AUTOMATIC SYNTHESIS OF FAULT-TOLERANCE

By

Ali Ebnenasir

A Dissertation

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

Doctor of Philosophy

Department of Computer Science and Engineering

2005

ABSTRACT

AUTOMATIC SYNTHESIS OF FAULT-TOLERANCE

By

Ali Ebnenasir

Fault-tolerance is an important property of today's software systems as we rely on computers in our daily affairs (e.g., medical equipments, transportation systems, etc). Since it is difficult (if not impossible) to anticipate all classes of faults that perturb a program while designing that program, it is desirable to incrementally add fault-tolerance concerns to an existing program as we encounter new classes of faults. Hence, in this dissertation, we concentrate on automatic addition of fault-tolerance to (distributed) programs; i.e., synthesizing fault-tolerant programs from their fault-intolerant version. Such automated synthesis generates a fault-tolerant program that is correct by construction, thereby alleviating the need for its proof of correctness. Also, there exists a potential for reusing the computations of the fault-intolerant program during the synthesis of its fault-tolerant version.

In the absence of faults, the synthesized fault-tolerant program should behave similar to the fault-intolerant program. In the presence of faults, the synthesized fault-tolerant program has to provide a desired level of fault-tolerance, namely failsafe, nonmasking, or masking fault-tolerance. A *failsafe* fault-tolerant program guarantees safety even in the presence of faults. In the presence of faults, a *nonmasking* fault-tolerant program recovers to states from where its safety and liveness specifications are satisfied. A *masking* fault-tolerant program always satisfies safety and recovers to states from where its safety and liveness specifications are satisfied.

To provide a foundation for automatic synthesis of fault-tolerant programs, we concentrate on two directions: theoretical aspects, and the development of a software framework for the synthesis of fault-tolerant programs. The main contributions of the dissertation regarding theoretical aspects are as follows:

- We identify the effect of safety specification modeling on the complexity of synthesizing fault-tolerant programs from their fault-intolerant version.
- We show the NP-completeness proof of synthesizing failsafe fault-tolerant distributed programs from their fault-intolerant version.
- We identify the sufficient conditions for polynomial-time synthesis of failsafe fault-tolerant distributed programs.
- We design a sound and complete synthesis algorithm for enhancing the fault-tolerance of high atomicity programs – where program processes can atomically read/write all program variables – from nonmasking to masking.
- We present a sound algorithm for enhancing the fault-tolerance of distributed programs – where program processes have read/write restriction with respect to program variables.
- We present a synthesis method for providing reuse in the synthesis of different programs where we automatically specify and add pre-synthesized fault-tolerance components to programs.
- We define and address the problem of synthesizing *multitolerant* programs that are subject to multiple classes of faults and provide (possibly) different levels of fault-tolerance corresponding to each fault-class.

To validate our theoretical results, we develop an extensible software framework, called Fault-Tolerance Synthesizer (FTSyn), where *developers of fault-tolerance* can interactively synthesize fault-tolerant programs. Also, FTSyn provides a platform for *developers of heuristics* to extend FTSyn by integrating their heuristics for the addition of fault-tolerance in FTSyn. Using FTSyn, we have synthesized several fault-tolerant distributed programs that demonstrate the applicability of FTSyn for the cases where we have different types of faults, and for the cases where a program is subject to multiple simultaneous faults.

To my parents and my wife for all their love and sacrifices.

Acknowledgments

All thanks go to the almighty God who has endowed us the blessing of existence. I extend my regards to all people who have contributed to my education in anyway from primary school to higher educations. First, I am truly grateful to Dr. Sandeep Kulkarni whose guidance was always enlightening throughout my PhD program. Also, I would like to thank the members of my PhD committee, Dr. Laura Dillon, Dr. Betty Cheng, and Dr. Jonathan Hall, who have always supported me by their valuable comments. Furthermore, I appreciate all the efforts of the Computer Science and Engineering Department at Michigan State University towards creating a productive environment for research and education.

I am also obliged to thank my teachers and advisors to whom I am in debt for all their hard work and sacrifices for educating me and my fellow students: Dr. Mohsen Sharifi my advisor in my Master program at Iran University of Science and Technology, Tehran - Iran, Dr. Abbas Vafaei and Dr. Mustafa Kermani my professors at the University of Isfahan, Isfahan - Iran, Mr. Fereydani, Mr. Khayyam, Mr. Nahvi, Mr. Riazi, and Mr. Nasr my teachers in high school, Mr. Saljooghian in middle school, and finally my first grade teacher Mrs. Afshari.

Last but not least, I would like to thank my fellow graduate students at the Software Engineering and Network Systems Laboratory at Michigan State University who have always supported me by (i) proofreading my manuscripts, (ii) providing valuable feedback on my research work, (iii) engaging in discussions, and (iv) attending my not-so-attractive talks. In particular, I appreciate the sincere collaboration of Laura Anne Campbell, Karun Biyani, Bru Bezawada, Sascha Konrad, Mahesh Arumugam, and Borzoo Bonakdarpour.

Thank you.

Table of Contents

List of Figures

Chapter 1

Introduction

The anticipation of all classes of faults that may perturb a program is difficult (if not impossible). Thus, it is desirable to synthesize fault-tolerant programs from their fault-intolerant version upon finding new classes of faults. Although there exist efficient approaches [1] for the synthesis of high atomicity fault-tolerant programs – where processes can read/write all program variables in an atomic step, there exists a well-defined need for developing efficient techniques for the synthesis of (i) fault-tolerant *distributed* programs – where processes have read/write restrictions with respect to program variables, and (ii) *multitolerant* programs – where a program simultaneously provides different levels of fault-tolerance to different classes of faults. In this dissertation, we concentrate on the *theoretical* and the *practical* aspects of synthesizing fault-tolerant distributed programs and multitolerant programs.

To synthesize a fault-tolerant program from its fault-intolerant version, Kulkarni and Arora [1] present a synthesis method that takes a given class of faults and a fault-intolerant program, and generates a program that is fault-tolerant to that class of faults. The fault-intolerant program satisfies its (safety and liveness) specification in the absence of faults and provides no guarantees in the presence of faults. The synthesized fault-tolerant program provides a desired level of fault-tolerance in the presence of faults, and satisfies the safety and liveness specification of the fault-intolerant program in the absence of faults.

Such synthesis approach has the potential to reuse the computations of the fault-intolerant program during the synthesis of its fault-tolerant version. As a result, reusing the computations of a fault-intolerant program preserves its important properties (e.g., efficiency) that are difficult to specify in a *specification-based* approach (e.g., [2, 3, 4]) where one synthesizes a fault-tolerant program from its temporal logic (respectively, automata-theoretic [5, 6, 7]) specification.

The synthesized fault-tolerant program provides one of the three levels of fault-tolerance namely, failsafe, nonmasking, and masking [1]. Intuitively, in the presence of faults, a failsafe fault-tolerant program ensures that its safety specification is satisfied. In the presence of faults, a nonmasking fault-tolerant program recovers to states from where its safety and liveness specification is satisfied. A masking fault-tolerant program guarantees that in the presence of faults it recovers to states from where its safety and liveness specification is satisfied while preserving safety during recovery.

The complexity of the synthesis presented in [1] depends on the program model. The authors of [1] show that the complexity of synthesis is polynomial in the state space of the fault-intolerant program in the high atomicity model. For distributed programs (i.e., low atomicity model), Kulkarni and Arora show that the complexity of synthesizing *masking* fault-tolerance is exponential. Also, in the specification-based approach, the synthesis of fault-tolerant distributed programs (with particular architectures) from their specification is known to be non-elementary decidable [6, 7].

A survey of the literature [7, 8] reveals that the complexity of synthesis and the inefficiency of the synthesized programs construct the main obstacles in the automated synthesis of fault-tolerant programs. Moreover, to the best of our knowledge, no automated approach has been presented for adding *multitolerance* to programs where a multitolerant program is subject to multiple classes of faults and provides (possibly) different levels of fault-tolerance corresponding to different classes of faults. Hence, in this dissertation, we focus our attention on theoretical and practical problems in the synthesis of fault-tolerant *distributed* programs and *multitolerant* programs.

Theoretical problems. Regarding theoretical aspects of synthesis, we address the following problems:

- *Identify the effect of safety specification model on the complexity of synthesis*

  It is shown in the literature that the complexity of adding fault-tolerance to high atomicity programs is polynomial in the state space of the fault-intolerant program if the safety specification is represented as a set of bad transitions [1]. In [9], the authors conjecture that representing safety specification as a set of *sequences of transitions* results in exponential complexity for adding fault-tolerance. They validate their claim in the context of some examples. However, to the best of our knowledge, there exist no significant result to verify the claim made in [9]. Thus, it is desirable to explore the complexity of synthesis in the case where safety specification is represented as a set of *sequences of transitions*. The significance of such complexity analysis is in that it identifies the appropriate approach for modeling safety specification where automatic addition of fault-tolerance can be done efficiently.

- *Find sufficient conditions for polynomial-time synthesis of distributed programs*

  Since the complexity of synthesizing fault-tolerant distributed programs from their fault-intolerant version is exponential [1], we shall identify properties of programs and specifications where the synthesis can be done in polynomial time.

- *Reduce the complexity of synthesis by reusing the computations of the fault-intolerant program*

  During the synthesis of fault-tolerant programs, there exist situations where the computational structure of the fault-intolerant program provides necessary means for satisfying fault-tolerance requirements in the presence of faults. Thus, it is desirable to design synthesis algorithms that take advantage of such situations to reduce the complexity of synthesis.

- *Identify and reuse pre-synthesized fault-tolerance components*

  There exist *recurring* sub-problems of synthesis that arise in the synthesis of different programs (e.g., resolving deadlock states). Thus, it is desirable to generalize the solution to common synthesis problems so that we can develop generic solution strategies that are independent of the program at hand. In other words, we would like to reuse the effort put in the synthesis of one program for the synthesis of another program. To achieve this goal, we plan to identify commonly encountered patterns in the synthesis of programs in order to encapsulate those patterns in the form of pre-synthesized fault-tolerance components. Also, we would like to devise a synthesis method where we automatically specify and add the required pre-synthesized components to the fault-intolerant programs.

- *Synthesize programs that tolerate multiple classes of faults and provide different levels of fault-tolerance to each fault-class*

  Dependable and fault-tolerant systems are often subject to multiple classes of faults, and hence, these systems need to provide appropriate level of fault-tolerance to each class of faults. Often it is undesirable or impractical to provide the same level of fault-tolerance to each class of faults. Hence, these systems need to tolerate multiple classes of faults, and provide a (possibly) different level of fault-tolerance to each class. To characterize such systems, the notion of multitolerance was introduced in [10]. The importance of such multitolerant systems can be easily observed from the fact that several methods for designing multitolerant programs as well as several instances of multitolerant programs can be found (e.g., [11, 12, 13, 10]) in the literature.

Automated synthesis of multitolerant programs has the advantage of generating fault-tolerant programs that (i) are correct by construction, and (ii) tolerate multiple classes of faults. However, the complexity of such synthesis is an obstacle in the synthesis of multitolerant programs. Specifically, there exist situations where satisfying a specific fault-tolerance requirement for one class of faults conflicts with providing a different level of fault-tolerance to another fault-class. Hence, it is necessary to identify situations where synthesis of multitolerant programs can be performed efficiently and where heuristics need to be developed for adding multitolerance.

Practical problems. To reduce the exponential complexity of synthesis for practical purposes and to enable the synthesis of programs that have large state space, heuristic-based approaches are proposed in [14, 15, 9]. These heuristic-based approaches reduce the complexity of synthesis by forfeiting the completeness of synthesizing fault-tolerant distributed programs. In other words, if heuristics are applicable then a heuristic-based algorithm will generate a fault-tolerant program efficiently. However, if the heuristics are not applicable then the synthesis algorithm will declare failure even though it is possible to synthesize a fault-tolerant program from the given fault-intolerant program.

The development and the implementation of heuristics are complicated by the fact that, for a given heuristic, we need to determine how that heuristic reduces the complexity of synthesizing fault-tolerant distributed programs. Furthermore, we need to identify if a heuristic is so restrictive that its use will cause the synthesis algorithm to declare failure very often. Also, in order to provide maximum efficiency, there exist situations where we need to apply heuristics in a specific order. Moreover, the developers of a fault-tolerant program may have additional insights about the order in which heuristics should be applied. Thus, we have to provide the possibility of changing the order of available heuristics (respectively, adding new heuristics) for the developers of fault-tolerance.

Therefore, there exists a substantial need for an extensible software framework where (i) *developers of fault-tolerant programs* can synthesis fault-tolerant programs from their fault-intolerant version; (ii) *developers of heuristics* can integrate new heuristics into the framework or modify exiting heuristics, and (iii) developers can benefit from existing automated reasoning tools (e.g., SAT solvers) in the synthesis of fault-tolerant distributed programs.

1.1 The Outline of the Dissertation

In Chapter 2, we present preliminary concepts of programs, specifications, faults, and fault-tolerance. We also describe synthesis algorithms presented by Kulkanri and Arora [1] in Chapter 2 as we reuse those algorithms in this dissertation. Then, we identify the effect of specification modeling on the complexity of synthesis in Chapter 3. Subsequently, in Chapter 4, we show that synthesizing a failsafe fault-tolerant distributed program from its fault-intolerant version is NP-complete. We also present sufficient conditions for polynomial synthesis of failsafe fault-tolerant distributed programs. In Chapter 5, we define the enhancement problem where we enhance the level of fault-tolerance from nonmasking to masking in polynomial time. We introduce the concept of pre-synthesized fault-tolerance components in Chapter 6, where we present a synthesis method for automatic specification and addition of pre-synthesized fault-tolerance components to programs during synthesis. Afterwards, in Chapter 7, we formally state the problem of adding multitolerance to programs, and we show that, in general, synthesizing multitolerant programs from their fault-intolerant version is NP-complete even in the high atomicity model. In Chapter 8, we present the design of our software framework for automatic synthesis of fault-tolerant distributed programs. In Chapter 9, we present some ongoing research work.

Finally, in Chapter 10, we discuss related work, contributions, and the impact of this dissertation, and then we make concluding remarks.

Chapter 2

Preliminaries

In this chapter, we present formal definitions of programs, problem specifications, faults, fault-tolerance, and addition of fault-tolerance. Specifically, in Section 2.1, we present the formal definition of programs, state predicates, and projection of program transitions on a state predicate. In Section 2.2, we present the issues of modeling distributed programs that is adapted from [1, 4]. Then, in Section 2.3, we adapt the definition of specifications from Alpern and Schneider [16]. In Sections 2.4 and 2.5, we adapt the definition of faults and fault-tolerance from Arora and Gouda [17] and Kulkarni [18]. We represent the problem of adding fault-tolerance to fault-intolerant programs in Section 2.6. We have adapted the problem statement of fault-tolerance addition from [1]. In Section 2.7, we reiterate the results presented in [1] for the synthesis of fault-tolerant programs in high atomicity model – where processes can read/write all program variables in an atomic step. Finally, in Section 2.8, we recall the results presented in [1] for the synthesis of distributed programs – where processes have read/write restrictions with respect to program variables.

2.1 Program

A program p is specified by a finite set of variables, say $V = \{v_0, v_2, .., v_q\}$, and a finite set of processes, say $P = \{P_0, \cdots, P_n\}$, where q and n are positive integers. Each variable is associated with a finite domain of values. Let $v_0, v_2, .., v_q$ be variables of p, and let $D_0, D_2, .., D_q$ be their respective domains.

A state of p is obtained by assigning each variable a value from its respective domain. Thus, a state s of p has the form: $\langle l_0, l_1, .., l_q \rangle$ where $\forall i : 0 \leq i \leq q : l_i \in D_i$. The state space of p, S_p, is the set of all possible states of p.

A process, say P_j, consists of a set of transitions δ_j; each transition has the form (s_0, s_1) where $s_0, s_1 \in S_p$. A process P_j in p is associated with a set of variables, say r_j, that P_j can read and a set of variables, say w_j, that P_j can write. The transitions of program p, δ_p, is the union of the transitions of its processes. In most situations in this dissertation, we focus on the entire state space of a program and all its transitions. Hence, for simplicity, we rewrite program p as the tuple $\langle S_p, \delta_p \rangle$, where S_p is a finite set of states and δ_p is a subset of $S_p \times S_p$.

A state predicate X of p is any subset of S_p. We denote the cardinality of X by $|X|$, where $|X|$ represents the number of states in X. A state predicate X is closed in a program p (respectively, δ_p) iff (if and only if) the following condition holds.

$$\forall s_0, s_1 :: ((s_0, s_1) \in \delta_p) \Rightarrow (s_0 \in X \Rightarrow s_1 \in X)$$

A transition predicate Δ_p of p is any subset of $S_p \times S_p$. We denote the cardinality of Δ_p by $|\Delta_p|$, where $|\Delta_p|$ represents the number of transitions in Δ_p.

A sequence of states, $\sigma = \langle s_0, s_1, ... \rangle$, is a computation of p iff the following two conditions are satisfied (i.e., a computation is *maximal*):

1. If σ is infinite then $\forall j : j > 0 : (s_{j-1}, s_j) \in \delta_p$, and

2. If σ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in \delta_p$, and $\forall j : 0 < j \leq l : (s_{j-1}, s_j) \in \delta_p$.

A sequence of states, $\langle s_0, s_1, ..., s_n \rangle$, is a computation prefix of p iff $\forall j : 0 < j \leq n : (s_{j-1}, s_j) \in \delta_p$; i.e., a computation prefix need not be maximal.

The projection of program p on state predicate X, denoted as $p|X$, is the program $\langle S_p, \{(s_0,s_1) : (s_0,s_1)\in\delta_p \ \wedge \ s_0, s_1 \in X\}\rangle$. In other words, $p|X$ consists of transitions of p that start in X and end in X. Given two programs, $p=\langle S_p, \delta_p\rangle$ and $p'=\langle S_{p'}, \delta_{p'}\rangle$, we say $p' \subseteq p$ iff $S_{p'}=S_p$ and $\delta_{p'} \subseteq \delta_p$.

Notation. When it is clear from the context, we use p and δ_p interchangeably. Also, we say that a state predicate X is true in a state s iff $s\in X$.

2.2 Issues of Distribution

In this section, we present the issues that distribution introduces during the addition of fault-tolerance. More specifically, we identify how read/write restrictions on a process affect its transitions.

Write restrictions. Given a transition (s_0, s_1) of a program p, we can easily identify the variables that need to be changed in order to modify the state of p from s_0 to s_1. Hence, if process P_j can write only the variables in w_j and the value of a variable $x \notin w_j$ is changed in transition (s_0, s_1) then (s_0, s_1) cannot be used in obtaining the transitions of P_j. In other words, if P_j can write only variables in w_j then P_j cannot use the transitions in $nw(w_j)$, where

$$nw(w_j) = \{(s_0, s_1) : (\exists x : x \notin w_j : x(s_0) \neq x(s_1))\}$$

w_j is the set of variables that process P_j is allowed to write.

Notation. $x(s_0)$ represents the value of a variable x in state s_0.

Read restrictions. Given a single transition (s_0, s_1), the program p must read all the variables in order to execute (s_0, s_1). For this reason, read restrictions require us to group transitions and ensure that the entire group is included or the entire group is excluded. As an example, consider a program consisting of two variables a and b, with domains $\{0, 1\}$. Suppose that we have a process that cannot read b. Now, observe that the transition from the state $\langle a = 0, b = 0\rangle$ to $\langle a = 1, b = 0\rangle$ can be included iff the transition from $\langle a=0, b=1\rangle$ to $\langle a=1, b=1\rangle$ is also included. If we were to include only one of these transitions then we would need to read both a and b. However, when these two transitions are *grouped*, the value of b becomes irrelevant, and hence, we do not need read it.

More generally, consider the case where r_j is the set of variables that P_j can read, w_j is the set of variables that P_j can write, and $w_j \subseteq r_j$. (In this dissertation, we assume that $w_j \subseteq r_j$; i.e., j cannot *blindly* write any variable. A more general case is discussed in [1]; we omit it here as this case suffices for our presentation.) Now, process P_j can include the transition (s_0, s_1) iff P_j also includes the transition (s_0', s_1') where s_0 (respectively, s_1) and s_0' (respectively, s_1') are identical as far as the variables in r_j are considered, and s_0 (respectively, s_0') and s_1 (respectively, s_1') are identical as far as the variables not in r_j are considered. We define these transitions as $group(r_j)(s_0, s_1)$ for the case $w_j \subseteq r_j$, where

$$\begin{aligned} group(r_j)(s_0, s_1) = \{(s_0', s_1') : (\forall x : x \in r_j : x(s_0) = x(s_0') \ \wedge x(s_1) = x(s_1')) \ \wedge \\ (\forall x : x \notin r_j : x(s_0') = x(s_1') \ \wedge \ x(s_0) = x(s_1)) \ \} \end{aligned}$$

2.3 Specification

A specification is a set of infinite sequences of states that is suffix-closed and fusion-closed. Suffix-closure of the set means that if a state sequence σ is in that set then so are all the suffixes of σ. Fusion-closure of the set means that if state sequences $\langle \alpha, s, \gamma\rangle$ and $\langle \beta, s, \delta\rangle$ are in that set then so are the state sequences $\langle \alpha, s, \delta\rangle$ and $\langle \beta, s, \gamma\rangle$, where α and β are finite prefixes of state sequences, γ and δ are suffixes of state sequences, and s is a program state. Intuitively, fusion closure of the specification means that an implementation of the specification must execute its next transition only based on its current state; i.e., the history of a computation does not affect the next move of the program.

Following Alpern and Schneider [16], we rewrite the specification as a conjunction of a safety specification and a liveness specification. For a suffix-closed specification, the safety specification can be specified as a set of bad transitions [18] that must not occur in program computations; that is, for program p, its safety specification is a subset of $S_p \times S_p$. To investigate the effect of the safety specification model on the complexity of synthesis, we show, in Chapter 3, that if the specification is represented as a set of computation prefixes (i.e., a set of finite sequences of transitions), the complexity of synthesis significantly increases to a higher complexity class. Hence, except in Chapter 3, in the rest of this dissertation, we represent safety specification of programs as a set of bad transitions.

In the synthesis algorithms presented in this dissertation, we do not require the explicit specification of the liveness properties. More specifically, we require that, in the absence of faults, the synthesized fault-tolerant program satisfies the liveness specification of the fault-intolerant program. In the presence of faults, the fault-tolerant program must satisfy desired fault-tolerance properties defined in Section 2.5.

Given a program p, a state predicate S, and a specification *spec*, we say that p satisfies *spec* from S iff (1) S is closed in p, and (2) every computation of p that starts in a state of S is in *spec*. If p satisfies *spec* from S and $S \neq \{\}$, we say that S is an invariant of p for spec.

For a finite sequence (of states) α, we say that α maintains (does not violate) *spec* iff there exists a sequence of states β such that $\alpha\beta \in$ *spec*. We say that p maintains (does not violate) *spec* from a state predicate X iff (1) X is closed in p, and (2) every computation prefix of p that starts in a state in X maintains *spec*.

2.4 Fault

We systematically represent the faults that perturb a program by a set of transitions. A class of faults f for program $p = \langle S_p, \delta_p \rangle$ is a subset of the set $S_p \times S_p$. We use $p[]f$ to denote the transitions obtained by taking the union of the transitions in p and the transitions in f (i.e., $\delta_p \cup f$). We say that a state predicate T is an f-span (read as fault-span) of p from S iff the following two conditions are satisfied: (1) $S \subseteq T$, and (2) T is closed in $p[]f$. Observe that for all computations of p that start at states where S 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 transitions in f.

Now, we define the computations of p in the presence of faults, f. We say that a sequence of states, $\sigma = \langle s_0, s_1, ... \rangle$, is a computation of p in the presence of f iff the following three conditions are satisfied.

1. If σ is infinite then $\forall k : k > 0 : (s_{k-1}, s_k) \in (\delta_p \cup f)$,
2. If σ 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. $\exists n : n \geq 0 : (\forall k : k > n : (s_{k-1}, s_k) \in \delta_p)$.

The first requirement captures that in each step, either a program transition or a fault transition is executed. The second requirement captures that faults do not have to execute; i.e., if the program reaches a state where only a fault transition can be executed then the fault transition need not be executed. It follows that fault transitions cannot be used to deal with deadlocked states. Finally, the third requirement captures that the number of fault occurrences in a computation is finite. Such assumption also appears in previous work [19, 20, 17, 21].

Program and faults representation. We use Dijkstra's guarded commands [22] to represent the transitions of programs and faults. A guarded command (action) is of the form $grd \rightarrow st$, where grd is a state predicate and st is a function from S_p to S_p (i.e., an assignment) that updates program variables. Specifically, the guarded command $grd \rightarrow st$ represents the following set of transitions:

$$\{(s_0, s_1) : grd \text{ is true at } s_0 \text{ and the } \textit{atomic} \text{ execution of } st \text{ at } s_0 \text{ takes the program to state } s_1\}$$

2.5 Fault-Tolerance

In this section, we formally define what it means for a program to be fault-tolerant. We define three levels of fault-tolerance; failsafe, nonmasking, and masking. In the absence of faults, irrespective of the level of fault-tolerance, a program should satisfy its specification, say *spec*, from its invariant. The level of fault-tolerance characterizes the extent to which the program satisfies *spec* in the presence of faults. Intuitively, a failsafe fault-tolerant program ensures that in the presence of faults, the safety of *spec* is maintained. A nonmasking fault-tolerant program ensures that in the presence of faults, the program recovers to states from where *spec* is satisfied. A masking fault-tolerant program ensures that in the presence of faults the safety of *spec* is maintained and the program recovers to states from where *spec* is satisfied. Thus, we formally define these three levels of fault-tolerance for a program p, its invariant S, its specification *spec*, and a class of faults f as follows:

Program p is failsafe f-tolerant for *spec* from S iff the following two conditions hold: (1) p satisfies *spec* from S, and (2) there exists T such that T is an f-span of p from S and $p[]f$ maintains *spec* from T.

Program p is nonmasking f-tolerant for *spec* from S iff the following two conditions hold: (1) p satisfies *spec* from S, and (2) there exists T such that T is an f-span of p from S and every computation of $p[]f$ that starts from a state in T has a state in S.

Program p is masking f-tolerant for *spec* from S iff the following two conditions hold: (1) p satisfies *spec* from S, and (2) there exists T such that T is an f-span of p

from S, $p[]f$ maintains *spec* from T, and every computation of $p[]f$ that starts from a state in T has a state in S .

Note that a specification is a set of infinite sequences of states. Hence, if p satisfies *spec* from S then all computations of p that start in S must be infinite. In the context of nonmasking and masking fault-tolerance, every computation from the fault-span reaches a state in its invariant. Hence, if fault-span T is used to show that p is nonmasking (respectively, masking) f-tolerant for *spec* from S then all computations of p that start in a state in T must also be infinite. Also, note that p is allowed to contain a self-loop of the form (s_0, s_0); we use such a self-loop whenever s_0 is an *acceptable fixpoint* of p.

Notation. Henceforth, whenever the program p is clear from the context, we will omit it; thus, "S is an invariant" abbreviates "S is an invariant of p" and "f is a fault" abbreviates "f is a fault for p". Also, whenever the specification *spec* and the invariant S are clear from the context, we omit them; thus, "f-tolerant" abbreviates "f-tolerant for *spec* from S".

2.6 The Problem of Adding Fault-Tolerance

In this section, we reiterate the problem of adding fault-tolerance presented in [1]. The addition problem requires a fault-tolerant program p' (with its invariant S') to behave similar to its fault-intolerant version, say p, in the absence of a given class of faults f. In the presence of f, p' must provide a desired level of fault-tolerance, say $\mathcal{L}$, where $\mathcal{L}$ could be failsafe, nonmasking, or masking. Since p' must behave similar to p in the absence of faults, Kulkarni and Arora [1] stipulate the following conditions:

1. S' must be a subset of S. Otherwise, if there exists a state $s \in S'$ where $s \notin S$ then, *in the absence of faults*, p' can reach s and create new computations that do not belong to p. Thus, p' will include new ways of satisfying *spec* from s in the absence of faults.

2. $p'|S'$ must be a subset of $p|S'$. If $p'|S'$ includes a transition that does not belong to $p|S'$ then p' can include new ways for satisfying *spec in the absence of faults*.

Thus, the formal definition of the problem of adding fault-tolerance is as follows:

The Addition Problem

Given p, S, *spec*, and faults f, identify p' and S' such that

$S' \subseteq S$,

$p'|S' \subseteq p|S'$, and

p' is $\mathcal{L}$ f-tolerant for *spec* from S', where

$\mathcal{L}$ can be failsafe, nonmasking, or masking. $\square$

The decision problem of adding fault-tolerance to fault-intolerant programs (from [1]) is as follows:

The Decision Problem

For a given fault-intolerant program p, its invariant S, the specification spec, and faults f, does there exist a fault-tolerant program p' and the invariant S' such that $S' \subseteq S$, $p'|S' \subseteq p|S'$, and p' is failsafe/nonmasking/masking fault-tolerant for spec from S'?

Remark. Given a program p' and its invariant S' that meet the requirements of the decision problem, every computation of $p'[]f$ that starts in the fault-span reaches a state in S'. From that state in S', a computation of p' is also a computation of p (since $S' \subseteq S$ and $p'|S' \subseteq p|S'$). Since the fault-intolerant program p satisfies its liveness specification from S, every computation of p has a suffix that is in the liveness specification. It follows that every computation of p' that starts in its fault-span will eventually reach a state from where it continuously satisfies its liveness specification. For this reason, liveness specification is not included in the above problem statement.

2.7 Synthesis of Fault-Tolerance in High Atomicity

The properties of synthesized *high atomicity* fault-tolerant programs identify an upper bound on the abilities of fault-tolerant distributed programs. As a result, in the synthesis of fault-tolerant distributed programs, there exist situations where we need to verify the possibility of solving a problem in the high atomicity model (e.g., see Chapter 5). Hence, we recall synthesis algorithms presented by Kulkarni and Arora [1] for the synthesis of fault-tolerant programs in the high atomicity model.

We represent three synthesis algorithms presented in [1] for adding three different levels of fault-tolerance to fault-intolerant programs. These algorithms synthesize a (failsafe/nonmasking/masking) fault-tolerant program in the high atomicity model where there exist no read/write restrictions for the program processes with respect to program variables. In particular, we present Add_Failsafe algorithm in Subsection 2.7.1. Then, in Subsection 2.7.2, we show how one synthesizes a nonmasking fault-tolerant program. Finally, in Subsection 2.7.3, we describe the algorithm Add_Masking where one adds masking fault-tolerance to fault-intolerant programs.

Throughout this section, we denote a fault-intolerant program with p, its invariant with S, its specification with *spec*, and a given class of faults with f. Also, we denote a synthesized fault-tolerant program and its invariant with p' and S'.

2.7.1 Synthesizing Failsafe Fault-Tolerance

The algorithm Add_Failsafe (cf. Figure 2.1) takes p, S, *spec*, and faults f. It calculates program p' with the invariant S' where p' is failsafe f-tolerant for *spec* from S'.

To synthesize a fault-tolerant program p' from the given fault-intolerant program p, Add_Failsafe calculates a set of states, say ms, from where fault transitions alone may violate safety of *spec*. The fault-tolerant program p' must never reach a state in ms, otherwise, faults may directly violate the safety of *spec*. Thus, p' should not

```
Add_failsafe($p, f$ : transitions, $S$ : state predicate, $spec$ : specification)
{
        $ms := \{s_0 : \exists s_1, s_2, ...s_n :$
                    $(\forall j : 0 \leq j < n : (s_j, s_{(j+1)}) \in f) \quad \wedge \quad (s_{(n-1)}, s_n)$ violates $spec$ };
        $mt := \{(s_0, s_1) : ((s_1 \in ms) \ \vee \ (s_0, s_1)$ violates $spec) \}$;
        $S'$ := ConstructInvariant($S - ms, p - mt$);
        if ($S' = \{\}$)  declare no failsafe $f$-tolerant program $p'$;
                    return $\emptyset, \emptyset$;
        else $p'$ :=ConstructTransitions($p - mt, S'$)
        return $p', S'$;
}

ConstructInvariant($S$ : state predicate, $p$ : transitions)
// Returns the largest subset of $S$ such that computations of $p$ within that subset are infinite
    { while ($\exists s_0 : s_0 \in S : (\forall s_1 : s_1 \in S : (s_0, s_1) \notin p)$) $S := S - \{s_0\}$   }

ConstructTransitions($p$ : transitions, $S$ : set of states)
    { return $p - \{(s_0, s_1) : s_0 \in S \ \wedge \ s_1 \notin S\}$ }
```

Figure 2.1: Synthesizing failsafe fault-tolerance in the high atomicity model.

include the transitions that reach ms or directly violate safety of $spec$ (i.e., set of mt transitions).

To calculate the invariant S', the algorithm Add_Failsafe returns the largest subset of $S - ms$ where the computations of $p - mt$ are infinite and include no transitions of mt (cf. Figure 2.1). The routine Construct_Invariant calculates such a subset of S as the invariant of p'. Since S' must be closed in transitions of p', Add_Failsafe removes transitions that start in S' and end outside S' using the routine Construct_Transition.

Soundness and completeness. The algorithm Add_Failsafe is sound; i.e., the synthesized program p' and its invariant S' satisfy the requirements of the addition problem stated in Section 2.6. Also, Add_Failsafe is complete; i.e., if there exists a failsafe fault-tolerant program p'' derived from p that satisfies the requirements of the addition problem then Add_Failsafe will find p'' and its invariant S'' [1].

2.7.2 Synthesizing Nonmasking Fault-Tolerance

To add nonmasking fault-tolerance to fault-intolerant programs, Kulkarni and Arora present algorithm Add_Nonmasking (cf. Figure 2.2). The Add_Nonmasking algorithm takes p, S, $spec$, and faults f, and then, synthesizes program p' with its invariant S'.

```
Add_nonmasking($p, f$ : transitions, $S$ : state predicate, $spec$ : specification)
{
    $S' := S$;
    $p' := (p|S) \cup \{(s_0, s_1) : s_0 \notin S \wedge s_1 \in S\}$
    return $p', S'$;
}
```

Figure 2.2: Synthesizing nonmasking fault-tolerance in the high atomicity model.

The invariant S' is equal to S since Add_Nonmasking only adds recovery transitions to S. The set of transitions of p' is the union of transitions of $p|S$ and recovery transitions.

Soundness and completeness. The algorithm Add_Nonmasking is sound; i.e., the synthesized program p' and its invariant S' satisfy the requirements of the addition problem (cf. Section 2.6). Also, Add_Nonmasking is complete; i.e., if there exists a nonmasking fault-tolerant program p'' derived from p that satisfies the requirements of the addition problem then Add_Nonmasking will find p'' and its invariant S'' [1].

2.7.3 Synthesizing Masking Fault-Tolerance

In the presence of faults, a masking fault-tolerant program must maintain safety of *spec* and provide *safe* recovery to its invariant. The Add_Masking algorithm (cf. Figure 2.3) takes p, S, *spec*, and faults f, and then generates masking fault-tolerant program p' with its invariant S' and its f-span T'.

Since no masking fault-tolerant program is allowed to reach a state from where fault transitions may violate safety, the invariant of the masking fault-tolerant program must include no ms state. Moreover, the fault-span of the masking program p' must not include any state of ms. Hence, Add_Masking sets the initial value of fault-span T_1 to $true - ms$ (cf. Line 4 in Figure 2.3). Also, since a masking fault-tolerant program should satisfy safety of *spec* from every state in its fault-span (i.e., in the presence of faults), the set of transitions of the masking program must not include a transition of mt. Thus, Add_Masking calculates the initial invariant S_1 (cf. Figure 2.3) by removing ms states from S and mt transitions from the set of transitions of

p (cf. Line 3 in Figure 2.3).

```
Add_masking($p, f$ : transitions, $S$ : state predicate, $spec$ : specification)
{
    $ms := \{s_0 : \exists s_1, s_2, ...s_n :$                                                        (1)
            $(\forall j : 0 \le j < n : (s_j, s_{(j+1)}) \in f) \;\; \wedge \;\; (s_{(n-1)}, s_n)$ violates $spec$ $\}$;
    $mt := \{(s_0, s_1) : ((s_1 \in ms) \; \vee \; (s_0, s_1)$ violates $spec)$ $\}$;                    (2)
    $S_1 :=$ ConstructInvariant$(S - ms, p - mt)$;                                                      (3)
    $T_1 := true - ms$;                                                                              (4)

    repeat                                                                                           (5)
        $T_2, S_2 := T_1, S_1$;                                                                      (6)
        $p_1 := p|S_1 \cup \{(s_0, s_1) : s_0 \notin S_1 \; \wedge \; s_0 \in T_1 \; \wedge \; s_1 \in T_1\} - mt$;    (7)
        $T_1 :=$ ConstructFaultSpan$(T_1 - \{s : S_1$ is not reachable from $s$ in $p_1$ $\}, f)$;         (8)
        $S_1 :=$ ConstructInvariant$(S_1 \wedge T_1, p_1)$;                                           (9)
        if $(S_1 = \{\} \; \vee \; T_1 = \{\})$                                                       (10)
            declare no masking $f$-tolerant program $p'$ exists;                                      (11)
            return $\emptyset, \emptyset, \emptyset$;                                                 (12)
    until $(T_1 = T_2 \; \wedge \; S_1 = S_2)$;                                                       (13)

    For each state $s : s \in T_1$ :                                                                  (14)
        $Rank(s)$ = length of the shortest computation prefix of $p_1$                                (15)
                    that starts from $s$ and ends in a state in $S_1$;
        $p' := \{(s_0, s_1) : ((s_0, s_1) \in p_1) \; \wedge (s_0 \in S_1 \; \vee \; Rank(s_0) > Rank(s_1)\})$;   (16)
        $S' := S_1$;                                                                                  (17)
        $T' := T_1$                                                                                   (18)
        return $p', S', T'$;                                                                          (19)
}

ConstructFaultSpan($T$ : state predicate, $f$ : transitions)
// Returns the largest subset of $T$ that is closed in $f$.
{
    while $(\exists s_0, s_1 : s_0 \in T \wedge s_1 \notin T \wedge (s_0, s_1) \in f)$       $T := T - \{s_0\}$
}
```

Figure 2.3: Synthesizing masking fault-tolerance in the high atomicity model.

In the iterative steps between Lines 5 to 13 in Figure 2.3, the Add_Masking algorithm searches for a valid invariant and its corresponding fault-span for the masking fault-tolerant program. Towards this end, in each iteration, Add_Masking identifies the set of transitions of p_1 that consists of transitions of p on the current invariant S_1 (i.e., $p|S_1$) and every transition in the fault-span T_1 that does not violate the closure of S_1 and does not belong to mt (cf. Line 7 in Figure 2.3). Afterwards, using Construct_FaultSpan routine, the Add_Masking algorithm calculates the largest subset of T_1 that is closed in $p_1[]f$. Since the invariant of the masking program must be a subset of its fault-span, Add_Masking recalculates the invariant S_1 considering the recalculated fault-span T_1 (cf. Line 9 in Figure 2.3).

The Add_Masking algorithm continues the above iterative procedure until there exist no more changes in S_1 and T_1, or S_1 becomes empty. When S_1 becomes empty, the Add_Masking algorithm declares that there exists no masking fault-tolerant program synthesized from p. Otherwise, there must exist a non-empty subset of S that satisfies the requirements of the addition problem (cf. Section 2.6). If there exists such subset S' of S then Add_Masking will guarantee safe recovery from states outside invariant S' to S', and there will be no cycles in $T' - S'$ (cf. Lines 14-16 in Figure 2.3).

Soundness and completeness. The algorithm Add_Masking is sound; i.e., the synthesized program p' and its invariant S' satisfy the requirements of the addition problem. Also, Add_Masking is complete; i.e., if there exists a masking fault-tolerant program p'' derived from p that satisfies the requirements of the addition problem then Add_Masking will find p'' and its invariant S'' [1].

2.8 Synthesis of Fault-Tolerant Distributed Programs

In this section, we represent the non-deterministic algorithm presented by Kulkarni and Arora [1] for the synthesis of distributed fault-tolerant programs. We also recall a theorem from [1] about the complexity of synthesizing fault-tolerant distributed programs.

