPEC_{bs}$ and $SPEC_{bt}$), liveness specification $Lv_p$ (consisting of multiple $\mathcal{F} \rightsquigarrow \mathcal{T}$ proprieties)
Output: failsafe fault-tolerant program $p_r$.
    // Find the legitimate states $I_w$
 1: $I_w = wLsp\ (p,\ Sf_p, Lv_p)$;
 2: $Self\_loops = \{(s_0,s_0) | s_0 \in I_w \wedge \forall s_1 :: (s_0,s_1) \notin p\}$;
 3: $p = p \vee Self\_loops$;

 4: repeat
 5:     $ms' := ms$;
 6:     $ms := ms \cup \{s_0 :: \exists s_1 : (s_0,s_1) \in f \wedge (\ ((s_0,s_1) \in SPEC_{bt})\ \vee\ (s_1 \in ms)\ )\}$;
 7: until $(ms\ =\ ms')$

 8: $mt := \{(s_0,s_1) ::\ (\ ((s_0,s_1) \in SPEC_{bt})\ \vee\ (s_1 \in ms)\ )\ \}$;

    // compute the largest subset of $I_w$ from where all computations of $p$ are infinite
 9: $I'_w := I_w - ms$;
10: repeat
11:     $I'_{tmp}\ :=\ I'_w$;
12:     $I'_w\ :=\ I'_w - \{s_0 :: s_0 \in I'_w : (\forall s_1 :: s_1 \in I'_w : (s_0,s_1) \notin (p - mt))\}$;
13: until $(I'_w\ =\ I'_{tmp})$

14: if $(I'_w = \{\})$ then
15:     print No failsafe $f$-tolerant program $p_r$ exists;
16:     return $\{\}$;
17: else
18:     $p_r := p - mt$;
19:     $p_r = p_r - \{(s_0,s_1) :: s_0 \in I'_w\ \wedge\ s_1 \notin I'_w\}$;
20: end if
21: return $p_r - Self\_loops$;
```

that start in a state in I_w also end in a state in I_w. If not such transitions are removed from p_r.

Remark: Note that since this section focuses on failsafe fault-tolerant programs, there is no recovery requirement for the program in the presence of faults. However, for other levels of fault-tolerance, e.g., nonmasking and masking, where the program needs to satisfy its liveness properties as well, we would need an additional requirement that states that eventually faults stop for a long enough time to ensure that liveness properties can be met.

Theorem 7.4.5 *Algorithm* Add_fs_fr_spec *is sound, i.e., the output p_r of* Add_fs_fr_spec *satisfies the constraints of Problem Statement 7.2.*

Proof. Let I_w from the problem statement 7.2 be instantiated with the value of I'_w at the end of Add_fs_fr_spec. Now, the first constraint of the Problem Statement 7.2 is satisfied by construction. Moreover, the satisfaction of the first constraint implies correctness of p_r in the absence of faults. Regarding the behavior in the presence of faults, we can observe that by construction, the program does not reach a state in $SPEC_{bs}$ or execute a transition in $SPEC_{bt}$. Moreover, the construction of *ms* implies that the program does not reach states from where faults can violate the safety specification. Thus, the revised program is failsafe fault-tolerant. ∎

Theorem 7.4.6 *Algorithm* Add_fs_fr_spec *is complete, i.e., if it declares failure, then there does not exist a fault-tolerant program that satisfies the constraints in Problem Statement 7.2.*

Proof. Suppose that a program, say p'', satisfies the constraints of Problem Statement 7.2. Let I'' be the predicate used in demonstrating that p'' satisfies the constraints of Problem Statement 7.2. Now, we show that at any time in the use of Add_fs_fr_spec, it must be the case that $I'' \subseteq I'_w$. In particular, on Line 1, this follows from the correctness of the algorithm that computes the weakest legitimate state predicate. On Line 9, this follows from the fact that no state in *ms* can be legitimate state, as faults alone can violate safety from those states. Likewise, since I'' cannot have deadlock states, $I'' \subseteq I'_w$ is true on Line 12. Since the algorithm declares failure when $I'_w = \{\}$, it follows that $I'' = \{\}$, which is a contradiction. ∎

Theorem 7.4.7 *The algorithm* Add_fs_fr_spec *is in* P.

Proof. Let us consider the complexity of each statement in Add_fs_fr_spec. (1) From Chapter 6, the complexity of computing the weakest legitimate state predicate is in P. (2-3) The complexity of statements 2, and 3 is clearly in P. (4-7) Calculating ms is in P as we can use the following algorithm: For each fault transition (s_0, s_1) such that (s_0, s_1) violates safety of $SPEC$, include s_0 in ms. Now, in each iteration, check if there exists a fault transition (s_0, s_1) such that $s_0 \notin ms$ and $s_1 \in ms$. If such a transition exists add s_0 to ms. Since the size of ms increases by at least one in each iteration, the number of iterations is polynomial in the state space, namely, S_p. (8) Calculating mt is in P as we need to check each transition only once. (9) This statement is in P. The while loop in (10-13) can execute only $|S_p|$ number of times. (14-21) The complexity of these statements is clearly polynomial. ∎

The above result shows that in the context of failsafe fault-tolerance, when we reduce the designer's burden by not requiring them to identify the legitimate states explicitly, there is no significance in terms of the complexity class of the problem involved or in terms of the soundness and completeness property of the corresponding algorithms.

7.4.5 Summary of Complexity Results

In Section 7.4.4, we showed that the problem of total model revision for failsafe fault-tolerance is in P. In this section, we list the complexity for other levels of fault-tolerance for both total and partial revision.

Recall from Section 7.4.1 that, for partial revision, the problem of adding failsafe and masking fault-tolerance is NP-complete. For distributed programs, it is shown (in [101]) that revising the program for adding failsafe and masking fault-tolerance is NP-complete when the set of legitimate states is specified explicitly. A variation of that proof also works for model revision without explicit legitimate states. Revising the program for adding nonmasking fault-tolerance is in NP. However, it is not known whether it is NP-complete or whether it is in P.

For high atomicity programs, i.e., where a program can read and write all its variables atomically, it is possible to perform total revision in P. To show this, we note that the algorithm Add_fs_fr_spec first identifies the weakest legitimate state predicate. Then it utilizes the set of legitimate states in Add_failsafe (from [101]) which requires that the legitimate states be explicitly specified. Likewise, we can utilize the algorithms Add_nonmasking and Add_masking (from [101]) to obtain the corresponding algorithms for total revision for adding nonmasking and masking fault-tolerance.

		Revision *Without* Explicit Legitimate States		**Revision *With* Explicit Legitimate States**	
		Partial	**Total**	**Partial**	**Total**
High Atomicity	Failsafe	?	$P^{\ddagger}$	P^{*}	P^{*}
	nonmasking	?	$P^{\ddagger}$	P^{*}	P^{*}
	masking	$NP-C^{\dagger}$	$P^{\ddagger}$	P^{*}	P^{*}
Distributed Programs	Failsafe	$NP-C^{\Delta}$	$NP-C^{\Delta}$	$NP-C^{*}$	$NP-C^{*}$
	nonmasking	?	?	?	?
	masking	$NP-C^{\Delta}$	$NP-C^{\Delta}$	$NP-C^{*}$	$NP-C^{*}$

Table 7.1: The complexity of different types of automated revision (NP-C = NP-Complete).

In summary, the results for complexity comparison are as shown in Table 7.1. Results marked with † follow from NP-completeness results from Section 7.4.1. Results marked ‡ follow from Section 7.4.2, 7.4.3, and 7.4.4. Results marked Δ are stated without proof. Results marked ? indicate that the complexity of the corresponding problem is open. And, finally, results marked $*$ are from [101].