Kulkarni and Arora [1] present the non-deterministic algorithm Add_ft (cf. Figure 2.4) for the addition of fault-tolerance to distributed programs in polynomial time. The Add_ft algorithm takes the transition groups $g_0, \cdots, g_{max}$ (that represent a fault-intolerant distributed program p), its invariant S, its specification $spec$, and a class of faults f. Afterwards, Add_ft calculates the set of ms states from where safety can be violated by the execution of fault transitions alone. Also, Add_ft computes the set of transitions mt that violate safety or reach a state in ms. Then, the Add_ft algorithm non-deterministically guesses the fault-tolerant program, p', its invariant, S' and its fault-span, T'.

```
Add_ft($p, f$ : set of transitions, $S$ : state predicate, $spec$ : specification,
                                                  $g_0, g_1, ..., g_{max}$ : groups of transitions)
{
    $ms := \{s_0 : \exists s_1, s_2, ...s_n : (\forall j : 0 \leq j < n : (s_j, s_{(j+1)}) \in f) \wedge (s_{(n-1)}, s_n)$ violates $spec$ $\}$;
    $mt := \{(s_0, s_1) : ((s_1 \in ms) \vee (s_0, s_1)$ violates $spec)\}$;

    Guess $S', T'$, and $p' := \bigcup (g_i : g_i$ is chosen to be included in the fault-tolerant program);
    Verify the following
        $(F1)$ $p'|S' \subseteq p|S'$;
        $(F2)$ $S' \subseteq T'$; $T'$ is closed in $p'[]f$;           // $T'$ is a fault-span of $p'$.
        $(F3)$ $T' \cap ms = \{\}$; $(p'|T') \cap mt = \{\}$;  // Safety cannot be violated from states in $T'$.
        $(F4)$ $(\forall s_0 : s_0 \in T' : (\exists s_1 :: (s_0, s_1) \in p'))$;  // $T'$ does not have deadlocks.
        $(F5)$ $S' \neq \{\}$; $S' \subseteq S$; $S'$ is closed in $p'$;      // $S'$ is an invariant of $p'$.
        $(F6)$ $p'|(T' - S')$ is acyclic;                    // $p'$ cannot stay in $(T' - S')$ forever.
}
```

Figure 2.4: A non-deterministic algorithm for adding fault-tolerance to distributed programs.

The algorithm Add_ft verifies that the synthesized (guessed) fault-tolerant program satisfies the three conditions of the addition problem (cf. Section 2.6) depending on the required level of fault-tolerance. This goal is achieved by verifying the six formulae $F1$-$F6$. The first formula $F1$ verifies that $p'|S' \subseteq p|S'$ is true. The second formula, $F2$, checks that T' is a valid fault-span. The third formula, $F3$, ensures that safety is not violated from any state in T'. The fourth formula, $F4$, verifies that the program does not deadlock in a state in T'. The fifth formula, $F5$, checks that S' is a valid invariant, i.e., S' is nonempty and S' is closed in p'. The formula $F5$ also verifies if S' is a subset of S. Finally, the formula $F6$ verifies that the program cannot stay in $T' - S'$ forever.

For synthesizing failsafe fault-tolerant programs, we do not need verify $F4$ and $F6$ as a failsafe program need not provide recovery to S'. Likewise, in the synthesis of a nonmasking fault-tolerant program, there exists no need to verify $F3$ as a nonmasking program is allowed to temporarily violate safety of *spec* in the presence of faults.

Since the algorithm Add_ft is non-deterministic, there exists no specific order in the verification of $F1$-$F6$. However, a deterministic implementation of Add_ft imposes a specific order for verifying (a subset of) $F1$-$F6$ in order to satisfy one of the requirements of the addition problem. We call such a deterministic strategy a *heuristic*.

Regarding the complexity of Add_ft, Kulkarni and Arora [1] show that each one of the conditions $F1$-$F6$ can be verified in polynomial time in the state space of p. As a result, Add_ft is in NP. We reiterate this result in the following theorem.

Theorem 2.1 The problem of synthesizing failsafe/nonmasking/masking fault-tolerant distributed programs is in NP. $\square$

Chapter 3

The Effect of Safety Specification Model on the Complexity of Synthesis

In this chapter, we focus on the effect of safety specification model on the complexity of adding masking fault-tolerance to high atomicity programs. We consider two approaches for modeling safety specifications. The first approach is based on the modeling used in [1], where the safety specification is specified in terms of a set of bad transitions that must not occur in program computations. In other words, intuitively, a program computation violates safety specification if there exists a bad transition in that computation. We denote this model as the *bad transition* (BT) model (cf. Section 2.3 for precise definition).

The second approach is a restricted model of the safety specification specified by Alpern and Schneider [23]. In [23], the safety specification is specified as a set of computation prefixes, where a computation prefix is a finite sequence of transitions. A computation violates the safety specification if one of its prefixes is ruled out by the safety specification. This model is more general than that in [1]; given the safety specification specified in terms of ‘bad transitions’ that should not occur in program computations, we can obtain the corresponding set of prefixes that should not occur in program computations.

As a special case of the model presented by Alpern and Schneider [23], we introduce a model where safety specification is specified in terms of a set of sequences of *at most* two transitions. In this model, a computation violates the safety specification if and only if it contains any sequence ruled out by the safety specification. We denote this model as the *bad pair* (BP) model. It is straightforward to observe that the BP model is a generalization of the BT model and a specialization of the model presented by Alpern and Schneider.

We show that synthesizing a masking fault-tolerant program from its fault-intolerant version in the BP model is significantly more complex than synthesizing a fault-tolerant program in the BT model. Specifically, for high atomicity programs, the synthesis in the BT specification model can be performed in polynomial time. (This result has been previously shown in [1].) However, for the same program model, the synthesis in the BP specification model is NP-complete. (This result is shown in this chapter.) It follows that the problem of adding fault-tolerance for the case where safety is represented as a set of computation prefixes that should not occur in a program computation is NP-hard. With this result, we argue that the synthesis of fault-tolerant programs will be more successful if we focus on more restrictive specifications from the BT model. Hence, in the rest of this dissertation, we represent safety specification in the BT model.

The organization of this chapter is as follows: In Section 3.1, we show that adding masking fault-tolerance to high atomicity programs is NP-complete for the BP model. In Section 3.2, we present a summary of this chapter.

3.1 NP-Completeness Proof

In this section, we show that, in general, the problem of synthesizing masking fault-tolerant programs from their fault-intolerant version becomes NP-complete if the safety specification is specified in the BP model. Towards this end, in Section 3.1.1, we present a mapping between a given instance of the 3-SAT problem and an instance of the (decision) problem of adding masking fault-tolerance. Then, in Section 3.1.2, we show that the given 3-SAT instance is satisfiable iff the answer to the decision problem is affirmative.

3.1.1 Mapping 3-SAT to the Addition of Masking Fault-Tolerance

The problem statement for the 3-SAT problem is as follows:

Given is a set of propositional variables, $x_1, x_2, ..., x_n$, and a Boolean formula $y = y_1 \wedge y_2 \wedge ... \wedge y_M$, 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? $\square$

Next, we identify each entity of the instance of the problem of adding fault-tolerance, based on the given 3-SAT formula. The instance of the decision problem of synthesizing masking fault-tolerance consists of the fault-intolerant program, p, its invariant, S, its specification *spec*, and a class of faults f.

The state space and the invariant of the fault-intolerant program, p. The invariant, S, of the fault-intolerant program, p, includes only one state, say s. Corresponding to the propositional variables and disjunctions of the given 3-SAT instance, we include additional *states outside the invariant* (cf. Figure 3.1). Specifically, for each propositional variable x_i, we introduce three states a_i, b_i, and c_i $(1 \leq i \leq n)$. Also, for simplicity, we introduce a propositional variable x_{n+1} which is always true, and corresponding to x_{n+1}, we introduce two states a_{n+1} and b_{n+1}. For each disjunction y_j, we introduce a state d_j outside the invariant ($1 \leq j \leq M$).

The transitions of the fault-intolerant program. For the convenience of representing safety specification, we classify transitions as *short*, *long*, and *medium* transitions. The only transition inside the invariant of the fault-intolerant program is the medium transition (s, s). Also, we introduce *short* transitions (a_i, b_i) and (b_i, c_i) for each propositional variable x_i. We also introduce a short transition (a_{n+1}, b_{n+1}) for x_{n+1}.

Moreover, corresponding to each propositional variable x_i, we introduce *long* transitions (b_i, a_{i+1}), (b_i, b_{i+1}), (c_i, a_{i+1}), and (c_i, b_{i+1}) ($1 \leq i \leq n$). From b_{n+1}, we introduce a long transition (b_{n+1}, s) to the invariant. Corresponding to each disjunction y_j, we have the following long transitions:

- If x_i is a literal in y_j then we include the long transition (d_j, a_i).
- If $\neg x_i$ is a literal in y_j then we include the long transition (d_j, b_i).

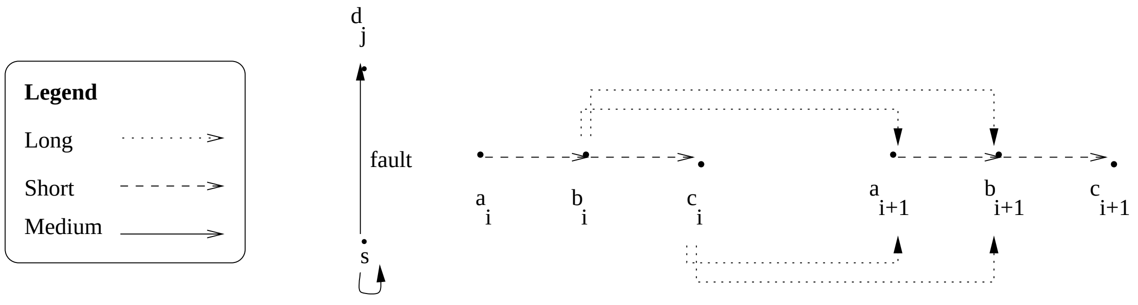

Figure 3.1: The states and the transitions corresponding to the propositional variables in the 3-SAT formula. (Except for transitions marked as fault all are program transitions. Also, note that the program has no long transitions that originate from a_i and no short transitions that originate from c_i.)

Fault transitions. The class of faults f is equal to the set of *medium* transitions $\{(s, d_j) : 1 \leq j \leq M\}$.

The safety specification of the fault-intolerant program, p. Safety will be violated if a short (respectively, long) transition is followed by another short (respectively, long) transition. Note that (s, s) and fault transitions are medium transitions (cf. Figure 3.1). Hence, they can be followed by (respectively, preceded by) any transition. Also, all transitions except those identified above violate the safety specification. This is to ensure that transitions such as $(d_j, s), (a_i, s), (b_i, s)$, and (c_i, s) $((1 \leq j \leq M) \wedge (1 \leq i \leq n))$ cannot be used for recovery.

3.1.2 Reduction from 3-SAT

In this section, we show (with Lemmas 3.1 and 3.2) that the given instance of 3-SAT is satisfiable iff masking fault-tolerance can be added to the problem instance identified in Section 3.1.1.

Lemma 3.1 If the given 3-SAT formula is satisfiable then there exists a masking fault-tolerant program for the instance of the decision problem identified in Section 3.1.1.

Proof. Since the 3-SAT formula is satisfiable, there exists an assignment of truth values to the propositional variables x_i, $1 \leq i \leq n$, such that each y_j, $1 \leq j \leq M$, is *true*. Now, we identify a masking fault-tolerant program, p', that is obtained by adding fault-tolerance to the fault-intolerant program p identified in Section 3.1.1.

The invariant of p' is the same as the invariant of p (i.e., $\{s\}$). We derive the transitions of the fault-tolerant program p' as follows. (As an illustration, we have shown the partial structure of p' where $x_1 = true$, $x_2 = false$, and $x_3 = true$ in Figure 3.2.)

- For each propositional variable x_i, $1 \leq i \leq n$, if x_i is *true* then we include the short transition (a_i, b_i). In this case, we also include the long transition (b_i, a_{i+1}) if x_{i+1} is *true*, or (b_i, b_{i+1}) if x_{i+1} is *false*.

- For each propositional variable x_i, $1 \leq i \leq n$, if x_i is *false* then we include the short transition (b_i, c_i). In this case, we also include the long transition (c_i, a_{i+1}) if x_{i+1} is *true*, or (c_i, b_{i+1}) if x_{i+1} is *false*.

- We include the transitions (a_{n+1}, b_{n+1}) and (b_{n+1}, s) corresponding to x_{n+1}.

- For each disjunction y_j that includes x_i, we include the transition (d_j, a_i) iff x_i is *true*.

- For each disjunction y_j that includes $\neg x_i$, we include the transition (d_j, b_i) iff x_i is *false*.

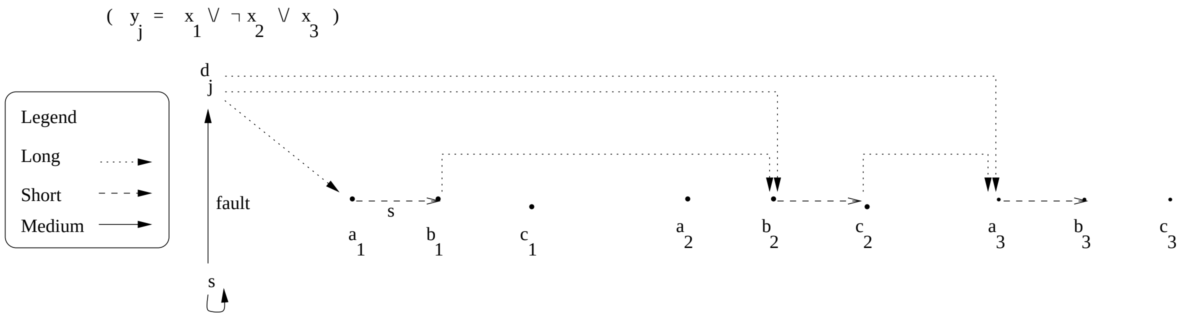

Figure 3.2: The partial structure of the masking fault-tolerant program

Now, we show that p' is masking fault-tolerant in the presence of faults f.

- **p' in the absence of faults.** $p'|S = p|S$. Thus, p' satisfies *spec* in the absence of faults.

- **p' is masking f-tolerant for *spec* from S.** To show this result, we let T' be the set of states reached in the computations of $p'[]f$ starting from s.

  - **p' satisfies its safety specification from T'.** Since the instance of the 3-SAT formula is satisfiable, each propositional variable x_i is assigned a unique truth value. Thus, for each pair of transitions (a_i, b_i) and (b_i, c_i), one of them is excluded in the set of transitions of p'. Hence, a computation of p' cannot include two consecutive short transitions. Also, the only way to execute two consecutive long transitions in the original fault-intolerant program is to execute a long transition that terminates in state b_i, $1 \leq i \leq n$, and then execute a long transition that originates in b_i. If the former transition is included then x_i is assigned the truth value false. However, in this case, no outgoing long transition from b_i is included. Thus, p' cannot execute two consecutive long transitions.

– **Starting from every state in T', a computation of p' reaches s.** By construction, p' contains no cycles outside the invariant. Hence, it suffices to show that p' does not deadlock in $T' - S'$. Now, let $y_j = x_i \vee \neg x_k \vee x_r$ be a disjunction in the 3-SAT formula. Since y_j evaluates to true, p' includes a transition from $\{(d_j, a_i), (d_j, b_k), (d_j, a_r)\}$. Also, by considering the truth values of x_i and x_{i+1}, $1 \leq i \leq n$, we observe that for every state in $\{a_i, b_i, c_i\}$ in T' there is a path that reaches a state in $\{a_{i+1}, b_{i+1}, c_{i+1}\}$. Finally, from a_{n+1} (respectively, b_{n+1}) there is an outgoing transition to b_{n+1} (respectively, s). It follows that p' does not deadlock in $T' - S$. □

Lemma 3.2 If there exists a masking fault-tolerant program for the instance of the decision problem identified earlier then the given 3-SAT formula is satisfiable.

Proof. Before we use the masking fault-tolerant program p' to identify the truth value assignment to the propositional variables in the 3-SAT formula, we make some observations about p'. Let S' be the invariant of p' and let T' be the fault-span used to show the masking fault-tolerance property of p'. Since $S' \neq \{\}$ and $S' \subseteq S$, the conditions $S' = S$ and $p|S' = p'|S'$ hold.

Since faults may directly perturb p' to d_j ($1 \leq j \leq M$), the condition $d_j \in T'$ holds. Thus, p' must provide safe recovery from each d_j. As a result, for each d_j, there exists $1 \leq i \leq n$ such that either (d_j, a_i) or $((d_j, b_i)$ and $(b_i, c_i))$ is included in $p'|T'$; i.e., either a_i or c_i must be reachable. Hence, we have

Observation 3.3. There exists $1 \leq i \leq n$ such that either $a_i \in T'$ or $c_i \in T'$. □

Now, consider the case where $a_i \in T'$ and $c_i \in T'$. In this case, (a_i, b_i) must be included as all transitions terminating in a_i are long transitions. Further, if $c_i \in T'$ then (b_i, c_i) must be included since it is the only transition that reaches c_i. In this case, $p'[]f$ can violate safety by executing (a_i, b_i) and (b_i, c_i). Hence, we have

Observation 3.4. If $a_i \in T'$ then $c_i \notin T'$. $\square$

Moreover, if $a_i \in T'$ then $(a_i, b_i) \in p'|T'$ since all transitions terminating in a_i are long transitions. Hence, $b_i \in T'$. Now, to guarantee safe recovery from b_i, p' must include either (b_i, a_{i+1}) or $((b_i, b_{i+1})$ and $(b_{i+1}, c_{i+1}))$. Thus, either $a_{i+1} \in T'$ or $c_{i+1} \in T'$. Also, if $c_i \in T'$ then either (c_i, a_{i+1}) or $((c_i, b_{i+1})$ and $(b_{i+1}, c_{i+1}))$ must be included. Thus, we have

Observation 3.5. If $(a_i \in T') \vee (c_i \in T')$ holds then we have $(\forall l : i < l \leq n : ((a_l \in T') \vee (c_l \in T')))$. $\square$

Now, let sm be the smallest value for which $((a_{sm} \in T') \vee (c_{sm} \in T'))$ holds. Based on the Observation 3.5, we have $(\forall l : sm < l \leq n : (a_l \in T') \vee (c_l \in T'))$. Hence, we make value assignment to the literals of the 3-SAT formula as follows:

- For $t < sm$, we assign $true$ to x_t.
- For $sm \leq t$, if $a_t \in T'$ then $x_t = true$. And, if $c_t \in T'$ then $x_t = false$.

Based on the observations 3.3-3.5, it is straightforward to observe that a unique value is assigned to each x_i $(1 \leq i \leq n)$. To complete the proof, we need to show that, with this truth-value assignment, the 3-SAT formula is satisfiable. We show this for a disjunction y_j $(1 \leq j \leq M)$. Wlog, let $y_j = x_i \vee x'_k \vee x_r$. Since state d_j can be reached by the occurrence of a fault from s, p' must provide safe recovery from d_j. Since the only safe transitions from d_j are those corresponding to states a_i, b_k and a_r, p' must include at least one of the transitions (d_j, a_i), (d_j, b_k), or (d_j, a_r). Now, if $(d_j, a_i) \in p'$ then $a_i \in T'$, and hence, x_i is assigned $true$. Further, if $(d_j, b_k) \in p'$ then no long transition from b_k can be included as it would allow p' to execute two long transitions successively. Hence, p' must include (b_k, c_k). Thus, $c_k \in T'$, and hence, x_k is assigned $false$. It follows that irrespective of which transition is included from d_j, y_j evaluates to $true$. Therefore, the 3-SAT formula is satisfiable. $\square$

Theorem 3.6 If the safety specification is specified in the BP model then the problem of adding masking fault-tolerance to high atomicity programs is NP-complete.

Proof. The NP-hardness of adding masking fault-tolerance in the BP model follows from Lemmas 3.1 and 3.2. To show that this problem is in NP, we proceed as follows: Given an input for the problem of adding fault-tolerance, we guess fault-tolerant program p', its invariant S' and its fault-span T'. Now, we need to verify that (1) $S' \subseteq S$, (2) S' is closed in p', (3) $p'|S' \subseteq p|S'$, (4) T' is closed in $p'[]f$, (5) $p'[]f$ does not violate safety in T', (6) p' does not deadlock in $T' - S'$, (7) $p'|(T' - S')$ is acyclic. Since each of these conditions can be verified in polynomial time in the state space, the theorem follows. $\square$

Corollary 3.7 If the safety specification is specified by a set of computational prefixes that should not occur in program computations (as in [23]) then the problem of adding masking fault-tolerance is NP-hard in the program state space. $\square$

3.2 Summary

In this chapter, we investigated the effect of the representation of the safety specification on the complexity of adding masking fault-tolerance. It is shown in the literature [1] that if one represents the safety specification as a set of *bad transitions* (denoted BT model) that must not occur in program computations then adding fault-tolerance to high atomicity programs – where processes can read/write all program variables in an atomic step – can be done in polynomial time in the state space of the input fault-intolerant program. However, in this chapter, we showed that if safety is represented by a set of sequences of transitions, where each sequence contains at most two transitions (denoted *bad pair* (BP) model), then adding fault-tolerance to programs is NP-complete. With this result, we argue that adding fault-tolerance to existing programs can be done more efficiently if we focus on the BT model.

Although the BT model is a restricted version of the BP model, it is general enough to capture other representations for modeling safety considered in the literature. For example, in the *bad state* (BS) model (e.g., [2, 4]), a computation violates safety if it reaches a state that is ruled out by the safety specification. The BS model is a restrictive version of the BT model. Hence, the algorithms in [1] can be extended to the BS model. Thus, the complexity for the BS model is (approximately) in the same complexity class as that of the BT model.

Also, we observe that the expressiveness of the BT model has the potential to capture the safety specification of practical problems. As an illustration, we model the safety specification of several examples including a simplified version of an aircraft altitude switch (cf. Section 8.5) throughout this dissertation. As a result, we argue that although the results of this chapter limit the applicability of *efficient* addition of fault-tolerance to the BT model, this model can capture a broad range of interesting problems in the synthesis of fault-tolerant programs. Therefore, in the rest of this dissertation, we represent safety specification of programs in the BT model.

Chapter 4

Synthesizing Failsafe Fault-Tolerant Distributed Programs

In this chapter, we focus on the synthesis of failsafe fault-tolerant distributed programs from their fault-intolerant versions. First, we show that synthesizing a failsafe fault-tolerant distributed program from its fault-intolerant version (i.e., adding failsafe fault-tolerance to distributed fault-intolerant programs) is NP-complete. To achieve this goal, we reduce the 3-SAT problem to the decision problem of synthesizing a failsafe fault-tolerant program. Second, we identify the restrictions that can be imposed on specifications and fault-intolerant programs in order to ensure that failsafe fault-tolerance can be synthesized in polynomial time. Towards this end, we identify a class of specifications, namely *monotonic specifications*, and a class of programs, namely *monotonic programs*. We show that failsafe fault-tolerance can be synthesized in polynomial time if monotonicity restrictions on the program and the specification are met.

As another important contribution of this chapter, we evaluate the role of restrictions imposed on specification and fault-intolerant program. In this context, we show that if monotonicity restrictions are imposed only on the specification (respectively, the fault-intolerant program) then the problem of adding failsafe fault-tolerance will remain NP-complete. Finally, we show that the class of monotonic specifications contains well-recognized [24, 25, 26, 27, 28] problems of distributed consensus, atomic commitment and Byzantine agreement.

We proceed as follows: In Section 4.1, we state the problem of adding failsafe fault-tolerance to fault-intolerant programs. In Section 4.2, we show the NP-completeness of the problem of adding failsafe fault-tolerant distributed programs. In Section 4.3, we precisely define the notion of monotonic specifications and monotonic programs, and identify their role in reducing the complexity of synthesizing failsafe fault-tolerance. Finally, we give examples of monotonic specifications and monotonic programs in Section 4.4, and summarize this chapter in Section 4.5.

4.1 Problem Statement

In this subsection, we formally state the problem of synthesizing failsafe fault-tolerance. Our goal is to *only* add failsafe fault-tolerance to generate a program that *reuses* a given fault-intolerant program. In other words, we require that any new computations that are added in the fault-tolerant program are solely for the purpose of dealing with faults; no new computations are introduced when faults do not occur.

Now, consider the case where we begin with the fault-intolerant program p, its invariant S, its specification, *spec*, and faults f. Let p' be the fault-tolerant program derived from p, and let S' be an invariant of p'. Since S is an invariant of p, all the computations of p that start from a state in S satisfy the specification, *spec*. Since we have no knowledge about the computations of p that start outside S and we are interested in deriving p' such that the correctness of p' in the absence of faults is derived from the correctness of p, we must ensure that p' begins in a state in S; i.e., the invariant of p', say S', must be a subset of S (cf. Figure 4.1).

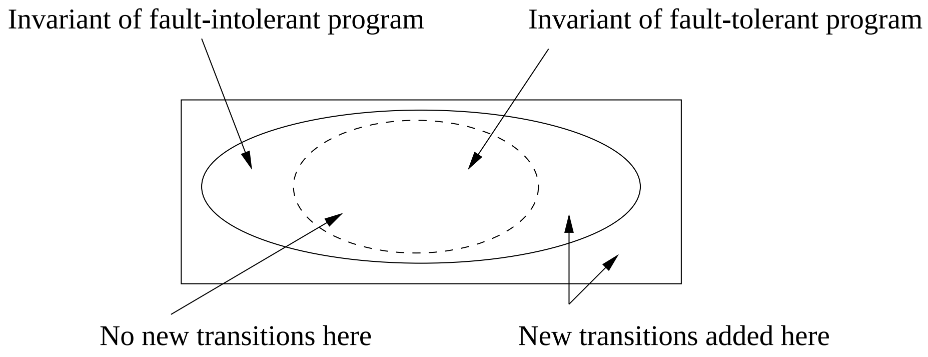
Invariant of fault-intolerant program
Invariant of fault-tolerant program
No new transitions here
New transitions added here

Figure 4.1: The relation between the invariant of a fault-intolerant program p and a fault-tolerant program p'.

Likewise, to show that p' is correct in the absence of faults, we need to show that the computations of p' that start in states in S' are in *spec*. We only have knowledge about computations of p that start in a state in S (cf. Figure 4.1). Hence, we must not introduce new transitions in the absence of faults. Thus, we define the problem of synthesizing failsafe fault-tolerance as follows:

The Problem of Synthesizing Failsafe Fault-Tolerance

Given p, S, *spec* and f such that p satisfies *spec* from S

Identify p' and S' such that

$S' \subseteq S$,

$p'|S' \ \subseteq \ p|S'$, and

p' is *failsafe* fault-tolerant to *spec* from S'. $\square$

This problem statement is taken from [1]. In [1], a generalized definition that applies to other types of fault-tolerance is presented. However, we use this restrictive definition as it suffices in this chapter. Also, to show that the problem of synthesizing failsafe fault-tolerance is NP-complete, we state the corresponding decision problem: *for a given fault-intolerant program p, its invariant S, the specification spec, and faults f, does there exist a failsafe fault-tolerant program p' and the invariant S' that satisfy the three conditions of the synthesis problem?*

Notation. Given a fault-intolerant program p, specification *spec*, invariant S and faults f, we say that program p' and predicate S' solve the synthesis problem for a given input iff p' and S' satisfy the three conditions of the synthesis problem. We say p' (respectively, S') solves the synthesis problem iff there exists S' (respectively, p') such that p', S' solve the synthesis problem.

4.2 NP-Completeness Proof

In this section, we prove that the problem of synthesizing failsafe fault-tolerant distributed programs from their fault-intolerant version is NP-complete. Towards this end, we reduce the 3-SAT problem to the problem of synthesizing failsafe fault-tolerance. In Subsection 4.2.1, we present the mapping of the given 3-SAT formula into an instance of the synthesis problem. Afterwards, in Subsection 4.2.2, we show that the 3-SAT formula is satisfiable iff a failsafe fault-tolerant program can be synthesized from this instance of the synthesis problem. Before presenting the mapping, we state the 3-SAT problem:

The 3-SAT problem.

Given is a set of propositional variables, $b_1, b_2, ..., b_n$, and a Boolean formula $c = c_1 \wedge c_2 \wedge ... \wedge c_M$, where each c_j is a disjunction of exactly three literals.

Does there exist an assignment of truth values to $b_1, b_2, ..., b_n$ such that c is satisfiable?

4.2.1 Mapping 3-SAT to an Instance of the Synthesis Problem

In this subsection, we map the given 3-SAT formula into an instance of the synthesis problem. The instance of the synthesis problem includes the fault-intolerant program, its specification, its invariant, and a class of faults. Corresponding to each propositional variable and each disjunction in the 3-SAT formula, we specify the states and the set of transitions of the fault-intolerant program. Then, we identify the fault transitions of this instance. Subsequently, we identify the safety specification and the invariant of the fault-intolerant program and determine the value of each program variable in every state.

The states of the fault-intolerant program. Corresponding to each propositional variable b_i, we introduce the following states (see Figure 4.2): $x_i, x'_i, a_i, y_i, y'_i, z_i$, and z'_i.

For each disjunction, $c_j = b_m \vee \neg b_k \vee b_l$ (cf. Figure 4.3), we introduce the following states ($k \neq m$): $c'_{jm}, d'_{jm}, c_{jk}, d_{jk}, c'_{jl}$, and d'_{jl}.

The transitions of the fault-intolerant program. In the fault-intolerant program, corresponding to each propositional variable b_i, we introduce the following transitions (cf. Figure 4.2): $(a_{i-1}, x_i), (x_i, a_i), (y'_i, z'_i)$, $(a_{i-1}, x'_i), (x'_i, a_i)$, and (y_i, z_i).

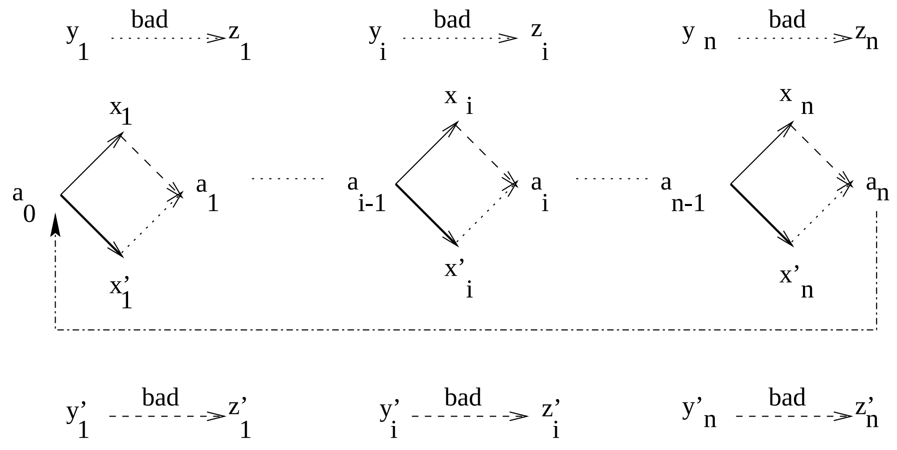

Figure 4.2: The transitions corresponding to the propositional variables in the 3-SAT formula.

Also, we introduce a transition from a_n to a_0 in the fault-intolerant program. Corresponding to each $c_j = b_m \vee \neg b_k \vee b_l$, we introduce the following program transitions (cf. Figure 4.3): $(c'_{jm}, d'_{jm}), (c_{jk}, d_{jk})$, and (c'_{jl}, d'_{jl}).

Fault transitions. We introduce the following fault transitions: From state x_i, the fault-intolerant program can reach y_i by the execution of faults. From state x'_i the faults can perturb the program to state y'_i. Thus, for each propositional variable b_i, we introduce the following fault transitions: (x_i, y_i), and (x'_i, y'_i). In addition, for each

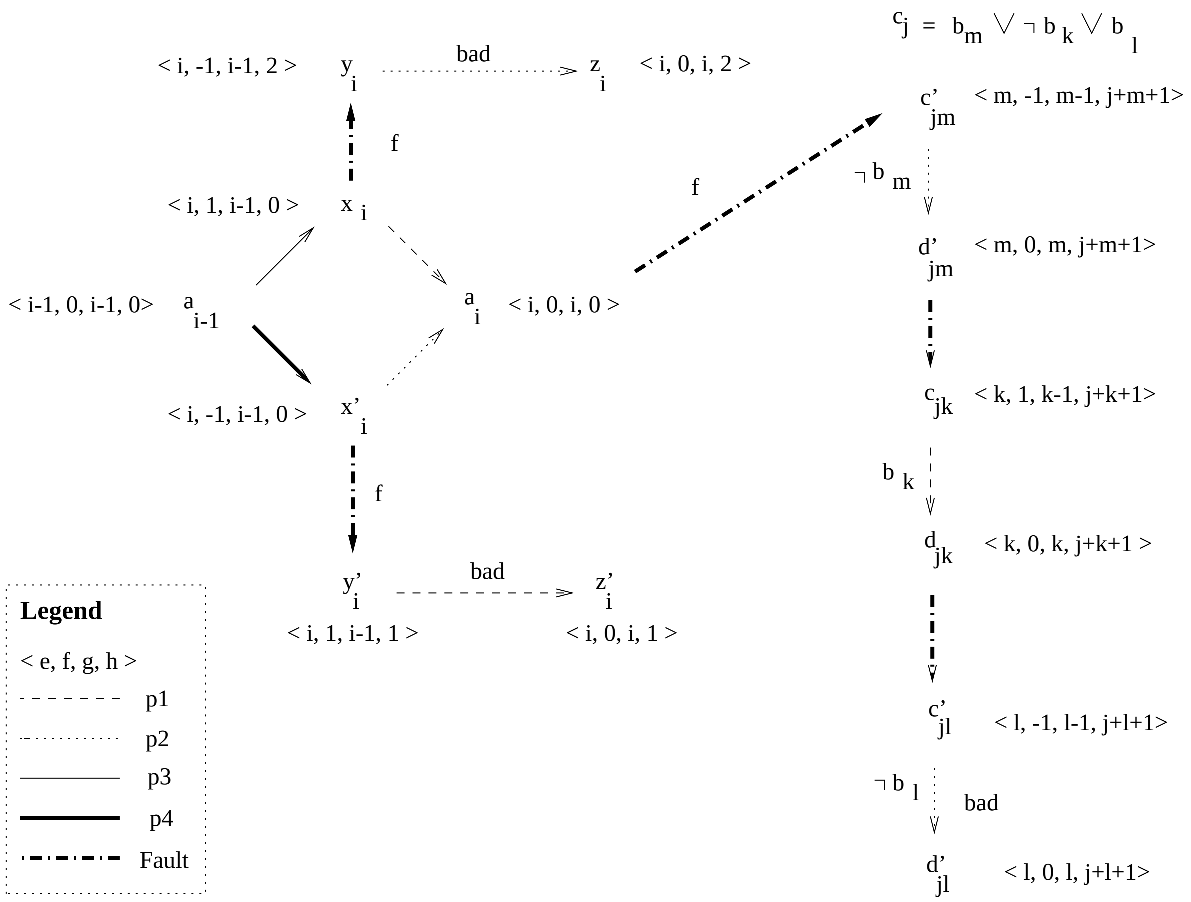

Figure 4.3: The structure of the fault-intolerant program for a propositional variable b_i and a disjunction $c_j = b_m \vee \neg b_k \vee b_l$.

disjunction $c_j = (b_m \vee \neg b_k \vee b_l)$, we introduce a fault transition that perturbs the program from state a_i, $0 \leq i < n$, to c'_{jm}. We also introduce the fault transition that perturbs the program from d'_{jm} to c_{jk}, and the transition that perturbs the program from d_{jk} to c'_{jl}. Thus, the fault transitions for c_j are as follows: (a_i, c'_{jm}), (d'_{jm}, c_{jk}), and (d_{jk}, c'_{jl}). (Note that the fault transition can perturb the program from state a_i only to the *first* state introduced for c_j; i.e., c'_{jm}.)

The invariant of the fault-intolerant program. The invariant of the fault-intolerant program consists of the following set of states: $\{x_1, \cdots, x_n\} \cup \{x'_1, \cdots, x'_n\} \cup \{a_0, \cdots, a_{n-1}\}$.

Safety specification of the fault-intolerant program. For each propositional variable b_i, the following two transitions violate the safety specification: (y_i, z_i), and

(y'_i, z'_i). Observe that in state x_i (respectively, x'_i) safety may be violated if the fault perturbs the program to y_i (respectively, y'_i) and then the program executes the transition (y_i, z_i) (respectively, (y'_i, z'_i)) (cf. Figure 4.3). For each disjunction $c_j = b_m \vee \neg b_k \vee b_l$, only the *last* program transition (c'_{jl}, d'_{jl}) added for c_j violates the safety of specification. Thus, if all three program transitions corresponding to c_j are included then safety may be violated by the execution of program and fault transitions (cf. Figure 4.3).

Variables. Now, we specify the variables used in the fault-intolerant program and their respective domains. These variables are assigned in such a way that allows us to group transitions appropriately. The fault-intolerant program has 4 variables: *e, f, g,* and *h*. The domains of these variables are respectively as follows: $\{0, \cdots, n\}$, $\{-1, 0, 1\}$, $\{0, \cdots, n\}$, and $\{0, \cdots, M + n + 1\}$.

Value assignments. The value assignments are as follows (cf. Figure 4.4):

State/Variable name	e	f	g	h
a_i	i	0	i	0
x_i	i	1	$i-1$	0
x'_i	i	-1	$i-1$	0
y'_i	i	1	$i-1$	1
y_i	i	-1	$i-1$	2
z'_i	i	0	i	1
z_i	i	0	i	2

State/Variable name	e	f	g	h
c'_{ji}	i	-1	$i-1$	$j+i+1$
d'_{ji}	i	0	i	$j+i+1$
c_{ji}	i	1	$i-1$	$j+i+1$
d_{ji}	i	0	i	$j+i+1$

Figure 4.4: The value assignment to variables.

Processes and read/write restrictions. The fault-intolerant program consists of five processes, P_1, P_2, P_3, P_4, and P_5. The read/write restrictions on these processes are as follows:

- Processes P_1 and P_2 can read and write variables f and g. They can only read variable e and they cannot read or write h.
- Processes P_3 and P_4 can read and write variables e and f. They can only read variable g and they cannot read or write h.
- Process P_5 can read all program variables and it can only write e and g.

Remark. We could have used one process for transitions of P_1 and P_2 (respectively, P_3 and P_4) however, we have separated them in two processes in order to simplify the presentation.

Grouping of Transitions. Based on the above read/write restrictions, we identify the transitions that are grouped together. We illustrate the grouping of the program transitions and the values assigned to the program variables in Figure 4.3.

Observation 4.1 Based on the inability of P_3 and P_4 to write g, the transitions (x_i, a_i), (x'_i, a_i), (y_i, z_i) and (y'_i, z'_i) can only be executed by P_1 or P_2. $\square$

Observation 4.2 Based on the inability of P_1 and P_2 to write e, the transitions (a_{i-1}, x_i) and (a_{i-1}, x'_i) can only be executed by P_3 or P_4. $\square$

Observation 4.3 Based on the inability of P_1 to read h, the transitions (x_i, a_i) and (y'_i, z'_i) are grouped in P_1. Moreover, this group also includes the transition (c_{ji}, d_{ji}) for each c_j that includes $\neg b_i$. $\square$

Observation 4.4 Based on the inability of P_2 to read h, the transitions (x'_i, a_i) and (y_i, z_i) are grouped in P_2. Moreover, this group also includes the transition (c'_{ji}, d'_{ji}) for each c_j that includes b_i. $\square$

Observation 4.5 (a_{i-1}, x_i) is grouped in P_3. $\square$

Observation 4.6: (a_{i-1}, x'_i) is grouped in P_4. $\square$

Observation 4.7: Since process P_5 cannot write f, it cannot execute the following transitions: $(a_{i-1}, x_i), (a_{i-1}, x'_i), (x_i, a_i), (x'_i, a_i), (y_i, z_i)$, and (y'_i, z'_i), for $1 \leq i \leq n$. Process P_5 can only execute transition (a_n, a_0). $\square$

For i, $1 \leq i \leq n$, the set of transitions for each process is the union of the transitions mentioned above.

4.2.2 Reduction from 3-SAT

In this subsection, we show that 3-SAT has a satisfying truth value assignment if and only if there exists a failsafe fault-tolerant program derived from the instance introduced in Section 4.2.1. Towards this end, we prove the following lemmas:

Lemma 4.8 If the given 3-SAT formula is satisfiable then there exists a failsafe fault-tolerant program that solves the instance of the addition problem identified in Section 4.2.1.

Proof. Since the 3-SAT formula is satisfiable, there exists an assignment of truth values to the propositional variables b_i, $1 \leq i \leq n$, such that each c_j, $1 \leq j \leq M$, is *true*. Now, we identify a fault-tolerant program, p', that is obtained by adding failsafe fault-tolerance to the fault-intolerant program, p, identified earlier in this section. The invariant of p' is:

$$S' = \{a_0, .., a_{n-1}\} \cup \{x_i \mid \text{propositional variable } b_i \text{ is } \textit{true} \text{ in 3-SAT } \} \cup \{x'_i \mid \text{propositional variable } b_i \text{ is } \textit{false} \text{ in 3-SAT } \}$$

The transitions of the fault-tolerant program p' are obtained as follows:

- For each propositional variable b_i, $1 \leq i \leq n$, if b_i is *true*, we include the transition (a_{i-1}, x_i) that is grouped in process P_3. We also include the transition (x_i, a_i). Based on Observation 4.3, as we include (x_i, a_i), we have to include (y'_i, z'_i). Also, based on Observation 4.3, for each disjunction c_j that includes $\neg b_i$, we have to include the transition (c_{ji}, d_{ji}).
- For each propositional variable b_i, $1 \leq i \leq n$, if b_i is *false*, we include the transition (a_{i-1}, x'_i) that is grouped in process P_4. We also include the transition (x'_i, a_i). Based on Observation 4.4, as we include (x'_i, a_i), we have to include (y_i, z_i). Also, for each disjunction c_j that includes b_i, we have to include the transition (c'_{ji}, d'_{ji}).
- We include the transition (a_n, a_0) to ensure that p' has infinite computations in its invariant.

Now, we show that p' does not violate safety even if faults occur. Note that we introduced safety-violating transitions for each propositional variable and for each disjunct. We show that none of these can be executed by p'.

- *Safety-violating transitions related to propositional variable b_i .* If the value of propositional variable b_i is *true* then the safety-violating transition (y'_i, z'_i) is included in p'. However, in this case, we have removed the state x'_i from the invariant of p' and, hence, p' cannot reach state y'_i. It follows that p' cannot execute the transition (y'_i, z'_i). By the same argument, p' cannot execute transition (y_i, z_i) when b_i is *false*.

- *Safety-violating transitions related to disjunction c_j.* Since the 3-SAT formula is satisfiable, every disjunction in the formula is true. Let $c_j = b_m \vee \neg b_k \vee b_l$. Without loss of generality, let b_m be true in c_j. Therefore, the transition (c'_{jm}, d'_{jm}) is not included in p'. It follows that p' cannot reach the state c'_{jl} and, hence, it cannot violate safety by executing the transition (c'_{jl}, d'_{jl}).

Since $S' \subseteq S$, $p' \,|\, S' \subseteq p \,|\, S'$, p' does not deadlock in the absence of faults, and p' does not violate safety in the presence of faults, p' and S' solve the synthesis problem.

□

Lemma 4.9 If there exists a failsafe fault-tolerant program that solves the instance of the addition problem identified in Section 4.2.1 then the given 3-SAT formula is satisfiable.

Proof. Suppose that there exists a failsafe fault-tolerant program p' derived from the fault-intolerant program, p, identified in Section 4.2.1. Since the invariant of p', S', is not empty and $S' \subseteq S$, S' must have at least one state in S. Since the computations of the fault-tolerant program in S' should not deadlock, for $0 \leq i \leq n-1$, every a_i must be included in S'. For the same reason, since P_5 cannot execute from a_{i-1} (cf. Observation 4.7), one of the transitions (a_{i-1}, x_i) or (a_{i-1}, x'_i) should be in p' $(1 \leq i \leq n)$. If p' includes (a_{i-1}, x_i) then we will set $b_i = true$ in the 3-SAT formula. If p' contains the transition (a_{i-1}, x'_i) then we will set $b_i = false$. Hence, each propositional variable will be assigned a truth value. Now, we show that it is not the case that b_i is assigned *true* and *false* simultaneously, and that each disjunction is true.

- *Each propositional variable gets a unique truth assignment.* We prove this by contradiction. Suppose that there exists a propositional variable b_i, which is assigned both *true* and *false*; i.e., both (a_{i-1}, x_i) and (a_{i-1}, x'_i) are included in p'. Based on the Observations 4.1 and 4.3, the transitions $(a_{i-1}, x_i), (x_i, a_i)$ and (y'_i, z'_i) must be included in p'. Likewise, based on the Observations 4.2 and 4.4, the transitions $(a_{i-1}, x'_i), (x'_i, a_i)$ and (y_i, z_i) must also be included in p'. Hence, in the presence of faults, p' may reach y_i and violate safety by executing the transition (y_i, z_i). This is a contradiction since we assumed that p' is failsafe fault-tolerant.

- *Each disjunction is true.* Suppose that there exists a $c_j = b_m \vee \neg b_k \vee b_l$, which is not *true*. Therefore, $b_m = false, b_k = true$ and $b_l = false$. Based on the grouping discussed earlier, the transitions $(c'_{jm}, d'_{jm}), (c_{jk}, d_{jk}), (c'_{jl}, d'_{jl})$ are included in p'. Thus, in the presence of faults, p' can reach c'_{jl} and violate safety specification by executing the transition (c'_{jl}, d'_{jl}). Since this is a contradiction, it follows that each disjunct in the 3-SAT formula is true. □

Theorem 4.10 The problem of synthesizing failsafe fault-tolerant distributed programs from their fault-intolerant version is NP-complete.

Proof. The NP-hardness of synthesizing failsafe fault-tolerant distributed programs follows from Lemmas 4.8 and 4.9. Also, using Theorem 2.1 presented in Section 2.8, it follows that the problem of synthesizing failsafe fault-tolerant distributed programs is NP-complete. □

4.3 Monotonic Specifications and Programs

Since the synthesis of failsafe fault-tolerance is NP-complete, as discussed earlier, we focus on this question: *What restrictions can be imposed on specifications, programs* *and faults in order to guarantee that the addition of failsafe fault-tolerance can be done in polynomial time*?

As seen in Section 4.2, one of the reasons behind the complexity involved in the synthesis of failsafe fault-tolerance is the inability of the fault-intolerant program to execute certain transitions even when no faults have occurred. More specifically, if a group of transitions includes a transition within the invariant of the fault-intolerant program and a transition that violates safety, then it is difficult to determine whether that group should be included in the failsafe fault-tolerant program.

To identify the restrictions that need to be imposed on the specification, the fault-intolerant program and the faults, we begin with the following question: *Given a program p with invariant S, under what conditions, can we design a failsafe fault-tolerant program, say p', that includes all transitions in $p|S$*? If all transitions in $p|S$ are included then it follows that p' will not deadlock in any state in S. Moreover, p' will satisfy its specification from S; if a computation of p' begins in S then it is also a computation of p. Now, we need to ensure that safety will not be violated due to fault transitions and the transitions that are grouped with those in $p|S$.

In this section, we identify the situations under which the addition of failsafe fault-tolerance can be achieved in polynomial time. Towards this end, in Subsection 4.3.1, we define a class of specifications, *monotonic specifications*, and a class of programs, *monotonic programs*, for which failsafe fault-tolerance can be synthesized in polynomial time. The intent of these definitions is to identify conditions under which a process can make *safe estimates* of variables that it cannot read. Also, we introduce the concept of *fault-safe specifications*. Subsequently, in Subsection 4.3.2, we show the role of monotonicity restrictions imposed on specifications and programs in adding failsafe fault-tolerance. When these restrictions are satisfied, we show the transitions in $p|S$ and the transitions grouped with them form the failsafe fault-tolerant program.

4.3.1 Sufficiency of Monotonicity

In this section, we identify sufficient conditions for polynomial-time synthesis of fail-safe fault-tolerant distributed programs from their fault-intolerant version. In a program with a set of processes $\{P_0, \cdots, P_n\}$, consider the case where process P_j $(0 \leq j \leq n)$ cannot read the value of a Boolean variable x. The definition of (positive) monotonicity captures the case where P_j can safely assume that x is *false*, and even if x were *true* when P_j executes, the corresponding transition would not violate safety. Thus, we define monotonic specification as follows:

Definition. A specification *spec* is *positive monotonic* on a state predicate Y with respect to a Boolean variable x iff the following condition is satisfied:

$$\begin{array}{ll}
\forall s_0, s_1, s_0', s_1' :: & x(s_0) = false \ \wedge x(s_1) = false \ \wedge \ x(s_0') = true \ \wedge x(s_1') = true \\
& \wedge \text{the value of all other variables in } s_0 \text{ and } s_0' \text{ are the same} \\
& \wedge \text{the value of all other variables in } s_1 \text{ and } s_1' \text{ are the same} \\
& \wedge (s_0, s_1) \text{ does not violate } \textit{spec} \ \wedge s_0 \in Y \wedge s_1 \in Y \\
& \Rightarrow \\
& \quad (s_0', s_1') \text{ does not violate } \textit{spec}
\end{array}$$

Likewise, we define monotonicity for programs by considering transitions within a state predicate, and define monotonic programs as follows:

Definition. A program p is positive monotonic on a state predicate Y with respect to a Boolean variable x iff the following condition is satisfied.

$$\begin{array}{ll}
\forall s_0, s_1, s_0', s_1' :: & x(s_0) = false \ \wedge x(s_1) = false \ \wedge \ x(s_0') = true \ \wedge x(s_1') = true \\
& \wedge \text{the value of all other variables in } s_0 \text{ and } s_0' \text{ are the same} \\
& \wedge \text{the value of all other variables in } s_1 \text{ and } s_1' \text{ are the same} \\
& \wedge (s_0, s_1) \in p|Y \\
& \Rightarrow \\
& \quad (s_0', s_1') \in p|Y
\end{array}$$

Negative monotonicity and monotonicity with respect to non-Boolean variables. We define negative monotonicity by swapping the words $false$ and $true$ in the above definitions. Also, although we defined monotonicity with respect to Boolean variables, it can be extended to deal with non-Boolean variables. One approach is to replace $x = false$ with $x = 0$ and $x = true$ with $x \neq 0$ in the above definition. In this case, the estimate for x is 0. We use this definition later in the section where we discuss the necessity of the monotonic programs and specifications.

Definition. Given a specification *spec* and faults f, we say that *spec* is f-safe iff the following condition is satisfied.

$$\forall s_0, s_1 :: \quad ((s_0, s_1) \in f \wedge (s_0, s_1) \text{ violates } spec) \Rightarrow (\forall s_{-1} :: (s_{-1}, s_0) \text{ violates } spec)$$

The above definition states that if a fault transition (s_0, s_1) violates *spec* then all transitions that reach state s_0 violate *spec*. The goal of this definition is to capture the requirement that if a computation prefix violates safety and the last transition in that prefix is a fault transition then the safety is violated even before the fault transition is executed. Another interpretation of this definition is that if a computation prefix maintains safety then the execution of a fault action cannot violate safety. Yet another interpretation is that the first transition that causes safety to be violated is a program transition.

We would like to note that for most problems, the specifications being considered are fault-safe. To understand this, consider the problem of mutual exclusion where a fault may cause a process to fail. In this problem, failure of a process does not violate the safety; safety is violated if some process subsequently accesses its critical section even though some other process is already in the critical section. Thus, the first transition that causes safety to be violated is a program transition. We also note that the specifications for Byzantine agreement, consensus and commit are f-safe for the corresponding faults (cf. Section 4.4). In fact, given a specification *spec* and a class of fault f, we can obtain an *equivalent* specification $spec_f$ that prohibits the execution of the following transitions.

$\{(s_0, s_1) : (s_0, s_1)$ violates *spec* $\;\vee\;$ $(\exists s_2 :: (s_1, s_2) \in f \;\wedge\; (s_1, s_2)$ violates *spec*$)\ \}$

We leave it to the reader to verify that 'p is failsafe f-tolerant to *spec* from S' iff 'p is failsafe f-tolerant to $spec_f$ from S'. With this observation, in the rest of this section, we assume that the given specification, *spec*, is f-safe. If this is not the case, Theorem 4.11 and Corollary 4.12 can be used if one replaces *spec* with $spec_f$.

Using monotonicity of specifications/programs for polynomial time synthesis. We use the monotonicity of specifications and programs to show that even if the fault-intolerant program executes after faults occur, safety will not be violated. More specifically, we prove the following theorem:

Theorem 4.11 Given is a fault-intolerant program p, its invariant S, faults f and an f-safe specification *spec*,
If

$\forall P_j, x : P_j$ is a process in p, x is a Boolean variable such that P_j cannot read x :

spec is positive monotonic on S with respect to x

$\wedge$ The program consisting of the transitions of P_j is negative monotonic on S with respect to x

Then

Failsafe fault-tolerant program that solves the synthesis problem can be obtained in polynomial time.

Proof. Let (s_0, s_1) be a transition of process P_j and let (s_0, s_1) be in $p|S$. Let x be a Boolean variable that P_j cannot read. Since we are considering programs where a process cannot blindly write a variable, it follows that $x(s_0)$ equals $x(s_1)$. Now, we consider the transition (s_0', s_1') where s_0' (respectively, s_1') is identical to s_0 (respectively, s_1) except for the value of x. We show that (s_0', s_1') does not violate *spec* by considering the value of $x(s_0)$.

- $x(s_0) = false$. Since $(s_0, s_1) \in p|S$, it follows that (s_0, s_1) does not violate safety. Hence, from the positive monotonicity of *spec* on S, it follows that (s_0', s_1') does not violate *spec*.

- $x(s_0) = true$. From the negative monotonicity of p on S, (s'_0, s'_1) is in $p|S$. Hence, (s'_0, s'_1) does not violate *spec*.

The above discussion leads to a special case of solving the synthesis problem where the transitions in $p|S$ and the transitions grouped with them can be included in the failsafe fault-tolerant program. Since $p'|S$ equals $p|S$ and p satisfies *spec* from S, it follows that p' satisfies *spec* from S. Moreover, as shown above, no transition in p' violates *spec*. And, since *spec* is f-safe, execution of fault actions alone cannot violate *spec*. It follows that p' is failsafe f-tolerant to *spec* from S. □

We generalize Theorem 4.11 as follows:

Corollary 4.12 Given is a fault-intolerant program p, its invariant S, faults f and an f-safe specification *spec*,

If

$\forall P_j, x : P_j$ is a process in p, x is a Boolean variable such that P_j cannot read x :

(*spec* is positive monotonic on S with respect to x

$\wedge$ The program consisting of the transitions of P_j is negative monotonic on S with respect to x)

$\vee$

(*spec* is negative monotonic on S with respect to x

$\wedge$ The program consisting of the transitions of P_j is positive monotonic on S with respect to x)

Then

Failsafe fault-tolerant program that solves the synthesis problem can be obtained in polynomial time. □

4.3.2 Role of Monotonicity in Complexity of Synthesis

In Section 4.3.1, we showed that if the given specification is positive (respectively, negative) monotonic and the fault-intolerant program is negative (respectively, positive) monotonic then the problem of adding failsafe fault-tolerance can be solved in polynomial time. In this section, we consider the question: *What can we say about the complexity of adding failsafe fault-tolerance if only one of these conditions is satisfied?* Specifically, in Observations 4.13 and 4.14, we show that if only one of these conditions is satisfied then the problem remains NP-complete.

Observation 4.13 Given is a fault-intolerant program p, its invariant S, faults f and an f-safe specification *spec*. If the monotonicity restrictions (from Corollary 4.12) are satisfied for p and no restrictions are imposed on the monotonicity of *spec* then the problem of adding failsafe fault-tolerance to p remains NP-complete.

Proof. This proof follows from the fact that the program obtained by mapping the 3-SAT problem in Section 4.2 is negative monotonic with respect to h. Moreover, all processes can read all variables except h (i.e., e, f, and g). It follows that the proof in Section 4.2 maps an instance of the 3-SAT problem to an instance of the problem of adding failsafe fault-tolerance where the monotonicity restrictions from Corollary 4.12 holds for the program and no assumption is made about the monotonicity of the specification. Therefore, based on Lemmas 4.8 and 4.9, the proof follows. $\square$

Furthermore, the specification obtained by mapping the 3-SAT problem in Section 4.2 is negative monotonic with respect to h. Hence, similar to Observation 4.13, we have

Observation 4.14 Given is a fault-intolerant program p, its invariant S, faults f and an f-safe specification *spec*. If the monotonicity restrictions (from Corollary 4.12) are satisfied for *spec* and no restrictions are imposed on the monotonicity of p on S then the problem of adding failsafe fault-tolerance to p remains NP-complete.

Proof. The proof is similar to the proof of Observation 4.13. $\square$

Based on the above discussion, it follows that monotonicity of both programs and specifications is necessary in the proof of Theorem 4.11. If only one of these properties is satisfied then the problem of adding failsafe fault-tolerance remains NP-complete.

Comment on the monotonicity property. The monotonicity requirements are simple and if a program and its specification meet the monotonicity requirements then the synthesis of failsafe fault-tolerance will be simple as well. Nevertheless, the significance of such sufficient conditions lies in developing heuristics by which we transform specifications (respectively, programs) to monotonic specifications (respectively, programs) so that polynomial-time addition of failsafe fault-tolerance becomes possible. While the issue of designing such heuristics is outside the scope of this paper, we note that we have developed such heuristics in Chapter 9 and [29], where we automatically transform specifications (respectively, programs) to monotonic specifications (respectively, programs) for the sake of polynomial-time addition of failsafe fault-tolerance to distributed programs.

4.4 Examples of Monotonic Specifications

In this section, we present three problems, Byzantine agreement, consensus and commit, for which the specifications and fault-intolerant programs are monotonic. In the case of Byzantine agreement, we first identify the variables and their respective domains. Then, we provide the fault-intolerant program and its invariant. Subsequently, we present the specification and faults. Finally, we show the monotonicity with respect to appropriate variables. Since the arguments for consensus and commit are similar to those in the Byzantine agreement problem, we simply sketch the arguments for those two problems.

4.4.1 Byzantine Agreement

For simplicity, we consider the canonical version where there are 4 distributed processes g, j, k, and l such that g is the general and j, k, l are the non-generals. (An identical explanation is applicable if we consider arbitrary number of non-generals.) In the agreement program, the general sends its decision to non-generals and subsequently non-generals output their decisions. Hence each process has a variable d to represent its decision, a boolean variable b to represent if that process is Byzantine, and a variable f to represent that process has finalized (output) its decision. The program variables and their domains are as follows:

$$\begin{array}{ll} d.g : \{0,1\} & \\ d.j, d.k, d.l : \{0,1,\bot\} & // \bot \text{ denotes uninitialized decision} \\ b.g, b.j, b.k, b.l : \{true, false\} & // \, b.j{=}true \text{ iff } j \text{ is Byzantine} \\ f.j, f.k, f.l : \{0,1\} & // \, f.j{=}1 \text{ iff } j \text{ has finalized its decision} \end{array}$$

The fault-intolerant Byzantine Agreement, *IB*. Each non-Byzantine process j is represented by the following actions:

$$\begin{array}{lcl} d.j = \bot \ \wedge \ f.j = 0 & \longrightarrow & d.j := d.g \\ d.j \neq \bot \ \wedge \ f.j = 0 & \longrightarrow & f.j := 1 \end{array}$$

Invariant of *IB*. The invariant of IB, S_{IB}, is as follows:

$$S_{IB} = (\forall p :: \quad \neg b.p \ \wedge \ (d.p = \bot \vee d.p = d.g) \ \wedge \ (f.p \Rightarrow d.p \neq \bot))$$

Safety specification of Byzantine agreement. The safety specification requires that *Validity* and *Agreement* be satisfied. Validity requires that if the general is not Byzantine and a non-Byzantine non-general has finalized its decision then the decision of that non-general process is the same as that of the general. Agreement requires that if two non-Byzantine non-generals have finalized their decisions then their decisions are identical. Hence, the program should not reach a state in S_{sf}, where

$$\begin{array}{rl} S_{sf} = & (\exists p,q :: \neg b.p \wedge \neg b.q \wedge d.p \neq \bot \wedge d.q \neq \bot \wedge d.p \neq d.q \wedge f.p \wedge f.q) \\ & \qquad \vee \quad (\exists p :: \neg b.g \wedge \neg b.p \wedge d.p \neq \bot \wedge d.p \neq d.g \wedge f.p) \end{array}$$

In addition, when a non-Byzantine process finalizes, it is not allowed to change it decision. Therefore, the set of transitions that should not be executed is as follows:

$$\begin{array}{rl} t_{sf} = & \{(s_0,s_1) : s_1 \in S_{sf}\} \ \cup \ \{(s_0,s_1) : \neg b.j(s_0) \wedge \neg b.j(s_1) \wedge f.j(s_0) = 1 \\ & \qquad \wedge \ (d.j(s_0) \neq d.j(s_1) \vee f.j(s_0) \neq f_j(s_1))\} \end{array}$$

Faults. The Byzantine faults, f_B, can affect one process at most and a Byzantine process can change its decision arbitrarily. Hence, the Byzantine faults are shown by the following actions:

$$\begin{array}{lcl} \neg b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l & \longrightarrow & b.j := true \\ b.j & \longrightarrow & d.j, f.j := 0|1, 0|1 \end{array}$$

The read/write restrictions: Each non-general non-Byzantine process j is allowed to read $r_j = \{b.j, d.j, f.j, d.k, d.l, d.g\}$ and it can only write $w_j = \{d.j, f.j\}$. Hence, in this case $w_j \subseteq r_j$. And, the variables that j is not allowed to read are $nr_j = \{b.g, b.k, b.l, f.k, f.l\}$.

Monotonicity of the specification and the program. We make the following observations.

Observation 4.15 The specification of Byzantine agreement is positive monotonic with respect to $b.k$ (respectively, $b.j$ and $b.l$)

Proof. Consider a transition (s_0, s_1) of some non-general process, say j, where validity and agreement are not violated when k is not Byzantine. Let (s_0', s_1') be the corresponding transition where k is Byzantine. Since validity and agreement impose no restrictions on what a Byzantine process may do, it follows that (s_0', s_1') does not violate validity and agreement. $\square$

Observation 4.16 The specification of Byzantine agreement is negative monotonic with respect to $f.k$ (respectively, $f.j$ and $f.l$)

Proof. Consider a transition (s_0, s_1) of some non-general process, say j, where validity and agreement are not violated when $f.k$ is 1, i.e., k has finalized its decision. Let (s_0', s_1') be the corresponding transition where $f.k$ is 0. Since validity and agreement impose no restrictions on processes that have not finalized their decision, it follows that (s_0', s_1') does not violate validity and agreement. $\square$

Observation 4.17 The program IB_j, consisting of the transitions of j, with invariant S_{IB} is negative monotonic with respect to $b.k$ (respectively, $b.j$ and $b.l$)

Proof. Follows from the fact that $IB|S_{IB}$ contains no transitions when $b.k$ is true.

$\square$

Observation 4.18 The program IB_j, consisting of the transitions of j, with invariant S_{IB} is positive monotonic with respect to $f.k$ (respectively, $f.j$ and $f.l$)

Proof. We leave it to the reader to observe this by considering all transitions in j.

$\square$

Observation 4.19 The specification of Byzantine agreement is f_B-safe.

Proof. Follows from the fact that a fault only affects the variables of a Byzantine process and, hence, cannot violate safety; safety may only be violated if a non-Byzantine process changes its state based on the variables of the Byzantine process.

$\square$

Now, using Observations 4.15-4.19 and Corollary 4.12, we have

Theorem 4.20 Failsafe fault-tolerant Byzantine agreement program can be obtained in polynomial time. $\square$

To obtain the failsafe fault-tolerant program, we calculate the transitions of the fault-tolerant program inside the invariant S_{IB}. The groups of transitions associated with them form the failsafe fault-tolerant program, FSB. Thus, the actions of a non-general process P_j in the fault-tolerant program are as follows:

$$\begin{array}{lllll}
FSB_1: & d.j = \bot \;\wedge\; f.j = 0 & & \longrightarrow & d.j := d.g \\
FSB_2: & (d.j = 0) \;\wedge\; ((d.k \neq 1) \wedge (d.l \neq 1)) \;\wedge\; f.j = 0 & & \longrightarrow & f.j := 1 \\
FSB_3: & (d.j = 1) \;\wedge\; ((d.k \neq 0) \wedge (d.l \neq 0)) \;\wedge\; f.j = 0 & & \longrightarrow & f.j := 1
\end{array}$$

The first action remains unchanged, and the second and the third actions determine when a process can safely finalize its decision so that the validity and agreement are preserved. Note that if the general is Byzantine and casts two different decisions to two non-general processes then the non-general processes may never finalize their decisions. Nonetheless, the program FSB will never violate the safety of specification (i.e., FSB is failsafe fault-tolerant).

4.4.2 Consensus and Commit

We now discuss the problems of distributed consensus and atomic commit to show that their specifications and fault-intolerant programs satisfy the monotonicity requirements. Since the arguments involved in these problems are similar to those in Byzantine agreement, we simply outline the reasoning behind the monotonicity.

Consensus. In distributed consensus, each process begins with a *vote*. Initially, the votes of processes may be different. It is required that all non-faulty processes agree on the same value (agreement) and that if the vote of every process is v then the agreed value be the same as v (validity). A fault can cause a process to crash (undetectably). Upon failure, the vote (and the decision) of the failed process is reset to $\perp$ so that other processes cannot distinguish between the failed process and a process that has yet to vote.

In this problem, we introduce a variable, *up.j* for every process j; j can read its own *up* value but not the *up* value of other processes. It is straightforward to see that the specification of consensus is negative monotonic with respect to *up*. Likewise, in the absence of faults, all *up* values are true and, hence, in the absence of faults, a fault-intolerant program has no transitions that execute when an *up* value is false. It follows that a fault-intolerant program for consensus is positive monotonic with respect to *up*.

Commit. In the commit problem, the agreement requirement is the same as that in consensus. However, validity requires that if the vote of any process is 0 then the agreed value must be 0. And, if all processes vote 1 and no failures occur then it is required that the agreed value must be 1. Again, the fault considered for this problem is the crash fault and, hence, we introduce the variable *up* for every process to denote whether the process is up or not. The argument that monotonicity requirements are met in the commit problem is the same as that in the consensus problem.

4.5 Summary

In this chapter, we focused on the problem of adding failsafe fault-tolerance to an existing fault-intolerant distributed program. A failsafe fault-tolerant program satisfies its specification (including safety and liveness) when no faults occur. However, if faults occur, it satisfies at least the safety specification. We showed, in Section 4.2, that the problem of adding failsafe fault-tolerance to distributed programs is NP-complete. Towards this end, we reduced the 3-SAT problem to the problem of adding failsafe fault-tolerance.

In a broader perspective, we are interested in identifying the problems for which the synthesis of fault-tolerant programs can be designed efficiently (in polynomial time) and the problems for which exponential complexity is inevitable (unless $P = NP$). By identifying such a boundary, we can determine the problems that can reap the benefits of automation and the problems for which heuristics need to be developed in order to benefit from automation. This chapter helps to make this boundary more precise than [1] in three ways. For one, the proof in [1] is for masking fault-tolerance where both safety and liveness need to be satisfied. By contrast, the NP-completeness in this chapter applies to the class of programs where only safety is satisfied. Also, the proof in [1] relies on the ability of a process to *blindly* write some variables. By contrast, the proof in this chapter does not rely on such an assumption.

The third –and the most important– step in identifying the boundary is addressed in Section 4.3 where we identified a class of specifications and a class of programs for which failsafe fault-tolerance can be added in polynomial time. Essentially, this class captures the intuition that to obtain a failsafe fault-tolerant program, we can let the fault-intolerant program execute in the presence of faults and ensure that a program transition is executed only if its execution will be safe even if faults have occurred. Towards this end, we imposed two restrictions: positive monotonicity of the specification and negative monotonicity of the fault-intolerant program. We showed that these restrictions are sufficient for polynomial synthesis of failsafe fault-tolerant distributed programs.

To show the sufficiency, in Section 4.3, we showed how a failsafe fault-tolerant program can be designed if one begins with a positive monotonic specification and a negative monotonic program. Also, we proved that if only the input program (respectively, specification) is monotonic and there exist no assumption about the monotonicity of the specification (respectively, program) then the synthesis of failsafe fault-tolerance remains NP-complete.

Chapter 5

Fault-Tolerance Enhancement

In this chapter, we concentrate on automated techniques to enhance the fault-tolerance level of a program from nonmasking to masking. Given the complexity of adding fault-tolerance to a fault-intolerant distributed program, in this chapter, we address the following question. *Is it possible to reduce the complexity of adding masking fault-tolerance if we begin with a program that provides additional guarantees about its behavior in the presence of faults?* Towards this end, we formally define the problem of enhancing the fault-tolerance of nonmasking programs to masking. Then, we present a sound and complete algorithm for the enhancement of fault-tolerance in high atomicity model. We also present a sound algorithm for enhancing the fault-tolerance of nonmasking distributed programs. We illustrate our algorithms by enhancing the fault-tolerance of the triple modular redundancy (TMR) program and the Byzantine agreement program.

This chapter is organized as follows: In Section 5.1, we state the problem of enhancing the fault-tolerance from nonmasking to masking. In Section 5.2, we present our solution for the high atomicity model. In Section 5.3, we present our solution for distributed programs. Finally, we summarize this chapter in Section 5.6.

5.1 Problem Statement

In this section, we formally define the problem of enhancing fault-tolerance from nonmasking to masking. The input to the enhancement problem includes the (transitions of) nonmasking program, p, its invariant, S, faults, f, and specification, *spec*. Given p, S, and f, we can calculate an f-span, say T, of p by starting at a state in S and identifying states reached in the computations of $p[]f$. Hence, we include fault-span T in the inputs of the enhancement problem. The output of the enhancement problem is a masking fault-tolerant program, p', its invariant, S', and its f-span, T'.

Since p is nonmasking fault-tolerant, in the presence of faults, p may temporarily violate safety. More specifically, faults may perturb p to a state in $T-S$. After faults stop occurring, p will eventually reach a state in S. However, p may violate *spec* while it is in $T-S$. By contrast, a masking fault-tolerant program p' must satisfy its safety specification even during recovery from $T-S$ to S.

The goal of the enhancement problem is to separate the tasks involved in adding recovery transitions and the tasks involved in ensuring safety. The enhancement problem deals only with adding safety to a nonmasking fault-tolerant program. With this intuition, we define the enhancement problem in such a way that only safety may be added while adding masking fault-tolerance. In other words, we require that during the enhancement, no new transitions are added to deal with functionality or to deal with recovery. Towards this end, we identify the relation between state predicates T and T', and the relation between the transitions of p and p'.

If $p'[]f$ reaches a state that is outside T then new recovery transitions must be added while obtaining the masking fault-tolerant program. Hence, we require that the fault-span of the masking fault-tolerant program, T', be a subset of T. Likewise, if p' does not introduce new recovery transitions then all the transitions included in $p'|T'$ must be a subset of $p|T'$. Thus, the enhancement problem is as follows:

The Enhancement Problem

Given p, S, *spec*, f, and T such that p satisfies *spec* from S and

T is an f-span used to show that p is nonmasking fault-tolerant for *spec* from S

Identify p' and T' such that

$T' \subseteq T$,

$p'|T' \ \subseteq \ p|T'$, and

p' is masking f-tolerant from T' for *spec*. $\square$

Comments on the Problem Statement

1. While the invariant, S, of the nonmasking fault-tolerant program is an input to the enhancement problem, it is not used explicitly in the requirements of the enhancement problem. The knowledge of S permits us to identify the transitions of p that provide functionality and the transitions of p that provide recovery. We find that such classification of transitions is useful in solving the enhancement problem. Hence, we include S in the problem statement.

2. If S' is an invariant of p', $S' \subseteq T'$, every computation of p' that starts from a state in T' maintains safety, and every computation of p' that starts from a state in T' eventually reaches a state in S' then every computation of p' that starts in a state in T' also satisfies its specification. In other words, in this situation, T' is also an invariant of p'. (This result has been previously shown in [18]; we repeat the proof in Section 5.2.) Hence, we do not explicitly identify an invariant of p'. Predicates T' and $T' \cap S$ can be used as the invariants of p'.

3. The above problem statement assumes that no new states/variables are added while enhancing fault-tolerance. This assumption can be removed by allowing systematic addition of new variables [1]. Another approach is to pretend that a process can read certain *private* variables of other processes. Then, we design a masking program that uses such private variables. The transitions of such

a masking program will require the detection of predicates involving the private variables of other processes; one can use refinement techniques to detect these non-local predicates appropriately. These refinement techniques, in turn, will determine the new variables that need to be added to detect these non-local predicates. Several such refinement techniques have been discussed in the literature (e.g., [30, 18]).

5.2 Enhancement in High Atomicity Model

In this section, we present our algorithm for solving the enhancement problem in high atomicity model. Thus, given a high atomicity nonmasking fault-tolerant program p, our algorithm derives masking fault-tolerant program p' that ensures that safety is added while the recovery provided by p is preserved. The goal of the enhancement problem is to add safety while preserving recovery. Hence, we obtain a solution for the enhancement problem by tailoring the algorithm $Add\_failsafe$ (see Section 2.7.1); $Add\_failsafe$ deals with the addition of safety to a fault-intolerant program in the presence of faults.

In our algorithm (cf. Figure 5.1), first, we compute the set of states, ms, from where fault actions alone violate safety. Clearly, we must ensure that the program never reaches a state in ms. Hence, in addition to the transitions that violate safety, we cannot use the transitions that reach a state in ms. We use mt to denote the transitions that cannot be used while adding safety. Using ms and mt, we compute the fault-span of p', T', by calling function HighAtomicityConstructInvariant ($HACI$). The first guess for T' is $T-ms$. However, due to the removal of transitions in mt, it may not be possible to provide recovery from some states in $T-ms$. Hence, we remove such states while obtaining T'. If the removal of such states causes other states to become deadlocked, we remove those states as well. Moreover, if (s_0, s_1) is a fault transition such that s_1 was removed from T' then we remove s_0 to ensure that T' is closed in f. We continue the removal of states from T' until a fixed point is established. After computing T', we compute the transitions of p' by removing all the transitions of $p-mt$ that start in a state in T' but reach a state outside T'. Thus, our algorithm is as follows:

```
High_Atomicity_Enhancement($p, f$: set of transitions, $T$: state predicate,
                                                    $spec$: specification)
{     $ms := \{s_0 : \exists s_1, s_2, ...s_n :$
                    $(\forall j : 0 \leq j < n : (s_j, s_{(j+1)}) \in f) \quad \wedge \quad (s_{(n-1)}, s_n)$ violates $spec$ };
      $mt := \{(s_0, s_1) : ((s_1 \in ms) \vee (s_0, s_1)$ violates $spec)$ };
      $T' :=$ HACI$(T - ms, p - mt, f)$;
      if $(T' = \{\})$ declare no masking $f$-tolerant program $p'$ exists;
      else $p' := \quad (p - mt) \quad - \quad \{(s_0, s_1) : s_0 \in T' \quad \wedge \quad s_1 \notin T'\}$
}

$HACI(T$: state predicate, $p$, $f$: set of transitions)
   { while $(\exists s_0 : s_0 \in T : (\forall s_1 : s_1 \in T : (s_0, s_1) \notin p) \vee (\exists s_1 : s_1 \notin T : (s_0, s_1) \in f))$
         $T := T - \{s_0\}$   }
```

Figure 5.1: The enhancement of fault-tolerance in high atomicity.

Before showing that the algorithm High_Atomicity_Enhancement is sound and complete and its complexity is polynomial in the state space of the nonmasking fault-tolerant program, we present a set of observations about our high atomicity algorithm. We use these observations to prove two lemmas about the computations of the synthesized masking fault-tolerant program in the presence of faults. Then, we use these lemmas to prove the soundness and completeness of our algorithm in the high atomicity model. To prove the soundness of our algorithm, we have to show that p' and T' satisfy the conditions of the enhancement problem. To prove the completeness, we show that if there exists any masking fault-tolerant program that enhances the fault-tolerance of the given nonmasking program then our algorithm will succeed in finding one.

We use the following notation in the rest of this section: Given a fault-intolerant program p, specification $spec$, invariant S, faults f, and fault-span T, we say that program p' and predicate T' solve the enhancement problem for the given input iff p' and T' satisfy the three conditions of the enhancement problem. We say p' (respectively, T') solves the enhancement problem iff there exists T' (respectively, p') such that p', T' solve the enhancement problem.

In the high atomicity algorithm, based on the the construction of T', we have:

Observation 5.1 $T' \cap ms = \{\}$. $\square$

By the construction of T', T' is obtained by removing zero or more states in T. Thus, we have:

Observation 5.2 $T' \subseteq T$. $\square$

The transitions of p' are a subset of the transitions of p. Thus, we have:

Observation 5.3 $(p'|T') \subseteq (p|T')$. $\square$

From the definition of $HACI$, T' is closed in p' and f. Thus, we have:

Observation 5.4 T' is closed in $p'[]f$. $\square$

If faults perturb p to a state in T, eventually p will return to a state in S. Also, by definition, $S \subseteq T$ and by Observation 5.2, $T' \subseteq T$. Now, if $T' \cap S = \{\}$, and a computation c of $p'[]f$ reaches a state in $T' - S$ then p' will never have a chance to return to a state of S. By Observation 5.3, c is also a computation of p. Thus, if $T' \cap S = \{\}$ then there exists a computation of $p[]f$ that starts in a state in T and never reaches a state in S. Since this is a contradiction, we have

Observation 5.5 $T' \cap S \neq \{\}$. $\square$

Definition. For the rest of the section, we let S' to be equal to $T' \cap S$. $\square$

Now, we use these observations to present two lemmas that are used in the soundness proof of the algorithm. First, in Lemma 5.6, we show that in the presence of faults safety specification is not violated. Then, in Lemma 5.7, we show that if faults perturb p' to a state in T' then every computation of p' starting at T' will reach a state in S'.

Lemma 5.6 $p'[]f$ maintains *spec* from T'.

Proof. By construction, T' is closed in $p'[]f$. Let c be a computation of $p'[]f$ that starts from a state in T'. If c violates the safety of *spec*, there exists a prefix, say $\langle s_0, s_1, ..., s_n\rangle$, that violates the safety of *spec*. Wlog, let $\langle s_0, s_1, ..., s_n\rangle$ be the smallest such prefix. It follows that $(s_{(n-1)}, s_n)$ violates the safety of *spec* and, hence, $(s_{(n-1)}, s_n) \in mt$. By construction, p' does not contain any transition in mt. Thus, $(s_{(n-1)}, s_n)$ is a transition of f. If $(s_{(n-1)}, s_n)$ is a transition of f then $s_{(n-1)} \in ms$ and $(s_{(n-2)}, s_{(n-1)}) \in mt$ and, hence, $(s_{(n-2)}, s_{(n-1)})$ is a transition of f. By induction, if $\langle s_0, s_1, ..., s_n\rangle$ violates the safety of *spec*, $s_0 \in ms$, which is not possible since $s_0 \in T'$ (cf. Observation 5.1). Thus, $p'[]f$ maintains *spec* from T'. $\square$

Lemma 5.7 Every computation of p' that starts from a state in T' contains a state in S'.

Proof. Consider a computation of p', say c, that starts from a state s_0 in T'. Since c is also a computation of p, it eventually reaches in a state, say s_n, in S $(0 \leq n)$. By the definition of S' and the closure of T' in p', it follows that s_n is in S'. $\square$

Theorem 5.8 T' is an invariant of p' for *spec*.

Proof. Let c be a computation of p' that starts from a state in T'. By Lemma 5.6, c maintains *spec* and by Lemma 5.7, c contains a state s_n, where $s_n \in S'$. Thus, c is of the form $\langle s_0, s_1, ..., s_n, s_{n+1}, ...\rangle$, where the prefix $\langle s_0, s_1, ..., s_n\rangle$ maintains *spec* and $\langle s_n, s_{n+1}, ...\rangle$ is in *spec*. By definition of *maintains*, there exists a suffix, say β, such that $\langle s_0, s_1, ..., s_n\rangle\beta$ is in *spec*. Now, from fusion closure, it follows that $\langle s_0, s_1, ..., s_n, s_{n+1}, ...\rangle$ is also in *spec*. Thus, every computation of p' that starts in a state in T' is in *spec*. Also, T' is closed in p' (cf. Observation 5.4). It follows that T' is an invariant of p' for *spec*. $\square$

Theorem 5.9 (Soundness) The algorithm *High_Atomicity_Enhancement* is sound.

Proof. To prove that our algorithm is sound, we have to show that the conditions of the enhancement problem are satisfied.

1. $T' \subseteq T$. (cf. Observation 5.2).

2. $p'|T' \ \subseteq \ p|T'$. (cf. Observation 5.3).

3. p' is masking f-tolerant to *spec* from T'. By letting the fault-span to be T' itself, the proof follows.

Theorem 5.10 (Completeness) The algorithm *High_Atomicity_Enhancement* is complete.

Proof. Let program p'' and predicate T'' solve the enhancement problem. Clearly, $T'' \cap ms = \{\}$; if $s_0 \in (T'' \cap ms)$ then the execution of faults alone from s_0 can violate the safety of *spec*. It follows that $T'' \subseteq (T - ms)$. Moreover, $p''|T''$ cannot include any transitions in mt; if $p''|T''$ contains a transition in mt then the execution of this transition followed by zero or more fault transitions can violate the safety of *spec*. Thus, $p''|T'' \subseteq (p - mt)$. Finally, every computation of p'' that starts from a state in T'' must be an infinite computation, if it were to be in *spec*, and T'' must be closed in f. It follows that there exists a nonempty subset of T (namely, T'') such that all computations of $p - mt$ within that subset are infinite.

Our algorithm declares that no solution for the enhancement problem exists only when there is no nonempty subset of $T - ms$ such that all the computations of $p - mt$ within that subset are infinite, and that set is closed. It follows that the algorithm is complete. $\Box$

Theorem 5.11 The algorithm *High_Atomicity_Enhancement* is sound and complete and the complexity of *High_Atomicity_Enhancement* is polynomial in the state space of the nonmasking fault-tolerant program.

Proof. The soundness and completeness proof follows from Theorems 5.9 and 5.10. Regarding complexity, note that the computation of ms as well as computation of $HACI$ are both polynomial in the state space of the input program. $\Box$

5.2.1 Example: Triple Modular Redundancy

As an illustration of our high atomicity algorithm, we show how the masking triple modular redundancy (TMR) program can be designed by enhancing the fault-tolerance level of the corresponding nonmasking program.