7.5 Relative Computation Cost (Q. 3)

As mentioned in Section 7.1, the increased cost of model revision in the absence of explicit legitimate states needs to be studied in two parts: complexity class and relative increase in the execution time. We considered the former in Section 7.4. In this section, we consider the latter. As we can see from Section 7.4.4, if the legitimate states are not specified explicitly, the increased cost of model revision is essentially that of computing $wLsp(p, Sf_p, Lv_p)$. Hence, we analyze the complexity of computing $wLsp(p, Sf_p, Lv_p)$ in the context of a case study. We choose the classic example from the literature, namely, Byzantine Agreement [107]. We explain this case study in detail and show the time required to generate the weakest legitimate state predicate for different numbers of processes. This case study illustrates that the increased cost when explicit legitimate states are unavailable is very small compared to the overall time required for the addition of fault-tolerance. In particular, we show that reducing the burden of the designer in terms of not requiring the explicit legitimate states increases the computation cost by approximately 1%.

Throughout this section, the experiments are run on a MacBook Pro with 2.6 Ghz Intel Core 2 Duo processor and 4 GB RAM. The OBDD representation of the Boolean formula has been done using the C++ interface to the CUDD package developed at the University of Colorado [125].

The amount of time required for computing this set of legitimate states for a different number of processes is as shown in Table 7.2. We would like to note that the set of legitimate states computed in these case studies is the same as that used in the addition of fault-tolerance.

We use this case study to illustrate that computing the set of legitimate states to be those that are reachable from initial states is not relatively complete. In particular, for the Byzantine agreement example, the initial state is one where all processes are non-Byzantine and the decision of all non-general processes is equal to $\perp$. Clearly, all processes are non-Byzantine in all states reached by the program from these initial states. It follows that

No.of Process	Reachable States	Leg. States Generation Time(Sec)	Total Revision Time(Sec)
10	10^9	0.57	6
20	10^{15}	1.34	199
30	10^{22}	4.38	1836
40	10^{30}	9.25	9366
50	10^{36}	26.34	> 10000
100	10^{71}	267.30	> 10000

Table 7.2: The time comparison for the Byzantine Agreement program.

recovery to these reachable states is not always feasible in the presence of faults. Hence, these reachable states are insufficient to obtain the fault-tolerant program. By contrast, the weakest legitimate state predicate can be utilized to find the fault-tolerant program.

7.6 Summary

We devoted this chapter to study the problem of automated model revision without explicit legitimate states. In particular, we compared performing the revision when the legitimate states are explicitly specified with that when they are not explicitly specified. We considered three different aspects in our comparison: relative completeness, qualitative complexity class comparison, and quantitative change of the time for model revision. We illustrated that our approach for model revision without explicit legitimate states is relatively complete. This is important, since it implies that the reduction in the human effort required for model revision does not reduce the class of the problems that could be solved. Additionally, we found some surprising and counterintuitive results. Specifically, for total revision, we found that the complexity class remains unchanged. However, for partial revision, the complexity class changes substantially. Finally, we found that quantitative change of the time for model revision without explicit legitimate states is negligible.

Chapter 8

Related Work

During the past three decades, automation of the software verification tools evolved significantly. Currently, verification tools are widely used in several applications. In particular, they are used in the verification of the high assurance and mission critical systems, where the consequences of any failure can end with catastrophic results.

Formal verification of distributed and concurrent program focuses on the use of mathematical logic and formal methods to verify the correctness of the properties of a specific program. Initially, the focus was on developing techniques to verify full functional correctness. However, most of the tools developed for this purpose were incapable of handling complex systems. This limitation encouraged many to focus on the verification of the properties that are more important than others. In most of the verification techniques the system and the desired properties are described via a logical model. The verification algorithm answers with (yes/no) to the question of whether the model satisfies the desired property.

Unlike the automated verification techniques, the goal of the automated model revision is to automatically revise an existing model to generate a new model, which is correct-by-construction. Such revised model will preserve the existing model properties as well as satisfy new properties. The basic form of the problem of automated model revision focuses on modifying an existing model, say M, into a new model, say M'. It is required that M' satisfies the new property of interest. Additionally, M' continues to satisfy existing properties of M using the same transitions that M used.

In this chapter, we briefly review some of the automated verification techniques and discuss their relation to our approach. Currently, there is a wide range of tools available for verifying the correctness of distributed programs. Those tools are based on different techniques, which makes them useful for different types of applications. We believe that no single approach is suitable for the verification of all types of distributed programs. However, some approaches may be more appropriate to some applications more than others. Out of the wide range of the available techniques for automated verification of the finite state distributed and concurrent programs, we focus on those that are closely related to our approach.

8.1 Model Checking

Model checking is a technique for verifying the correctness of finite state programs. The idea is based on exploring the state space of the program, described using temporal logic, in an efficient manner. In model checking, the program is represented using Kripke structure, say M, and a formula, say f, represent one of the program properties. The model checker determines if M is a model for f, i.e., whether the formula holds or not.

One of the advantages of the model checking technique is that it provides a push button approach. Model checking is effective in verifying whether the system meets the desired properties. Furthermore, if the model does not satisfy the property of interest, then the model checker, typically, provides a counterexample and the corresponding execution trace. Moreover, it supports partial verification, e.g., it does not require the complete specification of the program being verified. Due to this push-button approach, model checking techniques have become very popular for detecting errors in the early stages of the design. Also, it has helped in transferring the formal verification of correctness from research to practice. Next, we briefly review the evolution of model checking tools and techniques.

As early as the 1970's, Tadao Murata and Kurt Jensen started working on the verification of Petri nets; however, there were no actual verification tools created prior to 1981 [42]. The initial work on state exploration started in the 1980 when Bochmann presented a method for verifying communication protocols [24]. Later, Holzmann presented a technique for automatic protocol verification. Burstall [37], Kröger [99] and Pnueli used the temporal logic to describe the program behavior and the proof of correctness was done manually.

In their early work on concurrency, E. M. Clarke et al. [45,50,110] focused on the fixed point theory and abstract interpretation. They emphasized on the connection between Branching Time Logic and Mu-Calculus [62]. Clarke also presented how program text is used to extract the invariant of a given program. In 1980, Emerson and Clarke [62] developed a technique based on branching time logic. Later they adopted more elegant presentation of temporal logic that was presented in [46]. In a milestone step in the evolution of verification techniques, Clarke, Emerson and Sistla [43,48] presented the EMC Model Checker. This was the first model checker that could handle fairness constraints. Although, the EMC Model Checker can only check models with state space of a size not more than 10^5, it was able to detect errors in several systems. In [63] Emerson and Halpern presented framework CTL* for investigating the expressive power of temporal logic. Their framework was a combination of branching-time and linear-time operators.

The most significant improvement in model checking was in the early 90's. To this end, symbolic model checking and partial order reduction were used in building the model checker. McMillan used a symbolic representation based on the ordered binary decision diagrams (OBDDs) to develop the SMV [112]. The compact representation of the state space and the transition graph made it possible to verify sophisticated programs with very large state space [36,112]. Since then, the SMV model checker has been used in verifying several systems. In 2000, a new version of SMV was released [41].

The second important improvement in model checking techniques is the exploitation of the partial order reduction of the state space [74]. The basic idea of the partial order reduction is as follows. If two events are independent, then the system will reach the same global state with no regards to which event execute first. This way, less space is needed to represent the system, which in turn reduces the effect of the state explosion problem.

Since the early 90's, many techniques have been developed to extend the capabilities of the model checking tools. These techniques include Abstraction [80], where the data values of the system, usually reactive systems, are mapped to smaller set of abstracted values, Compositional Reasoning [8,51,79], where the behavior of the system, which is composed of many similar process, can be represented by few processes, Symmetry Reduction [44, 117], where the model checker exploits the symmetrical characteristics of the program to obtain smaller model, and Induction and Parameterized Verification [106,131], where the behavior of the system is represented in a way that can be used for an arbitrary number of processes.