First, we present the nonmasking version of TMR program, the specification of TMR, and the fault actions for TMR. Then, we show how our high atomicity algorithm is used to enhance the level of fault-tolerance to masking.

Nonmasking TMR program. Nonmasking version of TMR program consists of three processes j, k, and l that share an output variable *out*. Each process j has an input variable *in.j*. The values of these input variables are obtained from a common sensor. The domain of each input variable is $\{0, 1\}$ and the domain of *out* is $\{0, 1, \perp\}$ ($\perp$ means no value has been assigned to *out*). For each process j, if the value of *out* is not yet assigned, j copies (using guarded command $N1$) its input *in.j* to *out*. And, if *out* is assigned a wrong value, i.e., the value other than the majority value, and the value of *in.j* is not corrupted then process j *corrects* (by guarded command $N2$) *out* by copying *in.j* to *out*. Both nonmasking and masking programs for TMR include a self-loop for states in which *out* has been assigned a *correct* value. However, for brevity, in this section, we keep such self-loops implicit. Thus, the actions of each process j in the nonmasking version of TMR are as follows (in this section, $\oplus$ denotes modulo 3 addition):

$$N1 : (out = \perp) \quad \longrightarrow \quad out := in.j$$

$$N2 : (out \neq \perp) \wedge (out \neq in.j) \wedge ((in.j = in.(j \oplus 1)) \vee (in.j = in.(j \oplus 2))) \quad \longrightarrow \quad out := in.j$$

Faults. Faults may perturb one of the inputs when all of them are equal. Thus, the fault action that affects j is represented by the following action:

$$F : (\forall p :: in.j = in.p) \quad \longrightarrow \quad in.j := 0 \mid 1$$

Invariant. The following state predicate is an invariant of TMR.

$$S_{TMR} = (out = \perp \wedge (\forall p, q :: in.p = in.q)) \quad \vee \quad (\exists p, q : p \neq q : out = in.p = in.q)$$

Safety specification. The safety specification of TMR requires the program not to reach states in which there exist two processes whose input values are equal but these inputs are not equal to *out* (where $out \neq \perp$). The safety specification also stipulates that variable *out* cannot change if it is different from $\perp$. Thus, safety specification requires that following transitions are not included in a program computation.

$sf_{TMR} = sf_1 \cup sf_2$, where

$sf_1 = \{(s_0, s_1) \mid (\exists p, q : (p \neq q) : (in.p(s_1) = in.q(s_1)) \wedge$
$(in.q(s_1) \neq out) \wedge (out(s_1) \neq \perp))\}$, and

$sf_2 = \{(s_0, s_1) \mid (out(s_0) \neq \perp) \wedge (out(s_0) \neq out(s_1))\}$

Fault-span. If all the inputs are equal then the value of *out* is either $\perp$ or equal to those inputs. Thus, fault-span of the nonmasking version of TMR is T, where

$$T_{TMR} = (\forall p, q :: in.p = in.q) \Rightarrow ((out = \perp) \quad \vee \quad (\forall p :: out = in.p))$$

Remark. The TMR program consists of three variables whose domain is $\{0, 1\}$ and one variable whose domain is $\{0, 1, \perp\}$. Enumerating the states associated with these variables, the state space of TMR program includes 24 states. Of these, 10 states are in the invariant, 12 additional states are in the fault-span, and two states are outside the fault-span.

The program consisting of actions $N1$ and $N2$ is nonmasking fault-tolerant in that if it begins in a state where S_{TMR} is *true* then it satisfies its specification. However, if the faults perturb it to a state in $T_{TMR} - S_{TMR}$ then it eventually recovers to a state where S_{TMR} is true. Nonetheless, until such a state is reached, safety specification may be violated.

Enhancing the tolerance of TMR. We trace the execution of our high atomicity algorithm for nonmasking TMR program.

1. **Compute ms.** ms includes all the states from where one or more fault transitions violate safety. In case of TMR, fault transitions do not violate safety if they execute in a state in T_{TMR} . Faults only change the value of one of the inputs and then safety may be violated if the corresponding process executes guarded command $N1$. Thus, $T_{TMR} \cap ms = \{\}$.

2. **Compute mt.** From the definition of ms, $mt = sf_{TMR}$.

3. **Construct T'_{TMR} and p'.** After removing transitions in mt, states where out differs from $\perp$ and out differs from the majority of the inputs are deadlocked. Hence, we need to remove those states while obtaining T'_{TMR}. After removal of those states, there are no other deadlock states. Hence, our algorithm will let T'_{TMR} to be the state predicate:

$$T'_{TMR} = T_{TMR} - \{s : (\exists p, q : (p \neq q) : (in.p(s) = in.q(s)) \wedge (out(s) \neq \perp) \wedge (out(s) \neq in.p(s)))\}$$

Moreover, to obtain the transitions of masking version of TMR, we consider the transitions of p that preserve the closure property of T'_{TMR}. Thus, the masking version of TMR consists of the following guarded command:

$$M1 : (out = \perp) \wedge ((in.j = in.(j \oplus 1)) \vee (in.j = in.(j \oplus 2))) \quad \longrightarrow \quad out := in.j$$

The predicate T'_{TMR} computed by our algorithm is both an invariant and a fault-span for the above program; every computation of the above program satisfies the specification if it begins in a state in T'_{TMR}. Moreover, T'_{TMR} is closed in both the program and fault transitions.

Remark. Note that transitions included in $N2$ are removed from the above masking fault-tolerant program as those transitions violate sf_2. However, if safety consisted of only sf_1 then the fault-tolerant program would include the transitions included in $N2$. While a masking fault-tolerant program can be obtained without using the transitions in $N2$, their inclusion follows from the heuristic in [1] that the output program should be maximal. In [1], Kulkarni and Arora have argued that if the output of a synthesis algorithm is to be used as an input, say to add fault-tolerance for a new fault, it is desirable that the intermediate program be maximal.

5.3 Enhancement for Distributed Programs

In this section, we present an algorithm to enhance the fault-tolerance level of a distributed nonmasking fault-tolerant program to masking. First, we discuss the issues involved in the enhancement problem for distributed programs. Then, we present our algorithm. As a case study, we apply our algorithm to the Byzantine agreement problem.

In high atomicity model, the main issue in enhancing the fault-tolerance level of a nonmasking fault-tolerant program p was to ensure that p does not execute a safety violating transition (s_0, s_1). In order to achieve this goal, we can either (i) ensure that p will never reach s_0, or (ii) remove (s_0, s_1). For the high atomicity model, we chose the latter option as it was strictly a better choice. However, for distributed programs, we cannot simply remove a safety violating transition (s_0, s_1) as (s_0, s_1) could be grouped with some other transitions (due to read restrictions). Thus, removal of (s_0, s_1) will also remove other transitions that are potentially useful recovery transitions. In other words, for distributed programs, the second choice is not necessarily the best option. Since an appropriate choice from the above two options cannot be identified easily for distributed programs, the synthesis of distributed programs becomes more difficult.

We develop our low atomicity algorithm (cf. Figure 5.3) by tailoring the high atomicity algorithm to deal with the grouping of transitions. More specifically, given a nonmasking fault-tolerant program p, we first start by calculating a high atomicity fault-span, T'_{high} , which is closed in $p[]f$. Since the low atomicity model is more restrictive than the high atomicity model and T'_{high} is the largest fault-span for a high atomicity program, we use T'_{high} as the domain of the states that may be included in the fault-span of our low atomicity program. In other words, if a transition, say (s_0, s_1) violates the safety specification and $s_0 \notin T'_{high}$ then we include the group associated with (s_0, s_1) and ensure that state s_0 is never reached.

Then, we call function LowAtomicityConstructInvariant ($LACI$) to calculate a low atomicity invariant S'_{low} for p' (cf. Figure 5.2). In the body of the algorithm in Figure 5.3, to calculate S'_{low}, we first call function $LACI$ with $T'_{high} \cap S$ as its first argument. Inside $LACI$, we ignore the fault transitions during the call to $HACI$; we consider the effect of fault transitions subsequently. In this call to $HACI$, we also ignore the grouping of transitions. These requirements are checked on the value of S'_{high} returned by $HACI$. Specifically, if there exists a group containing transitions (s_0, s_1) and (s'_0, s'_1) such that $s_0, s'_0, s'_1 \in S'_{high}$ and $s_1 \notin S'_{high}$, we remove s_0 from S'_{high} and recalculate the invariant. If no such group exists, $LACI$ returns S'_{high}. Thus, the function $LACI$ is as follows:

```
LACI($S$ : state predicate, $p$: transitions, $g_0, \cdots, g_m$: groups of transitions )
{   $S'_{high} = \text{HACI}(S, p, \emptyset)$;
    if ($\exists g_i, s_0, s_1, s'_0, s'_1 : (s_0, s_1), (s'_0, s'_1) \in g_i : (s_0, s'_0, s'_1 \in S'_{high} \ \wedge \ s_1 \notin S'_{high})$ )
        then return LACI($S'_{high} - \{s_0\}, p, g_0, \cdots, g_m$);
        else return $S'_{high}$;
}
```

Figure 5.2: Constructing an invariant in the low atomicity model.

In Figure 5.3, the value returned by $LACI$, S'_{init}, is used as an estimate of the invariant of the masking fault-tolerant distributed program. To compute T', we identify the effect of the fault transitions and the program transitions from states in S'_{init}. We use the variable S'_{low} to keep track of states reached in the execution of the program and fault transitions from S'_{init}. Our first estimate for S'_{low} is the same as S'_{init}. Now, we compute S_2 as the set of states reached in one step (of program or fault). Regarding fault transitions, if (s_0, s_1) is a fault transition, $s_0 \in S'_{low}$ and $s_1 \in (T'_{high} - S'_{low})$ then we add state s_1 to the set S_2. Regarding program transitions, we only consider a group if the following three conditions are satisfied: (1) at least one of the transitions in it begins and ends in S'_{low}, (2) if a transition in that group begins in a state in T'_{high} then it terminates in a state in T'_{high} and it does not violate safety, and (3) if a transition in that group begins in a state in S'_{init} then it terminates in a state in S'_{init}. If such a group has another transition (s'_0, s'_1) such that $s'_0 \in S'_{low}$ and $s'_1 \notin S'_{low}$ then we include state s'_1 in the set S_2. (Note that in the first iteration, S'_{init} equals S'_{low}. Hence, expansion by program transitions need not be considered. However, this expansion may be necessary in subsequent iterations.) Thus, S_2 identifies states from where recovery must be preserved.

```
Low_Atomicity_Enhancement($p$ : transitions, $g_0, \cdots, g_m$: groups of transitions,
                          $f$: faults, $T$, $S$ : state predicate, $spec$ : specification)
// $p = g_0 \cup g_1 \cup ... \cup g_m$
{   Calculate $ms$ and $mt$ as in High_Atomicity_Enhancement
    $T'_{high}$ = HACI($T - ms, p - mt, f$);
    $S'_{init} = S'_{low}$ = LACI($S \cap T'_{high}, p - mt, g_0, \cdots, g_m$);
    repeat {
        $S_2 = \{s_1 : s_1 \in (T'_{high} - S'_{low}) : (\exists s_0 : s_0 \in S'_{low} : (s_0, s_1) \in f \vee$
                  $(\exists g_i : (s_0, s_1) \in g_i : (((g_i | S'_{low}) \cap (p - mt)) \neq \phi) \wedge$
                      $(\forall s_2, s_3 : (s_2, s_3) \in g_i \wedge s_2 \in T'_{high} : s_3 \in T'_{high} \wedge (s_2, s_3) \notin mt) \wedge$
                      $(\forall s_2, s_3 : (s_2, s_3) \in g_i \wedge s_2 \in S'_{init} : s_3 \in S'_{init})) )\}$
        $S_3 = \{s_0 : s_0 \in (T'_{high} - S'_{low}) : (\exists s_1, g_i : (s_0, s_1) \in g_i \wedge s_1 \in S'_{low} :$
                  $(\forall s_2, s_3 : (s_2, s_3) \in g_i \wedge s_2 \in T'_{high} : s_3 \in T'_{high} \wedge (s_2, s_3) \notin mt) \wedge$
                  $(\forall s_2, s_3 : (s_2, s_3) \in g_i \wedge s_2 \in S'_{init} : s_3 \in S'_{init}))\}$
        $S'_{low} = S'_{low} \cup S_3$;
    } until ($S_3 = \emptyset$);
    if ($S_2 \neq \emptyset$) then *declare fault-tolerance cannot be enhanced;* exit().
    $T' = S'_{low}$;
    $p' = \{g_i : (\forall s_0, s_1 : (s_0, s_1) \in g_i : (s_0 \in T' \Rightarrow (s_1 \in T' \wedge (s_0, s_1) \in (p - mt))) \wedge$
                                          $(s_0 \in S'_{init} \Rightarrow s_1 \in S'_{init}))\}$;
    return $p', T'$;
}
```

Figure 5.3: The enhancement of fault-tolerance for distributed programs.

We then calculate the set of states from where recovery can be added, in one step. Specifically, if there is a transition (s_0, s_1) such that $s_0 \notin S'_{low}$ and $s_1 \in S'_{low}$ then we include s_0 in set S_3. We require that T'_{high} and S'_{init} are closed in the group being considered for recovery and that safety is not violated by any transition (that starts in a state in T'_{high}) in that group (see the constraints of S_3 in Figure 5.3). Subsequently, we add S_3 to S'_{low}. The goal of this step is to ensure that infinite computations are possible from all states in S'_{low}. This result is true about the initial value (S'_{init}) of S'_{low}. Moreover, this property continues to be true since there is an outgoing transition from every state in S_3.

We continue this calculation until no new states can be added to S'_{low}. At this point, if S_2 is nonempty, i.e., there are states from where recovery needs to be added but no new recovery transitions can be added, we declare failure. Otherwise, we identify the transitions of fault-tolerant program p' by considering transitions of p–mt that start in a state in S'_{low}. Hence, our low atomicity algorithm is as shown in Figure 5.3.

Before we discuss the soundness and the complexity of Low_Atomicity_Enhancement, we first make some observations about our low atomicity algorithm. Then, we present three lemmas that are used in the soundness proof. Similar to the proof in the high atomicity algorithm, we have

Observation 5.12 $T' \subseteq (T - ms)$, $T' \cap ms = \{\}$, and $(p \mid T') \cap mt = \emptyset$. $\square$

Observation 5.13 $S'_{init} \subseteq S$, and $S'_{low} \cap ms = \{\}$. $\square$

Observation 5.14 $T'_{high} \subseteq T$. $\square$

Observation 5.15 $(p' \mid T') \subseteq (p \mid T')$. $\square$

In the main loop of the algorithm, S_2 and S_3 are subsets of T'_{high}. Hence, the relations $S'_{low} \subseteq T'_{high}$ remains true throughout our algorithm. The value of T' equals the value of S'_{low} when the loop terminates. Hence, we have

Observation 5.16 $T' \subseteq T'_{high} \subseteq T$. □

Lemma 5.17 $p'[]f$ maintains *spec* from T'.

Proof. By construction, when T' is assigned the value S'_{low}, the value of S_2 is the empty set. Thus, starting from a state in T', $p'[]f$ cannot perturb p' to a state that is outside T'. It follows that T' is closed in $p'[]f$. Now, let c be a computation of $p'[]f$ that starts from a state in T'. Just as in the proof of Lemma 5.6, it can be shown that each prefix of c maintains *spec*. Thus, $p'[]f$ maintains *spec* from T'. □

Lemma 5.18 p' satisfies *spec* from S'_{init} .

Proof. Since S'_{init} is a subset of S, $S'_{init} \subseteq S'_{low} \subseteq T'$, $(p'|T') \subseteq (p|T')$ and every computation of p' that starts from a state in S'_{init} is also a computation of p. Hence, every computation of p' that starts from a state in S'_{init} is in *spec*. Also, by construction of p', S' is closed in p'. Thus, p' satisfies *spec* from S'_{init} . □

Lemma 5.19 Every computation of p' that starts in a state in T' is infinite.

Proof. By construction of $LACI$, this property is true about S'_{init}. Now, a state, say s, is added to S_3 only if there is a recovery transition, say t, from that state. Moreover, when transitions of p' are computed, the value of S_2 is the empty set. Hence, the group(s) of transitions containing t is included in p'. Thus, from every state in T', there is an outgoing transition in p'. It follows that every computation of p' that starts in a state in T' is infinite. □

Theorem 5.20 T' is (also) an invariant of p' for *spec*.

Proof. From Observation 5.15, every computation of p' that starts in a state in T' is a computation of p. Thus, every computation of p' that starts from a state in T' reaches a state in S. Thus, a computation of p from T is of the form $\langle s_0, s_1, ..., s_n, s_{n+1}, ...\rangle$ where $s_n \in S$. By Lemma 5.17, $\langle s_0, s_1, ..., s_n\rangle$ maintains *spec* and $\langle s_n, s_{n+1}, ...\rangle$ is in *spec*. Now, similar to the proof in Theorem 5.8, we can show that c is in *spec*. Thus, T' is also an invariant of p' for *spec*. □

Theorem 5.21 The algorithm Low_Atomicity_Enhancement is sound and its complexity is polynomial in the state space of the nonmasking fault-tolerant program.

Regarding soundness, we have to show that the conditions of the enhancement problem are satisfied.

1. $T' \subseteq T$. (cf. Observation 5.16).
2. $p'|T' \subseteq p|T'$. (cf. Observation 5.15).
3. p' is masking f-tolerant to *spec* from T'. By letting the fault-span to be T' itself, the proof follows.

Regarding, complexity, we observe that the number of iterations for the main loop are at most $|T'_{high}|$ and each statement in the low atomicity algorithm requires only polynomial time. □

Modifications/Improvements for Low_Atomicity_Enhancement. There are several improvements that can be made for the above algorithm. We discuss these improvements and issues related to completeness below.

1. In the low atomicity enhancement algorithm, if the value of S_2 is the empty set then we can break out of the loop before computing S_3. Subsequently, we can use value of S'_{low} at that time to compute p' and T'. However, we continue in the loop to determine whether recovery can be added from new states. This allows the possibility that a larger fault-span is computed and additional transitions are included in the masking fault-tolerant program. As mentioned in [1], if the output of a synthesis algorithm is used as an input to another synthesis algorithm, say to add fault-tolerance for a new fault, then it is desirable that the fault-span and the transitions of the intermediate program be maximal. For this reason, we have allowed the algorithm to expand the fault-span and to add new transitions.

2. In the low atomicity enhancement algorithm, in the calculation of S_3, we calculate states from where recovery is possible. One heuristic is to focus on states in S_2 first as recovery must be added from states in S_2. If recovery from states in S_2 is not possible then other states in $T'_{high}-S'_{low}$ should be considered. However, considering states in S_2 alone may be insufficient as it may not be possible to add recovery from those states in one step; adding recovery from other states can help in recovering from states in S_2.

3. Our algorithm is incomplete in that it may be possible to enhance the fault-tolerance of a given nonmasking program although our algorithm fails to find it. One of the causes for incompleteness is in our calculation of $LACI$; when $LACI$ needs to remove states/transitions to deal with grouping of transitions, the choice is non-deterministic. Since this choice may be inappropriate, the algorithm is incomplete. As we showed in Chapter 4 that adding failsafe fault-tolerance to distributed programs is NP-complete, it is expected that the complexity of a deterministic sound and complete algorithm for enhancing the fault-tolerance of a distributed nonmasking program will be exponential unless $P=NP$.

5.3.1 Example: Byzantine Agreement

We show how our algorithm for the low atomicity model is used to enhance the fault-tolerance level of a nonmasking Byzantine agreement program to masking. First, we present the nonmasking program, its invariant, its safety specification, faults, the fault-span for the given faults, and read/write restrictions. Finally, we show how our algorithm is used to obtain the masking program (in [26]) for Byzantine agreement.

Variables for Byzantine agreement. The nonmasking program consists of three non-general processes j, k, l and a general g. Each non-general process has three variables d, f, and b. Variable $d.j$ represents the decision of a non-general process j, $f.j$ denotes whether j has finalized its decision, and $b.j$ denotes whether j is Byzantine or not. Process g also has a variable $d.g$ and $b.g$. Thus, the variables in the Byzantine agreement program are as follows:

- $d.g : \{0, 1\}$
- $d.j, d.k, d.l : \{0, 1, \bot\}$
- $b.g, b.j, b.k, b.l : \{true, false\}$
- $f.j, f.k, f.l : \{0, 1\}$

Transitions of the nonmasking program. If process j has not copied a value from the general, action $NB1$ copies the decision of the general. If j has copied a decision and as a result $d.j$ is different from $\bot$ then j can finalize its decision by action $NB2$. If process j reaches a state, where its decision is not equal to the majority of decisions and all the non-general processes have decided then j corrects its decision by actions $NB3$ or $NB4$. Thus, the actions of each process j in the nonmasking program are as follows:

$$\begin{array}{llcl}
NB1: & d.j = \bot \ \wedge \ f.j = 0 & \longrightarrow & d.j := d.g \\
NB2: & d.j \neq \bot \ \wedge \ f.j = 0 & \longrightarrow & f.j := 1 \\
NB3: & (d.j = 1) \ \wedge \ (d.k = 0) \ \wedge \ (d.l = 0) & \longrightarrow & d.j := 0 \\
NB4: & (d.j = 0) \ \wedge \ (d.k = 1) \ \wedge \ (d.l = 1) & \longrightarrow & d.j := 1
\end{array}$$

Safety specification. The safety specification requires that if g is Byzantine, all the non-general processes should finalize with the same decision (*agreement*). If g is not Byzantine, then the decision of every non-general non-Byzantine process that has finalized should be the same as $d.g$ (*validity*). Thus, safety is violated if the program reaches a state in S_{sf}, where (in this section, unless otherwise specified, quantifications are on non-general processes)

$$\begin{aligned}
S_{sf} = \quad & (\exists p, q :: \neg b.p \wedge \neg b.q \wedge d.p \neq \bot \wedge d.q \neq \bot \wedge \ \ d.p \neq d.q \wedge f.p \wedge f.q) \\
& \qquad \vee (\exists p :: \neg b.g \wedge \neg b.p \wedge d.p \neq \bot \wedge d.p \neq d.g \wedge f.p)
\end{aligned}$$

Also, a transition violates safety if it changes the decision of a process after it has finalized. Thus, the set of transitions that violate safety is equal to t_{sf}, where

$$\begin{aligned} t_{sf} = \ & \{(s_0, s_1) : s_1 \in S_{sf}\} \ \cup \ \{(s_0, s_1) : \exists p :: \neg b.p(s_0) \wedge \neg b.p(s_1) \wedge f.p(s_0) = 1 \\ & \qquad \wedge \ (d.p(s_0) \neq d.p(s_1) \vee f.p(s_0) \neq f.p(s_1))\} \end{aligned}$$

Invariant. The invariant of nonmasking Byzantine agreement is the state predicate $S_{NB} = S_{NB_1} \ \vee \ S_{NB_2}$, where

$$\begin{aligned} S_{NB_1} = \ & \neg b.g \wedge (\neg b.j \vee \neg b.k) \wedge (\neg b.k \vee \neg b.l) \wedge (\neg b.l \vee \neg b.j) \\ & \wedge (\forall p :: \neg b.p \Rightarrow (d.p = \bot \vee d.p = d.g)) \ \wedge \ (\forall p :: (\neg b.p \wedge f.p) \Rightarrow (d.p \neq \bot)) \\ S_{NB_2} = \ & b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l \ \wedge \ (d.j = d.k = d.l \ \wedge \ d.j \neq \bot) \end{aligned}$$

Read/Write restrictions. Each non-general process j is allowed to read $\{b.j$, $d.j$, $f.j$, $d.k$, $d.l$, $d.g\}$. Thus, j can read the d values of other processes and all its variables. The set of variables that j can write is $\{d.j, f.j\}$.

Faults for Byzantine agreement. A fault transition can cause a process to become Byzantine if no process is initially Byzantine. A fault can also change the d and f values of a Byzantine process. Thus, the fault transitions that affect j are as follows (We include similar fault-transitions for k, l, and g):

$$\begin{aligned} &F1 : \neg b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l && \longrightarrow && b.j := true \\ &F2 : b.j && \longrightarrow && d.j, f.j := 0|1, 0|1 \end{aligned}$$

Fault-Span. Starting from a state in S_{NB_1}, if no process is Byzantine then a fault transition can cause one process to become Byzantine. Then, faults can change d and f values of the Byzantine process. Now, if the faults do not cause g to become Byzantine then the set of states reached from S_{NB_1} is the same as S_{NB_1}. However, if the faults cause g to become Byzantine then d and f values of non-general processes may be arbitrary. Nonetheless, the b values of non-general processes will remain false. Thus, the set of states reached from S_{NB_1} is $(S_{NB_1} \cup (b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l))$.

Starting from S_{NB_2}, no process can become Byzantine. Hence, the d values of non-general processes will remain unchanged. It follows that the set of states reached from S_{NB_2} is S_{NB_2}. Finally, since S_{NB_2} is a subset of $(b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l)$, the set of states reached from S_{NB} is T_{NB}, where

$$T_{NB} = S_{NB_1} \cup (b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l)$$

Application of our algorithm. First, we compute ms and mt that are needed by our algorithm. Every fault transition originating at S_{sf} reaches S_{sf} because it only affects the Byzantine process and the destination state will remain in S_{sf}. Since the destination of these fault transitions is S_{sf}, they violate the safety. Thus, the set of states from where faults alone violate safety is equal to S_{sf}, and as a result $ms = S_{sf}$. Since t_{sf} includes all the transitions that reach S_{sf} (which is equal to ms) or violate safety, $mt = t_{sf}$.

To calculate T'_{high}, we use the $HACI$ function of our high atomicity algorithm. This function removes deadlock states and states from where the closure of T'_{high} is violated by fault transitions. Since we have removed ms states and no fault transition can reach a state in ms from a state outside ms, there exists no state from where the closure of T'_{high} can be violated by fault transitions. Now, consider a state, say s_0, where $d.j = 0, d.k = 0, d.l = 1, b.l = false$, and $f.l = 1$. Clearly, s_0 is a deadlock state as no process can execute a *safe* transition from s_0. Hence, such states must be removed while obtaining T'_{high}.

Now, consider a state, say s_1, where $d.j = \perp, d.k = 0, d.l = 1, b.l = false$, and $f.l = 1$. In state s_1, only process j can execute a transition (by copying $d.g$) without violating safety. However, if j copies the value of the general and $d.g = 0$, the program reaches a state that was removed earlier. Hence, such states must also be removed while obtaining T'_{high}. Continuing thus, we remove all states where a process in the *minority* has finalized its decision. In other words, T'_{high} is equal to $T_{NB} - X$, where

$$X = \{s : \quad (\exists p :: f.p(s) = 1 \wedge (\forall q : p \neq q : d.p(s) \neq d.q(s)))\}$$

After this step, function $LACI$ returns $S'_{init} = T'_{high} \cap S_{NB}$. Now, we trace two iterations of the main loop in our algorithm in order to illustrate the way that our algorithm works.

1. **First iteration.** To calculate S_2, we search for states in S'_{init} from where we can directly reach a state in $T'_{high} - S'_{init}$ by fault transitions or by program transitions. From S'_{init}, no program transition can reach a state that is outside S'_{init}. However, from a state s, where $(\neg(d.j(s) = d.k(s) = d.l(s)) \vee (\exists p :: d.p(s) = \bot))$, a fault transition can cause the general to become Byzantine and then the program is outside S'_{init}. Hence, in the first iteration, $S_2 = \{s : s \in (T'_{high} - S'_{init}) : b.g(s) \wedge (\neg(d.j(s) = d.k(s) = d.l(s)) \vee (\exists p :: d.p(s) = \bot)) \wedge (\forall p : (d.p(s) \neq \bot) \Rightarrow (d.p(s) = d.g(s)))\}$.

   Now, we compute S_3. Consider a state, say s_0, where $d.j = 0, d.k = 0, d.l = 1, b.l = false$, and $f.l = 0$. In s_0, l can change $d.l$ to 0 and reach a state in S'_{init}. Hence, such states are included in S_3. Also, consider a state, say s_1, where $d.j = \bot, d.k = 1, d.l = 1$, and $d.g = 1$. In s_1, process j can copy the value of $d.g$ and take the program to S'_{init}. Therefore, in this iteration $S_3 = P_1 \cup P_2$, where

   $$\begin{aligned} P_1 = \{s : s \in (T'_{high} - S'_{init}) &: (\exists p : (d.p(s) \neq \bot) \wedge (f.p(s) = 0) : \\ &(\forall q : (q \neq p) : (d.q(s) \neq \bot) \wedge d.p(s) \neq d.q(s)))\}, \text{ and} \\ P_2 = \{s : s \in (T'_{high} - S'_{init}) &: \\ &(\exists p : d.p(s) = \bot : (\forall q : q \neq p : (d.q(s) \neq \bot) \;\wedge\; (d.q(s) = d.g(s))))\} \end{aligned}$$

   Then, we add S_3 states to S'_{low}.

   Remark. In the case of Byzantine agreement, the only states from where recovery to S'_{init} can be achieved in a single step are the states of S_3 in the first iteration. Every other recovery path includes these states as its final step to S'_{init}.

2. **Second iteration.** In the second iteration $S'_{low} = S'_{init} \cup S_3$ (S_3 in the first iteration.). To calculate S_2 in the second iteration, we search for states in S'_{low} from where we can directly reach a state in $T'_{high} - S'_{low}$ by fault transitions or by program transitions.

   To calculate S_2 in the second iteration, we need to calculate the set of states in $T'_{high} - S'_{low}$ that are reachable by a fault transition from S'_{low}. From the first iteration, we already know the set of states reachable from S'_{init}. Thus, we only need to calculate the states of $T'_{high} - S'_{low}$ that are reachable by a fault transition from recently joined states (i.e., $S_3 = P_1 \cup P_2$ of the first iteration) to S'_{low}. Since in P_1 the general process is Byzantine and all non-generals have decided, P_1 is closed in fault transitions. However, in a state in P_2, since g is Byzantine, faults may change the value of $d.g$ and take the program outside S'_{low}. In these states, the condition $(\exists p : d.p = \bot : (\forall q : q \neq p : (d.q \neq \bot) \wedge (d.q \neq d.g)))$ holds. Therefore, in this iteration, the program can reach states of S_2 by a fault transition, where

$$\begin{aligned} S_2 = \{s : s \in (T'_{high} - S'_{low}) :& \\ & b.g(s) \ \wedge \ (\exists p : d.p = \bot : (\forall q : q \neq p : (d.q \neq \bot) \wedge (d.q \neq d.g)))\} \end{aligned}$$

   To calculate S_3, we find states from where recovery is possible to S'_{low}. Thus, we search for states from where we can reach the states of S_3 calculated in the first iteration. Hence, in this iteration, single-step recovery to S'_{low} is possible from S_3, where

$$\begin{aligned} S_3 = \{s : s \in (T'_{high} - S'_{low}) : \ & (\exists p : (d.p(s) = \bot) : \ (\forall q : q \neq p : d.q(s) \neq \bot)) \vee \\ & (\exists p : (d.p(s) \neq \bot) \wedge (d.p(s) = d.g(s)) : (\forall q : q \neq p : d.q(s) = \bot))\} \end{aligned}$$

Continuing thus, we get the masking fault-tolerant Byzantine agreement; this program is the same as that in [26]. The actions of this program are as follows:

$$\begin{array}{lll}
MB1: \ d.j = \bot \ \wedge \ f.j = 0 & \longrightarrow & d.j := d.g \\
MB2: \ d.j \neq \bot \ \wedge \ f.j = 0 \ \wedge \ ((d.j = d.k) \vee (d.j = d.l)) & \longrightarrow & f.j := 1 \\
MB3: \ (d.j = 1) \ \wedge \ (d.k = 0) \ \wedge \ (d.l = 0) \ \wedge \ (f.j = 0) & \longrightarrow & d.j := 0 \\
MB4: \ (d.j = 0) \ \wedge \ (d.k = 1) \ \wedge \ (d.l = 1) \ \wedge \ (f.j = 0) & \longrightarrow & d.j := 1
\end{array}$$

5.4 Using Monotonicity for the Enhancement of Fault-Tolerance

In this section, we illustrate how we use monotonicity of programs and specifications to enhance the fault-tolerance of nonmasking fault-tolerant distributed programs to masking fault-tolerance in polynomial-time (in the state space of the nonmasking program). Towards this end, in Subsection 5.4.1, we present a theorem that identifies the sufficient conditions for enhancing the fault-tolerance of nonmasking programs in polynomial time. Then, in Subsection 5.4.2, we present an example to illustrate the application of the theorem presented in Section 5.4.1.

5.4.1 Monotonicity of Nonmasking Programs

In this section, our goal is to identify properties of programs and specifications where enhancing the fault-tolerance of nonmasking fault-tolerant programs can be done in polynomial time. Specifically, we present a theorem that identifies the sufficient conditions for polynomial-time enhancement of the fault-tolerance of nonmasking distributed programs to masking. As we have shown in Section 4.2, in general, adding failsafe fault-tolerance to a distributed program is NP-complete. Thus, it is expected that the enhancement problem is also NP-complete. Hence, we focus on the following question:

Given is a nonmasking program, p, its specification, *spec*, its invariant, S, a class of faults f, and its fault-span, T:

Under what conditions can one derive a masking fault-tolerant program p' from a nonmasking fault-tolerant program p in polynomial time?

To address the above question, we sketch a simple scenario where we can easily derive a masking fault-tolerant program from p. Specifically, we investigate the case where we only remove groups of transitions of p that include safety-violating transitions and the remaining groups of transitions construct the set of transitions of the masking fault-tolerant program p'. However, removing a group of transitions may result in creating states with no outgoing transitions (i.e., deadlock states) in the fault-span T or the invariant S. In order to resolve deadlock states, we need to add recovery transitions, and as a result, adding recovery transitions may create non-progress cycles in $(T - S)$. When we remove a non-progress cycle, we may create more deadlock states. This way, removing a group of safety-violating transitions may lead us to a cycle of complex actions of adding and removing (groups of) transitions.

To address the above problem, we require the set of transitions of p to be structured in such a way that removing safety-violating transitions (and their associated group of transitions) does not create deadlock states. Towards this end, we define *potentially safe* nonmasking programs as follows:

Definition. A nonmasking program p with the invariant S and the specification *spec* is *potentially safe* iff the following condition is satisfied.

$$\forall s_0, s_1 :: \quad ((s_0, s_1) \notin p|S \;\wedge\; ((s_0, s_1) \text{ violates } spec)\ )$$
$$\Rightarrow (\ \exists s_2 :: ((s_0, s_2) \in p) \;\wedge\; (s_0, s_2) \text{ does not violate } spec\ ) \qquad \Box$$

Moreover, we require that the removal of a safety-violating transition and its associated group of transitions does not remove good transitions that are useful for the purpose of recovery. Thus, if a transition violates the safety of *spec* then we require that no good transition exists in its associated group of transitions. To address this issue (i.e., safety-violating transitions are not grouped with good transitions), we use the monotonicity property to define *independent* programs and specifications as follows.

Definition. A nonmasking program p is *independent* of a Boolean variable x on a predicate Y iff p is both positive and negative monotonic on Y with respect to x. $\square$

Intuitively, the above definition captures that if there exists a transition $(s_0, s_1) \in p|Y$ and (s_0, s_1) belongs to a group of transitions g that is created due to inability of reading x then for all transitions $(s'_0, s'_1) \in g$ we will satisfy $(s'_0, s'_1) \in p|Y$, regardless of the value of the variable x in s'_0 and s'_1. Likewise, we define the notion of independence for specifications as follows:

Definition. A specification *spec* is *independent* of a Boolean variable x on a predicate Y iff *spec* is both positive and negative monotonic on Y with respect to x. $\square$

Based on the above definition, if a transition (s_0, s_1) belongs to a group of transitions g that is created due to inability of reading x, and (s_0, s_1) does not violate safety then no transition $(s'_0, s'_1) \in g$ will violate safety, regardless of the value of the variable x in s'_0 and s'_1.

Now, using the above definitions, we present the following theorem.

Theorem 5.22 Given is a nonmasking fault-tolerant program p, its invariant S, its fault-span T, faults f and f-safe specification *spec*,

If p is potentially safe, and

$\forall P_j, x :$ P_j is a process in p, x is a Boolean variable such that P_j cannot read x :

spec is independent of x on T

$\wedge$ The program consisting of the transitions of P_j is independent of x on S

Then

A masking fault-tolerant program p' can be derived from p in polynomial time.

Proof. Let (s_0, s_1) be a transition of process P_j. We consider two case where $(s_0, s_1) \in (p|S)$ or $(s_0, s_1) \notin (p|S)$.

1. Let $(s_0, s_1) \in (p|S)$ and x be a variable that P_j cannot read. Since we consider programs where a process cannot blindly write a variable, it follows that $x(s_0)$ equals $x(s_1)$. Now, we consider the transition (s'_0, s'_1) where s'_0 (respectively, s'_1) is identical to s_0 (respectively, s_1) except for the value of x. Since p is independent of x on S, for every value of $x(s_0)$ we will have $(s'_0, s'_1) \in (p|S)$. Thus, we include the group associated with (s_0, s_1) in the set of transitions of p'.

2. Let $(s_0, s_1) \notin (p|S)$. Again, due to the inability of P_j to read x, we consider the transition (s'_0, s'_1) where s'_0 (respectively, s'_1) is identical to s_0 (respectively, s_1) except for the value of x. By the definition of *spec* independence, if (s_0, s_1) violates *spec* then regardless of the value of x every transition (s'_0, s'_1) in the group associated with (s_0, s_1) violates *spec*, and as a result, we exclude this group of transitions in the set of transitions of p'.

p' satisfies *spec* from S. Now, let p' be the program that consists of the transitions remained in $p|T$ after excluding some groups of transitions. Since $p'|S$ equals $p|S$ and p satisfies *spec* from S, it follows that p' satisfies *spec* from S in the absence of f.

Every computation prefix of $p'[]f$ that starts at T maintains *spec*. Since we have removed the safety-violating transitions in $p|T$, when f perturbs p to T every computation prefix of $p'[]f$ maintains safety of specification.

Every computation of $p'[]f$ that starts in T has a state in S. When we remove a safety-violating transition $(s_0, s_1) \in p|T$, we actually remove all transitions (s'_0, s'_1), where s'_0 (respectively, s'_1) is identical to s_0 (respectively, s_1) except for the value of x. Note that since *spec* is independent of x, all transitions (s'_0, s'_1) that are grouped with (s_0, s_1) violate the safety of *spec* if (s_0, s_1) violates the safety of *spec*. Now, since

p is potentially safe, by definition, for every removed transition (s_0, s_1) (respectively, (s'_0, s'_1)) there exist safe transitions (s_0, s_2) (respectively, (s'_0, s'_2)) that guarantee s_0 (respectively, s'_0) has at least one outgoing transition (i.e., s_0 (respectively, s'_0) is not a deadlock state). Thus, if we remove the safety-violating transitions then we will not create any deadlock state in T. It follows that the recovery from $T-S$ to S, provided by the nonmasking program p, is preserved. Also, we have shown that p' satisfies *spec* from S and every computation prefix of $p'[]f$ maintains *spec*. Therefore, p' is masking f-tolerant to *spec* from S. □

5.4.2 Example: Distributed Counter

In this section, we present an example for enhancing the fault-tolerance of nonmasking distributed programs to masking using the monotonicity property. Towards this end, we first introduce the nonmasking program, its invariant, its safety specification, and the faults that perturb the program. Then, we synthesize the masking fault-tolerant program using Theorem 5.22.

Nonmasking program. The nonmasking program p represents an even counter. The program p consists of two processes namely, P_0 and P_1, where P_0 is responsible to reset the least significant bit (denoted x_0) whenever it is not equal to zero. And, P_1 is responsible to toggle the value of the most significant bit (denoted x_1), continuously. Process P_0 can only read/write x_0, P_1 is able to read x_0 and x_1, and P_1 can only write x_1. The only action of P_0 is as follows:

$P_0: \ x_0 \neq 0 \quad \longrightarrow \quad x_0 := 0$

The following two actions represent the transitions of P_1.

$(x_1 = 1) \wedge (x_0 = 0) \quad \longrightarrow \quad x_1 := 0$

$x_1 = 0 \quad \longrightarrow \quad x_1 := 1$

For simplicity, we represent a state of the program by a tuple $\langle x_1, x_0 \rangle$.

Invariant. Since the program simulates an even counter, we represent the invariant of the program by the state predicate $S_{ctr} \equiv (x_0 = 0)$.

Faults. Fault transitions perturb the value of x_0 and arbitrarily change its value from 0 to 1 and vice versa. The following action represents the fault transitions.

$$true \quad \longrightarrow \quad x_0 := 0 \mid 1$$

Fault-span. The entire state space is the fault-span for faults that perturb x_0. Thus, we represent the fault-span of the program by the state predicate $T_{ctr} \equiv true$.

Safety specification. Intuitively, the safety specification specifies that whenever faults perturb the counter, the counting operation should stop until the program returns to its invariant. In other words, the counter must not count from an odd value to another odd value. We identify the safety of specification $spec_{ctr}$ by the following set of transitions that the program is not allowed to execute:

$$spec_{ctr} = \{(s_0, s_1) \mid (x_0(s_0) = 1) \wedge (x_0(s_1) = 1) \wedge (x_1(s_1) \neq x_1(s_0))\}$$

Observe that, p is potentially safe and $spec_{ctr}$ is f-safe.

The nonmasking program p is independent of x_1 on S_{ctr}. For two arbitrary transitions of P_0, say (s_0, s_1) and (s'_0, s'_1), that are grouped due to inability of P_0 to read x_1, we show that the nonmasking program is independent of x_1 on S_{ctr}. Towards this end, we first show that p is negative monotonic on S_{ctr} with respect to x_1, and then, we show that p is positive monotonic on S_{ctr} with respect to x_1.

1. **Negative monotonicity of p on S_{ctr} with respect to x_1.** Consider (s_0, s_1), where $(x_1(s_0) = 1)$ and $(x_1(s_1) = 1)$. Since there is no transition (s_0, s_1) in $p|S$, where $(x_1(s_0) = 1)$ and $(x_1(s_1) = 1)$, p is negative monotonic on S_{ctr} with respect to x_1.

2. **Positive monotonicity of p on S_{ctr} with respect to x_1.** Consider (s_0, s_1), where $(x_1(s_0) = 0)$ and $(x_1(s_1) = 0)$. Since there is no transition (s_0, s_1) in $p|S$, where $(x_1(s_0) = 0)$ and $(x_1(s_1) = 0)$, p is positive monotonic on S_{ctr} with respect to x_1.

As a result of the above argument, p is independent of x_1 on S_{ctr}. Now, we show that $spec_{ctr}$ is independent of x_1 on the fault-span T_{ctr}.

For a given transition (s_0, s_1) of process P_0, we let $(x_0(s_0) = 1)$ and $(x_0(s_1) = 0)$. Since P_0 cannot read x_1, the transition (s_0, s_1) is grouped with a transitions (s'_0, s'_1), where the value of x_1 remains unchanged in (s'_0, s'_1). Now, using the definition of program monotonicity,

***$spec_{ctr}$* is independent of x_1 on T_{ctr}.** Given two arbitrary transitions of P_0, say (s_0, s_1) and (s'_0, s'_1), that are grouped due to inability of P_0 to read x_1, we show that the specification is both negative and positive monotonic on T_{ctr} with respect to x_1.

1. **Positive monotonicity of *$spec_{ctr}$*.** Consider (s_0, s_1), where $(x_1(s_0) = 0)$ and $(x_1(s_1) = 0)$, and (s_0, s_1) does not violate safety. If $(x_1(s'_0) = 1)$ and $(x_1(s'_1) = 1)$ then (s'_0, s'_1) will not violate safety (because the value of x_1 does not change during this transition). Since we have chosen (s_0, s_1) and (s'_0, s'_1) arbitrarily, the specification is positive monotonic on T_{ctr} with respect to x_1.
2. **Negative monotonicity of *$spec_{ctr}$*.** A similar argument shows that the specification is negative monotonic on T_{ctr} with respect to x_1.

Based on the above discussion, the specification is independent of x_1 on T_{ctr}.

Masking fault-tolerant program. The nonmasking program presented in this section is potentially safe. Also, process P_0 is independent of x_1 on S_{ctr}. Moreover, the specification, $spec_{ctr}$ is f-safe and is independent of x_1 on T_{ctr}. Therefore, using Theorem 5.22, we can derive a masking fault-tolerant version of p in polynomial time. In the synthesis of masking program, we remove the transition from $\langle 0, 1\rangle$ to $\langle 1, 1\rangle$. The action of P_0 remains as is, and the actions of P_1 are as follows:

$$(x_1 = 1) \wedge (x_0 = 0) \quad \longrightarrow \quad x_1 := 0$$

$$(x_1 = 0) \wedge (x_0 = 0) \quad \longrightarrow \quad x_1 := 1$$

5.5 Enhancement versus Addition

In this section, we compare the complexity of enhancement with adding masking fault-tolerance. Specifically, we first discuss enhancement in high atomicity with respect to Add_Masking algorithm represented in Subsection 2.7.3. Subsequently, we compare the complexity of these two algorithms for distributed programs (i.e., low atomicity model).

Complexity of enhancement versus addition in high atomicity. Since Add_Masking tries to add both safety and recovery simultaneously, it is more complex than High_Atomicity_Enhancement presented in this chapter. More specifically, the asymptotic complexity of High_Atomicity_Enhancement is less than that of Add_masking. Thus, if the state space of the problem at hand prevents the addition of masking fault-tolerance to a fault-intolerant program, it may be possible to partially automate the design of a masking fault-tolerant program by manually designing a nonmasking fault-tolerant program and enhancing its fault-tolerance to masking using automated techniques.

The algorithm High_Atomicity_Enhancement adds safety to a nonmasking fault-tolerant program while ensuring that the recovery provided by it continues to be satisfied. We note that the asymptotic complexity of High_Atomicity_Enhancement is the same as the complexity of adding failsafe fault-tolerance to a fault-intolerant program. In other words, in High_Atomicity_Enhancement, the recovery is preserved for free!

Complexity of enhancement versus addition in low atomicity. We compare the cost of adding masking fault-tolerance to a fault-intolerant distributed program and the cost of enhancing the fault-tolerance of a nonmasking fault-tolerant distributed program to masking. Asymptotically speaking, adding masking (respectively, failsafe) fault-tolerance to a fault-intolerant distributed program is NP-complete [1, 31]. Therefore, it is expected that the enhancement problem —that adds safety while preserving recovery— for distributed programs will also be NP-complete.

Although the enhancement problem may not provide relief in terms of the worst-case complexity, we find that it helps in developing heuristics that determine if safe recovery is possible from states that are reached in the presence of faults. More specifically, consider a state, say s, that is reached in a computation of the fault-intolerant program in the presence of faults. While adding masking fault-tolerance to a fault-intolerant program, we need to exhaustively search all possible transition sequences from s to determine if recovery from s is possible. By contrast, while enhancing the fault-tolerance of a nonmasking fault-tolerant program, we reuse the recovery provided by the nonmasking fault-tolerant program. Hence, we need to check only the transition sequences that the nonmasking fault-tolerant program can produce. It follows that deriving heuristics that determine if safe recovery is possible from a given state is simpler in the enhancement problem.

The enhancement problem also allows us to deduce additional information about states by reasoning in the high atomicity model. We illustrate this by one example that occurs in Byzantine agreement. Consider a state s_0 where all processes are non-Byzantine, $d.j = d.k = \perp$, $d.g = 1$, $d.l = 1$ and $f.l = 0$. Let s_1 be a state that is identical to s_0 except that the value of $f.l$ in s_1 is 1. Now, consider the transition (s_0, s_1). Note that both s_0 and s_1 are in the invariant, S_{NB}. Hence, for a synthesis algorithm, this appears as a good transition that should be retained in the fault-tolerant program. However, from s_1, if g becomes Byzantine and changes $d.g$, we can reach a state where $d.g, d.j$, and $d.k$ become 0. The resulting state is a deadlock state.

While adding masking fault-tolerance to a fault-intolerant program, it is difficult to check that all computations that (1) start from s_1, (2) in which g becomes Byzantine, and (3) in which g changes $d.g$ to 0 are deadlock states. Moreover, if we ignore the grouping restrictions imposed by the low atomicity model, i.e., if we could read and write all variables in one atomic step then recovery would be possible from s_1.

However, in the context of the enhancement problem, we concluded that even in the high atomicity model, we could not recover from state s_1 by reusing the transitions of the nonmasking fault-tolerant program.

We expect that such high atomicity reasoning will play an important role in reducing complexity in the enhancement problem. To reduce the complexity of adding fault-tolerance in the low atomicity model, it is desirable to reason about the input program in the high atomicity model, obtain a high atomicity masking fault-tolerant program, and modify that high atomicity masking fault-tolerant program so that the restrictions of the low atomicity model are satisfied while preserving the masking fault-tolerance. As the Byzantine agreement example illustrates, this approach can be followed while enhancing the fault-tolerance of a nonmasking fault-tolerant program. However, this approach could not be used while adding masking fault-tolerance to a fault-intolerant program.

5.6 Summary

In this chapter, we defined the problem of enhancing the fault-tolerance level of a nonmasking program to masking. This problem separates (1) the task of adding recovery, and (2) the task of maintaining the safety specification during recovery. For the high atomicity model, we presented a sound and complete algorithm for the enhancement problem. We showed that the complexity of our high atomicity algorithm is asymptotically less than Add_Masking algorithm (cf. Subsection 2.7.3). For distributed programs, we presented a sound algorithm for the enhancement problem. We also showed that our fault-tolerance enhancement algorithm for distributed programs resolves some of the difficulties encountered in adding safe recovery transitions in [14].

As an illustration of our algorithms, we showed how masking fault-tolerant programs for TMR (in high atomicity model) and Byzantine agreement (for distributed programs) can be designed by enhancing the fault-tolerance of the corresponding nonmasking programs. We chose these examples as masking fault-tolerant versions of these programs have been *manually* designed from the corresponding nonmasking fault-tolerant versions [32]. The results in this chapter show that those enhancements can in fact be automated as well.

Also, we argued that enhancing the fault-tolerance of a distributed program is simpler than adding masking fault-tolerance to its fault-intolerant version. We validated this result by comparing the derivation of a masking fault-tolerant Byzantine agreement program from the corresponding fault-intolerant version and from the corresponding nonmasking version.

Moreover, we have used the monotonicity property (presented in Section 4.3) to identify sufficient conditions under which the enhancement of fault-tolerance can be done in polynomial time. Specifically, we presented a sufficiency theorem and we enhanced the fault-tolerance of a distributed counter to masking fault-tolerance using our sufficiency theorem.

Chapter 6

Pre-Synthesized Fault-Tolerance Components

In this chapter, we present a synthesis approach that adds pre-synthesized fault-tolerance components to a given fault-intolerant program in the synthesis of its fault-tolerant version. Techniques presented in [14] and Chapters 4 and 5 respectively reduce the complexity of synthesis by using heuristics and by identifying classes of programs and specifications for which efficient synthesis is possible. However, these techniques cannot apply the lessons learnt in synthesizing one fault-tolerant program while synthesizing another fault-tolerant program. The synthesis method presented in this chapter allows us to recognize the *patterns* that we often apply in the synthesis of fault-tolerant distributed programs. Then, we organize those patterns in terms of fault-tolerance components and reuse them in the synthesis of new problems.

To investigate the use of pre-synthesized fault-tolerance components in the synthesis of fault-tolerant programs from their fault-intolerant version, we use *detectors* and *correctors* identified in [33, 10]. Specifically, in [33, 10], Arora and Kulkarni have shown that detectors and correctors suffice in the *manual* design of a rich class of fault-tolerant programs. Hence, we expect to benefit from the generality of such components in automated synthesis of fault-tolerant programs. Thus, in this chapter, we present a synthesis approach that adds pre-synthesized detectors and correctors to a given fault-intolerant program in order to synthesize its fault-tolerant version. In particular, we focus on adding *masking* fault-tolerance where we address issues regarding the representation, the specification, and the addition of pre-synthesized fault-tolerance components. In general, our synthesis method is applicable for adding failsafe and nonmasking fault-tolerance as well.

As a running example, we synthesize a token ring program that consists of 4 processes and is subject to process-restart faults. The masking fault-tolerant (token ring) program can recover even from the situation where every process is corrupted. We note that the previous approaches that added fault-tolerance to the token ring program presented in this chapter assumed that at least one process is not corrupted.

We proceed as follows: in Section 6.1, we formally state the problem of adding fault-tolerance components to fault-intolerant programs. Then, in Section 6.2, we present a synthesis method that identifies when and how the synthesis algorithm decides to add a component. Subsequently, in Section 6.3, we formally describe how we represent a fault-tolerance component. In Section 6.4, we show how we automatically specify a component and add it to a program. In Section 6.5, we show how we reuse a *linear* pre-synthesized component in the synthesis of an alternation bit protocol. Afterwards, in Sections 6.6, we apply our synthesis method for adding nonmasking fault-tolerance to a diffusing computation program with a tree-like structure where we show that our synthesis method is applicable for programs with hierarchical topologies. In Section 6.7, we address some of the questions raised by the synthesis method presented in this chapter. Finally, we summarize our discussion in Section 6.8.

6.1 Problem Statement

In this section, we formally define the problem of adding fault-tolerance components to a fault-intolerant program. We identify the conditions of the addition problem by which we can verify the correctness of the synthesized fault-tolerant program after adding fault-tolerance components.

Given a fault-intolerant program p, its state space S_p, its invariant S, its specification *spec*, and a class of faults f, we add pre-synthesized fault-tolerance components (i.e., detectors and correctors) to p in order to synthesize a fault-tolerant program p' with the new invariant S'. When we add a fault-tolerance component to p, we also add the variables associated with that component. As a result, we expand the state space of p. The new state space, say $S_{p'}$, is actually the state space of the synthesized fault-tolerant program p'.

After the addition, we require the fault-tolerant program p' to behave similar to p in the absence of faults f. In the presence of faults f, p' should satisfy *masking* fault-tolerance. To ensure the correctness of the synthesized fault-tolerant program in the new state space, we need to identify the conditions that have to be met by the synthesized program, p'. Towards this end, we define a projection from $S_{p'}$ to S_p using onto function $H : S_{p'} \rightarrow S_p$. We apply H on states, state predicates, transitions, and groups of transitions in $S_{p'}$ to identify their corresponding entities in S_p.

Let the invariant of the synthesized program be $S' \subseteq S_{p'}$. If there exists a state $s'_0 \in S'$ where $H(s'_0) \notin S$ then in the absence of faults p' can start at s'_0 whose image, $H(s'_0)$, is outside S. As a result, in the absence of faults, p' will include computations in the new state space $S_{p'}$ that do not have corresponding computations in p. These new computations resemble new behaviors in the absence of faults, which is not desirable. Therefore, we require that $H(S') \subseteq S$. Also, if p' contains a transition (s'_0, s'_1) in $p'|S'$ that does not have a corresponding transition (s_0, s_1) in $p|H(S')$ (where $H(s'_0) = s_0$ and $H(s'_1) = s_1$) then p' can take this transition and create a new way for satisfying *spec* in the absence of faults. Therefore, we require that $H(p'|S') \subseteq p|H(S')$. Now, we present the problem of adding fault-tolerance components to p.

The Addition Problem.

Given p, S, *spec*, f, with state space S_p such that p satisfies *spec* from S,

$S_{p'}$ is the new state space due to adding fault-tolerance components to p,

$H : S_{p'} \to S_p$ is an onto function,

Identify p' and $S' \subseteq S_{p'}$ such that

$H(S') \subseteq S$,

$H(p'|S') \subseteq p|H(S')$, and

p' is masking f-tolerant for *spec* from S'. $\square$

6.2 The Synthesis Method

In this section, we present a synthesis method to solve the addition problem of Section 6.1. In Section 6.2.1, we present a high level description of our synthesis method and express our approach for combining heuristics from [14] (cf. Section 6.2.2 for an example heuristic) with pre-synthesized components. Then, in Section 6.2.2, we illustrate our synthesis method using a simple example, a token ring program with 4 processes. We use the token ring program as a running example in the rest of the chapter, where we synthesize a token ring program that is masking fault-tolerant to process-restart faults.

6.2.1 Overview of Synthesis Method

Our synthesis method takes as its input a fault-intolerant program p with a set of processes $P_0 \cdots P_n$ ($n > 1$), its specification *spec*, its invariant S, a set of read/write restrictions $r_0 \cdots r_n$ and $w_0 \cdots w_n$, and a class of faults f to which we intend to add fault-tolerance. The synthesis method outputs a fault-tolerant program p' and its invariant S'.

The heuristics in [14] (i) *add safety* to ensure that the masking fault-tolerant program never violates its safety specification, and (ii) *add recovery* to ensure that the masking fault-tolerant program never deadlocks (respectively, livelocks). Moreover, while adding recovery transitions, it is necessary to ensure that all the groups of transitions included along that recovery transition are safe unless it can be guaranteed (with the help from heuristics) that those transitions cannot be executed. Thus, adding recovery transitions from deadlock states is one of the important issues in adding fault-tolerance. Hence, the method presented in this chapter, focuses on adding pre-synthesized components for resolving deadlock states.

Now, in order to resolve a deadlock state, say s_d, using our hybrid approach, we proceed as follows: First, for each process P_i in the given fault-intolerant program, we introduce a high atomicity pseudo process PS_i. Initially, PS_i has no action to execute, however, we allow PS_i to read all program variables and write only those variables that P_i can write. Using these special processes, we now present the $ResolveDeadlock$ routine (cf. Figure 6.1) that is the core of our synthesis method. The input of $ResolveDeadlock$ consists of the deadlock state that needs to be resolved, s_d, and the set of high atomicity pseudo processes PS_i $(0 \leq i \leq n)$.

```
Resolve_Deadlock($s_d$: state, $PS_0, \cdots, PS_n$: high atomicity pseudo process)
{
  Step 1. If Add_Recovery ($s_d$) then return *true*.
  Step 2. Else non-deterministically choose a $PS_{index}$, where $0 \leq index \leq n$ and $PS_{index}$
            adds a high atomicity recovery action $grd \rightarrow st$
  Step 3. If (there exists a $PS_{index}$) and (there exists a detector $d$ in the component
            library that suffices to refine $grd \rightarrow st$ without interfering with the program)
            then add $d$ to the program, and return *true*.
            else return *false*.
                // Subsequently, we remove some transitions to make $s_d$ unreachable.
}
```

Figure 6.1: Overview of the synthesis method.

First, in Step 1, we invoke a heuristic-based routine *Add_Recovery* to add recovery from s_d under the distribution restrictions (i.e., in the low atomicity model) – where program processes have read/write restrictions with respect to the program variables.

Add_Recovery explores the ability of each process P_i to add recovery transition from s_d under the distribution restrictions. If *Add_Recovery* fails then we will choose to add a fault-tolerance component in Steps 2 and 3.

In Steps 2 and 3, we identify a fault-tolerance component and then add it to p in order to resolve s_d. To add a fault-tolerance component, the synthesis algorithm should (i) specify the required component; (ii) retrieve the specified component from a given library of components; (iii) ensure the interference freedom of the composition of the component and the program, and finally (iv) add the extracted component to the program. As a result, adding a pre-synthesized component is a costly operation. Hence, we prefer to add a component during the synthesis only when available heuristics for adding recovery fail in Step 1.

To identify the required fault-tolerance components, we use pseudo process PS_i that can read all program variables and write w_i (i.e., the set of variables that P_i can write). In other words, we check the ability of each PS_i to add high atomicity recovery – where we have no read restrictions – from s_d. If no PS_i can add recovery from s_d then our algorithm fails to resolve s_d. If there exist one or more pseudo processes that add recovery from s_d then we non-deterministically choose a process PS_{index} with high atomicity action $ac : grd \rightarrow st$. Since we give PS_{index} the permission to read all program variables for adding recovery from s_d, the guard grd is a global state predicate that we need to refine. If there exists a detector that can refine grd without interfering with the program execution then we will add that detector to the program. (We present the discussion about how to specify the required detector d and how to add d to the fault-intolerant program in Sections 6.3 and 6.4.)

In cases where *Resolve_Deadlock* returns *false*, we remove some transitions to make s_d unreachable. If we fail to make s_d unreachable then we will declare failure in the synthesis of the masking fault-tolerant program p'. Observe that by using pre-synthesized components, we increase the chance of adding recovery from s_d, and as a result, we reduce the chance of reaching a point where we declare failure to synthesize a fault-tolerant program.

6.2.2 Token Ring Example

In this subsection, we introduce a token ring program with 4 processes that is subject to process restart faults. Using our synthesis method (cf. Figure 6.1), we synthesize a token ring program that is masking fault-tolerant for the case where all processes are corrupted.

The token ring program. The fault-intolerant program consists of four processes P_0, P_1, P_2, and P_3 arranged in a ring. Each process P_i has a variable x_i $(0 \leq i \leq 3)$ with the domain $\{\perp, 0, 1\}$. Due to distribution restrictions, process P_i can read x_i and x_{i-1} and can only write x_i $(1 \leq i \leq 3)$. P_0 can read x_0 and x_3 and can only write x_0. We say, a process P_i $(1 \leq i \leq 3)$ has the token iff $x_i \neq x_{i-1}$ and fault transitions have not corrupted P_i and P_{i-1}. And, P_0 has the token iff $x_3 = x_0$ and fault transitions have not corrupted P_0 and P_3. A process P_i $(1 \leq i \leq 3)$ copies x_{i-1} to x_i if the value of x_i is different from x_{i-1}. Also, if $x_0 = x_3$ then process P_0 copies the value of $(x_3 \oplus 1)$ to x_0, where $\oplus$ is addition in modulo 2. This way, a process passes the token to the next process.

We represent a state s of the token ring program by a 4-tuple $\langle x_0, x_1, x_2, x_3 \rangle$. Each element of the 4-tuple $\langle x_0, x_1, x_2, x_3 \rangle$ represents the value of x_i in s $(0 \leq i \leq 3)$. Thus, if we start from initial state $\langle 0, 0, 0, 0 \rangle$ then process P_0 has the token and the token circulates along the ring. We represent the transitions of the fault-intolerant program TR by the following actions $(1 \leq i \leq 3)$.

$TR_0 : \ (x_0 = 1) \ \wedge \ (x_3 = 1) \longrightarrow x_0 := 0;$ $\qquad$ $TR_i : \ (x_i = 0) \ \wedge \ (x_{i-1} = 1) \longrightarrow x_i := 1;$

$TR'_0 : \ (x_0 = 0) \ \wedge \ (x_3 = 0) \longrightarrow x_0 := 1;$ $\qquad$ $TR'_i : \ (x_i = 1) \ \wedge \ (x_{i-1} = 0) \longrightarrow x_i := 0;$

Faults. Faults can restart a process P_i. Thus, the value of x_i becomes unknown. Hence, we model faults by setting the value of x_i to an unknown value $\perp$.

Specification. The problem specification requires that the corrupted value of one process does not affect a non-corrupted process, and there is only one process that has the token.

Invariant. The invariant of the above program includes states $\langle 0,0,0,0 \rangle$, $\langle 1,0,0,0 \rangle$, $\langle 1,1,0,0 \rangle$, $\langle 1,1,1,0 \rangle$, $\langle 1,1,1,1 \rangle$, $\langle 0,1,1,1 \rangle$, $\langle 0,0,1,1 \rangle$, and $\langle 0,0,0,1 \rangle$.

A heuristic for adding recovery. In the presence of faults, the program TR may reach states where there exists at least a process P_i $(0 \leq i \leq 3)$ whose x_i is corrupted (i.e., $x_i = \perp$). In such cases, processes P_i and $P_{((i+1)\ mod\ 4)}$ cannot take any transition, and as a result, the propagation of the token stops (i.e., the whole program deadlocks).

In order to recover from the states where there exist some corrupted processes, we apply the heuristic for single-step recovery from [14] in an iterative fashion. Specifically, we identify states from where single-step recovery to a set of states $RecoverySet$ is possible. The initial value of $RecoverySet$ is equal to the program invariant. At each iteration, we include a set of states in $RecoverySet$ from where single-step recovery to $RecoverySet$ is possible.

In the first iteration, we search for deadlock states where there is only one corrupted process in the ring. For example, consider a state $s_0 = \langle 1, \perp, 1, 0 \rangle$. In state s_0, P_1 and P_2 cannot take any transitions. However, P_3 can copy the value of x_2 and reach $s_2 = \langle 1, \perp, 1, 1 \rangle$. Subsequently, P_0 changes x_0 to 0, and as a result, the program reaches state $s_3 = \langle 0, \perp, 1, 1 \rangle$. The state s_3 is a deadlock state since no process can take any transition at s_3. To add recovery from s_3, we allow P_1 to correct itself by copying the value of x_0, which is equal to 0. Thus, by copying the value of x_0, P_1 adds a recovery transition to an invariant state $\langle 0,0,1,1 \rangle$. Therefore, we include s_3 in the set of states $RecoverySet$ in the first iteration. Note that this recovery transition is added in low atomicity in that all the transitions included in action $(x_0 = 0) \wedge (x_1 = \perp) \rightarrow x_1 := 0$ can be included in the fault-tolerant program without violating safety.

In the second and third iterations, we follow the same approach and add recovery from states where there are two or three corrupted processes to states that we have already resolved in the previous iterations. Adding recovery up to the fourth iteration of our heuristic results in the intermediate program ITR $(1 \leq i \leq 3)$.

$ITR_0 : \ ((x_0 = 1) \vee (x_0 = \bot)) \ \wedge \ (x_3 = 1) \quad \longrightarrow x_0 := 0;$

$ITR'_0 : \ ((x_0 = 0) \vee (x_0 = \bot)) \ \wedge \ (x_3 = 0) \quad \longrightarrow x_0 := 1;$

$ITR_i : \ ((x_i = 0) \vee (x_i = \bot)) \ \wedge \ (x_{i-1} = 1) \quad \longrightarrow x_i := 1;$

$ITR'_i : \ ((x_i = 1) \vee (x_i = \bot)) \ \wedge \ (x_{i-1} = 0) \quad \longrightarrow x_i := 0;$

Using above heuristic, we can only add recovery from the states where there exists at least one uncorrupted process. If there exists at least one uncorrupted process P_j $(0 \leq j \leq 3)$ then $P_{((j+1)\ mod\ 4)}$ will initiate the token circulation throughout the ring, and as a result, the program recovers to its invariant. However, in the fourth iteration of the above heuristic, we reach a point where we need to add recovery from the state where all processes are corrupted; i.e., we reach the program state $s_d = \langle \bot, \bot, \bot, \bot \rangle$. In such a state, the program ITR deadlocks as an action of the form $(x_0 = \bot) \wedge (x_1 = \bot) \rightarrow x_1 := 0$ cannot be included in the fault-tolerant program. Such an action can violate safety if x_2 and x_3 are not corrupted. In fact, no process can add safe recovery from s_d in low atomicity. Thus, *Add_Recovery* returns false for $\langle \bot, \bot, \bot, \bot \rangle$.

Adding the actions of the high atomicity pseudo process. In order to add masking fault-tolerance to the program ITR, a process P_{index} $(0 \leq index \leq 3)$ should set its x value to 0 (respectively, 1) when all processes are corrupted. Hence, we follow our synthesis method (cf. Figure 6.1), where the pseudo process PS_0 takes the high atomicity action HTR and recovers from s_d. Thus, the actions of the masking program MTR are as follows $(1 \leq i \leq 3)$.

$MTR_0 : \quad ((x_0 = 1) \vee (x_0 = \bot)) \ \wedge \ (x_3 = 1) \qquad \longrightarrow x_0 := 0;$

$MTR'_0 : \quad ((x_0 = 0) \vee (x_0 = \bot)) \ \wedge \ (x_3 = 0) \qquad \longrightarrow x_0 := 1;$

$$\begin{array}{lll}
MTR_i: & ((x_i = 0) \vee (x_i = \bot)) \ \wedge \ (x_{i-1} = 1) & \longrightarrow x_i := 1; \\
MTR'_i: & ((x_i = 1) \vee (x_i = \bot)) \ \wedge \ (x_{i-1} = 0) & \longrightarrow x_i := 0; \\
HTR: & (x_0 = \bot) \ \wedge \ (x_1 = \bot) \ \wedge \ (x_2 = \bot) \ \wedge \ (x_3 = \bot) & \longrightarrow x_0 := 0;
\end{array}$$

In order to refine the high atomicity action HTR, we need to add a detector that detects the state predicate $(x_0 = \bot) \wedge \ (x_1 = \bot) \wedge \ (x_2 = \bot) \wedge \ (x_3 = \bot)$. In Section 6.3, we describe the specification of fault-tolerance components, and we show how we use a distributed detector to refine high atomicity actions.

Remark. Had we non-deterministically chosen to use PS_i $(i \neq 0)$ as the process that adds the high atomicity recovery action then the high atomicity action HTR would have been different in that HTR would write x_i. (We refer the reader to Section 6.7 for a discussion about this issue.)

6.3 Specifying Pre-Synthesized Components

In this section, we describe the specification of fault-tolerance components (i.e., detectors and correctors). Specifically, we concentrate on detectors and we consider a special subclass of correctors where a corrector consists of a detector and a write action on the local variables of a single process.

6.3.1 The Specification of Detectors

We recall the specification of a detector component presented in [34, 33]. Towards this end, we describe detection predicates, and witness predicates. A detector, say d, identifies whether or not a global state predicate, X, holds. The global state predicate X is called a *detection* predicate in the global state space of a distributed program [34, 33].

It is often difficult to evaluate the truth value of X in an atomic action. Thus, we (i) decompose the detection predicate X into a set of smaller detection predicates $X_0 \cdots X_n$ where the compositional detection of $X_0 \cdots X_n$ leads us to the detection of X, and (ii) provide a state predicate, say Z, whose value leads the detector to the conclusion that X holds. Since when Z becomes true its value witnesses that X is true, we call Z a *witness* predicate. If Z holds then X will have to hold as well. If X holds then Z will eventually hold and continuously remain *true*. Hence, corresponding to each detection predicate X_i, we identify a witness predicate Z_i such that if Z_i is *true* then X_i will be *true*.

The detection predicate X is either the conjunction of X_i $(0 \leq i \leq n)$ or the disjunction of X_i. Since the detection predicates that we encounter represent deadlock states, they are inherently in conjunctive form where each conjunct represents the valuation to program variables at some process. Hence, in the rest of this chapter, we consider the case where X is a conjunction of X_i, for $0 \leq i \leq n$.

Specification. Let X and Z be state predicates. Let 'Z detects X' be the problem specification. Then, 'Z detects X' stipulates that

- (*Safety*) When Z holds, X must hold as well.
- (*Liveness*) When the predicate X holds and continuously remains *true*, Z will eventually hold and continuously remain *true*. □

We represent the safety specification of a detector as a set of transitions that a detector is not allowed to execute. Thus, the following set of transitions represents the safety specification of a detector.

$$spec_d = \{(s_0, s_1) : (Z(s_1) \ \wedge \ \neg X(s_1))\}$$

6.3.2 The Representation of Detectors

In this section, we describe how we formally represent a distributed detector. While our method allows one to use detectors of different topologies (cf. Section 6.4.1), in this section, we comprehensively describe the representation of a linear (sequential) detector as such a detector will be used in our token ring example.

The composition of detectors. A detector, say d, with the detection predicate $X \equiv X_0 \wedge \ldots \wedge X_n$ is obtained by composing d_i, $0 \leq i \leq n$, where d_i is responsible for the detection of X_i using a witness predicate Z_i $(0 \leq i \leq n)$. The elements of d can execute in parallel or in sequence. More specifically, parallel detection of X requires $d_0 \cdots d_n$ to execute concurrently. As a result, the state predicate $(Z_0 \wedge \cdots \wedge Z_n)$ is the witness predicate for detecting X.

A sequential detector requires the detectors $d_0, \cdots, d_n$ to execute one after another. For example, given a linear arrangement $d_n \cdots d_0$, a detector d_i $(0 \leq i < n)$ detects its detection predicate, using Z_i, after d_{i+1} witnesses. Thus, when Z_i becomes *true*, it shows that Z_{i+1} already holds. Since when Z_i becomes *true* X_i must be also *true*, it follows that the detection predicates $X_n \cdots X_i$ hold. Therefore, we can atomically check the witness predicate Z_0 in order to identify whether or not $X \equiv (X_n \wedge \cdots \wedge X_0)$ holds.

The detection of global state predicates of programs that have a hierarchical topology (e.g., tree-like structures) requires parallel and sequential detectors. In this section, we demonstrate our method in the context of a linear detector as such a detector suffices for the token ring example. In Section 6.6, we apply our synthesis method for the synthesis of a diffusing computation program using components with hierarchical topology.

A linear detector. We consider a detector d with linear topology. The detector d consists of $n+1$ elements $(n > 0)$, its specification $spec_d$, its variables, and its invariant U. Since the structure of the detector is linear, without loss of generality, we consider an arrangement $d_n \cdots d_0$ for the elements of the distributed detector, where the left-most element is d_n and the right-most element is d_0.

Component variables. Each element d_i, $0 \leq i \leq n$, of the detector has a Boolean variable y_i.

Read/write restrictions. Element d_i can read y_i and y_{i+1}, and can only write y_i $(0 \leq i < n)$. d_n reads and writes y_n. Also, d_i is allowed to read all variables that P_i can read (i.e., the process with which d_i is composed).

Witness predicates. The witness predicate of each d_i, say Z_i, is equal to $(y_i = true)$.

The detector actions. The actions of the linear detector are as follows $(0 \leq i < n)$.

$$DA_n : (LC_n) \wedge (y_n = false) \longrightarrow y_n := true;$$
$$DA_i : (LC_i) \wedge (y_i = false) \wedge (y_{i+1} = true) \longrightarrow y_i := true;$$

Using action DA_i $(0 \leq i < n)$, each element d_i of the linear detector witnesses (i.e., sets the value of y_i to *true*) whenever (i) the condition LC_i becomes *true*, where LC_i represents a local condition that d_i atomically checks (by reading the variables of P_i), and (ii) its neighbor d_{i+1} has already witnessed. The detector d_n witnesses (using action DA_n) when LC_n becomes true.

Detection predicates. The detection predicate X_i for element d_i is equal to $(LC_n \wedge \cdots \wedge LC_i)$ $(0 \leq i \leq n)$. Therefore, d_0 detects the global detection predicate $LC_n \wedge \cdots \wedge LC_0$.

Invariant. During the detection, when an element d_i sets y_i to true, the elements d_j, for $i < j \leq n$, have already set their y values to true. Hence, we represent the invariant of the linear detector by the predicate U, where

$$U = \{s : (\forall i : (0 \leq i \leq n) : (y_i(s) \Rightarrow (\forall j : (i < j \leq n) : LC_j))\}$$

Faults. We model the fault transitions that affect the linear detector using the following action (cf. Section 6.7 for a discussion about the way that we have modeled the faults).

$$F : true \longrightarrow y_i := false;$$

Theorem 6.1 The linear detector is masking F-tolerant for 'Z detects X' from U.

Proof. The linear detector satisfies 'Z detects X' from U. Also, in the presence of F, no element d_i $(0 \leq i \leq n)$ of the detector will reach a state where d_i witnesses incorrectly. As a result, the linear detector never violates the safety of 'Z detects

X' in the presence of F. Also, when faults stop occurring, the actions of the linear detector correct the corrupted values of y_i if necessary. Thus, every computation of the linear detector in the presence of F will eventually reach a state in U. Therefore, the linear detector component is masking F-tolerant for 'Z detects X' from U. $\square$

6.3.3 Token Ring Example Continued

In Section 6.2.2, we added the following high atomicity action to the token ring program ITR that is executed by the pseudo process PS_0.

$$HTR: \quad (x_0 = \bot) \ \wedge \ (x_1 = \bot) \ \wedge \ (x_2 = \bot) \ \wedge \ (x_3 = \bot) \quad \longrightarrow \quad x_0 := 0$$

In order to synthesize a distributed program (that includes low atomicity actions), we need to refine the guard of the above action. The read/write restrictions of the processes in the token ring program identify the underlying communication topology of the fault-intolerant program, which is a ring. Hence, we select a linear detector, d, so that we can organize its elements, d_3, d_2, d_1, d_0, in the ring. Each detector d_i is responsible to detect whether or not the local conditions LC_3 to LC_i hold ($LC_i \equiv (x_i = \bot)$), for $0 \leq i \leq 3$. Thus, the detection predicate X_i is equal to $((x_3 = \bot) \wedge \cdots \wedge (x_i = \bot))$, for $0 \leq i \leq 3$. As a result, the global detection predicate of the linear detector is $((x_3 = \bot) \wedge (x_2 = \bot) \wedge (x_1 = \bot) \wedge (x_0 = \bot))$. The witness predicate of each d_i, say Z_i, is equal to $(y_i = true)$, and the actions of the sequential detector are as follows ($0 \leq i \leq 2$).

$$DA_3: \ (x_3 = \bot) \ \wedge \ (y_3 = false) \quad \longrightarrow \quad y_3 := true;$$
$$DA_i: \ (x_i = \bot) \ \wedge \ (y_i = false) \ \wedge \ (y_{i+1} = true) \quad \longrightarrow \quad y_i := true;$$

Note that we replace (LC_i) with $(x_i = \bot)$ in the above actions. During the synthesis, after the synthesis algorithm acquires the actions of its required component, it replaces each (LC_i) with the appropriate condition in order to create the transition groups corresponding to each action of the component.

6.4 Using Pre-Synthesized Components

In this section, we describe how we perform the second and the third step of our synthesis approach presented in Figure 6.1. In particular, in Section 6.4.1, we show how we automatically specify the required components during the synthesis. Then, in Section 6.4.3, we show how we ensure that no interference exists between the program and the fault-tolerance component. Afterwards, we present an algorithm for the addition of fault-tolerance components. In Sections 6.4.2 and 6.4.4, we respectively present the algorithmic specification and the algorithmic addition of a linear detector to the token ring program.

6.4.1 Algorithmic Specification of the Fault-Tolerance Components

We present the Component_Specification algorithm (cf. Figure 6.2) that takes a deadlock state s_d, the distribution restrictions (i.e., the read/write restrictions) of the program being synthesized, and the set of high atomicity pseudo processes PS_i $(0 \leq i \leq n)$. First, the algorithm searches for a high atomicity process PS_{index} that is able to add a high atomicity recovery action, $ac : grd \rightarrow st$, from s_d to a state in the state predicate S_{rec}, where S_{rec} represents the set of states from where there exists a safe recovery path to the invariant. Also, we verify the closure of $S_{rec} \cup s_d$ in the computations of $p[]f$. If there exists such a process PS_{index} then the algorithm returns a triple $\langle X, R, index \rangle$, where (i) X is the detection predicate that should be refined in the refinement of the action ac; (ii) R is a relation that represents the topology of the program, and (iii) the $index$ is an integer that identifies the process that should detect grd and execute st.

The Component_Specification algorithm constructs the state predicate X using the LC_i conditions. Each LC_i condition is by itself a conjunction that consists of the program variables readable for process P_i. Therefore, the predicate X will be the conjunction of LC_i conditions ($0 \leq i \leq n$).

```
Component_Specification($s_d$: state, $S_{rec}$: state predicate, $PS_0, \cdots, PS_n$: high atomicity pseudo
      process, $spec$: safety specification, $r_0, \cdots, r_n$: read restrictions, $w_0, \cdots, w_n$: write restrictions)
{ // $n$ is the number of processes.
  if ( $\exists index : 0 \leq index \leq n : (\exists s : s \in S_{rec} : (s_d, s) \in PS_{index} \ \wedge \ ((s_d, s)$ does not violate $spec) \wedge$
              $(\forall x : (x(s_d) \neq x(s)) : x \in w_{index}))$ )
  then     $X := \wedge_{i=0}^{n}(LC_i)$, where $LC_i = (\wedge^{|r_i|}(x = x(s_d)))$;
           $R = \{\langle i, j\rangle : (0 \leq i \leq n) \wedge (0 \leq j \leq n) : w_i \subseteq r_j\}$;
           return $X$, $R$, $index$;
  else return $false$, $\emptyset$, -1;
}
```

Figure 6.2: Automatic specification of a component.

The relation $R \subseteq (P \times P)$ identifies the communication topology of the distributed program, where P is the set of program processes. We represent R by a finite set $\{\langle i, j\rangle : (0 \leq i \leq n) \wedge (0 \leq j \leq n) : w_i \subseteq r_j\}$ that we create using the read/write restrictions among the processes. The presence of a pair $\langle i, j\rangle$ in R shows that there exists a communication link between P_i and P_j. Since we internally represent R by an undirected graph, we consider the pair $\langle i, j\rangle$ as an unordered pair.

The interface of the fault-tolerance components. The format of the interface of each component is the same as the output of the Component_Specification algorithm, which is a triple $\langle X, R, index\rangle$ as described above. We use this interface to extract a component from the component library using a pattern-matching algorithm. To achieve this goal, we use existing specification-matching techniques [35] for extracting components from the component library.

The output of the component library. Given the interface $\langle X, R, index\rangle$ of a required component, the component library returns the witness predicate, Z, the invariant, U, and the set of transition groups, $gd_0 \cup \cdots \cup gd_k \cup g_{index}$, of the pre-synthesized component ($k \geq 0$). The group of transitions g_{index} represents the low atomicity write action that should be executed by process P_{index}.

Complexity. Since the algorithm Component_Specification checks the possibility of adding a high atomicity recovery action to each state of S_{rec}, its complexity is polynomial in the number of states of S_{rec}.

6.4.2 Token Ring Example Continued

We trace the algorithm of Figure 6.2 for the case of the token ring program. First, we non-deterministically identify PS_0 as the process that can read every program variable and can add a high atomicity recovery transition from the deadlock state $s_d = \langle \bot, \bot, \bot, \bot \rangle$. Thus, the value of *index* will be equal to 0. Second, we construct the detection predicate X, where $X \equiv ((x_0 = \bot) \wedge (x_1 = \bot) \wedge (x_2 = \bot) \wedge (x_3 = \bot))$. Finally, using the read/write restrictions of the processes in the token ring program, the relation R will be equal to $\{\langle 0, 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, \langle 3, 0\rangle\}$.