The development of more effective methods for program verification continued over the past few years. Also, it resulted in the creation of more innovative technique, to handle specific problems in more customized settings.

One application of our approach is to be complementary to existing approaches [36,41, 93] for verifying program correctness in early stages of system design. In particular, the techniques in [98,119,130] aim to identify unacceptable system behavior to find the root causes that makes the system behave incorrectly. However, these approaches do not address what to do when new faults or bugs are identified. Generally, it is left to the designer to address this with some guidance or with trial and error. Moreover, manual revision has the potential to introduce new errors.

Our approach focuses on automating such model revisions. Therefore, once the model checker identifies an instance where the model does not satisfy the property of interest, we can use the automated model revision techniques to automatically revise the existing model (c.f. Figure 8.1). The revised model will continue to satisfy the original properties as well as the new property. Such automated revision is highly desirable since it enables system designers to automatically and incrementally add properties to the models. Some of the advantages of this approach are that the revised model is correct by construction and there is no need to re-verify the revised model. Also, the original model properties are preserved. Furthermore, there is a potential for this approach to require less time and space complexities since it does not require the revision of the entire model specification.

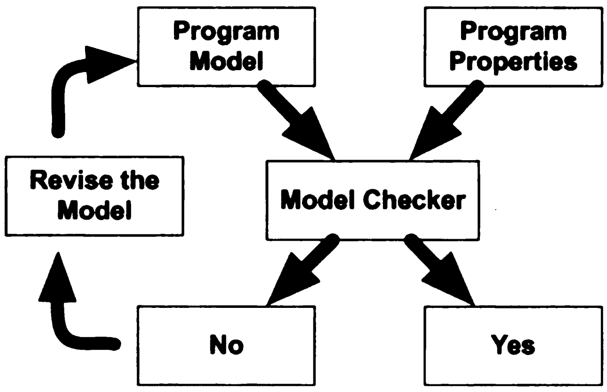

Figure 8.1: Model Checking and Automated Model Revision.

In another context, we have adopted many techniques, which were used to advance the development of better model checking tools, in the development of our model revision tools. For example, one way to reduce the complexity further is to integrate advances from model checking, as incremental synthesis involves several tasks that are also considered in model checking. We considered two approaches from model checking: (1) the use of symmetry and (2) the parallelism of the algorithm with multiple processors/cores.

8.2 Controller Synthesis and Game Theory

Our work is closely related to the work on controller synthesis (e.g. [16,17,32]) and game theory (e.g., [70]). In this work, 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*. In particular, Jobstmann, Griesmayer, and Bloem [96] used an approch based on the concepts of the game theory. They presented the problem of program repair by two players playing the Büchi game. They modeled the program and its environment as the two players. More specifically, the program takes a move in response to a move taken by the environment. Our formulation for the automated model revision is similar to that used by Ramadge and Wonham [122] in the discrete controller synthesis problem. In both approaches the goal is to restrict the program actions to the desired behaviors.

These techniques require highly expressive specifications. Hence, the complexity is also high (EXPTIME-complete or higher). 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 our work.

8.3 Model Revision and Automated Program Synthesis

In this section, we review the history and the evolution of the automated model revision techniques [101]. Also, we show how our work in this dissertation is related to the previous work done in this regard.

Automated program synthesis and revision have been studied from various perspectives. Inspired by the seminal work by Emerson and Clarke [64], Arora, Attie, and Emerson [11] propose an algorithm for synthesizing fault-tolerant programs from CTL specifications. Their method, however, does not address the issue of the addition of fault-tolerance to existing programs. Initially, Kulkarni and Arora presented an automated algorithm for the automated addition of fault-tolerance for centralized programs as well as distributed programs. Their approach depends on the existence of an original program that is correct in the absence of faults, i.e., the existing program satisfies its specification as far as no faults exists. Their goal is to modify (i.e., revise) the existing program and generate modified (i.e., revised) version of the program such that the revised program is fault-tolerant as well as it does not introduce any new behavior in the absence of faults [101]. The authors also analyzed the complexity of adding fault-tolerance in different setting. We used some of their results in the table in Section 7. For instance, they proved that the problem of the automated addition of masking fault-tolerance is NP-complete.

Kulkarni, Arora, and Chippada [102] developed a polynomial time algorithm for automated synthesis of fault-tolerant distributed programs. Since this problem was proven to be NP-hard in [101], the authors presented an algorithm that relies on heuristics to reduce the complexity. Moreover, they demonstrated that the algorithm suffices to synthesize an agreement program that tolerates a Byzantine fault.

In their effort to automate the synthesis of fault-tolerant programs, Ebnenasir and Kulkarni developed a framework, called Fault-Tolerance Synthesizer (FTSyn) [60]. The FTSyn framework implemented most of the heuristics that have been proposed to synthesize fault-tolerant programs. The main reasons for developing FTSyn were to validate the theoretical results as well as to provide developers with an interactive tool for automated synthesize. The authors use FTSyn to synthesize several fault-tolerant distributed programs. For instance, they used FTSyn to synthesis an altitude switch that controls the altitude of an aircraft. The input of FTSyn consists of an abstract program consisting of a set of processes described in a guarded command language. And, the output is masking fault-tolerant program also in guarded commands. The authors used FTSyn to demonstrate the applicability of their approach and also to show that with automation it can be applied to the cases where there are different types of faults. However, similar to other enumerative implementations, FTSyn was subject to the state explosion problems and was only suitable for synthesizing small programs.

Recently, Bonakdarpour and Kulkarni presented a symbolic-based implementation for the synthesis algorithm [27,30]. In their tool (SYCRAFT), the components of the synthesis algorithm are constructed using Boolean formulae represented by Bryants Ordered

Binary Decision Diagrams [33]. This was the first time where moderate to large sized programs (a state space of 10^{50} and beyond) have been synthesized. Although, both FTSyn and SYCRAFT implement similar synthesis heuristics from [102], there are several difference between them. For instance, the symbolic representation made SYCRAFT capable of handling programs with larger state space. Moreover, the grammar of the input language of SYCRAFT has more constructs which can assist the designer in describing the abstract program. Also, one of the characteristics of SYCRAFT is that it describes the output in an optimized representation. Using SYCRAFT, authors also have identified several bottlenecks that can slow down the synthesis. In particular, they identified the following bottlenecks: the deadlock resolution, computation of recovery action, computation of the fault-span and the cycle resolution. In this dissertation, we focused on two major complexity obstacles in deadlock resolution, namely computation of the recovery actions and the deadlock elimination. We used parallelism and symmetry to overcome these bottlenecks.

Our work in this dissertation is closely related to the tool SYCRAFT. In particular, we have implemented most of the techniques we presented in this dissertation and added them to SYCRAFT.

8.4 Parallelization and Symmetry

In the model checking community, various techniques have been proposed to implement the symbolic state space generation and exploration using parallel computing. Some of those approaches targeted the state explosion problem by focusing on data parallelism by distributing the computation among a group of workstations, e.g., NOWs [77,78,92,115, 126]. Their goal was mainly providing more memory resources to handle the expanding state-space. Obviously, the speed was not an issue here and the time complexity was not the target. Others focused on enhancing the time-efficiency by using parallelism. For this group, the goal was to use the ever-expanding parallel infrastructure of multi-core PCs and multi-processers platforms in expediting model checking. Most notably was the work on parallelizing the *Saturation* algorithm [39]. Unfortunately, the symbolic state exploration has proven to be notoriously resistant to parallelization.

In [66,67,69], the authors propose solutions and analyze different approaches to parallelization of the *saturation*-based generation of state space in model checking. In particular, in [67], the authors show that in order to gain speedups in saturation-based parallel symbolic verification, one has to pay a penalty for memory usage of up to 10 times, that of the sequential algorithm. Other efforts range from simple approaches that essentially implement BDDs as two-tiered hash tables [115,127], to sophisticated approaches relying on *slicing* BDDs [78] and techniques for *workstealing* [77]. However, the resulting implementations show only limited speedups. Ezekiel j., Luttgen G., and Siminiceanu R. [68] argue that a heavily optimized symbolic algorithm such as Saturation may be more efficient than a parallel version of the same algorithm.