6.4.3 Algorithmic Addition of The Fault-Tolerance Components

In this section, we present an algorithm for adding a fault-tolerance component to a fault-intolerant distributed program to resolve a deadlock state s_d. Before the addition, we ensure that no interference exists between the program and the fault-tolerance component that we add. We show that our addition algorithm is sound; i.e., the synthesized program satisfies the requirement of the addition problem (cf. Section 6.1).

We recall the structure of the fault-intolerant program, p, from the first paragraph of Section 6.2.1. We represent the transitions of p by the union of its groups of transitions (i.e., $\cup_{i=0}^{m} g_i$). We also assume that we have extracted the required pre-synthesized component, c, as described in Section 6.4.1. The component c consists of a detector d that includes a set of transition groups $\cup_{i=0}^{k} gd_i$, and the write action of the pseudo process PS_{index} represented by a group of transitions g_{index} in the low atomicity.

The state space of the composition of p and d is the new state space $S_{p'}$. We introduce an onto function $H_1 : S_{p'} \rightarrow S_p$ (respectively, $H_2 : S_{p'} \rightarrow S_d$, where S_d is the state space of the detector d) that maps the states in the new state space $S_{p'}$ to the states in the old state space S_p (respectively, S_d). Now, we show how we verify the interference-freedom of the composition of c and p.

Interference-freedom. We say the program p and the fault-tolerance component c interfere iff the execution of one of them violates the (safety or liveness) specification of the other one. In order to ensure that no interference exists between p and c, we verify the following three conditions in the new state space $S_{p'}$: (i) transitions of p do not interfere with the execution of d; (ii) transitions of d do not interfere with the execution of p, and (iii) the low atomicity write action associated with c does not interfere with the execution of p and d. Towards this end, we present the algorithm Interfere in Figure 6.3.

```
Interfere($S, S_{rec}, U$ : state predicate, $H_1, H_2$: onto mapping function,
                                    $spec, spec_d$: safety specification,
                                    $g_0, \cdots, g_m, gd_0, \cdots, gd_k, g_{index}$: groups of transitions)
// Checks the interference-freedom between the program and
// the fault-tolerance component.
{ // $p = g_0 \cup \cdots \cup g_m$, and $d = gd_0 \cup \cdots \cup gd_k \cup g_{index}$
// $P_0 \cdots P_n$ are the processes of $p$, and $d_0 \cdots d_n$ are the elements of $d$

  $I_1 = \{g : (\exists g_j : (g_j \in p) \wedge (0 \leq j \leq m) : (H_1(g) = g_j) \wedge$
                   $(\exists (s'_0, s'_1) : (s'_0, s'_1) \in g : ((s'_0, s'_1)$  violates  $spec_d) \vee$
                                                $(H_2(s'_0) \in U \ \wedge \ H_2(s'_1) \notin U))\}$
  if ($I_1 \neq \emptyset$) then return true;
  $I_2 = \{gd : (\exists gd_j : (gd_j \in d) \wedge (0 \leq j \leq k) : (H_2(gd) = gd_j) \wedge$
                   $(\exists (s'_0, s'_1) : (s'_0, s'_1) \in gd : ((s'_0, s'_1)$  violates  $spec) \vee$
                                                $(H_1(s'_0) \in S \ \wedge \ H_1(s'_1) \notin S))\}$
  if ($I_2 \neq \emptyset$) then return true;
  $I_3 := \{g : (H_2(g) = g_{index}) \ \wedge \ (\exists (s'_0, s'_1) : (s'_0, s'_1) \in g : ((s'_0, s'_1)$  violates  $spec_d) \vee$
                   $(H_1(s'_1) \notin S_{rec}) \ \vee \ (H_1(s'_0) \in S \ \wedge \ H_1(s'_1) \notin S) \vee$
                   $(H_2(s'_0) \in U \ \wedge \ H_2(s'_1) \notin U) \ \vee \ ((s'_0, s'_1)$  violates  $spec))\}$
  if ($I_3 \neq \emptyset$) then return true;
  return false;
}
```

Figure 6.3: Verifying the interference-freedom conditions.

First, we ensure that the set of transitions of p do not interfere with the execution of d by constructing the set of groups of transitions I_1, where I_1 contains those groups of transitions in the new state space $S_{p'}$ that violate either the safety of d or the closure of its invariant U. The transitions of p do not interfere with the liveness of d because d executes only when p is deadlocked in the state s_d. Hence, we are only concerned with the safety of the detector d and the closure of U. When we map the transitions of p to the new state space, the mapped transitions should preserve the safety of d. Moreover, if the image of a transition (s'_0, s'_1) starts in U (i.e., $H_2(s'_0) \in U$) then the image of (s'_0, s'_1) will have to end in U (i.e., $H_2(s'_1) \in U$). The emptiness of I_1 shows that the transitions of p do not interfere with the execution of d.

Second, using a similar argument, we construct the set of groups of transitions I_2 in the new state space $S_{p'}$ whose every transition is a mapping of the transitions of d that violate either the safety of *spec* or violate the closure of the program invariant S.

Third, if I_1 and I_2 are empty then it will follow that the detector d is able to detect s_d without interfering with p. However, after d detects its detection predicate, the component c performs a write action to change the state of the program from s_d to a state $s \in S_{rec}$, where S_{rec} is the set of states from where safe recovery has already been added. If a transition in the group associated with the write transition (s_d, s) violates (i) the safety of the detector; (ii) the safety of the program; (iii) the closure of U, or (iv) the closure of S then the recovery action will interfere with the program (see the construction of I_3 in Figure 6.3). If I_1, I_2, and I_3 are empty then the Interfere algorithm declares that no interference will happen due to the addition of c to p.

Addition. We present the Add_Component algorithm for an interference-free addition of the fault-tolerance component c to p. Thus, if the Interfere algorithm returns *false* then we will invoke Add_Component (cf. Figure 6.4). In the new state space $S_{p'}$, we construct a set of transition groups p_{H_1} (respectively, d_{H_2}) that includes all groups of transitions,

g, whose images in S_p (respectively, S_d) belong to p (respectively, d). Besides, no transition of $(s'_0, s'_1) \in g$ violates the safety specification of d (respectively, p) or the closure of the invariant of d (respectively, p), i.e., U (respectively, S). In the calculation of d_{H_2}, we note that the image of every group g in d and p must belong to the same process (cf. condition $(l = i)$ in the construction of d_{H_2}).

```
Add_Component($S, S_{rec}, U$: state predicate, $H_1, H_2$: onto mapping function,
                    $spec, spec_d$: safety specification,
                    $g_0, \cdots, g_m$, $gd_0, \cdots, gd_k, g_{index}$: groups of transitions)
{ // $p = g_0 \cup \cdots \cup g_m$, and $d = gd_0 \cup \cdots \cup gd_k \cup g_{index}$
// $P_0 \cdots P_n$ are the processes of $p$, and $d_0 \cdots d_n$ are the elements of $d$

  $p_{H_1} = \{g : (\exists g_j : (g_j \in p) \wedge (0 \leq j \leq m) : (H_1(g) = g_j) \wedge$
                $(\forall (s'_0, s'_1) : (s'_0, s'_1) \in g : ((s'_0, s'_1)$ does not violate $spec_d) \wedge$
                                        $(H_2(s'_0) \in U \Rightarrow H_2(s'_1) \in U))\}$

  $d_{H_2} = \{gd : (\exists gd_j : (gd_j \in d) \wedge (0 \leq j \leq k) : (H_2(gd) = gd_j) \wedge$
                $(\exists d_i, P_l : (0 \leq i \leq n) \wedge (0 \leq l \leq n) :$
                                $(H_2(gd) \in d_i) \wedge (H_1(gd) \in P_l) \wedge (l = i)) \wedge$
                $(\forall (s'_0, s'_1) : (s'_0, s'_1) \in gd : ((s'_0, s'_1)$ does not violate $spec) \wedge$
                                        $(H_1(s'_0) \in S \Rightarrow H_1(s'_1) \in S))\}$

  $p_c := \{g : (H_2(g) = g_{index}) \wedge (\forall (s'_0, s'_1) :$
                $(s'_0, s'_1) \in g : ((s'_0, s'_1)$ does not violate $spec) \wedge (H_1(s'_1) \in S_{rec}) \wedge$
                $(H_2(s'_0) \in U \Rightarrow H_2(s'_1) \in U) \vee ((s'_0, s'_1)$ does not violate $spec_d))\}$
  $S' := \{s : s \in S_{p'} : H_1(s) \in S \wedge H_2(s) \in U\}$
  $p' := p_{H_1} \cup d_{H_2} \cup p_c$;
  return $p', S'$;
}
```

Figure 6.4: The automatic addition of a component.

The set p_c includes all groups of transitions, g, whose every transition has an image in g_{index} under the mapping H_2. Further, no transition $(s'_0, s'_1) \in g$ violates the safety of *spec* or the closure of S.

The set of states of the invariant of the synthesized program, S', consists of those states whose images in S_p belong to the program invariant S and whose images in the state space of the detector, S_d, belong to the detector invariant U.

Theorem 6.2 The algorithm Add_Component is sound. $\square$

Theorem 6.3 The complexity of Add_Component is polynomial in $|S'_p|$. □

Before we show the soundness of Add_Component, we make some observations and present the following preliminary lemmas and theorems. Towards this end, we assume that we are given a program p, its specification *spec*, its invariant S, its state space S_p, faults f, and a deadlock state $s_{deadlock} \notin S$. We consider the case where we have already added safety to p and we only need to resolve $s_{deadlock}$ to synthesize the masking fault-tolerant program p' with the invariant S' in the new state space $S_{p'}$. Towards this end, we use Add_Component algorithm for adding a fault-tolerance component c to p.

The component c consists of a distributed detector d, with the detection predicate X, the witness predicate Z, an invariant U, and a low atomicity write action $Z \rightarrow st$ that takes p from state $s_{deadlock}$ to a state $s \in S_{rec}$. The state predicate S_{rec} represents the set of states from where a *safe* recovery to the invariant S is guaranteed. By definition, the set of states S_{rec} includes the invariant S; i.e., $S \subseteq S_{rec}$. Also, the set $S_{rec} \cup s_{deadlock}$ is closed in the computations of $p[]f$. However, because of the deadlock state $s_{deadlock}$, recovery to S is not guaranteed from $S_{rec} \cup s_{deadlock}$.

We define two mapping functions H_1 and H_2 respectively from $S_{p'}$ to S_p and from $S_{p'}$ to S_d, where S_d is the state space of the distributed detector d included in c. In the Add_Component algorithm, based on the construction of S', we include those states in S' whose images in S_p belong to S. Thus,

Observation 6.4 $\forall s : s \in S' : H_1(s) \in S$ □

Now, we present the following theorem.

Theorem 6.5 $H_1(S') \subseteq S$.

Proof. The proof follows from Observation 6.4. □

By construction, for every arbitrary group of transitions $g \in p_{H_1}$ (cf. Figure 6.4) there exists a group of transitions $g_j \in p$ $(0 \leq j \leq m)$. Now, if we consider a transition $(s'_0, s'_1) \in g$ such that $s'_0 \in S'$ and $s'_1 \in S'$ then using Observation 6.4, $H_1(s'_0) \in S$ and

$H_1(s'_2) \in S$. As a result, the condition $(H_1(s'_0), H_1(s'_0)) \in p|H_1(S')$ holds. Thus, we have

Observation 6.6 $\forall (s'_0, s'_1) : (s'_0, s'_1) \in p_{H_1} : (((s'_0, s'_1) \in p'|S') \Rightarrow (H_1((s'_0, s'_1)) \in p|H_1(S')))$ $\square$

$(H_1((s'_0, s'_1))$ denotes the transition $(H_1(s'_0), H_1(s'_1))$ in the state space S_p.)

Using a similar argument, we present the following observation.

Observation 6.7 $\forall (s'_0, s'_1) : (s'_0, s'_1) \in d_{H_2} : (((s'_0, s'_1) \in p'|S') \Rightarrow (H_1((s'_0, s'_1)) \in p|H_1(S')))$ $\square$

The transition groups of p_c add recovery to $s_{deadlock}$. Also, by construction, for every transition $(s'_{deadlock}, s'_1) \in p_c$, $Z(s'_{deadlock})$ holds. Thus, at $s'_{deadlock}$ the detector detects the deadlock state $s_{deadlock}$. Since $s_{deadlock} \notin S$, the state $s'_{deadlock}$ does not belong to S'. It follows that $(s'_{deadlock}, s'_1) \notin p'|S'$. Therefore, we observe that

Observation 6.8 $\forall (s'_0, s'_1) : (s'_0, s'_1) \in p_c : (((s'_0, s'_1) \in p'|S') \Rightarrow (H_1((s'_0, s'_1)) \in p|H_1(S')))$ $\square$

Using above observations, we present the second theorem.

Theorem 6.9 $H_1(p'|S') \subseteq p|H_1(S')$.

Proof. By the construction of p', the proof follows from Observations 6.6, 6.7, and 6.8. $\square$

To show that p' is masking f-tolerant for *spec*, we prove the following lemmas.

Lemma 6.10 From every state of S'_{rec} safe recovery to S' with respect to *spec* is guaranteed.

Proof. By definition, from every state of S_{rec} safe recovery to S is guaranteed with respect to *spec*. Now, let *cmp* be a computation of $p'[]f$ that starts from a state in S'_{rec}. If *cmp* violates *spec* then there exists a computation prefix of *cmp* that violates *spec*. Let $\langle s'_0, s'_1, ..., s'_n \rangle$ be the smallest such prefix. It follows that $(s'_{(n-1)}, s'_n)$ violates the safety of *spec*. As a result, $(H_1(s'_{(n-1)}), H_1(s'_n))$ is a transition of program p that violates *spec*. Thus, the corresponding computation prefix $\langle H_1(s'_0), H_1(s'_1), ..., H_1(s'_n)\rangle$ violates *spec*. Hence, we find a computation prefix in S_{rec} that is not safe. This contradicts with the assumption that from every state of S_{rec} safe recovery to S with respect to *spec* is guaranteed.

If $(s'_{(n-1)}, s'_n)$ is a fault transition then the corresponding fault transition $(H_1(s'_{(n-1)}), H_1(s'_n))$ violates *spec*. Hence, we could find a state of p in the state space S_p (i.e., $H_1(s'_{(n-1)})$) from where faults alone violate *spec*. This contradicts with the assumption that we have already added safety to p.

Now, let *cmp* be a computation of p' that starts from a state in S'_{rec} and never reaches S'. Since the computations of p' are infinite, there must exist a prefix $\langle s'_0, s'_1, ..., s'_n, s'_0\rangle$ of *cmp* that includes a cycle. Now, using function H_1, we calculate the computation prefix $\langle s_0, s_1, ..., s_n, s_0\rangle$ in the old state space S_p, where $H_1(s'_i) = s_i$ $(0 \leq i \leq n)$. As a result, starting at $s_0 \in S_{rec}$, we find a computation prefix that includes a cycle and never reaches S, which is a contradiction with the definition of S_{rec}. Therefore, from every state of S'_{rec} safe recovery to S' with respect to *spec* is guaranteed. □

Lemma 6.11 From every state of S'_{rec}, no computation prefix of $p'[]f$ that ends in S' violates the safety specification of the detector d (i.e., $spec_d$).

Proof. Let *cmp* be a computation of $p'[]f$ that starts from a state in S'_{rec}. If *cmp* violates $spec_d$ then there exists a computation prefix of *cmp* that violates *spec*. Let $\langle s'_0, s'_1, ..., s'_n\rangle$ be the smallest such prefix. It follows that $(s'_{(n-1)}, s'_n)$ violates $spec_d$. Thus, the transition $(H_2(s'_{(n-1)}), H_2(s'_n))$ violates $spec_d$; i.e., the detector d and the program p interfere. By the construction of the transitions of p', no transition of p' interferes with the execution of d. Thus, the computation prefix *cmp* does not violate $spec_d$.

Also, since we showed (cf. Theorem 6.1) that the fault-tolerance component d is by itself F-tolerant, $(H_2(s'_{(n-1)})$, $H_2(s'_n))$ cannot be a fault transition that violates

$spec_d$. Therefore, starting from every state in S'_{rec}, every computations of $p'[]f$ satisfy $spec_d$. □

Lemma 6.12 $T' = S'_{rec} \cup \{s'_{deadlock}\}$ is a valid fault-span for p' in the new state space $S_{p'}$ (i.e., $H_1(T') = S_{rec} \cup \{s_d\}$).

Proof. By construction, we have $S \subseteq S_{rec}$. Hence, using function H_1, we have $S' \subseteq S'_{rec}$. Otherwise, if there exists a state $s'_0 \in S'$ such that $s'_0 \notin S'_{rec}$ then we will have a state $s_0 \in S$, where $H_1(s'_0) = s_0$, that is not in S_{rec}, which is a contradiction with $S \subseteq S_{rec}$. Hence, we have $S' \subseteq S'_{rec}$. Also, by assumption, the set $S_{rec} \cup s_{deadlock}$ is closed in the computations of $p[]f$. As a result, $S'_{rec} \cup s'_{deadlock}$ is closed in the computations of $p'[]f$. It follows that T' is a valid fault-span since it is closed in $p'[]f$, and $S' \subseteq T'$. □

Using T', we present the following lemmas.

Lemma 6.13 $p'[]f$ satisfies *spec* and $spec_d$ from T'.

Proof. Using Lemmas 6.10 and 6.11, $p'[]f$ satisfies *spec* and $spec_d$ from S'_{rec}. We only need to show that $p'[]f$ satisfies *spec* and $spec_d$ from $s'_{deadlock}$, where $H_1(s'_{deadlock}) = s_{deadlock}$. By the construction of p_c, no transition originated at $s'_{deadlock}$ violates *spec* or $spec_d$. Therefore, starting from every state at T', $p'[]f$ satisfies *spec* and $spec_d$. □

Lemma 6.14 Every computation of $p'[]f$ that starts from a state in T', where $H_1(T') = S_{rec} \cup \{s_d\}$, contains a state in S'.

Proof. Using Lemma 6.10, it follows that every computation of $p'[]f$ that starts from a state in S'_{rec}, where $H_1(S'_{rec}) = S_{rec}$, reaches a state in S'. Moreover, by the construction of p', transitions of p_c provide safe recovery from $s'_{deadlock}$ to a state in S'_{rec}, where $H_1(s'_{deadlock}) = s_{deadlock}$. Since safe recovery from every state of S'_{rec} to S' is guaranteed, every computation of p' that starts from a state in T' contains a state in S'. □

Theorem 6.15 p' is masking f-tolerant for *spec* from S'.

Proof. First, we show that S' is an invariant of p'. We consider a transition (s'_0, s'_1) of p' that starts in S' and ends outside S'. Since $s'_0 \in S'$, by Observation 6.4, we have $H_1(s'_0) \in S$. Also, from the construction of S', we have $H_1(s'_1) \notin S$. As a result, we find a transition $(H_1(s'_0), H_1(s'_1))$ of p that starts in S and ends outside S, which is a contradiction with the closure of S in p. Thus, the execution of p' is closed in S'.

From Theorem 6.9, it follows that p' satisfies *spec* from S'. Thus, S' is an invariant of p'. Therefore, using S' as an invariant and T' as a fault-span, and based on Lemmas 6.13, and 6.14, we have shown that p' is masking f-tolerant for *spec* from S'. □

Theorem 6.2 (Soundness) The algorithm Add_Component is sound.

Proof. To prove that our algorithm is sound, we have to show that the conditions of the addition problem are satisfied.

1. $H_1(S') \subseteq S$. (cf. Theorem 6.5).

2. $H_1(p'|S') \subseteq p|H_1(S')$. (cf. Theorem 6.9).

3. p' is masking f-tolerant for *spec* from S'. (cf. Theorem 6.15). □

Theorem 6.3 The complexity of Add_Component is polynomial in $S_{p'}$.

Proof. The Add_Component algorithm consists of three parts where we construct the set of transitions p_{H_1}, d_{H_2}, and p_c. Respectively, each one of these sets contains a set of transition groups in the new state space $S_{p'}$. The size of the new state space is in the order of $|S_p| \cdot |S_d|$ (i.e., $|S_{p'}| = |S_p| \cdot |S_d|$). As a result, the size of each transition group cannot be more than $|S_{p'}| \cdot |S_{p'}|$ in $S_{p'}$.

To construct p_{H_1}, we process all groups of transitions that belong to p_{H_1}. Thus, in the worst case, we need to process m groups of transitions in the new state space $S_{p'}$, where m is the number of groups. As a result, the worst-case complexity for constructing p_{H_1} is in the order of $m \cdot |S_{p'}|^2$. The same reasoning holds for the worst-case complexity for constructing d_{H_2} and p_c. Therefore, the complexity of the Add_Component algorithm is polynomial in the size of the $S_{p'}$; i.e., $|S_{p'}|$. $\square$

6.4.4 Token Ring Example Continued

Using Add_Component, we add the detector specified in Section 6.4.2 to the token ring program MTR introduced in Section 6.2.2. The resulting program, consisting of the processes $P_0 \cdots P_3$ arranged in a ring, is masking fault-tolerant to process-restart faults. We represent the transitions of P_0 by the following actions.

$MTR_0 :$ $((x_0 = 1) \vee (x_0 = \bot)) \ \wedge \ (x_3 = 1)$ $\longrightarrow x_0 := 0;$

$MTR'_0 :$ $((x_0 = 0) \vee (x_0 = \bot)) \ \wedge \ (x_3 = 0)$ $\longrightarrow x_0 := 1;$

$D_0 :$ $(x_0 = \bot) \ \wedge \ (y_0 = false) \ \wedge \ (y_1 = true)$ $\longrightarrow y_0 := true;$

$C_0 :$ $(y_0 = true)$ $\longrightarrow x_0 := 0; \ y_0 := false;$

The actions MTR_0 and MTR'_0 are the same as the actions of the MTR program presented in Section 6.2.2. The action D_0 belongs to the sequential detector that sets the witness predicate Z_0 to true. The action C_0 is the recovery action that P_0 executes whenever the witness predicate $(y_0 = true)$ becomes *true*. Now, we present the actions of P_3.

$MTR_3 :$ $((x_3 = 0) \vee (x_3 = \bot)) \ \wedge \ (x_2 = 1)$ $\longrightarrow x_3 := 1; y_3 := false;$

$MTR'_3 :$ $((x_3 = 1) \vee (x_3 = \bot)) \ \wedge \ (x_2 = 0)$ $\longrightarrow x_3 := 0; y_3 := false;$

$D3 :$ $(x_3 = \bot) \ \wedge \ (y_3 = false)$ $\longrightarrow y_3 := true;$

The action D_3 belongs to the detector that sets Z_3 to *true*. We present the actions of P_1 and P_2 as the following parameterized actions (for $i = 1, 2$).

$MTR_i :$ $((x_i = 0) \vee (x_i = \bot)) \ \wedge \ (x_{i-1} = 1)$ $\longrightarrow x_i := 1; y_i := false;$

$MTR'_i :$ $((x_i = 1) \vee (x_i = \bot)) \ \wedge \ (x_{i-1} = 0)$ $\longrightarrow x_i := 0; y_i := false;$

$D_i :$ $(x_i = \bot) \ \wedge \ (y_i = false) \ \wedge \ (y_{i+1} = true)$ $\longrightarrow y_i := true;$

The above program is masking fault-tolerant for the faults that corrupt one or more processes. Note that when a process P_i ($1 \leq i \leq 3$) changes the value of x_i to a non-corrupted value, it falsifies Z_i (i.e., y_i). The falsification of Z_i is important during the recovery from $s_d = \langle \bot, \bot, \bot, \bot \rangle$ in that when x_i takes a non-corrupted value, the detection predicate X_i no longer holds. Thus, if Z_i remains true then the detector d_i witnesses incorrectly, and as a result, violates the safety of the detector. However, P_0 does not need to falsify its witness predicate Z_0 in actions MTR_0 and MTR'_0 because the action C_0 has already falsified Z_0 during a recovery from s_d.

Remark. One could argue that we could have selected a different linear order $d_0 \cdots d_3$ for the detector added to the token ring program. To address this issue, we note that in the case of token ring program a detector with such linear arrangement would interfere with the execution of the program (cf. Section 6.7 for details).

6.5 Example: Alternating Bit Protocol

In this section, we reuse the linear component used in the synthesis of the token ring program presented in this chapter in the synthesis of a fault-tolerant alternating bit protocol (ABP). The ABP program consists of a sender and a receiver processes connected by a communication link that is subject to message loss faults. Using the synthesis method presented in this chapter, we add pre-synthesized components to synthesize an alternating bit protocol that is nonmasking fault-tolerant; i.e., when faults occur the program guarantees recovery to its invariant. However, during recovery, the nonmasking fault-tolerant protocol may violate its safety specification.

The alternating bit protocol (ABP). The fault-intolerant program consists of two processes: a sender and a receiver. The sender reads from an infinite input stream of data packets and sends the newly read packet to the receiver. The receiver copies each received packet into an infinite output stream. When the sender sends a data packet, it waits for an acknowledgement from the receiver before it sends the next packet.

Also, when the receiver receives a new data packet, it sends an acknowledgment bit back to the sender. A one-bit message header suffices to identify the data packet currently being sent since at every moment there exists at most one unacknowledged data packet. Using this identifier bit, the sender (respectively, the receiver) does not need count the total number of packets sent (respectively, received).

Both processes have read/write access to a send channel and a receive channel. The send channel is represented by an integer variable cs and the variable cr models the receive channel. The domain of cs (respectively, cr) is $\{-1, 0, 1\}$, where 0 and 1 represent the value of the data bit in the channel and -1 represents an empty channel. Since we are only concerned about the synchronization between the sender and the receiver, we do not explicitly consider the actual data being sent. Thus, we consider the contents of cs and cr to be a single binary digit. The sender process has a Boolean variable bs that stores the data bit that identifies the data packet currently being sent to the receiver. Correspondingly, the receiver process has a Boolean variable br that represents the value that is supposed to be received. When the sender process transmits a data packet, it waits for a confirmation from the receiver before it sends the next packet. To represent the mode of operation, the sender process uses a Boolean variable rs. The value of rs is 0 iff the sender is waiting for an acknowledgement. Likewise, the receiver process uses a Boolean variable rr such that the value of rr is 0 iff the receiver is waiting for a new packet.

We represent a state s of the ABP program by a 6-tuple $\langle rs, bs, rr, br, cs, cr\rangle$. Thus, if we start from initial state $\langle 1, 1, 0, 0, -1, -1\rangle$, then the sender process begins to send a data bit 1 while the receiver waits to receive it. We represent the transitions of the sender process in the fault-intolerant program ABP by the following actions.

$Send_0 : \ (rs = 1) \quad \longrightarrow \quad rs := 0;\ cs := bs;$

$Send_1 : \ (cr \neq -1) \quad \longrightarrow \quad rs := 1;\ cr := -1;\ bs := (bs + 1) \mod 2;$

Using action $Send_0$, the sender sends another packet to the receiver when it is not waiting for an acknowledgment. Thus, by setting rs to 0, the sender moves to the sate where it waits for an acknowledgment from the receiver. If the receive channel is non-empty (i.e., $(cr \neq -1)$) then the sender reads the receive channel and becomes ready for sending the next packet. The actions of the receiver process in the fault-intolerant program ABP are as follows:

$$
\begin{array}{llll}
Rec_0: & (cs \neq -1) & \longrightarrow & cs := -1;\ rr := 1;\ br := (br+1) \mod 2; \\
Rec_1: & (rr = 1) & \longrightarrow & rr := 0;\ cr := br;
\end{array}
$$

The receiver reads the send channel cs when it is non-empty (cf. Action Rec_0). Then, the receiver toggles the value of br where it becomes ready to send an acknowledgment to the receiver (in Action Rec_1).

Read/Write restrictions. The sender can read/write rs, cs, bs, and cr, but it is not allowed to read rr and br. The receiver is allowed to read/write rr, cs, br, and cr. The receiver is not allowed to read rs and bs.

Faults. Faults can remove a data bit from either one of the communication channels causing the loss of that data bit. Hence, we model faults by setting the value of cs (respectively, cr) to -1.

$$
\begin{array}{llll}
F_0: & (cs \neq -1) & \longrightarrow & cs := -1; \\
F_1: & (cr \neq -1) & \longrightarrow & cr := -1;
\end{array}
$$

We assume that the fault actions will be executed a finite number of times; i.e., eventually faults stop occurring.

Safety specification. The problem specification requires that the receiver receives no duplicate packets.

Invariant. The state of the ABP program should satisfy the following conditions: (i) If the receiver is ready to send an acknowledgement message or it has already sent an acknowledge then the receive bit br and the send bit bs must be equal; (ii) If the sender is ready to send a new packet or it has already sent a new packet then the bs and br must not be equal; (iii) It is always the case that either the send channel cs is empty or it contains the sent bit bs; (iv) If both channels are empty then only one of the processes (i.e., the sender or the receiver) should be waiting; (v) If one of the channels is empty and the other one contains some data then both processes are waiting. Hence, we specify the invariant of the ABP program, S_{ABP}, as follows:

$$\begin{aligned}
S_{ABP} = \{s \mid & (((rr(s) = 1) \lor (cr(s) \neq -1)) \Rightarrow (br(s) = bs(s))) \land \\
& (((rs(s) = 1) \lor (cs(s) \neq -1)) \Rightarrow (br(s) \neq bs(s))) \ \land \ ((cs(s) = -1) \lor (cs(s) = bs(s))) \land \\
& (((cs(s) = -1) \land (cr(s) = -1)) \Rightarrow ((rr(s) + rs(s)) = 1)) \land \\
& (((cs(s) \neq -1) \land (cr(s) = -1)) \Rightarrow ((rr(s) + rs(s)) = 0)) \land \\
& (((cs(s) = -1) \land (cr(s) \neq -1)) \Rightarrow ((rr(s) + rs(s)) = 0)) \ \}
\end{aligned}$$

Fault-span. The state of the ABP program may be perturbed to the state predicate T_{ABP} due to fault transitions, where

$$\begin{aligned}
T_{ABP} = \{s \mid & ((cs(s) = -1) \lor (cs(s) = bs(s))) \land \\
& (((cs(s) = -1) \lor (cr(s) = -1)) \Rightarrow (((rr(s) + rs(s)) = 1) \lor ((rr(s) + rs(s)) = 0)))\}
\end{aligned}$$

The state predicate T_{ABP} includes states where (i) the send channel is empty or it is equal to the sent bit bs, and (ii) if at least one of the channels is empty then at least one of the processes is waiting.

Adding the actions of the high atomicity pseudo process. Faults may perturb the program in the states where sender has sent a new packet and the receiver is waiting for its arrival. As a result, the sent message is lost in the sender channel (i.e., cs becomes -1) and the receiver is waiting for a lost message. Likewise, the acknowledgement sent by the receiver might be lost in cr. Thus, the program may reach states where both channels are empty and both processes are waiting. For example, when the sent message is lost, the receiver is waiting for the lost message and the sender is waiting for its acknowledgement. In such states the program takes no action; i.e., deadlock state. Since the processes are not allowed to read the global state of the program, they cannot detect such global deadlock states. Using our synthesis method, we use high atomicity processes to identify the following high atomicity actions that are added to the program for recovery.

$$
\begin{array}{llll}
HAC_0: & (rs=0)\wedge(rr=0)\wedge(bs=1)\wedge(br=0)\wedge(cs=-1)\wedge(cr=-1) & \longrightarrow & cs:=1; \\
HAC_1: & (rs=0)\wedge(rr=0)\wedge(bs=0)\wedge(br=1)\wedge(cs=-1)\wedge(cr=-1) & \longrightarrow & cs:=0; \\
HAC_2: & (rs=0)\wedge(rr=0)\wedge(bs=1)\wedge(br=1)\wedge(cs=-1)\wedge(cr=-1) & \longrightarrow & cr:=1; \\
HAC_3: & (rs=0)\wedge(rr=0)\wedge(bs=0)\wedge(br=0)\wedge(cs=-1)\wedge(cr=-1) & \longrightarrow & cr:=0;
\end{array}
$$

The guards of the above actions are global state predicates that we refine using linear distributed detectors. Let G_i be the guard of the action HAC_i, where $0 \leq i \leq 3$. For example, we have $G_0 \equiv ((rs=0)\wedge(rr=0)\wedge(bs=1)\wedge(br=0)\wedge(cs=-1)\wedge(cr=-1))$. Corresponding to each global state predicate G_i, we use a distributed detector with two elements ds_i and dr_i, where ds_i is the local detector installed in the sender side and dr_i is the local detector installed in the receiver side. Next, we show how we add a linear distributed detector for the detection of G_0. We omit the presentation of the refinement of G_1, G_2, and G_3 as it is similar to the refinement of G_0.

Adding fault-tolerance components. Due to read restriction the sender (respectively, the receiver) cannot atomically detect G_0. However, the sender can detect a local condition $LC_s \equiv ((rs=0)\wedge(bs=1)\wedge(cs=-1))$. Respectively, the receiver can detect a local condition $LC'_r \equiv ((rr=0)\wedge(br=0)\wedge(cr=-1))$, where $G_0 \equiv (LC_s \wedge LC'_r)$. Now, we instantiate the required distributed detector by reusing the code of the pre-synthesized linear detectors presented in Section 6.3.

$$
\begin{array}{llll}
DAr_0: & (LC'_r)\ \wedge\ (y'_r = false) & \longrightarrow & y'_r := true; \\
DAs_0: & (LC_s)\ \wedge\ (y_s = false)\ \wedge\ (y'_r = true) & \longrightarrow & y_s := true;
\end{array}
$$

The action DAs_0 belongs to detector ds_0 that is allowed to read the witness predicate y'_r of the detector element dr_0 in the receiver side. If the detector element

dr_0 detects its local predicate LC'_r then it will set its witness predicate y'_r to true. Then, if the condition LC_s holds in the sender side then the detector element ds_0 will detect the global state predicate G_0 by setting its witness predicate y_s to true. Afterwards, the synthesis algorithm adds the following write action to the sender process.

$$Cs_0 : \ (y_s = true) \qquad \longrightarrow \qquad cs := 1; \ \ y_s := false;$$

The synthesis algorithm adds similar distributed detectors to ABP in order to refine the global state predicates G_1, G_2, and G_3. Given the local conditions $LC'_s \equiv ((rs = 0) \wedge (bs = 0) \wedge (cs = -1))$ and $LC_r \equiv ((rr = 0) \wedge (br = 1) \wedge (cr = -1))$, we have the following logical equivalences:

- $G_1 \equiv (LC'_s \wedge LC_r)$
- $G_2 \equiv (LC_s \wedge LC_r)$
- $G_3 \equiv (LC'_s \wedge LC'_r)$.

Corresponding to global detection predicates $G_1 \cdots G_3$, we respectively add the following linear distributed detectors and also the necessary correcting action for recovery to the invariant. Note that each added component has its own variables for representing the witness predicates.

Detecting G_1. This linear detector refines the guard of the action HAC_1 added by our synthesis algorithm.

$$DAr_1 : \ (LC_r) \ \wedge \ (y_r = false) \qquad \longrightarrow \qquad y_r := true;$$
$$DAs_1 : \ (LC'_s) \ \wedge \ (y'_s = false) \ \wedge \ (y_r = true) \qquad \longrightarrow \qquad y'_s := true;$$

Correcting G_1. After the detection of G_1, the following write action takes place.

$$Cs_1 : \ (y'_s = true) \qquad \longrightarrow \qquad cs := 0; \ \ y'_s := false;$$

Detecting G_2. We use the following linear detector to refine the guard of the action HAC_2.

$$DAr_2 : \ (LC_r) \ \wedge \ (u_r = false) \ \wedge \ (u_s = true) \qquad \longrightarrow \qquad u_r := true;$$
$$DAs_2 : \ (LC_s) \ \wedge \ (u_s = false) \qquad \longrightarrow \qquad u_s := true;$$

Correcting G_2. The following action, composed with the receiver, recovers the state of the ABP program to the invariant S_{ABP} after the detection of the global state predicate G_2.

$$Cr_2 : \ (u_r = true) \qquad \longrightarrow \qquad cr := 1; \ u_r := false;$$

Detecting G_3. To detect the global state predicate G_3 (i.e., the guard of the high atomicity action HAC_3), we add the following detector to ABP.

$$DAr_3 : \ (LC'_r) \ \wedge \ (u'_r = false) \ \wedge \ (u'_s = true) \qquad \longrightarrow \qquad u'_r := true;$$
$$DAs_3 : \ (LC'_s) \ \wedge \ (u'_s = false) \qquad \longrightarrow \qquad u'_s := true;$$

Correcting G_3. This action changes the state of the ABP program to a state in S_{ABP} after the detection of G_3.

$$Cr_3 : \ (u'_r = true) \qquad \longrightarrow \qquad cr := 0; \ u'_r := false;$$

The fault-tolerant ABP program. Next, we present the actions of the sender process in the resulting nonmasking fault-tolerant program.

$$\begin{array}{lll}
Send'_0 : & (rs = 1) & \longrightarrow \ rs := 0; cs := bs; cs := bs; \\
 & & \ u'_s := false; u_s := false; \\
Send'_1 : & (cr \neq -1) & \longrightarrow \ rs := 1; cr := -1; bs := (bs+1) \mod 2; \\
 & & \ u'_s := false; u_s := false; \\
DAs_0 : & (LC_s) \ \wedge \ (y_s = false) \ \wedge \ (y'_r = true) & \\
 & & \longrightarrow \ y_s := true; \\
Cs_0 : & (y_s = true) & \longrightarrow \ cs := 1; \ y_s := false;
\end{array}$$

$DAs_1: \quad (LC'_s) \wedge (y'_s = false) \wedge (y_r = true)$

$\qquad\qquad\qquad\qquad\qquad \longrightarrow \quad y'_s := true;$

$Cs_1: \quad (y'_s = true) \quad \longrightarrow \quad cs := 0;\ y'_s := false;$

$DAs_2: \quad (LC_s) \wedge (u_s = false) \longrightarrow \quad u_s := true;$

$DAs_3: \quad (LC'_s) \wedge (u'_s = false) \longrightarrow \quad u'_s := true;$

The synthesis algorithms has added new assignments to the actions $Send'_0$ and $Send'_1$ for the falsification of the witness predicates. For example, in action $Send'_0$, when cs is assigned a value other than -1, the predicates LC_s and LC'_s no longer hold. Thus, the witness predicates u'_s and u_s must be falsified. The actions of the receiver in the synthesized fault-tolerant program are as follows:

$Rec_0: \quad (cs \neq -1) \quad \longrightarrow \quad cs := -1; rr := 1; br := (br+1) \mod 2;$

$\qquad\qquad\qquad\qquad\qquad y_r := false; y'_r := false;$

$Rec_1: \quad (rr = 1) \quad \longrightarrow \quad rr := 0; cr := br;$

$\qquad\qquad\qquad\qquad\qquad y_r := false; y'_r := false;$

$DAr_0: \quad (LC'_r) \wedge (y'_r = false) \quad \longrightarrow \quad y'_r := true;$

$DAr_1: \quad (LC_r) \wedge (y_r = false) \quad \longrightarrow \quad y_r := true;$

$DAr_2: \quad (LC_r) \wedge (u_r = false) \wedge$

$\qquad\qquad (u_s = true) \quad \longrightarrow \quad u_r := true;$

$Cr_2: \quad (u_r = true) \quad \longrightarrow \quad cr := 1;\ u_r := false;$

$DAr_3: \quad (LC'_r) \wedge (u'_r = false) \wedge$

$\qquad\qquad (u'_s = true) \quad \longrightarrow \quad u'_r := true;$

$Cr_3: \quad (u'_r = true) \quad \longrightarrow \quad cr := 0;\ u'_r := false;$

Observe that in actions Rec_0 (respectively, Rec_1), we falsify the witness predicate y_r and y'_r once the program changes the value of rr to 1 (respectively, cr to 0 or 1). This falsification is necessary since once the condition $(rr = 1)$ holds, the predicates LC_r and LC'_r no longer hold. Also, this example illustrates the case where we simultaneously add multiple pre-synthesized components to a distributed program to add fault-tolerance. We have verified the interference-freedom requirements using the SPIN model checker [36] to gain more confidence with the implementation of our synthesis framework, FTSyn (see Appendix A for the Promela [37] code of this example).

6.6 Adding Hierarchical Components

In this section, we show how we add components with *hierarchical* topology to a diffusing computation program to provide recovery in the presence of faults. In earlier sections, we showed how we apply the synthesis algorithm presented in this chapter to programs where the underlying communication topology between processes is linear. In this section, we show how we add *hierarchical* pre-synthesized components to distributed programs. Specifically, we add tree-like structured components to a diffusing computation program where processes are arranged in an out-tree, where the indegree of each node is at most one. A diffusing computation starts at the root and propagates throughout the tree, and then, reflects back up to the root of the tree. The fault-intolerant program is subject to faults that perturb the state of the diffusing computation and the topology of the program (i.e., the parenting relationship amongst processes).