Ebnenasir presented a divide-and-conquer method [58] for synthesizing *failsafe* fault-tolerant distributed programs. In failsafe fault-tolerance, the program is not required to maintain any liveness requirements when faults occur. Therefore, resolving deadlock states in the fault-span is not needed.

In this dissertation, we focused on two major complexity obstacles in deadlock resolution, namely computation of the recovery actions and the deadlock elimination. We used parallelism and symmetry to reduces the time complexity. Our work utilizes parallelization of group computation as well as symmetry for expediting the automated model revision. Unlike other parallelization algorithms for the symbolic based representation of models, we were able to achieve speedup up to multiple orders of magnitude. By focusing on parallelizing the group operation, we were able to harness the benefits of the multi-core infrastructure.

8.5 Nonmasking and Stabilizing Fault-Tolerance

Automated program synthesis is studied from different perspectives. One approach (e.g., [11]) focuses on synthesizing fault-tolerant programs from their specification in a temporal logic (e.g., CTL, LTL, etc.). Our approach for adding nonmasking and stabilizing fault-tolerance is based on satisfying constraints that should be true in legitimate states.

In masking fault-tolerance, when faults occur, the program cannot violate the safety property during recovery. Therefore, this approach will not be able to synthesize nonmasking fault-tolerant programs where safety can be violated during recovery. Furthermore, while our algorithm accounts for weak-fairness among program actions and allows for recovery actions to be added under this assumption, the heuristic-based approach does not account for fairness assumptions.

Katz and Perry [97] proposed an algorithm to extend an arbitrary asynchronous distributed message-passing system into a self-stabilizing system. They also gave a formal definition of the self-stabilizing extension of a non-stabilizing program and they defined the set of properties that must be maintained by the new extension. Their algorithm superimposes a control program on the original non-stabilizing program. The control program repeatedly takes a global snapshot and then checks if the snapshot indicates an illegal state. If an illegal state is found, the control program resets the memory of each process to a legal default state.

Arora, Gouda, and Varghese [13] proposed a manual approach to design nonmasking fault-tolerant programs. In this approach, a program is intended to satisfy a set of constraints during normal operation (i.e., no faults). Program actions are categorized into "closure" actions and "convergence" actions. When faults occur and violate one or more of the program constraints, convergence actions are responsible for correcting program behavior and reestablishing those constraints again. This method, however, does not address the issue of automated addition of nonmasking fault-tolerance to existing fault-intolerant programs.

Our approach for adding nonmasking fault-tolerance and self-stabilization is based on satisfying constraints that should be true in legitimate states. An orthogonal approach is to utilize primitives such as distributed reset [97] where one detects whether the system is in a consistent state and resets it to a legitimate state, if needed. Examples of these approaches include [97, 128]. Our approach can be utilized to design the distributed reset protocol itself.

The verification of self-stabilizing properties has been studied by several researchers. One method to verify the correctness of self-stabilizing algorithms is by using mechanical theorem proving. In [121], Qadeer and Shankar used PVS [118] to verify the correctness of Dijekstra's algorithm. Another approach to verify self-stabilizing algorithms was done using model checking. In [129], Tsuchiya et al. applied CTL symbolic model checking techniques to verify several distributed algorithms against self-stabilization properties. They used SMV [113] to overcome the state explosion problem. They showed that the state space can be efficiently reduced using OBDDs. However, they concluded that their approach is applicable only when the number of process is modest.

8.6 Legitimate States Discovery

Several techniques have been developed to verify program correctness [35,36,47,89,93, 113]. For most of those methods, the program is translated into a logical formula that describes the program behavior and properties. Then, tools are used to verify the correctness of the program. For many of these tools, identifying the program legitimate states (i.e., legal or invariant states) is an essential step. Several approaches have been proposed to improve the automatic generation of the legitimate states [19,20,23,109,116]. These methods can be widely classified as either *top-down* or *bottom-up* approaches. The top-down approach starts with the weakest possible invariant and uses program specification to strengthen that invariant. The bottom-up approach performs forward propagations of the program actions to derive the invariant. Our algorithm is a top-down approach since it starts by initializing the largest set of legitimate states to be the whole state space and later removes states that violate the predefined safety and liveness specifications.

Rustan, Leino, and Barnett [19,109] presented methods for forming an efficient weakest precondition to enhance the performance of the verification tools like ESC/Java and ESC/Modula3. Their goal is to simplify the presentation of the weakest pre-condition to avoid redundancy and to avoid exponential growth of the condition size. Our definition of largest set of legitimate states is equivalent to their definition of the weakest conservative preconditions in which the execution of a program statement does not go wrong and it terminates. However, in their work they address the problem of redundancy in describing such conditions while we focus on the automatic generation of such conditions from the program specification.

Jeffords and Heitmeyer [94,116] described an algorithm to automate the generation of the invariant. Their technique is based on deriving the invariant based on propositional formulas derived from the SCR tables. Their algorithm is intended for detecting errors at early stages of program design. By contrast, our algorithm is intended to discover the largest set of legitimate states of programs assumed to be correct for the purpose of adding fault-tolerance to such programs.

The accurate and complete identification of the legitimate states is an essential step that enables designers to apply the algorithms and tools for the automated model revision of fault-tolerant programs from a fault-intolerant programs [27,30,101,111]. Unlike the traditional approaches, that require the explicit specification of the Legitimate States, our approach does not require explicit specification of the Legitimate States but it generates the largest set of legitimate states from program transitions and specification. Therefore, it will significantly improve and simplify the process of automated addition of fault-tolerance. Furthermore, our approach is relatively complete when compared to traditional approaches. Moreover, it does not introduce any significant cost.

Chapter 9

Conclusion and Future Work

In this dissertation, we focused on the problem of automated model revision. We derived theories, developed algorithms, and built tools to make the model revisions more comprehensive, efficient, and designer-friendly. In particular, we reduced the automated model revision learning curve by utilizing existing design tools. Also, we developed algorithms and tools to apply model revision in adding new types of fault-tolerance properties and to automate the generation of the legitimate states of the original model. Finally, we utilized both symmetry and parallelism to speedup the automated revision and to overcome its bottlenecks to reduce its time complexity.

In this chapter, we present a summary of our contributions. In Section 9.1, we summarize the contributions of this dissertation. Then, in Section 9.2 we list some of the future research directions.

9.1 Contributions

This dissertation makes four main contributions:

1. **Reducing the Learning Curve of the Automated Model Revision:** To reduce the learning curve of automated model revision, we focused on utilizing existing design

tools. We combined the automated model revision tool SYCRAFT with the SCR tool set. To achieve successful coupling, we developed a middle layer that translated the SCR specification into SYCRAFT input as well as from SYCRAFT output back to SCR. Thus, our approach gives designers the ability to perform the tasks of the model revision under-the-hood (i.e., while working within the SCR toolset). In this way, they do not need to know all the details required to perform automated model revision.

We expect that the ability to add fault-tolerance under-the-hood is especially useful, as it allows designers to continue to use the design tools they were already using. This reduces the learning curve of the model revision techniques. In the context of SCR, this is especially useful since the SCR toolset has already been adopted by the industry and is used in the development of many real world applications. Furthermore, the SCR toolset integrates several tools for consistency checking, verification, etc. Since synthesized fault-tolerant SCR specification can be viewed/modified using the SCR toolset, one can analyze the revised fault-tolerant SCR specification for various other properties.

With case studies we showed that, for our approach to be effective, certain changes need to be made to the SCR interface. In particular, we demonstrated that the SCR toolset would have to be modified to include the description of faults. However, we showed that the changes required for describing faults in the SCR toolset are straightforward. In particular, the faults themselves could be represented using tables. We also demonstrated that the designer needs to specify the requirements that should be met in the presence of faults. Once again, this is similar to how other requirements (not related to fault-tolerance) are specified in the SCR toolset. These changes to the SCR toolset are reasonable in that they essentially require the designer to specify what the faults are and the requirements for fault-tolerance in the presence of faults.

Additionally, automated revision with SYCRAFT also provides the possibility of detecting errors in the requirements themselves. In particular, one can identify errors caused due to a missing requirement on how recovery can be added. Since SYCRAFT tries to provide maximum non-determinism in the revised program, if a requirement is missing, then there is a high potential that it would be detected. Therefore, this approach provides the ability to reduce cost since it detects errors and missing specifications early in the design stage.

2. **Automating the Discovery of the Legitimate States:** To further reduce the effort required by the designer in automated model revision, we focused on generating one of the inputs - legitimate states - automatically. In particular, the inputs to the model revision algorithms includes: (1) the existing model, (2) the specification of the model, (3) the faults, and (4) the legitimate states of the original model.

   Clearly, specifying the existing model is unavoidable. Moreover, the task required in identifying it is easy, as model revision is expected to be used in contexts where designers already have an existing model. Specification is also already available to the designer when model revision is used in contexts where, existing model fails to satisfy the desired specification. Likewise, the new property that is to be added to the existing model is also easy to identify. In the context of fault-tolerance, this requires the designers to identify the faults that need to be tolerated.

   Based on our experience, the hardest input to identify is the set of legitimate states from where the original model satisfies its specification. In part, it is because of the fact that identifying these legitimate states explicitly is often not required during the evaluation of the original model. Hence, we focused on the problem of automated model revision of an existing model without the use of explicit legitimate states. Moreover, as shown by the example in Section 5.5, typical algorithms for computing legitimate states based on initial states do not work in the context of automated model revision.

We presented an algorithm for automated discovery of the weakest legitimate state predicate of the given program. Our algorithm uses the program actions and specification to automatically generate the weakest legitimate state predicate.

To evaluate this algorithm, we compared the automated model revision when the legitimate states are explicitly specified with that when they are not. We considered three questions in this context: (1) relative completeness, (2) qualitative complexity class comparison, and (3) quantitative change of the time for model revision. We illustrated that our approach for model revision without explicit legitimate states is relatively complete, i.e., if model revision can be solved with explicit legitimate states, then it could also be solved without explicit legitimate states. This is important since it implies that the reduction in the human effort required for model revision does not reduce the class of the problems that could be solved.

Regarding the second question, we found some surprising and counterintuitive results. Specifically, for total revision, we found that the complexity class remains unchanged. However, for partial revision, the complexity class changes substantially. In particular, we showed that problems that could be solved in P when legitimate states are available explicitly become NP-complete if explicit legitimate states are unavailable. This result is especially surprising since this is the first instance where complexity levels for total and partial revision have been found to be different. Even though the general problem of partial revision becomes NP-complete without the explicit legitimate states, we found a subset of these problems that can be solved in P. Specifically, this subset included all instances where model revision was possible when legitimate states are specified explicitly.

Regarding the third question, we showed that the extra computation cost obtained by reducing the human effort for specifying the legitimate states is negligible. Towards this end, we considered four case studies that included Byzantine agreement, mutual exclusion, token ring and diffusing computation. In each of these examples, the generated set of legitimate states was the same as the one specified explicitly in automated addition of fault-tolerance. Moreover, the time to generate the legitimate states was negligible (less than 1%) when compared with the time for performing the corresponding addition of fault-tolerance.

Also, we have integrated the automated revision without explicit legitimate states in the tool SYCRAFT. We note that this result can also be extended to other problems in model revision where one adds safety properties, liveness properties and timing constraints.

3. **Exploiting Parallelism and Symmetry to Expedite the Automated Model Revision:**

   Another contribution of this dissertation is directed towards making the automated model revision more efficient. Specifically, we worked on improving the performance of the automated model revision to synthesize fault-tolerant programs from their fault-intolerant version. Towards this end, we developed techniques that utilize the (1) multi-core processors and (2) the symmetry among the processes of the program being revised to expedite the automated model revision.

   In the case of parallelism, we focused on one of the main complexity barriers, resolution of deadlock states, in automated model revision to add fault-tolerance to distributed programs. Our approach was based on parallelization with multiple threads on a multi-core architecture. We considered parallelization in two scenarios: (1) adding recovery transitions, and (2) eliminating deadlock states. Our approach provides each thread its own copy of shared variables. Although this has a potential to increase the memory usage, in general, automated model revision problems tend to have a higher time complexity than the corresponding verification problems. Hence, we expect that the automated model revision algorithm will *run out of time* before it *runs out of memory*. Hence, the increased space complexity is unlikely to be the bottleneck during revision.

Initially, we showed that the approach of partitioning deadlock states provides a small improvement. And, the approach based on parallelizing the group computation – that is caused by distribution constraints of the program being synthesized– provides a significant benefit that is close to the ideal, i.e., equal to the number of threads used. Additionally, we demonstrated that there is a potential to gain superlinear speedup due to the partitioning of the group computation that reduces the size of corresponding BDDs. Since the configuration used to evaluate performance was on an 8-core (4 dual-cores) machine, we considered the case where up to 16 threads are used. We find that as the number of threads increases, the revision time decreases. In fact, because the parallelism is fine-grained, using more threads than available cores has the potential to improve the performance slightly. This demonstrates that we have not yet reached the bottleneck involved in parallelization. Furthermore, there is potential for further reduction in revision time if the level of parallelism is increased (e.g., if there are more processors). Although, the level of parallelism is fine-grained, we showed that the overhead of parallel computation is small.

In the case of symmetry, we showed that symmetry provides a substantial benefit in reducing the time involved in the revision. More specifically, we observed that multiple processes in a distributed program are symmetric in nature, i.e., their actions are similar (except for the renaming of variables). Thus, if our algorithm finds recovery transitions for a process, then it utilizes symmetry to identify other recovery transitions that should also be included for other processes in the system. Likewise, if some transitions of a process violate safety in the presence of faults, then it identifies similar transitions of other processes that would also violate safety. Since, the cost of identifying these similar transitions with the knowledge of symmetry among processes is less than the cost of identifying these transitions explicitly, then the use of symmetry reduces the overall time required for the revision. Moreover, the speedup increases as the number of symmetric processes increases.

4. **Automating the Model Revision to Add Nonmasking and Stabilizing Fault-Tolerance:** The tools for automated model revision need to be comprehensive and include techniques to automate the addition of different levels of fault-tolerance. In this dissertation, we also focused on the automated revision to add nonmasking and stabilizing fault-tolerance to hierarchical distributed systems. In particular, we considered systems where legitimate states are specified in terms of constraints that are true in legitimate states. The goal of adding nonmasking and stabilizing fault-tolerance was to ensure that if these constraints are violated by faults, then eventually the program would reach a state where all the constraints are satisfied and subsequent behavior would be correct.

   Our approach was to utilize an order among the constraints. With this order, we ensured that correction actions that correct constraint C_i did not cause violation of any of the previous constraints $C_0, C_1 \ldots C_{i-1}$ although they may violate constraints $C_j, j > i$. In our case studies from Chapter 5, we considered different possible orderings and in most cases, we were able to synthesize a nonmasking fault-tolerant program. Therefore, identifying an order among these predicates does not appear to be a critical concern. Moreover, the number of orderings that need to be considered for a group of n constraints will be at most $O(n^2)$. Finally, we find that this approach is especially suited for synthesizing stabilizing programs, since it eliminates one of the bottlenecks of the automated revision (evaluating fault-span).