This case study shows that the synthesis method presented in this chapter handles pre-synthesized components (respectively, distributed programs) with different topologies as we have already reused a particular linear component in the synthesis of a token ring program and an alternating bit protocol in this chapter. Next, in Subsection 6.6.1, we describe how we formally represent a hierarchical fault-tolerance component. Subsequently in Subsection 6.6.2, we show how we automatically add a hierarchical component to a diffusing computation program.

6.6.1 Specifying Hierarchical Components

In this section, we describe the representation of hierarchical fault-tolerance components (i.e., detectors and correctors). We focus on the representation of a detector with a tree-like structure as a special case of hierarchical detectors. The hierarchical detector d consists of n elements d_i $(0 \leq i < n)$, its specification $spec_d$ (specified in Subsection 6.3.1), its variables, and its invariant U. We introduce a relation $\preceq$ on the elements d_i that represents the parenting relation between the nodes of the tree; e.g., $i \preceq j$ means d_i is the parent of d_j.

The element d_0 is placed at the root of the tree and other elements of the detector are placed in other nodes of the tree. Each node d_i has its own detection predicate X_i and witness predicate Z_i. The siblings of a node can detect their detection predicate in parallel. However, the truth-value of the detection predicate of each node depends on the truth-value of its children. In other words, node d_i can witness if all its children have already witnessed.

Each element d_i, $0 \leq i < n$, of the detector has a Boolean variable y_i that represents its witness predicate; i.e., the witness predicate of each d_i, say Z_i, is equal to $(y_i = true)$. Also, the element d_i can read/write the y values of its children and its parent $(0 \leq i < n)$. Moreover, each element d_i is allowed to read the variables that P_i can read, where P_i is the process with which d_i is composed. Now, we present the *template* action of the detector d_i as follows $((0 \leq i, j, k < n) \wedge \ (j < k) \wedge (\forall r : j \leq r \leq k : i \preceq r))$:

$$DA_i : \ (LC_i) \ \wedge \ (y_j \wedge \cdots \wedge y_k) \ \wedge \ (y_i = false) \quad \longrightarrow \quad y_i := true;$$

Using action DA_i $(0 \leq i < n)$, each element d_i of the hierarchical detector witnesses (i.e., sets the value of y_i to *true*) whenever (i) the condition LC_i becomes *true*, where LC_i represents a local condition that d_i atomically checks (by reading the variables of P_i), and (ii) its children $d_j, \cdots, d_k$ have already witnessed

$((0 \leq j, k < n) \wedge (j < k))$. The detection predicate X_i for element d_i is equal to $(LC_i \wedge LC_j \wedge \cdots \wedge LC_k)$. Therefore, d_0 detects the global detection predicate $LC_0 \wedge \cdots \wedge LC_{n-1}$.

The above action is an abstract template that should be instantiated by the synthesis algorithm during the synthesis of a specific program in such a way that the program and the detector do not interfere. For automatic addition of nonmasking fault-tolerance, the interference-freedom of the program and the detector requires that (i) in the absence of faults, the program specification and the safety specification of detectors are satisfied, and (ii) in the presence of faults, recovery is provided by the composition of the program and the detectors.

During the detection, when d_i sets y_i to true, its children have already set their y values to true. Hence, we represent the invariant of the hierarchical detector by the predicate U, where

$$U = \{s : (\forall i : (0 \leq i < n) : (y_i(s) \Rightarrow (\forall j : i \preceq j : LC_j))\}$$

6.6.2 Diffusing Computation

In this section, we present the addition of a hierarchical pre-synthesized component to a fault-intolerant diffusing computation. We have adapted the diffusing computation program from [38]. First, in Subsection 6.6.2.1, we give the specification of the diffusing computation program. Then, in Subsection 6.6.2.2, we present the synthesized nonmasking fault-tolerant program before the addition of the hierarchical component, which includes high atomicity recovery actions. Finally, in Subsection 6.6.2.3, we show how we add pre-synthesized components to refine the high atomicity actions added during synthesis.

6.6.2.1 Diffusing Computation Program

The diffusing computation (DC) program consists of four processes $\{P_0, P_1, P_2, P_3\}$ whose underlying communication is based on a tree topology. The process P_0 is the root of the tree. Processes P_1 and P_2 are the children of P_0 (i.e., $(0 \preceq 1) \wedge (0 \preceq 2)$) and P_3 is the child of P_2 (i.e., $2 \preceq 3$).

Starting from a state where every process is green, P_0 initiates a diffusing computation throughout the tree by propagating the red color towards the leaves. The leaves reflect the diffusing computation back to the root by coloring the nodes green. Afterwards, when all processes become green again, the cycle of diffusing computation repeats.

Each process P_j $(0 \leq j \leq 3)$ has a variable c_j that represents its color and whose domain is $\{0, 1\}$, where 0 represents the red and 1 represents the green. Also, process P_j has a Boolean variable sn_j that represents the session number of the diffusing computation where P_j is currently participating. Thus, we use sn_j to distinguish the case where P_j *has not started to participate in the current diffusing computation* from the case where P_j *has completed the current session of diffusing computation.* Moreover, each process has a variable par_j that represents the parent of P_j. The domain of par_j is equal to $\{0, 1, 2, 3\}$. The value of par_j identifies the node from where there exists an edge to P_j in the out-tree. For example, since the parent of P_0 is itself, we have $par_0 = 0$.

Program actions. The actions of the process P_j $(0 \leq j < 4)$ are as follows:

$$
\begin{array}{lll}
DC_{j1}: \ (c_j = 1) \ \wedge \ (par_j = j) & \longrightarrow & c_j := 0; \ sn_j = \neg sn_j; \\
DC_{j2}: \ (c_j = 1) \ \wedge \ (c_{par_j} = 0) \ \wedge \ (sn_j \not\equiv sn_{par_j}) & \longrightarrow & c_j := c_{par_j}; \ sn_j = sn_{par_j}; \\
DC_{j3}: \ (c_j = 0) \ \wedge \ (\forall k :: (par_k = j) \Rightarrow (c_k = 1 \wedge sn_j \equiv sn_k)) & \longrightarrow & c_j := 1;
\end{array}
$$

Read/write restrictions. Each process P_j is allowed to read/write the variables of its children and its parent. For example, process P_0 can read/write its local variables and the local variables of P_1 and P_2. However, P_0 is not allowed to read/write the variables of P_3. Also, P_3 cannot read/write the variables of P_0 and P_1.

Invariant. In each session of diffusing computation, every process P_j meets one of the following requirements: (i) P_j and P_{par_j} have both started participating in the current session of diffusing computation; (ii) P_j and P_{par_j} have both completed the current session of diffusing computation; (iii) P_j has not started participating in the current session whereas P_{par_j} has, and (iv) P_j has completed participating in the current session whereas P_{par_j} has not. Hence, the invariant of the program contains all state where S_{DC} holds, where

$$S_{DC} = (\forall j : (0 \leq j \leq 3) : ((c_j = c_{par_j} \;\wedge\; sn_j \equiv sn_{par_j}) \vee (c_j = 1 \;\wedge\; c_{par_j} = 0))) \wedge$$
$$(par_0 = 0 \wedge par_1 = 0 \wedge par_2 = 0 \wedge par_3 = 2)$$

Faults. Fault transitions can perturb the values of c_j and sn_j $(0 \leq j \leq 3)$, and the underlying communication topology of the program. We represent the fault transitions by the following actions:

$$F_{j_1}: \quad (true) \quad \longrightarrow \quad c_j = 0|1;$$
$$F_{j_2}: \quad (true) \quad \longrightarrow \quad sn_j = false|true;$$
$$F_0: \quad (true) \quad \longrightarrow \quad par_0 = 0|1|2;$$

The actions F_{j_1} and F_{j_2} represent the fault transitions that perturb a process P_j whereas action F_0 only affects P_0. The class of faults F_0 perturbs the parenting relationship by changing the value of par_0 to one of the values $\{0, 1, 2\}$. We have included fault-class F_0 since it perturbs the DC program to states where we can demonstrate the advantages of using pre-synthesized components in dealing with deadlock states.

6.6.2.2 Intermediate Nonmasking Program

Now, we present the intermediate nonmasking fault-tolerant program that includes high atomicity recovery actions. We have synthesized this intermediate program using our software framework FTSyn (cf. Chapter 8).

The faults may perturb the state of the DC program outside S_{DC} where the program may fall in a non-progress cycle or reach a deadlock state. For example, faults F_0 may perturb the program to states where the condition $T_{deadlock} \equiv ((c_0 = 1) \wedge (c_1 = 1) \wedge (c_2 = 1) \wedge (c_3 = 1)) \wedge (par_0 \neq 0)$ holds. The state predicate $T_{deadlock}$

represents states from where no program action is enabled; i.e., deadlock states. Now, to add recovery from a state in $T_{deadlock}$, FTSyn assigns a high atomicity process P_{high_j} to each process P_j $(0 \leq j < 4)$.

To illustrate our approach of adding hierarchical pre-synthesized detectors (respectively, correctors), we only focus on one of the high atomicity recovery actions added by process P_{high_0} as the refinement of other high atomicity actions is similar. The actions of other high atomicity processes in the intermediate nonmasking program are available in the Appendix A. The action HAC is as follows:

$$\begin{aligned} HAC : (c_0 = 1) \wedge (c_1 = 1) \wedge (c_2 = 1) \wedge (c_3 = 1) \wedge (sn_0 = 1) \wedge \ ((par_0 = 2) \vee (par_0 = 1)) \wedge \\ ((sn_3 = 0) \vee (sn_1 = 0) \vee (sn_2 = 0)) \quad \longrightarrow \quad sn_0 := 0; \end{aligned}$$

The guard of HAC identifies a subset of $T_{deadlock}$ for which HAC provides recovery to states from where recovery to S_{DC} has been already established. The write action performed by HAC is a local write operation in process P_0, whereas the guard of HAC is a global state predicate that should be refined in the distributed program. Thus, we only need to add detectors for the refinement of the guard of HAC. In the next subsection, we show how FTSyn uses the guard of HAC to automatically specify the required detectors.

6.6.2.3 Adding Pre-synthesized Detectors

To refine the guard of HAC, the synthesis algorithm presented in this chater automatically identifies the interface of the required component. The component interface is a triple $\langle X, R, i \rangle$, where X is the detection predicate of the required component, R is a relation that represents the topology of the required component, and i is the index of the process that performs the local write action after the detection of X. For example, for action HAC, X is equal to the state predicate X_0 as we describe next in this section, R is a set of pairs where each pair represents the existence of a communication link between two processes, and i is equal to 0 since P_0 should perform the local write action.

Using the interface of the required pre-synthesized component, the synthesis algorithm queries an existing library of pre-synthesized components. At this step, we have the option of supervising the synthesis algorithm in that we can observe the guard of HAC and manually identify the required components. This manual intervention helps in minimizing the number of components added to the program since each component adds its associated variables to the program and expands the state space. For example, in the case of action HAC, the synthesis algorithm automatically identifies one component corresponding to each deadlock state in the set of states represented by the guard of HAC, whereas by manual intervention, we observe that the only variables that are not readable for P_0 are c_3 and sn_3. Hence, we add two distributed detectors d and d' to simultaneously detect the predicates X_0 and X_0', where

$$X_0 \equiv ((c_3 = 1) \wedge (c_0 = 1) \wedge (c_1 = 1) \wedge (c_2 = 1) \wedge (sn_0 = 1) \wedge ((par_0 = 2) \vee (par_0 = 1)))$$

$$X_0' \equiv ((sn_3 = 0) \wedge (c_0 = 1) \wedge (c_1 = 1) \wedge (c_2 = 1) \wedge (sn_0 = 1) \wedge ((par_0 = 2) \vee (par_0 = 1)))$$

The pre-synthesized detector d (respectively, d') includes four elements d_0, d_1, d_2, and d_3 (respectively, d_0', d_1', d_2', and d_3'), where d_i (respectively, d_i') is composed with P_i $(0 \leq i \leq 3)$. Thus, the topologies of the distributed detectors d and d' are similar to the topology of the DC program. Also, the parenting relationship (respectively, read/write restrictions) between d_0, d_1, d_2, d_3 (respectively, d_0', d_1', d_2', and d_3') follows the parenting relationship (respectively, read/write restrictions) of P_0, P_1, P_2, and P_3.

The synthesis algorithm automatically instantiates an instance of the template action presented in Section 6.6.1 with the appropriate local condition. The local conditions are automatically identified based on the set of readable variables of each process. For example, the part of X_0 that is readable for detector d_3 is identified as $LC_3 \equiv ((c_3 = 1) \wedge (c_2 = 1))$. Thus, the instantiation of the template action for detector d_3 results in the following action:

$$D_{31}:\ (c_3 = 1) \wedge (c_2 = 1) \wedge (y_3 = false) \quad \longrightarrow \quad y_3 := true;$$

Likewise, the part of X'_0 that is readable for detector d'_3 is automatically identified as $LC'_3 \equiv ((sn_3 = 0) \wedge (c_2 = 1))$. Hence, the action of d'_3 is as follows:

$$D'_{31}:\ (sn_3 = 0) \wedge (c_2 = 1) \wedge (y'_3 = false) \quad \longrightarrow \quad y'_3 := true;$$

The detector d_3 (respectively, d'_3) sets y_3 (respectively, y'_3) to *true* if the local condition LC_3 (respectively, LC'_3) holds and y_3 (respectively, y'_3) is *false*. The predicate $Z_3 \equiv (y_3 = true)$ (respectively, $Z'_3 \equiv (y'_3 = true)$) is the witness predicate of d_3 (respectively, d'_3), and the predicate $X_3 \equiv LC_3$ (respectively, $X'_3 \equiv LC'_3$) constructs the detection predicate of d_3 (respectively, d'_3). Note that since d_3 (respectively, d'_3) is the leaf of the tree, it does not have any children to wait for before it witnesses. Next, we present the actions of d_2 and d'_2 (i.e., actions D_{21} and D'_{21}) as follows:

$$D_{21}:\ (y_3 = true) \wedge (c_2 = 1) \wedge (sn_0 = 1) \wedge (c_0 = 1) \wedge ((par_0 = 2) \vee (par_0 = 1)) \wedge (y_2 = false) \longrightarrow y_2 := true;$$

$$D'_{21}:\ (y'_3 = true) \wedge (c_2 = 1) \wedge (sn_0 = 1) \wedge (c_0 = 1) \wedge ((par_0 = 2) \vee (par_0 = 1)) \wedge (y'_2 = false) \longrightarrow y'_2 := true;$$

The local condition of the action D_{21} (i.e., LC_2) is equal to $(c_2 = 1) \wedge (sn_0 = 1) \wedge (c_0 = 1) \wedge ((par_0 = 2) \vee (par_0 = 1))$. Thus, the detection predicate of d_2 is equal to $X_2 \equiv (LC_2 \wedge LC_3)$ and its witness predicate Z_2 is equal to $(y_2 = true)$. The local condition of the action D'_{21} (i.e., LC'_2) is also equal to LC_2. Hence, the detection predicate of d'_2 is equal to $X'_2 \equiv (LC'_2 \wedge LC'_3)$ and its witness predicate Z'_2 is equal to $(y'_2 = true)$.

Likewise, the synthesis algorithm identifies the detection (respectively, the witness) predicate of d_1 based on identifying $LC_1 \equiv (c_1 = 1)$. We omit the details of the actions of d_1 as it is straightforward and similar to the actions of d_2 and d_3. The local condition LC_0 of detector d_0 is equal to $(c_1 = 1) \wedge LC_2$. Also, the local condition LC'_0 of detector d'_0 is equal to $(c_1 = 1) \wedge LC'_2$. Thus, the actions of detectors d_0 and d'_0 are as follows:

$$\begin{array}{lll} D_{01}: & (y_1 = true) \wedge (y_2 = true) \wedge (c_1 = 1) \wedge LC_2 \wedge (y_0 = false) & \longrightarrow \; y_0 := true; \\ D'_{01}: & (y'_1 = true) \wedge (y'_2 = true) \wedge (c_1 = 1) \wedge LC'_2 \wedge (y'_0 = false) & \longrightarrow \; y'_0 := true; \end{array}$$

The truth-value of y_0 witnesses the truth-value of X_0 and y'_0 witnesses the truth-value of X'_0. Now, we add a recovery action that only reads the local variables of P_0 and d_0 and writes the local variables of P_0. The recovery action is as follows:

$$\begin{array}{lll} Rec: & (y_0 = true) \wedge ((y'_0 = true) \vee (sn_1 = 0) \vee (sn_2 = 0)) & \longrightarrow sn_0 := 0; y_0 := false; y'_0 := false; \\ & & \quad y_2 := false; y'_2 := false; \end{array}$$

When the program executes the above recovery action, the predicates X_0 and X_2 (respectively, X'_0 and X'_2) no longer hold. Thus, the witness predicates of d_0 and d_2 (respectively, d'_0 and d'_2) must be falsified; i.e., y_0 and y_2 (respectively, y'_0 and y'_2) should become $false$.

The composition of the DC program and the pre-synthesized detectors. Now, we present the actions of the process P_0 of the nonmasking DC program that is a composition of the actions of the pre-synthesized detectors and the actions of the processes in the intermediate fault-intolerant program. Since the actions of P_1 and P_2 are structurally similar to P_0's actions, we refer the interested reader to the Appendix A for the actions of P_1 and P_2. Note that since no detection is done by d_1, the synthesized program does not have any new actions in process P_1. Thus, the actions of P_1 remain similar to the fault-intolerant program. The actions of process P_0 composed with the actions of d_0, d'_0, and the recovery action Rec are as follows:

$$\begin{array}{lll} DC_{01}: & (c_0 = 1) \; \wedge \; (par_0 = 0) & \longrightarrow \; c_0 := 0; \\ & & \quad\; y_0 := false; y'_0 := false; \\ DC_{02}: & (c_0 = 1) \; \wedge \; (c_{par_0} = 0) \; \wedge \; (sn_0 \not\equiv sn_{par_0}) & \end{array}$$

```
                                   $\longrightarrow c_0 := c_{par_0};\ sn_0 = sn_{par_0};$
                                       if $((c_0 = 0) \wedge (y_0 = true))$
                                       then $y_0 := false; y'_0 := false;$
$DC_{03}:\ (c_0 = 0)\ \wedge\ (\forall k :: (par_k = 0) \Rightarrow (c_k = 1 \wedge sn_0 \equiv sn_k))$
                                   $\longrightarrow c_0 := 1;$
                                       if $(((y_1 = false) \vee (y_2 = false)) \wedge (y_0 = true))$
                                       then $y_0 := false;$
                                       if $(((y'_1 = false) \vee (y'_2 = false)) \wedge (y'_0 = true))$
                                       then $y'_0 := false;$


$D_{01}:\ (y_1 = true) \wedge (y_2 = true) \wedge LC_0 \wedge (y_0 = false)$
                                   $\longrightarrow y_0 := true;$
$D'_{01}:\ (y'_1 = true) \wedge (y'_2 = true) \wedge LC'_0 \wedge (y'_0 = false)$
                                   $\longrightarrow y'_0 := true;$
$Rec:\ (y_0 = true) \wedge ((y'_0 = true) \vee (sn_1 = 0) \vee (sn_2 = 0))$
                                   $\longrightarrow sn_0 := 0; y_0 := false; y'_0 := false;$
                                      $y_2 := false; y'_2 := false;$
```

The actions of process P_0 are composed with the actions of detectors d_0 and d'_0 (i.e., D_{01} and D'_{01}) and the recovery action *Rec* presented in this section. Observe that the statement of actions DC_{01} and DC_{02} of P_0 are composed with assignments that falsify the witness predicates of the corresponding detectors. Such falsification of the witness predicates is necessary so that program execution preserves the safety of detectors. For example, when c_0 becomes 0 the state predicate LC_0 no longer holds. Thus, the witness predicate y_0 must be falsified to ensure the interference-freedom of the program and the pres-synthesized detectors.

Interference-freedom. The interference-freedom requires the synthesized program to provide recovery in the presence of faults, and satisfy the specification of the DC program in the absence of faults. In the presence of faults, if faults perturb the program outside the invariant S_{DC} then the synthesized program satisfies the requirements of nonmasking fault-tolerance; i.e., recovery to S_{DC} is guaranteed. In the absence of faults, the added detectors do not interfere with the program execution. Thus, in the absence of faults, the above program satisfies the specification of diffusing computation program and the safety of detectors.

We would like to note that when faults occur, fault transitions may directly violate the safety specification of detectors; e.g., after d_3 witnesses that $(c_3 = 1)$ holds, faults may change the value of c_3 to 0, and as a result, d_3 witnesses incorrectly; i.e., the safety of d_3 will be violated by fault transitions. Since nonmasking fault-tolerance only requires recovery to the invariant, the violation of safety does not violate the nonmasking fault-tolerance property. Thus, the only requirement is that the composition of the program and the pre-synthesized detectors provides recovery in the presence of faults.

Although the synthesized nonmasking program is correct by construction, we verified the interference-freedom requirements of the above program in the SPIN model checker to gain more confidence on the implementation of the framework FTSyn presented in Chapter 8. We refer the reader to the Appendix A for the source of the Promela model.

6.7 Discussion

In this section, we address some of the questions raised by our synthesis method. Specifically, we discuss the following issues: the fault-tolerance of the components, the choice of detectors and correctors, and pre-synthesized components with nonlinear topologies.

Can the synthesis method deal with the faults that affect the fault-tolerance components?

Yes. The added component may itself be perturbed by the fault to which fault-tolerance is added. Hence, the added component must itself be fault-tolerant. For example, in our token ring program, we modeled the effect of the process restart on the added component and ensured that the component is fault-tolerant to that fault (cf. Theorem 6.1). For the fault-classes that are commonly used, e.g., process failure, process restart, input corruption, Byzantine faults, such modeling is always possible. For arbitrary fault-classes, however, some *validation* may be required to ensure that the modeling is appropriate for that fault.

How does the choice of detectors and correctors help in the synthesis of fault-tolerant programs?

While there are several approaches (e.g., [39]) that manually transform a fault-intolerant program into a fault-tolerant program, we use detectors and correctors in this chapter, based on their necessity and sufficiency for manual addition of fault-tolerance [18]. The authors of [18] have also shown that detectors and correctors are abstract enough to generalize other components (e.g., comparators and voters used in replication-based approaches) for the design of fault-tolerant programs. Hence, we expect that our synthesis method can benefit from the generality of detectors and correctors in the *automated* synthesis of fault-tolerant programs as there is a potential to provide a rich library of fault-tolerance components. Moreover, pre-synthesized detectors provide the kind of abstraction by which we can integrate efficient existing detections approaches (e.g., [40, 41]) in pre-synthesized fault-tolerance components.

Does the synthesis method support pre-synthesized components with non-linear topologies?

Yes. As we demonstrated in Sections 6.5 and 6.6.2, we have applied the synthesis method of this chapter to add pre-synthesized fault-tolerance components with linear and hierarchical topologies. These examples show the applicability of our synthesis method for distributed programs (respectively, distributed fault-tolerance components) with linear and hierarchical topologies.

In the token ring example, will the synthesis succeed if we select PS_{index} ($1 \leq index \leq$

3), instead of PS_0, as the pseudo process that adds a high atomicity recovery transition from the deadlock state $s_d = \langle\bot, \bot, \bot, \bot\rangle$?

Yes. We argue that if we select a detector d with the following arrangement, $d_{index-1}, \cdots, d_0, d_3, \cdots, d_{index}$, where $index \neq 0$, then the synthesis will succeed and the detector d will not interfere with the token ring program. In this arrangement, the element $d_{index-1}$ is allowed to read and write $y_{index-1}$. Every element d_j, $0 \leq j < index - 1$, is allowed to read y_j and y_{j+1}, and write y_j. d_3 is allowed to read d_0 and d_3, and write d_3. Elements d_k, $index \leq k < 3$, are allowed to read d_k and d_{k+1}, and write d_k.

Using the above arrangement, Z_{index} witnesses the detection predicate $X \equiv ((x_0 = \bot) \wedge (x_1 = \bot) \wedge (x_2 = \bot) \wedge (x_3 = \bot))$, and afterwards, the PS_{index} adds a high atomicity recovery action to the program. The proof of non-interference is similar to the case where PS_0 is selected as the pseudo process that adds the high atomicity action.

In the token ring example, will the synthesis succeed if we add a sequential detector with a different linear order $d_0 \cdots d_3$, where Z_3 witnesses for the detection predicate $X \equiv ((x_0 = \bot) \wedge (x_1 = \bot) \wedge (x_2 = \bot) \wedge (x_3 = \bot))$?

No. We show that if we use the above order then the Interfere algorithm returns true as I_1 becomes non-empty; i.e., the execution of the token ring program interferes with the added pre-synthesized component. In a state $s = \langle\bot, \bot, 0, 0\rangle$, the elements d_0 and d_1 of the linear detector witness their detection predicates X_0 and X_1, where $X_0 \equiv (x_0 = \bot)$ and $X_1 \equiv ((x_0 = \bot) \wedge (x_1 = \bot))$. Now, if P_0 executes and sets x_0 to 1 then X_1 no longer holds. As a result, the program reaches a state where d_1 incorrectly witnesses its detection predicate and violates the specification of the linear detector.

6.8 Summary

In this chapter, we presented an approach for the synthesis of a fault-tolerant program from its fault-intolerant version and pre-synthesized fault-tolerance components. Specifically, we presented an algorithm for automatic specification of the required fault-tolerance components during the synthesis. We also presented a sound algorithm for automatic addition of pre-synthesized fault-tolerance components to a distributed program. Before adding a component, we verified the interference-freedom of the composition of the program and the fault-tolerance component. Using our synthesis algorithm, we showed how we could add masking fault-tolerance to a token-ring program where all process might be corrupted. By contrast, previous work on automatic addition of fault-tolerance to the token ring program assumed that at least one process is not corrupted. Also, we demonstrated how we reuse the same component used in the synthesis of the token ring program for the synthesis of an alternating bit protocol that is nonmasking fault-tolerant to message loss faults. Moreover, we showed that our synthesis method is applicable for adding pre-synthesized components with different topologies (e.g., linear and hierarchical) where we added tree-like components to a diffusing computation program.

Chapter 7

Automated Synthesis of Multitolerance

In this chapter, we focus on automated synthesis of multitolerant programs. Such automated synthesis has the advantage of generating fault-tolerant programs that (i) tolerate multiple classes of faults, and (ii) are correct by construction. Automatic synthesis of multitolerance is desirable as (i) today's systems are often subject to multiple classes of faults, and (ii) it is often undesirable or impractical to provide the same level of fault-tolerance to each class of faults. Hence, these systems need to tolerate multiple classes of faults, and (possibly) provide a different level of fault-tolerance to each class. To characterize such systems, the notion of multitolerance was introduced in [34]. The importance of such multitolerant systems can be easily observed from the fact that several methods for designing multitolerant programs as well as several instances of multitolerant programs can be readily found (e.g., [11, 12, 13, 34]) in the literature.

We focus on automated synthesis of high atomicity multitolerant programs in a stepwise fashion. Specifically, we (i) present a sound and complete stepwise algorithm for the case where we add nonmasking fault-tolerance to one class of faults and masking fault-tolerance to another class of faults, and (ii) present a sound and complete stepwise algorithm for the case where we add failsafe fault-tolerance to one class of faults and masking fault-tolerance to another class of faults. The complexity of these algorithms is polynomial in the state space of the fault-intolerant program. For the case where failsafe fault-tolerance is added to one fault-class and nonmasking fault-tolerance is added to another fault-class, we find a somewhat surprising result. We find that this problem is NP-complete. This result is surprising in that automating the addition of failsafe and nonmasking fault-tolerance to the *same* class of faults can be performed in polynomial time. However, addition of failsafe fault-tolerance to one class of faults and nonmasking fault-tolerance to a *different* class of faults is NP-complete.

In the rest of this chapter, we proceed as follows: In Section 7.1, we present the formal definition of multitolerance and the problem of synthesizing a multitolerant program from a fault-intolerant program. Subsequently, in Section 7.2, we recall the relevant properties of algorithms in 2.7 that we use in automated addition of multitolerance. In Section 7.3, we present a sound and complete algorithm for the synthesis of multitolerant programs that provide nonmasking-masking multitolerance. Then, in Section 7.4, we present a sound and complete algorithm for the synthesis of multitolerant programs that provide failsafe-masking multitolerance. In Section 7.5, we present the NP-completeness proof for the case where failsafe-nonmasking multitolerance is added to fault-intolerant programs. Finally, in Section 7.6, we make concluding remarks and discuss future work.

7.1 Problem Statement

In this section, we formally define the problem of synthesizing multitolerant programs from their fault-intolerant versions. Before defining the synthesis problem, we present our definition of multitolerance; i.e., we identify what it means for a program to be multitolerant in the presence of multiple classes of faults.

As mentioned in Section 2.5, a failsafe/nonmasking/masking fault-tolerant program guarantees to provide a desired level of fault-tolerance (i.e., failsafe/nonmasking/masking) in the presence of a specific class of faults. Now, we consider the case where faults from multiple fault-classes, say $f1$ and $f2$, occur in a given program computation.

There exist several possible choices in deciding the level of fault-tolerance that should be provided in the presence of multiple fault-classes. One possibility is to provide no guarantees when $f1$ and $f2$ occur in the same computation. With such a definition of multitolerance, the program would provide fault-tolerance if faults from $f1$ occur or if faults form $f2$ occur. However, no guarantees will be provided if both faults occur simultaneously.

Another possibility is to require that the fault-tolerance provided for the case where $f1$ and $f2$ occur simultaneously should be equal to the minimum level of fault-tolerance provided when either $f1$ occurs or $f2$ occurs. For example, if masking fault-tolerance is provided to $f1$ and failsafe fault-tolerance is provided to $f2$ then failsafe fault-tolerance should be provided for the case where $f1$ and $f2$ occur simultaneously. However, if nonmasking fault-tolerance is provided to $f1$ and failsafe fault-tolerance is provided to $f2$ then no level of fault-tolerance will be guaranteed for the case where $f1$ and $f2$ occur simultaneously. We note that this assumption is not required in our proof of NP-completeness in Section 7.5.

In our definition, we follow the latter approach. The following table illustrates the minimum level of fault-tolerance provided for different combinations of levels of fault-tolerance provided to individual classes of faults.

Fault-Tolerance	Failsafe	Nonmasking	Masking
Failsafe	Failsafe	Intolerant	Failsafe
Nonmasking	Intolerant	Nonmasking	Nonmasking
Masking	Failsafe	Nonmasking	Masking

In a special case, consider the situation where failsafe fault-tolerance is provided to both $f1$ and $f2$. From the above description, failsafe fault-tolerance should be provided for the fault class $f1 \cup f2$. By taking the union of all the fault-classes for which failsafe fault-tolerance is provided, we get one fault-class, say $f_{failsafe}$, for which failsafe fault-tolerance needs to be added. Likewise, we obtain the fault-class $f_{nonmasking}$ (respectively, $f_{masking}$) for which nonmasking (respectively, masking) fault-tolerance is provided.

Now, given (the transitions of) a fault-intolerant program, p, its invariant, S, its specification, *spec*, and a set of distinct classes of faults $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$, we define what it means for a synthesized program p', with invariant S', to be multitolerant by considering how p' behaves when (i) no faults occur; (ii) only one class of faults happens, and (iii) multiple classes of faults happen.

Definition. Program p' is *multitolerant* to $f_{failsafe}, f_{nonmasking}$, and $f_{masking}$ from S' for *spec* iff (if and only if) the following conditions hold:

1. p' satisfies *spec* from S' in the absence of faults.
2. p' is masking $f_{masking}$-tolerant from S' for *spec*.
3. p' is failsafe $(f_{failsafe} \cup f_{masking})$-tolerant from S' for *spec*.
4. p' is nonmasking $(f_{nonmasking} \cup f_{masking})$-tolerant from S' for *spec*. □

Remark. Since every program is failsafe/nonmasking/masking fault-tolerant to a class of faults whose set of transitions is empty, the above definition generalizes the cases where one of the classes of faults is not specified (e.g., $f_{masking} = \{\}$).

Now, using the definition of multitolerant programs, we identify the requirements of the problem of synthesizing a multitolerant program, p', from its fault-intolerant version, p. The problem statement is motivated by the goal of simply adding multitolerance and introducing no new behaviors in the absence of faults. This problem statement is the natural extension to the problem statement in Section 2.6 where fault-tolerance is added to a single class of faults.

Since we require p' to behave similar to p in the absence of faults, we stipulate the following conditions: First, we require S' to be a subset of S (i.e., $S' \subseteq S$). Otherwise, if there exists a state $s \in S'$ where $s \notin S$ then, *in the absence of faults*, p' can reach s and create new computations that do not belong to p. Thus, p' will include new ways of satisfying *spec* from s in the absence of faults. Second, we require $(p'|S') \subseteq (p|S')$. If $p'|S'$ includes a transition that does not belong to $p|S'$ then p' can include new ways for satisfying *spec in the absence of faults*. Thus, the problem of multitolerance synthesis is as follows:

The Multitolerance Synthesis Problem

Given p, S, *spec*, $f_{failsafe}, f_{nonmasking}$, and $f_{masking}$

Identify p' and S' such that

$S' \subseteq S$,

$p'|S' \;\subseteq\; p|S'$, and

p' is multitolerant to $f_{failsafe}, f_{nonmasking}$, and $f_{masking}$ from S' for *spec*. □

We state the corresponding decision problem as follows:

The Decision Problem

Given p, S, *spec*, $f_{failsafe}, f_{nonmasking}$, and $f_{masking}$:

Does there exist a program p', with its invariant S' that satisfies the requirements of the synthesis problem? □

7.2 Addition of Fault-Tolerance to One Fault-Class

In the synthesis of multitolerant programs, we reuse algorithms Add_Failsafe, Add_Nonmasking, and Add_Masking, presented by Kulkarni and Arora [1] (cf. Section

2.7). These algorithms respectively add failsafe/nonmasking/masking fault-tolerance to a *single* class of faults. Hence, we recall the relevant properties of these algorithms in this section.

The algorithms represented in Section 2.7 take a program p, its invariant S, its specification *spec*, a class of faults f, and synthesize an f-tolerant program p' (if any) with the invariant S'. The synthesized program p' and its invariant S' satisfy the following requirements: (i) $S' \subseteq S$; (ii) $p'|S' \subseteq p|S'$, and (iii) p' is failsafe (respectively, nonmasking or masking) f-tolerant from S' for *spec*.

The invariant S', calculated by Add_Failsafe (respectively, Add_Masking), has the property of being the largest such possible invariant for any failsafe (respectively, masking) program obtained by adding fault-tolerance to the given fault-intolerant program. In other words, if there exists a failsafe fault-tolerant program p'', with invariant S'' that satisfies the above requirements for adding fault-tolerance then $S'' \subseteq S'$. Also, if no sequence of fault transitions can violate the safety of specification from any state inside S then Add_Failsafe (cf. Section 2.7) will not change the invariant of the fault-intolerant program. Hence, we make the following observations:

Observation 7.1. Let the input for Add_Failsafe be p, S, *spec* and f. Let the output of Add_Failsafe be fault-tolerant program p' and invariant S'. If any program p'' with invariant S'' satisfies (i) $S'' \subseteq S$; (ii) $p''|S'' \subseteq p|S''$, and (iii) p'' is failsafe f-tolerant from S' for *spec* then $S'' \subseteq S'$. $\square$

Observation 7.2. Let the input for Add_Failsafe be p, S, *spec* and f. Let the output of Add_Failsafe be fault-tolerant program p' and invariant S'. Unless there exists states in S from where a sequence of f transitions alone violates safety, $S' = S$. $\square$

Likewise, the f-span of the masking f-tolerant program, say T', synthesized by the algorithm Add_Masking (cf. Section 2.7) is the largest possible f-span. Thus, we make the following observation:

Observation 7.3. Let the input for Add_Masking be p, S, *spec* and f. Let the output of Add_Masking be fault-tolerant program p', invariant S', and fault-span T'. If any program p'' with invariant S'' satisfies (i) $S'' \subseteq S$; (ii) $p''|S'' \subseteq p|S''$, (iii) p'' is masking f-tolerant from S' for *spec*, and (iv) T'' is the fault-span used for verifying the masking fault-tolerance of p'' then $S'' \subseteq S'$ and $T'' \subseteq T'$. $\square$

The algorithm Add_Nonmasking only adds recovery transitions from states outside the invariant S to S. Thus, we make the following observations:

Observation 7.4. Add_Nonmasking does not add or remove any state of S. $\square$

Observation 7.5. Add_Nonmasking does not add or remove any transition of $p|S$. $\square$

Based on the Observations 7.1- 7.5, Kulkarni and Arora [1] show that the algorithms Add_Failsafe, Add_Nonmasking, and Add_Masking are sound and complete, i.e., the output of these algorithms satisfy the requirements for adding fault-tolerance to a *single* class of faults and these algorithms can find a fault-tolerant program if one exists.

Theorem 7.6. The algorithms Add_Failsafe, Add_Nonmasking, and Add_Masking are sound and complete. $\square$

7.3 Nonmasking-Masking Multitolerance

In this section, we present an algorithm for stepwise synthesis of multitolerant programs that are subject to two classes of faults $f_{nonmasking}$ and $f_{masking}$ for which respectively nonmasking and masking fault-tolerance is required. We also show that our synthesis algorithm is sound and complete.

Given a program p, with its invariant S, its specification *spec*, our goal is to synthesize a program p', with invariant S' that is multitolerant to $f_{nonmasking}$ and $f_{masking}$. By definition, p' must be masking $f_{masking}$-tolerant. In the presence of both $f_{nonmasking}$ and $f_{masking}$ (i.e., $f_{nonmasking} \cup f_{masking}$), p' must provide nonmasking $f_{nonmasking} \cup f_{masking}$-tolerance.

We proceed as follows: Using the algorithm Add_Masking, we synthesize a masking

$f_{masking}$-tolerant program p_1, with invariant S', and fault-span $T_{masking}$. Now, since program p_1 is masking $f_{masking}$-tolerant, it provides safe recovery to its invariant, S', from every state in $(T_{masking}{-}S')$. Thus, in the presence of $f_{nonmasking} \cup f_{masking}$, if p_1 is perturbed to $(T_{masking}{-}S')$ then p_1 will satisfy the requirements of nonmasking fault-tolerance (i.e., recovery to S'). However, if $f_{nonmasking} \cup f_{masking}$ transitions perturb p_1 to states s, where $s \notin T_{masking}$, then recovery must be added from those states. Based on the Observations 7.4 and 7.5, it suffices to add recovery to $T_{masking}$ as provided recovery by p_1 from $T_{masking}$ to S' can be reused *even after* adding nonmasking fault-tolerance. Thus, the synthesis algorithm Add_Nonmasking_Masking is as shown in Figure 7.1.

```
Add_Nonmasking_Masking($p$: transitions, $f_{nonmasking}, f_{masking}$: fault,
                                   $S$: state predicate, $spec$: safety specification)
{
  $p_1, S', T_{masking}$ := $Add\_Masking(p, f_{masking}, S, spec)$;
  if $(S' = \{\})$    declare no multitolerant program $p'$ exists;
                 return $\emptyset, \emptyset$;
  $p', T'$ := $Add\_Nonmasking(p_1, f_{nonmasking} \cup f_{masking}, T_{masking}, spec)$;
  return $p', S'$;
}
```

Figure 7.1: Synthesizing nonmasking-masking multitolerance.

Now, in Theorem 7.7, we show the soundness of Add_Nonmasking_Masking, i.e., we show that the output of Add_Nonmasking_Masking satisfies the requirements of the problem statement in Section 7.1. Subsequently, in Theorem 7.8, we show the completeness of Add_Nonmasking_Masking, i.e., we show that if a multitolerant program can be designed for the given fault-intolerant program then Add_Nonmasking_Masking will not declare failure.

Theorem 7.7. The algorithm Add_Nonmasking_Masking is sound.

Proof. Based on the soundness of Add_Masking (cf. Theorem 7.6), $S' \subseteq S$.

Also, using the soundness of Add_Masking, we have $p_1|S' \subseteq p|S'$. In addition, based on the Observation 7.5, we have $p_1|S' = p'|S'$. As a result, we have $p'|S' \subseteq p|S'$.

Now, we show that p' is multitolerant to $f_{nonmasking}$ and $f_{masking}$ from S' for *spec*:

1. **Absence of faults.** From the soundness of Add_Masking, it follows that p_1 satisfies *spec* from S' in the absence of faults. Since Add_Nonmasking does not add (respectively, remove) any transitions to (respectively, from) $p_1|S'$ (cf. Observation 7.5), it follows that p' satisfies *spec* from S'.

2. **Masking $f_{masking}$-tolerance.** From the soundness of Add_Masking, p_1 is masking $f_{masking}$-tolerant from S' for *spec*. Also, based on the Observation 7.4 and 7.5, Add_Nonmasking preserves masking $f_{masking}$-tolerance property of p_1 since $p_1|T_{masking} = p'|T_{masking}$. Therefore, p' is masking $f_{masking}$-tolerant from S' for *spec*.