   Also, we focused on improving the revision to add nonmasking and stabilizing fault-tolerant programs from their fault-intolerant version. We showed that the use of multi-core technology to parallelize the revision algorithm reduces the revision time substantially. We parallelized constraint satisfaction by: (1) partitioning the constraints and (2) utilizing the nature of distributed programs. We showed that parallelism provides a substantial benefit in reducing the time needed in the revision. We illustrated our approach with three case studies: stabilizing mutual exclusion, stabilizing diffusing computation, and a data dissemination problem for sensor networks. The complexity analysis demonstrated that automated model revision in these case studies was feasible and achieved in a reasonable time speedup in all case studies.

Furthermore, since our work is structured on constraint based (manual) design of nonmasking and stabilizing fault-tolerance from [13] that has been found to be useful in deriving several protocols manually (e.g., [73, 75, 128]), we expect that it will be highly valuable for automatically designing various stabilizing and nonmasking programs. We also showed that the hierarchical nature of the underlying system could be effectively utilized to reduce the complexity of synthesizing programs with larger number of processes while maintaining the *correct-by-construction* property of programs designed by automated model revision.

This work also advances the state-of-the-art of the automated model revision in yet another way. To our knowledge, this is the first instance where automated model revision to add fault-tolerance is achieved with fairness constraints. Without fairness constraints, a stabilizing mutual exclusion algorithm based on [124] is impossible. Moreover, the structure of the recovery actions in the first two case studies is too complex to successfully utilize previous heuristic based approaches [30].

9.2 Future Research Directions

During our work on automated model revision we have identified several possible directions of future work. Some of these are listed below.

In Chapter 3, we identified the requirements to complete the revision under-the-hood. Also, we developed middle layer that translate the SCR specification into the SYCRAFT specification. One future research direction in this context is to develop an enhanced version of the middle layer. In particular, a middle layer that can be more generic, i.e., capable of handling several types of specifications other than SCR. We believe that many activities of the automated model revision are not user centric and do not require direct involvement for the user. Furthermore, many software solutions require modification to some of the software properties at several stages in the software life cycle. Moreover, in many cases such software modification is required to be completed in an expedited fashion. These requirements make the ability to perform automated model revision under-the-hood more appealing to many design tools. Hence, one future research direction in this context is investigating of the possibility of integrating automated model revision to other design tools such as Simulink [52] and Rational Rhapsody [81,82]. The enhanced middle layer will also include complete description of the input and output fields. This will allow other developers/researchers to link their design tools with SYCRAFT.

Time complexity is one of the important factors in a successful automated model revision. One future research direction in this context is to combine other advances from program verification. We expect that by combining these advances along with characteristics of distributed systems, e.g., forward reachability analysis, hierarchical behavior, types of expected faults, etc., would be extremely beneficial. Specifically, it will make the automated revision of practical distributed programs to add new properties more feasible.

In Chapter 4, we listed some of the factors that contribute to the time complexity of the automated model revision. Of these, the deadlock resolution problem, is a unique bottleneck and does not exist in other verification methods. However, we recognize that there are other bottlenecks (e.g., forward reachability analysis) that are common with the other verification techniques. Hence, one future work in this context is to incorporate other techniques such as partitioning [35], clustering [123], and saturation-based reachability analysis [39,40] in the automated model revision tools. We expect these techniques to improve computation of many constructs in our tool.

In Chapter 4, we identified the importance of the group computations in automated model revision. In particular, we found that the revision time is often dedicated to computing such groups. Also, since the group computation is caused by distribution constraints of the program being synthesized, as discussed in Chapter 4 and 5, it is guaranteed to be required even with other techniques for expediting automated model revision. One future work is to combine the group parallelism with the techniques that partition the deadlock states among available threads. In particular, as discussed in Chapter 4, the parallelism that partitions the deadlock states is coarse-grained. However, it can permit threads to perform inconsistent behaviors that need to be resolved later. Thus, it provides a tradeoff between overhead of synchrony among threads and potential error resolutions. Hence, even when a large number of cores were available, this approach would be valuable together with other techniques that utilize those additional cores. Thus, one of the future works is to combine the partitioning of the deadlock states and the group parallelism. Also, another future research direction is to explore other approaches to expedite the group computation. For example, it can be used in conjunction with the approach that utilizes symmetry among processes being synthesized.

Another possible future work is developing more efficient algorithms for computing the groups. Due to the distributed nature of the programs being revised, it is most likely the case that the group associated with a given transition gets computed several times. Such repeated computation is not really necessary. In fact, the group associated with a given transition is fixed and does not change during the revision. Therefore, one approach for reducing the time required for computing the groups is as follows. In the initialization stage of the revision algorithm, we compute the groups associated with all the transitions of the program and store them in an efficient data structure. Later and during the revision, whenever it is required to compute the group associated with a given transition, such, group is retrieved from the storage. We expect this approach to significantly reduce the time complexity of the revision. However, it may require more memory and at that point some tradeoffs will need to be made to select the appropriate choices. We also expect that integrating our implementation with a SAT or SMT (satisfiability modulo theories) solver is beneficial. In SMT solvers, one can use other types such as abstract data types, integers, reals etc., in formulae that involve arithmetic and quantifiers as well.

In automated model revision tools, we used BDDs to efficiently represent the model being revised. However, the level of efficiency depends on the order in which we choose to list the variables of the model. Traditionally, such ordering is done manually based on some heuristics to achieve the minimal space required to describe the model. Such, manual approach is sufficient for other approaches for program verification (e.g., model checking) since in verification the model itself does not change. Therefore, the initial order chosen for the variables stays valid. Unlike verification, in model revision the model is modified. For instance, transitions can be removed if they violate the safety, on the other hand, transitions might be added to achieve recovery. Consequently, the initial order of the variables may need to be changed during the revision. One interesting future work is to look for solutions where the order of the variables is dynamic and changes during the revision.

Distributed programs often consist of processes with similar structure. In Chapter 4, we developed some simple yet effective techniques that utilize symmetry to expedite the revision. Also, we demonstrated that the use of symmetry could extremely lower the time required for automated model revision. However, one limitation for our technique is that it requires the designer to identify the symmetry patterns in the program. A future work in this area will involve searching for techniques that allow for automated discovery of such symmetry patterns. An interesting problem would be to exploit the symmetry in distributed programs by automatically identifying symmetrical processes and actions.

In Chapter 5, we demonstrated how the hierarchal structure of the processes could be used to reduce the complexity of the automated model revision. In particular, we showed how we could revise a small model and use the results to revise larger models. One future work in this context is to incorporate techniques that can automatically identify the network topology of the model being revised and use it to complete the revision efficiently.

In the automated model revision to add nonmasking fault-tolerance, we used a set of constraints to describe the legitimate states of the model being revised. The order in which we chose to satisfy these constraints is very important. More specifically, choosing a wrong order may result in the impossibility of finding the correct nonmasking fault-tolerant model. We briefly presented a heuristic that considers all possible combinations to order the constraints. Another future work in this context is to investigate other heuristics that takes into consideration the relation between the constraints them selves. For example, if the set of state identified by a constraint, say C_1, is included in the set of states identified by the constraints, say C_2, then we may need to satisfy C_1 before satisfying C_2.

BIBLIOGRAPHY

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(3):507–535, 1995.

[2] F. Abujarad, B. Bonakdarpour, and S. Kulkarni. Parallelizing Deadlock Resolution in Symbolic Synthesis of Distributed Programs. In *PDMC 2009*, 2009.

[3] F. Abujarad and S. Kulkarni. Automated Addition of Fault-Tolerance to SCR Toolset: A Case Study. In *Distributed Computing Systems Workshops, 2008. ICDCS'08. 28th International Conference on*, pages 539–544, 2008.

[4] F. Abujarad and S. Kulkarni. Constraint Based Automated Synthesis of Nonmasking and Stabilizing Fault-Tolerance. In *Reliable Distributed Systems, 2009. SRDS '09. 28th IEEE International Symposium on, Niagara Falls, New York, USA, Sep 27 - 30, 2009. In Proceedings*, pages 119 – 128, 2009.

[5] F. Abujarad and S. Kulkarni. Multicore Constraint-Based Automated Stabilization. In *Stabilization, Safety, and Security of Distributed Systems: 11th International Symposium, SSS 2009, Lyon, France, November 3-6, 2009. Proceedings*, page 47. Springer, 2009.

[6] F. Abujarad and S. Kulkarni. Weakest Invariant Generation for Automated Addition of Fault-Tolerance. *Electronic Notes in Theoretical Computer Science*, 258(2):3–15, 2009. Available as Technical Report MSU-CSE-09-29 at http://www.cse.msu.edu/cgi-user/web/tech/reports?Year=2009.

[7] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[8] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Computer Aided Verification*, pages 548–562. Springer, 2005.

[9] B. Aminof, T. Ball, and O. Kupferman. Reasoning about systems with transition fairness. *Proc. LPAR, LNCS 3452*, pages 194–208, 2004.

[10] A. Arora. Efficient reconfiguration of trees: A case study in methodical design of nonmasking fault-tolerant programs. In *Science of Computer Programming*. Springer, 1996.

[11] A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. In *Principles of Distributed Computing (PODC)*, pages 173–182, 1998.

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

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

[14] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, 1998.

[15] A. Arora and S. S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Transactions on Software Engineering*, pages 435–450, June 1998.

[16] E. Asarin and O. Maler. As soon as possible: Time optimal control for timed automata. In *Hybrid Systems: Computation and Control (HSCC)*, pages 19–30, 1999.

[17] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *IFAC Symposium on System Structure and Control*, pages 469–474, 1998.

[18] A. Avižienis, J. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, pages 11–33, 2004.

[19] M. Barnett and K. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87. ACM New York, NY, USA, 2005.

[20] S. BensMem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In *Proc. 8th Int. Conf. on Computer-Aided Verification, to appear in Lect. Notes in Comput. Sci.* Springer, 1996.

[21] R. Bharadwaj and C. Heitmeyer. Developing high assurance avionics systems with the SCR requirements method. In *Digital Avionics Systems Conference*, 2000.

[22] R. Bharadwaj and C. Heitmeyer. Developing high assurance avionics systems with the SCR requirements method. In *Digital Avionics Systems Conferences, 2000. Proceedings. DASC. The 19th*, volume 1, 2000.

[23] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[24] G. V. Bochmann. Hardware specification with temporal logic: An example. *IEEE Trans. Comput.*, 31(3):223–231, 1982.

[25] B. Bonakdarpour. *Automated Revision of Distributed and Real-Time Programs*. PhD thesis, Michigan State University, 2008.

[26] B. Bonakdarpour, A. Ebnenasir, and S. Kulkarni. Complexity results in revising UNITY programs. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(1):5, 2009.

[27] B. Bonakdarpour and S. Kulkarni. SYCRAFT: A Tool for Synthesizing Distributed Fault-Tolerant Programs. In *Proceedings of the 19th international conference on Concurrency Theory, August*, pages 19–22. Springer, 2008.

[28] B. Bonakdarpour and S. S. Kulkarni. SYCRAFT: SYmboliC synthesizeR and Adder of Fault-Tolerance. Available at `http://www.cse.msu.edu/~borzoo/sycraft`.

[29] B. Bonakdarpour and S. S. Kulkarni. Automated incremental synthesis of timed automata. In *International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, LNCS 4346, pages 261–276, 2006.

[30] B. Bonakdarpour and S. S. Kulkarni. Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 3–10, 2007.

[31] B. Bonakdarpour, S. S. Kulkarni, and F. Abujarad. Distributed synthesis of fault-tolerance. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2006. Full version available as a Technical Report MSU-CSE-06-27 at Computer Science and Engineering Department, Michigan State University, East Lansing, Michigan.

[32] P. Bouyer, D. D'Souza, P. Madhusudan, and A. Petit. Timed control with partial observability. In *Computer Aided Verification (CAV)*, pages 180–192, 2003.

[33] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

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

[35] J. Burch, E. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. In *International Conference on Very Large Scale Integration*, pages 49–58, 1991.

[36] 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.

[37] R. Burstall. Program proving as hand simulation with a little induction. *Information processing*, 74(308-312):448, 1974.

[38] K. M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[39] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state-space generation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 328–342, 2001.

[40] G. Ciardo and A. J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 146–161, 2005.

[41] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model checker. *Int. J. Softw. Tools Technol. Transf.*, 2(4):410–425, 2000.

[42] E. Clarke. The birth of model checking. *25 Years of Model Checking*, pages 1–26, 2008.

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

[44] E. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1):77–104, 1996.

[45] E. Clarke and L. Liu. Approximate algorithms for optimization of busy waiting in parallel programs (preliminary report). *20th Annual Symposium on Foundations of Computer Science*, pages 255–266, 1979.

[46] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.

[47] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 117–126, New York, NY, USA, 1983. ACM.

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

[49] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. Springer, 1999.

[50] E. Clarke Jr. Synthesis of resource invariants for concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(3):358, 1980.

[51] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *TACAS'03: Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, pages 331–346, Berlin, Heidelberg, 2003. Springer-Verlag.

[52] J. Dabney and T. Harman. *Mastering Simulink*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1997.

[53] E. Dijkstra. *A discipline of programming*. Prentice-Hall, Englewood Cliffs, NJ., 1976.

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

[55] R. Dimitrova and B. Finkbeiner. Synthesis of Fault-Tolerant Distributed Systems. In *Automated Technology for Verification and Analysis: 7th International Symposium, Atva 2009, Macao, China, October 14-16, 2009, Proceedings*, page 321. Springer, 2009.

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

[57] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[58] A. Ebnenasir. DiConic addition of failsafe fault-tolerance. In *Automated Software Engineering (ASE)*, pages 44–53, 2007.

[59] A. Ebnenasir, S. Kulkarni, and A. Arora. FTSyn: A framework for automatic synthesis of fault-tolerance. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(5):455–471, 2008.

[60] A. Ebnenasir, S. S. Kulkarni, and A. Arora. Ftsyn: a framework for automatic synthesis of fault-tolerance. *Int. J. Softw. Tools Technol. Transf.*, 10(5):455–471, 2008.

[61] A. Ebnenasir, S. S. Kulkarni, and B. Bonakdarpour. Revising UNITY programs: Possibilities and limitations. In *International Conference on Principles of Distributed Systems (OPODIS)*, LNCS 3974, pages 275–290, 2005.

[62] E. Emerson and E. Clarke. Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, page 181. Springer-Verlag, 1980.

[63] E. Emerson and J. Y. Halpern. "Sometimes" and "not never" revisited: On branching versus linear time temporal logic. *J. Assoc. Comput. Mach.*, 33:151–178, 1986.

[64] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[65] E. A. Emerson and C. L. Lei. Temporal model checking under generalized fairness constraints. In *Proc. 18th Hawaii International Conference on System Sciences*, pages 277–288, 1985.

[66] J. Ezekiel and G. Lüttgen. Measuring and evaluating parallel state-space exploration algorithms. In *International Workshop on Parallel and Distributed Methods in Verification (PDMC)*, 2007.

[67] J. Ezekiel, G. Lüttgen, and G. Ciardo. Parallelising symbolic state-space generators. In *Computer Aided Verification (CAV)*, pages 268–280, 2007.

[68] J. Ezekiel, G. Lüttgen, and R. Siminiceanu. Can Saturation be Parallelised? *Formal Methods: Applications and Technology*, pages 331–346.

[69] J. Ezekiel, G. Lüttgen, and R. Siminiceanu. Can Saturation be parallelised? on the parallelisation of a symbolic state-space generator. In *International Workshop on Parallel and Distributed Methods of Verification (PDMC)*, pages 331–346, 2006.

[70] M. Faella, S. LaTorre, and A. Murano. Dense real-time games. In *Logic in Computer Science (LICS)*, pages 167–176, 2002.

[71] F. Gartner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys (CSUR)*, 31(1):1–26, 1999.

[72] F. Gartner and A. Jhumka. Automating the addition of fail-safe fault-tolerance: Beyond fusion-closed specifications. *Lecture notes in computer science*, pages 183–198, 2004.

[73] F. Gartner and H. Pagnia. Self-stabilizing load distribution for replicated servers on a per-access basis. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilizing Systems*, pages 102–109, 1999.

[74] P. Godefroid. Using partial orders to improve automatic verification methods. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 176–185, London, UK, 1991. Springer-Verlag.

[75] M. Gouda. Multiphase stabilization. *IEEE Transactions on Software Engineering*, pages 201–208, 2002.

[76] M. G. Gouda. The triumph and tribulation of system stabilization. In *Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 1–18. Springer-Verlag London, UK, 1995.

[77] O. Grumberg, T. Heyman, N. Ifergan, and A. Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 129–145, 2005.

[78] O. Grumberg, T. Heyman, and A. Schuster. A work-efficient distributed algorithm for reachability analysis. *Formal Methods in System Design (FMSD)*, 29(2):157–175, 2006.

[79] O. Grumberg and D. Long. Model checking and modular verification. In *CONCUR'91*, pages 250–265. Springer, 1991.

[80] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, 1994.

[81] D. Harel and H. Kugler. The rhapsody semantics of statecharts. *Lecture notes in computer science*, pages 325–354, 2004.

[82] D. Harel and H. Kugler. The rhapsody semantics of statecharts. *Lecture notes in computer science*, pages 325–354, 2004.

[83] M. Heimdahl and N. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE transactions on Software Engineering*, 22(6):363–377, 1996.

[84] C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. Tools for constructing requirements specifications: The SCR toolset at the age of ten. *International Journal of Computer Systems Science and Engineering*, 20(1):19–35, 2005.

[85] C. Heitmeyer and R. Jeffords. Applying a Formal Requirements Method to Three NASA Systems: Lessons Learned. In *2007 IEEE Aerospace Conference*, pages 1–10, 2007.

[86] C. Heitmeyer, J. Kirby, and B. Labaw. Tools for formal specification, verification, and validation ofrequirements. In *Computer Assurance, 1997. COMPASS97. Are We Making Progress Towards Computer Assurance? Proceedings of the 12th Annual Conference on*, pages 35–47, 1997.

[87] C. Heitmeyer, J. Kirby, B. Labaw, R. Bharadwaj, et al. SCR*: A toolset for specifying and analyzing software requirements, 1998.

[88] C. Heitneter and J. McLean. Abstract requirements specification: A new approach and its application. *IEEE Transactions on Software Engineering*, pages 580–589, 1983.

[89] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[90] M. Herlihy. The future of distributed computing: Renaissance or reformation? In *Twenty-Seventh Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2008)*, 2008.

[91] S. Hester, D. Parnas, and D. Utter. Using documentation as a software design medium. *Bell System Tech. J*, 60(8):1941–1977, 1981.

[92] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Computer-Aided Verification (CAV)*, pages 20–35, 2000.

[93] G. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 1997.

[94] R. Jeffords and C. Heitmeyer. An algorithm for strengthening state invariants generated from requirements specifications. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE'01)*. IEEE Computer Society Washington, DC, USA, 2001.

[95] R. Jeffords and C. Heitmeyer. A strategy for efficiently verifying requirements. *ACM SIGSOFT Software Engineering Notes*, 28(5):28–37, 2003.

[96] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Computer Aided Verification (CAV)*, pages 226–238, 2005.

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

[98] T. Kletz. *Hazop and Hazan: Identifying and assessing process industry hazards*. Inst of Chemical Engineers, 1999.

[99] F. Kröger. Lar: A logic of algorithmic reasoning. *Acta Inf.*, 8:243–266, 1977.

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

[101] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 82–93, 2000.

[102] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of Byzantine agreement. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 130–140, 2001.

[103] S. S. Kulkarni, A. Arora, and A. Ebnenasir. *Software Engineering and Fault-Tolerance*, chapter Adding Fault-Tolerance to State Machine-Based Designs. World Scientific Publishing Co. Pte. Ltd, 2007.

[104] S. S. Kulkarni and M. Arumugam. Infuse: A TDMA based data dissemination protocol for sensor networks. *International Journal of Distributed Sensor Networks*, 2(1):55–78, 2006.

[105] S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. *International Conference on Distributed Computing Systems*, 2003.

[106] R. Kurshan and K. McMillan. A structural induction theorem for processes. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 239–247. ACM, 1989.

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

[108] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 1982.

[109] K. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[110] L. Liu and E. Clarke. Optimization of busy waiting in conditional critical regions. *13th Hawaii International Conference on System Sciences*, 1980.

[111] H. Mantel and F. C.Gärtner. A case study in the mechanical verification of fault-tolerance. Technical Report TUD-BS-1999-08, Department of Computer Science, Darmstadt University of Technology, 1999.

[112] K. L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[113] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[114] S. Meyer and S. White. Software requirements methodology and tool study for A6-E technology transfer. Technical Report MSU-CSE-09-21, Grumman Aerospace Corp., Bethpage, NY, 1983.

[115] K. Milvang-Jensen and A. J. Hu. BDDNOW: A parallel BDD package. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 501–507, 1998.

[116] J. Nimmer and M. Ernst. Automatic generation of program specifications. *ACM SIGSOFT Software Engineering Notes*, 27(4):229–239, 2002.

[117] C. Norris Ip and D. Dill. Better verification through symmetry. *Formal methods in system design*, 9(1):41–75, 1996.

[118] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[119] P. Palady. *Failure modes and effects analysis*. PT Publications Inc, 1995.

[120] D. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer programming*, 25(1):41–61, 1995.

[121] S. Qadeer and N. Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In D. Gries and W.-P. de Roever, editors, *IFIP International Conference on Programming Concepts and Methods (PROCOMET '98)*, pages 424–443, Shelter Island, NY, June 1998. Chapman & Hall.

[122] P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[123] R. Ranjan, A. Aziz, R. Brayton, B. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. In *IEEE/ACM International Workshop on Logic Synthesis*, 1995.

[124] K. Raymond. A tree based algorithm for mutual exclusion. *ACM Transactions on Computer Systems*, 7:61–77, 1989.

[125] F. Somenzi. CUDD: Colorado University Decision Diagram Package. `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`.

[126] T. Stornetta and F. Brewer. Implementation of an efficient parallel BDD package. In *Proceedings of the 33rd annual Design Automation Conference*, pages 641–644. ACM, 1996.

[127] T. Stornetta and F. Brewer. Implementation of an efficient parallel BDD package. In *Design automation (DAC)*, pages 641–644, 1996.

[128] O. Theel and F. Gartner. An exercise in proving convergence through transfer functions. In *Proc. 4th Workshop on Self-stabilizing Systems, Austin, Texas*, pages 41–47, 1999.

[129] T. Tsuchiya, S. Nagano, R. B. Paidi, and T. Kikuno. Symbolic model checking for self-stabilizing algorithms. *IEEE Trans. Parallel Distrib. Syst.*, 12(1):81–95, 2001.

[130] W. Vesely. Fault tree handbook. us nuclear regulatory committee report nureg-0492, us nrc, washington dc, united states., 1981.

[131] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 68–80, London, UK, 1990. Springer-Verlag.