3. **Nonmasking ($f_{nonmasking} \cup f_{masking}$)-tolerance.** From the soundness of Add_Nonmasking, we know that p' is nonmasking ($f_{nonmasking} \cup f_{masking}$)-tolerant from $T_{masking}$ for *spec*. Also, based on the Observation 7.4 and 7.5, Add_Nonmasking preserves masking $f_{masking}$-tolerance property of p_1 since $p_1|T_{masking} = p'|T_{masking}$. Thus, recovery from $T_{masking}$ to S' is guaranteed in the presence of $f_{nonmasking} \cup f_{masking}$. Therefore, p' is nonmasking ($f_{nonmasking} \cup f_{masking}$)-tolerant from S' for *spec*.

Based on the above discussion, it follows that p' is multitolerant to $f_{nonmasking}$ and $f_{masking}$ from S' for *spec*. Therefore, Add_Nonmasking_Masking is sound. □

Theorem 7.8. The algorithm Add_Nonmasking_Masking is complete.

Proof. Add_Nonmasking_Masking declares that a multitolerant program does not exist only when Add_Masking does not find a masking $f_{masking}$-tolerant program. Since the synthesized program must be masking $f_{masking}$-tolerant, from the completeness of Add_Masking, completeness of Add_Nonmasking_Masking follows. □

7.4 Failsafe-Masking Multitolerance

In this section, we investigate the stepwise synthesis of programs that are multitolerant to two classes of faults $f_{failsafe}$ and $f_{masking}$ for which we respectively require failsafe and masking fault-tolerance. We present a sound and complete algorithm for synthesizing failsafe-masking multitolerant programs.

Let p be the input fault-intolerant program with its invariant S, its specification *spec*, and p' be the synthesized multitolerant program with its invariant S'. Since the multitolerant program p' must maintain safety of *spec* from every reachable state in the computations of $p'[](f_{failsafe} \cup f_{masking})$, p' must not reach a state from where safety is violated by a sequence of $f_{failsafe} \cup f_{masking}$ transitions. Hence, we calculate a set of states, say ms (cf. Figure 7.2), from where safety of *spec* is violated by a sequence of transitions of $f_{failsafe} \cup f_{masking}$. Also, p' must not execute transitions that take p' to a state in ms. Hence, we define mt to include these transitions as well as the transitions that violate safety of *spec*.

Now, since p' should be masking $f_{masking}$-tolerant, we use the algorithm Add_Masking to synthesize a program p_1 given the input parameters $p-mt$, $f_{masking}$, $S-ms$, and mt. We only consider faults $f_{masking}$ because p_1 need not be masking fault-tolerant to $f_{failsafe}$. Since a multitolerant program must not reach a state of ms, we use the state predicate $S-ms$ as the input invariant to Add_Masking. Finally, we use mt transitions in place of the *spec* parameter (i.e., the fourth parameter of Add_Masking). Since Add_Masking treats mt as a set of safety-violating transitions, it does not include them in the synthesized program p_1. Thus, starting from a state in S', a computation of $p_1[]f_{masking}$ does not reach a state in ms. As a result, if $T_{masking}$ contains a state s in ms, s can be removed while preserving the masking $f_{masking}$-tolerance property of p_1. Hence, we make the following observation:

Observation 7.9. In the output of the algorithm Add_Masking (cf. Figure 7.2), removing ms states from $T_{masking}$ preserves masking $f_{masking}$-tolerance property of

p_1. □

Now, if faults $f_{failsafe} \cup f_{masking}$ perturb p_1 to a state s, where $s \notin T_{masking}$ then our synthesis algorithm will have to ensure that safety is maintained. To achieve this goal, we add failsafe $(f_{failsafe} \cup f_{masking})$-tolerance to p_1 from $(T_{masking} - ms)$ using the algorithm Add_Failsafe.

```
Add_Failsafe_Masking($p$: transitions, $f_{failsafe}, f_{masking}$: fault, $S$: state predicate,
                                                        $spec$: safety specification)
{
  $ms := \{s_0 : \exists s_1, s_2, ...s_n : (\forall j : 0 \leq j < n : (s_j, s_{(j+1)}) \in (f_{failsafe} \cup f_{masking})) \wedge$
                                                   $(s_{(n-1)}, s_n)$ violates $spec$ $\}$;
  $mt := \{(s_0, s_1) : ((s_1 \in ms) \vee (s_0, s_1)$ violates $spec)\}$;
  $p_1, S', T_{masking} := Add\_Masking(p - mt, f_{masking}, S - ms, mt)$;
  if $(S' = \{\})$   declare no multitolerant program $p'$ exists;
                 return $\emptyset, \emptyset$;
  $p', T' := Add\_Failsafe(p_1, f_{failsafe} \cup f_{masking}, T_{masking} - ms, mt)$;
  return $p', S'$;
```

Figure 7.2: Synthesizing failsafe-masking multitolerance.

The algorithm Add_Failsafe takes the program p_1, faults $f_{failsafe} \cup f_{masking}$, the state predicate $(T_{masking} - ms)$, and the set of mt transitions as the set of transitions that the multitolerant program is not allowed to execute. Since the input invariant to Add_Failsafe (i.e., $(T_{masking} - ms)$) has no ms state, based on the Observation 7.2, the algorithm Add_Failsafe does not remove any state of $(T_{masking} - ms)$. Also, Add_Failsafe does not remove any transition of $p_1|(T_{masking} - ms)$. Thus, we have $(p'|(T_{masking} - ms)) = (p_1|(T_{masking} - ms))$ and $p'|S' = p_1|S'$.

Theorem 7.10. The algorithm Add_Failsafe_Masking is sound.

Proof. Using the soundness of Add_Masking, we have $S' \subseteq (S - ms)$, and as a result, $S' \subseteq S$. Based on the Observation 7.2, it follows that Add_Failsafe preserves $S' \subseteq S$.

Also, from the soundness of Add_Masking, it follows that $p_1|S' \subseteq p|S'$. Using the Observation 7.9, we have $p'|S' \subseteq p|S'$.

Now, we show that p' (cf. Figure 7.2) is indeed multitolerant to $f_{failsafe}$ and

$f_{masking}$ from S' for *spec*.

1. **Absence of faults.** From the soundness of Add_Masking (cf. Theorem 7.6), it follows that p_1 satisfies *spec* from S' in the absence of faults. Thus, using Observations 7.2 and 7.9, it follows that p' satisfies *spec* from S' in the absence of faults.

2. **Masking $f_{masking}$-tolerance.** Based on the soundness of Add_Masking, p_1 is masking $f_{masking}$-tolerant from S' for *spec*. Also, using the Observations 7.2 and 7.9, it follows that p' is masking $f_{masking}$-tolerant from S' for *spec*.

3. **Failsafe ($f_{failsafe} \cup f_{masking}$)-tolerance.** From the soundness of Add_Failsafe, it follows that p' is failsafe ($f_{failsafe} \cup f_{masking}$)-tolerant from T' for *spec*. Using Observation 7.2 and 7.9, since $S' \subseteq (T_{masking} - ms)$, no $f_{failsafe} \cup f_{masking}$ transition can directly violate safety of *spec* from S'. Also, since $(p'|S') \subseteq (p'|(T_{masking} - ms))$, $p'|S'$ does not include any mt transitions. Thus, p' is failsafe ($f_{failsafe} \cup f_{masking}$)-tolerant from S' for *spec*.

Based on the above discussion, it follows that p' is multitolerant to $f_{failsafe}$ and $f_{masking}$ from S' for *spec*. □

Now, we present the completeness proof for Add_Masking algorithm.

Theorem 7.11. The algorithm Add_Failsafe_Masking is complete.

Proof. If there exists a program p'', with invariant S'', and fault-span T'' that is multitolerant to $f_{failsafe}$ and $f_{masking}$ then p'' must be masking $f_{masking}$-tolerant from S'' for *spec*. Thus, there must exist a program synthesized from p that is masking fault-tolerant to $f_{masking}$ faults. Also, since p'' is multitolerant, it must maintain the safety of *spec* in the presence of $f_{failsafe}$ and $f_{masking}$. Thus, we have $T'' \cap ms = \emptyset$ and $p''|T'' \cap mt = \emptyset$. Now, the completeness of Add_Failsafe_Masking follows from the completeness of Add_Masking and Add_Failsafe. □

7.5 Failsafe-Nonmasking-Masking Multitolerance

In this section, we show that, in general, the problem of synthesizing multitolerant programs from their fault-intolerant version is NP-complete. Towards this end, in Section 7.5.1, we show that the problem of synthesizing multitolerant programs from their fault-intolerant version is in NP by designing a non-deterministic polynomial algorithm. Afterwards, in Section 7.5.2, we present a mapping between a given instance of the 3-SAT problem and an instance of the (decision) problem of synthesizing multitolerance. Then, in Section 7.5.3, we show that the given 3-SAT instance is satisfiable iff the answer to the decision problem is affirmative; i.e., there exists a multitolerant program synthesized from the instance of the decision problem of multitolerance synthesis.

7.5.1 Non-Deterministic Synthesis Algorithm

In this section, we first identify the difficulties of adding multitolerance to three distinct classes of faults $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$. Then, we present a non-deterministic solution for adding multitolerance to fault-intolerant programs.

For a program p that is subject to three classes of faults $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$, consider the cases where there exists a state s such that (i) s is reachable in the computations of $p[](f_{failsafe} \cup f_{masking})$ from invariant, (ii) s is reachable in the computations of $p[](f_{nonmasking} \cup f_{masking})$ from invariant, and (iii) no safe recovery is possible from s to the invariant.

In such cases, we have the following options: (i) ensure that s is unreachable in the computations of $p[](f_{failsafe} \cup f_{masking})$ and add a recovery transition (that violates safety) from s to the invariant, or (ii) ensure that s is unreachable in the computations of $p[](f_{nonmasking} \cup f_{masking})$ and leave s as a deadlock state. Moreover, the choice made for this state affects other similar states. Hence, one needs to explore all possible choices for each such state s, and as a result, brute-force exploration of these options requires exponential time in the state space.

Now, given a program p, with its invariant S, its specification *spec*, and three classes of faults $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$, we present the non-deterministic algorithm Add_Multitolerance. In our non-deterministic algorithm, first, we guess a program p', its invariant S', and three fault-spans $T_{failsafe}$, $T_{nonmasking}$, and $T_{masking}$. Then, we verify a set of conditions that ensure the multitolerance property of p'. We have shown our algorithm in Figure 7.3.

```
Add_Multitolerance ($p$: transitions, $f_{failsafe}, f_{nonmasking}, f_{masking}$: fault, $S$: state predicate,
                                                                  $spec$: safety specification)
{
  $ms := \{s_0 : \exists s_1, s_2, ...s_n : (\forall j : 0 \leq j < n : (s_j, s_{(j+1)}) \in (f_{failsafe} \cup f_{masking})) \wedge$
                                          $(s_{(n-1)}, s_n)$ violates $spec$ };          (1)
  $mt := \{(s_0, s_1) : ((s_1 \in ms) \vee (s_0, s_1)$ violates $spec)$ };                (2)

  Guess $p', S', T_{failsafe}, T_{nonmasking}, T_{masking}$;                              (3)
  Verify the following conditions:
      $S' \subseteq S$; $S' \neq \{\}$; $S' \subseteq T_{failsafe}$; $S' \subseteq T_{nonmasking}$; $S' \subseteq T_{masking}$;   (4)
      $(\forall s_0 : s_0 \in S' : (\exists s_1 :: (s_0, s_1) \in p'))$;                  (5)
      $p'|S' \subseteq p|S'$; $S'$ is closed in $p'$;                                      (6)

      $T_{masking}$ is closed in $p'[]f_{masking}$;                                        (7)
      $T_{masking} \cap ms = \emptyset$; $(p'|T_{masking}) \cap mt = \emptyset$;           (8)
      $(\forall s_0 : s_0 \in T_{masking} : (\exists s_1 :: (s_0, s_1) \in p'))$; $(p'|(T_{masking} - S'))$ is acyclic;   (9)

      $T_{failsafe}$ is closed in $p'[](f_{failsafe} \cup f_{masking})$;                  (10)
      $T_{failsafe} \cap ms = \emptyset$; $(p'|T_{failsafe}) \cap mt = \emptyset$;         (11)

      $T_{nonmasking}$ is closed in $p'[](f_{nonmasking} \cup f_{masking})$;              (12)
      $(\forall s_0 : s_0 \in T_{nonmasking} : (\exists s_1 :: (s_0, s_1) \in p'))$; $(p'|(T_{nonmasking} - S'))$ is acyclic;   (13)
}
```

Figure 7.3: A non-deterministic polynomial algorithm for synthesizing multitolerance.

Theorem 7.12 The algorithm Add_Multitolerance is sound and complete. $\square$

Since this algorithm simply verifies the conditions needed for multitolerance in polynomial time in the state space of the program, the proof is straightforward.

Theorem 7.13 The problem of synthesizing multitolerant programs from their fault-intolerant versions is in NP. $\square$

7.5.2 Mapping 3-SAT to Multitolerance

In this section, we give an algorithm for polynomial-time mapping of any given instance of the 3-SAT problem into an instance of the decision problem defined in Section 7.1. The instance of the decision problem of synthesizing multitolerance consists of the fault-intolerant program, p, its invariant, S, its specification, and three classes of faults $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$ that perturb p. The problem statement for the 3-SAT problem is as follows:

3-SAT problem.

Given is a set of propositional variables, $a_1, a_2, ..., a_n$, and a Boolean formula $c = c_1 \wedge c_2 \wedge ... \wedge c_M$, where each c_j is a disjunction of exactly three literals.

Does there exist an assignment of truth values to $a_1, a_2, ..., a_n$ such that c is satisfiable?

Next, we identify each entity of the instance of the problem of multitolerance synthesis, based on the given instance of the 3-SAT formula.

The state space and the invariant of the fault-intolerant program, p. The invariant, S, of the fault-intolerant program, p, includes only one state, say s. Based on the propositional variables and disjunctions of the given 3-SAT instance, we include additional states outside the invariant. Specifically, for each propositional variable a_i, we introduce the following states (cf. Figure 7.4):

- x_i, x'_i, y_i, v_i

And, for each disjunction $c_j = (a_i \vee \neg a_k \vee a_r)$, where $1 \leq i \leq n$, $1 \leq k \leq n$, and $1 \leq r \leq n$, we introduce a state z_j outside the invariant $(1 \leq j \leq M)$.

The transitions of the fault-intolerant program. The only transition in the fault-intolerant program is a self-loop (s, s).

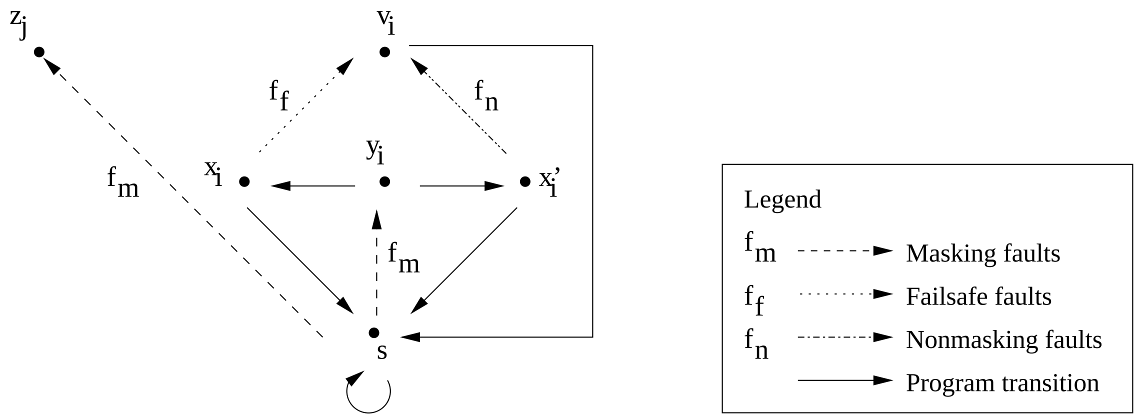

Figure 7.4: The states and the transitions corresponding to the propositional variables in the 3-SAT formula.

The transitions of $f_{failsafe}$. The transitions of $f_{failsafe}$ can perturb the program from x_i to v_i. Thus, the class of faults $f_{failsafe}$ is equal to the set of transitions $\{(x_i, v_i) : 1 \leq i \leq n\}$.

The transitions of $f_{nonmasking}$. The transitions of $f_{nonmasking}$ can perturb the program from x'_i to v_i. Thus, we have $f_{nonmasking} = \{(x'_i, v_i) : 1 \leq i \leq n\}$.

The transitions of $f_{masking}$. The transitions of $f_{masking}$ can take the program from s to y_i. Also, for each disjunction c_j, we introduce a fault transition that perturbs the program from state s to state z_j $(1 \leq j \leq M)$. Thus, the class of faults $f_{masking}$ is equal to the set of transitions $\{(s, y_i) : 1 \leq i \leq n\} \cup \{(s, z_j) : 1 \leq j \leq M\}$.

The safety specification of the fault-intolerant program, p. None of the fault transitions, namely $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$ identified above directly violate safety. In addition, for each propositional variable a_i, the following transitions do not violate safety (cf. Figure 7.4):

- $(y_i, x_i), (x_i, s), (y_i, x'_i), (x'_i, s)$

And, for each disjunction $c_j = a_i \vee \neg a_k \vee a_r$, the following transitions do not violate safety:

- $(z_j, x_i), (z_j, x'_k), (z_j, x_r)$

All transitions except those identified above violate safety of specification. Also, observe that the transition (v_i, s), shown in Figure 7.4, violates safety.

7.5.3 Reduction From 3-SAT

In this section, we show that the given instance of 3-SAT is satisfiable iff multitolerance can be added to the problem instance identified in Section 7.5.2. Specifically, in Lemma 7.14, we show that if the given instance of the 3-SAT formula is satisfiable then there exists a multitolerant program that solves the instance of the multitolerance synthesis problem identified in Section 7.5.2. Then, in Lemma 7.15, we show that if there exists a multitolerant program that solves the instance of the multitolerance synthesis problem, identified in Section 7.5.2, then the given 3-SAT formula is satisfiable.

Lemma 7.14 If the given 3-SAT formula is satisfiable then there exists a multitolerant program that solves the instance of the addition problem identified in Section 7.5.2.

Proof. Since the 3-SAT formula is satisfiable, there exists an assignment of truth values to the propositional variables a_i, $1 \leq i \leq n$, such that each c_j, $1 \leq j \leq M$, is *true*. Now, we identify a multitolerant program, p', that is obtained by adding multitolerance to the fault-intolerant program p identified in Section 7.5.2.

The invariant of p' is the same as the invariant of p (i.e., $\{s\}$). We derive the transitions of the multitolerant program p' as follows. (As an illustration, we have shown the partial structure of p' where $a_i = true$, $a_k = false$, and $a_r = true$ $(1 \leq i, k, r \leq n)$ in Figure 7.5.)

- For each propositional variable a_i, $1 \leq i \leq n$, if a_i is *true* then we will include the transitions (y_i, x_i) and (x_i, s). Thus, in the presence of $f_{masking}$ alone, p' provides safe recovery to s through x_i.

- For each propositional variable a_i, $1 \leq i \leq n$, if a_i is *false* then we will include (y_i, x'_i) and (x'_i, s) to provide safe recovery to the invariant. In this case, since state v_i can be reached from x'_i by faults $f_{nonmasking}$, we include transition (v_i, s) so that in the presence of $f_{masking}$ and $f_{nonmasking}$ program p' provides nonmasking fault-tolerance.

- For each disjunction c_j that includes a_i, we include the transition (z_j, x_i) iff a_i is *true*. And, for each disjunction c_j that includes $\neg a_i$, we include transition (z_j, x'_i) iff a_i is *false*.

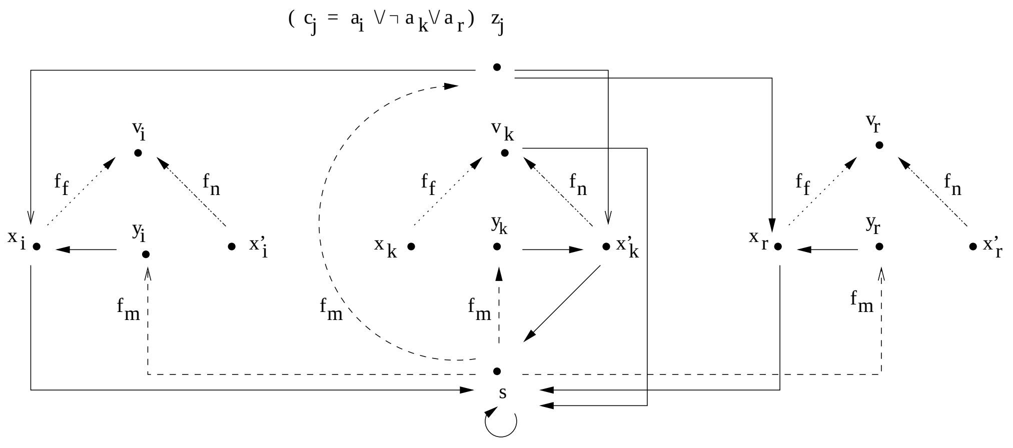

Figure 7.5: The partial structure of the multitolerant program

Now, we show that p' is multitolerant in the presence of faults $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$.

- p' **in the absence of faults.** $p'|S = p|S$. Thus, p' satisfies *spec* in the absence of faults.

- **Masking tolerance to $f_{masking}$.** If the faults from $f_{masking}$ occur then the program can be perturbed to (1) y_i, $1 \leq i \leq n$, or (2) z_j, $1 \leq j \leq M$.

  In the first case, if a_i is *true* then there exists exactly one sequence of transitions, $\langle (y_i, x_i), (x_i, s) \rangle$, in $p'[]f_{masking}$. Thus, any computation of $p'[]f_{masking}$ eventually reaches a state in the invariant. Moreover, starting from y_i the computations of

$p'[]f_{masking}$ do not violate the safety specification. And, if a_i is *false* then there exists exactly one sequence of transitions, $\langle(y_i, x_i'), (x_i', s)\rangle$, in $p'[]f_{masking}$. By the same argument, even in this case, any computation of $p'[]f_{masking}$ reaches a state in the invariant and does not violate the safety specification during recovery.

In the second case, since c_j evaluates to *true*, one of the literals in c_j evaluates to *true*. Thus, there exists at least one transition from z_j to some state x_k (respectively, x_k') where a_k (respectively, $\neg a_k$) is a literal in c_j and a_k (respectively, $\neg a_k$) evaluates to *true*. Moreover, the transition (z_j, x_k) is included in p' iff a_k evaluates to *true*. Thus, (z_j, x_k) (respectively, (z_j, x_k')) is included in p' iff (x_k, s) (respectively, (x_k', s)) is included in p'. Since from x_k (respectively, x_k'), there exists no other transition in $p'[]f_{masking}$ except (x_k, s), every computation of p' reaches the invariant without violating safety. Based, on the above discussion, p' is masking tolerant to $f_{masking}$.

- **Failsafe tolerance to** $f_{masking} \cup f_{failsafe}$**.** Clearly, based on the case considered above, if only faults from $f_{masking}$ occur then the program is also failsafe fault-tolerant. Hence, we consider only the case where at least one fault from $f_{failsafe}$ has occurred.

  Faults in $f_{failsafe}$ occur only in state x_i, $1 \leq i \leq n$. And, p' reaches x_i iff a_i is assigned *true* in the satisfaction of the given 3-SAT formula. Moreover, if a_i is *true* then there is no transition from v_i. Thus, after a fault transition of class $f_{failsafe}$ occurs p' simply stops. Therefore, p' does not violate safety.

- **Nonmasking tolerance to** $f_{masking} \cup f_{nonmasking}$**.** This proof is similar to the proof of failsafe fault-tolerance shown above. Specifically, we only need to consider the case where at least one fault transition of class $f_{nonmasking}$ has occurred.

Faults in $f_{nonmasking}$ occur only in state x'_i, $1 \leq i \leq n$. And, p' reaches x'_i iff a_i is assigned *false* in the satisfaction of the given 3-SAT formula. Moreover, if a_i is *false* then the only transition from v_i is (v_i, s). Thus, in the presence of $f_{masking}$ and $f_{nonmasking}$, p' recovers to its invariant. (Note that the recovery in this case violates safety.)

□

Lemma 7.15 If there exists a multitolerant program that solves the instance of the synthesis problem identified earlier then the given 3-SAT formula is satisfiable.

Proof. Suppose that there exists a multitolerant program p' derived from the fault-intolerant program, p, identified in Section 7.5.2. Since the invariant of p', S', is non-empty and $S' \subseteq S$, S' must include state s. Thus, $S' = S$. Also, since each y_i, $1 \leq i \leq n$, is directly reachable from s by a fault from $f_{masking}$, p' must provide safe recovery from y_i to s. Thus, p' must include either (y_i, x_i) or (y_i, x'_i). We make the following truth assignment as follows: If p' includes (y_i, x_i) then we assign a_i to be *true*. And, if p' includes (y_i, x'_i) then we assign a_i to be *false*. Clearly, each propositional variable in the 3-SAT formula will get at least one truth assignment. Now, we show that the truth assignment to each propositional variable is consistent and that each disjunct in the 3-SAT formula evaluates to *true*.

- *Each propositional variable gets a unique truth assignment.* Suppose that there exists a propositional variable a_i, which is assigned both *true* and *false*, i.e., both (y_i, x_i) and (y_i, x'_i) are included in p'. Now, v_i can be reached by the following transitions (s, y_i), (y_i, x'_i), and (x'_i, v_i). In this case, only faults from $f_{masking}$ and $f_{nonmasking}$ have occurred. Hence, p' must provide recovery from v_i to invariant. Also, v_i can be reached by the following transitions (s, y_i), (y_i, x_i), and (x_i, v_i). In this case, only faults from $f_{masking}$ and $f_{failsafe}$ have occurred. Hence, p' must ensure safety. Based on the above discussion, p' must provide a safe recovery to the invariant from v_i. Based on the definition of the safety specification identified in Section 7.5.2, this is not possible. Thus, propositional variable a_i is assigned only one truth value.

- *Each disjunction is true.* Let $c_j = a_i \vee \neg a_k \vee a_r$ be a disjunction in the given 3-SAT formula. The corresponding state added in the instance of the multitolerance problem is z_j. Note that state z_j can be reached by the occurrence of a fault from $f_{masking}$ from s. Hence, p' must provide safe recovery from z_j. Since the only safe transitions from z_j are those corresponding to states x_i, x'_k and x_r, p' must include at least one of the transitions (z_j, x_i), (z_j, x'_k), or (z_j, x_r).

  Now, we show that the transition included from z_j is consistent with the truth assignment of propositional variables. Specifically, consider the case where p' contains transition (z_j, x_i) and a_i is assigned *false*, p' can reach x_i in the presence of faults from $f_{masking}$ alone. Moreover, if a_i is assigned *false* then p' contains the transition (y_i, x'_i). Thus, x'_i can also be reached by the occurrence of faults from $f_{masking}$ alone. Based on the above proof for unique assignment of truth values to propositional variables, p' cannot reach x_i and x'_i in the presence of $f_{masking}$ alone. Hence, if (z_j, x_i) is included in p' then a_i must have been assigned truth value *true*. Likewise, if (z_j, x'_k) is included in p' then a_k must be assigned truth value *false*. Thus, with the truth assignment considered above, each disjunction must evaluate to *true*. $\square$

Theorem 7.16 The problem of synthesizing multitolerant programs from their fault-intolerant versions is NP-complete. $\square$

7.5.4 Failsafe-Nonmasking Multitolerance

In this section, we extend the NP-completeness proof of synthesizing multitolerance for the case where we add failsafe fault-tolerance to one class of faults, say $f_{failsafe}$, and we add nonmasking fault-tolerance to another class of faults, say $f_{nonmasking}$.

Our mapping for this case is similar to that in Section 7.5.2. We replace the $f_{masking}$ fault transition (s, y_i) with a sequence of transitions of $f_{failsafe}$ and $f_{nonmasking}$ as shown in Figure 7.6. Likewise, we replace fault transition (s, z_j) with a structure similar to Figure 7.6. Thus, y_i (respectively, z_i) is reachable by $f_{failsafe}$ faults alone and by $f_{nonmasking}$ faults alone. As a result, v_i is reachable in the computations of $p'[]f_{failsafe}$ and in the computations of $p'[]f_{nonmasking}$. Thus, to add multitolerance, safe recovery must be added from v_i to s (cf. Figure 7.4). Now, we note that with this mapping, the proofs of Lemmas 7.14 and 7.15 and Theorem 7.16 can be easily extended to show that synthesizing failsafe-nonmasking multitolerance is NP-complete. Thus, we have

Corollary 7.17. The problem of synthesizing failsafe-nonmasking multitolerant programs from their fault-intolerant version is NP-complete. □

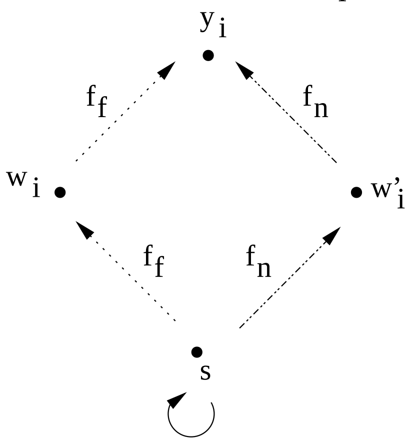

Figure 7.6: A proof sketch for NP-completeness of synthesizing failsafe-nonmasking multitolerance.

7.6 Summary

In this chapter, we investigated the problem of synthesizing multitolerant programs from their fault-intolerant versions. The input to the synthesis algorithm included the fault-intolerant program, different classes of faults to which fault-tolerance had to be added, and the level of tolerance provided for each class of faults. Our algorithms ensured that the synthesized program provided (i) the specified level of fault-tolerance if a fault from any single class had occurred, and (ii) the minimal level of fault-tolerance if faults from multiple classes occurred.

We presented a sound and complete algorithm for the case where failsafe (respectively, nonmasking) fault-tolerance would be added to one class of faults and masking fault-tolerance would be provided to another class of faults. Thus, in these cases, if a multitolerant program could be synthesized for the given input program, our algorithms would always produce one such fault-tolerant algorithm. The complexity of these algorithms is polynomial in the state space of the fault-intolerant program.

For the case where one needs to add failsafe fault-tolerance to one class of faults and nonmasking fault-tolerance to another class of faults, we showed that this problem is NP-complete. As mentioned earlier, this result was counterintuitive as adding failsafe and nonmasking fault-tolerance to the *same* class of faults can be done in polynomial time. However, adding failsafe fault-tolerance to one class of faults and nonmasking fault-tolerance to another class of faults is NP-complete.

Although the results focused in this chapter deal with the high atomicity model, we note that the algorithms in high atomicity model are important in synthesizing distributed fault-tolerant programs as well. Specifically, our algorithms identify a limit up to which even *highly powerful* processes can add the necessary multitolerance. Thus, the output of these algorithms can be used in identifying the limits that distributed processes –along with their limitation on reading and writing variables of the program– can achieve in terms of adding the necessary multitolerance. As an illustration, we note that in Chapter 5, we have identified how algorithms in high atomicity can be systematically used in enhancing the level of fault-tolerance to a single class of faults.

Chapter 8

FTSyn: A Software Framework for Automatic Synthesis of Fault-Tolerance

In this chapter, we present the design and the internal working of the framework Fault-Tolerance Synthesizer (FTSyn) that we have developed for the synthesis of fault-tolerant distributed programs. This framework allows the users to automatically (respectively, interactively) add fault-tolerance. We also show that our framework permits one to add new heuristics for adding fault-tolerance. Towards this end, we describe the addition of several heuristics (based on the algorithms proposed in [14] and in Chapter 5) for different steps involved in adding fault-tolerance. Further, we show how one can easily change the internal representation of different entities in the framework.

We have used our framework to synthesize several fault-tolerant programs among them (i) an altitude switch that controls the altitude of an aircraft by monitoring the altitude sensors and generating necessary command signals, where the altitude switch tolerates the corruption of altitude sensors; (ii) a token ring protocol that tolerates process-restart faults; (iii) an agreement protocol that tolerates Byzantine faults; (iv) an agreement program that tolerates both Byzantine faults and fail-stop faults; (v) an alternating bit protocol program that tolerates message-loss faults, and (vi) a Triple Modular Redundancy program that tolerates input-corruption faults. These examples illustrate the potential of our framework in adding fault-tolerance to different types of faults with different natures.

We proceed as follows: in Section 8.1, we illustrate how the developers of fault-tolerance can synthesize fault-tolerant programs using our framework. In Section 8.2, we present the design of the framework, and discuss the internal working of the framework. In Section 8.3, we show how one can integrate new heuristics into our framework. In Section 8.4, we present the way in which one can change the internal representation of entities involved in the framework. In Section 8.5, we present a simplified version of an altitude switch synthesized using our framework. We make concluding remarks and discuss future work in Section 8.6.

8.1 Adding Fault-Tolerance to Distributed Programs

In this section, we first describe the input and the output of our framework (cf. Section 8.1.1). Then, in Section 8.1.2, we give an overview of framework *fractions* that participate in the *automatic synthesis* of fault-tolerant programs. We implement a deterministic version of *Add_ft* algorithm (cf. Section 2.8) and a set of heuristics developed in [14, 15] to synthesize a fault-tolerant program. Further, in Section 8.1.3, we illustrate how the users can interact with the framework in order to *semi-automatically* synthesize a fault-tolerant program from its fault-intolerant version.

8.1.1 The Input/Output of the Framework

In this subsection, we explain how developers of fault-tolerance should prepare the input to our framework and how the framework provides the output to its users.

The input of our framework consists of the fault-intolerant program, its invariant, its safety specification, its initial states, and a class of faults.

We represent the input fault-intolerant program by Dijkstra's guarded commands [22]. A guarded command (action) is of the form $g \rightarrow st$, where g is a state predicate and st is a statement that updates the program variables. The guarded command $g \rightarrow st$ includes all program transitions $\{(s_0, s_1) : g$ holds at s_0 and the atomic execution of st at s_0 takes the program to state $s_1\}$. The output of our framework is also the *abstract* structure of the fault-tolerant program, represented by guarded commands.

We note that there exist automated techniques (e.g., [42, 43]) by which we can extract the abstract structure of programs written in common programming languages, and then provide our framework with the abstract structure of programs. Moreover, after the synthesis of a fault-tolerant program, there exist automated techniques (e.g., [44, 45, 46]) that allow us to refine the abstract structure of the fault-tolerant program while preserving its correctness and fault-tolerance properties. Next, we present a very simple example of a token ring program to illustrate the way developers can communicate with our framework to add fault-tolerance. Our goal is to provide an overall picture about the input/output of our framework. Afterwards, in Subsection 8.1.2, we show the internal working of our framework and how it synthesizes the fault-tolerant token ring program.

8.1.1.1 Token ring program

The fault-intolerant program consists of four processes P_0, P_1, P_2, and P_3 arranged in a ring. Each process P_i, $0 \leq i \leq 3$, has a variable x_i with the domain $\{-1, 0, 1\}$. We say that process P_i, $1 \leq i \leq 3$, has the token if and only if $(x_i \neq x_{i-1})$ and fault transitions have not corrupted P_i and P_{i-1}. And, P_0 has the token if $(x_3 = x_0)$ and fault transitions have not corrupted P_0 and P_3. Process P_i, $1 \leq i \leq 3$, copies x_{i-1} to x_i if the value of x_i is different than x_{i-1}. This action passes the token to the next process. Also, if $(x_0 = x_3)$ holds then process P_0 copies the value of $(x_3 \oplus 1)$ to x_0, where $\oplus$ is addition in modulo 2. Now, if we initialize every x_i, $0 \leq i \leq 3$, with 0 then process P_0 has the token and the token circulates along the ring. In the input file of our framework, we specify the actions of P_0 as follows (keywords are shown in *italic*):

```
1  process  P0
2  begin
3    (x0 == x3) ->  x0 = ((x3+1)%2);
4  read  x0, x3;
5  write x0;
6  end
```

Since processes P_1, P_2, and P_3 are similar, we present their actions in a parameterized format, where $1 \leq i \leq 3$.

```
1  process  Pi
2  begin
3      (xi != x(i-1)) ->  xi = x(i-1);
4  read  xi, x(i-1);
5  write xi;
6  end
```

Read/Write restrictions. Each process P_i, $1 \leq i \leq 3$, is only allowed to read x_{i-1} and x_i, and allowed to write x_i. Process P_0 is allowed to read x_3 and x_0, and write x_0. We specify the read/write restrictions of a process by *read* and *write* keywords inside the body of the process (cf. lines 4 and 5 in the body of P_i).

Faults. The faults are also modeled as a set of guarded commands that change the values of program variables. In the case of the token ring program, the faults may corrupt at most three processes. Also, in this example, the faults are detectable in that a process that is corrupted can detect if it is in a corrupted state. Hence, we model the fault at process P_i by setting $x_i = -1$. Thus, one of the fault actions that corrupts x_0 is represented as follows:

```
fault  TokenCorruption
begin
   ( ((x0!=-1)&&(x1!=-1)) || ((x0!=-1)&&(x2!=-1)) ||
     ((x0!=-1)&&(x3!=-1)) || ((x1!=-1)&&(x2!=-1)) ||
     ((x1!=-1)&&(x3!=-1)) || ((x2!=-1)&&(x3!=-1)) )
                                              -> x0 = -1;
end
```

Note that there exist no read/write restrictions for the fault transitions because we assume that fault transitions can read and write arbitrary program variables.

Safety specification. The safety specification of the fault-intolerant program is represented as a Boolean expression over program variables. In the token ring program, the problem specification stipulates that the fault-tolerant program is not allowed to take a transition where a non-corrupted process copies a corrupted value from its neighbor. Also, the program should not reach a state where there exists more than one token. In the input of the framework, we represent the specification as follows.

```
( ((x1s!=-1)&&(x1d==-1)) || ((x2s!=-1)&&(x2d==-1)) ||
  ((x3s!=-1)&&(x3d==-1)) || ((x3s==-1)&&(x0s!=x0d)) )
```

Note that we have added a suffix "s" (respectively, suffix "d") to the variable names that stands for *source* (respectively, *destination*). Since the above condition specifies a set of transitions t_{spec} using their source and destination states, we need to distinguish between the value of a specific variable xi in the source state of t_{spec} (i.e., xis means the value of xi in the source state of t_{spec}) and in the destination state of t_{spec} (i.e., xid means the value of xi in the destination state of t_{spec}).

Invariant. The invariant is also specified as a Boolean expression over program variables. The invariant of the token ring program consists of the states where no process is corrupted and there exists only one token in the ring. We represent the invariant of the program using the *invariant* keyword followed by a state predicate.

```
invariant
((x0==1)&&(x1==0)&&(x2==0)&&(x3==0)) ||
((x0==1)&&(x1==1)&&(x2==0)&&(x3==0)) ||
((x0==1)&&(x1==1)&&(x2==1)&&(x3==0)) ||
((x0==1)&&(x1==1)&&(x2==1)&&(x3==1)) ||
((x0==0)&&(x1==0)&&(x2==0)&&(x3==0)) ||
((x0==0)&&(x1==0)&&(x2==0)&&(x3==1)) ||
((x0==0)&&(x1==0)&&(x2==1)&&(x3==1)) ||
((x0==0)&&(x1==1)&&(x2==1)&&(x3==1))
```

Initial states. We also specify some initial states in the input of the synthesis framework. While these initial states are included in the invariant of the fault-intolerant program, we find that explicitly listing them assists in adding fault-tolerance. The initial states of the token ring program are as follows (*init* and *state* are keywords):

```
init
state x0 = 0; x1 = 0; x2 = 0; x3 = 0;
state x0 = 1; x1 = 1; x2 = 1; x3 = 1;
```

The output fault-tolerant program. Finally, the output of our framework is also generated in guarded commands. For the token ring program, the actions of process P_0 in the synthesized fault-tolerant program are as follows:

```
(x0==-1) && (x3==1) -> x0 := 0;
 |
(x0==1) && (x3==1)  -> x0 := 0;
 |
(x0==0) && (x3==0)  -> x0 := 1;
 |
(x0==-1) && (x3==0) -> x0 := 1;
```

The above actions mean that P_0 can copy the value of $(x_3 \oplus 1)$ to x_0 as long as $x_3 \neq -1$. We present the actions of other processes in a parameterized format.

```
(xi==1) && (x(i-1)==0)   -> xi := 0;
 |
(xi==-1) && (x(i-1)==0)  -> xi := 0;
 |
(xi==0) && (x(i-1)==1)   -> xi := 1;
 |
(xi==-1) && (x(i-1)==1)  -> xi := 1;
```

The above actions stipulate that each process P_i ($1 \leq i \leq 3$) can copy the value of x_{i-1} to x_i if $((x_{i-1} \neq -1) \wedge$ $(x_i \neq x_{i-1}))$ holds (i.e., P_{i-1} is not corrupted). We would like to note that the token ring program that we have *automatically* synthesized using our framework is the same as the program that was *manually* designed in [10].

8.1.2 Framework Execution Scenario

In this subsection, we discuss the sample execution scenario for the case where fault-tolerance is added without any user interaction. Also, we use the token ring example to illustrate the execution of the synthesis algorithm. In this scenario, the synthesis algorithm consists of four fractions: *Initialize, PreserveInvariant, ModifyInvariant*, and *ResolveCycles* (cf. Figure 8.1).

Expanding the reachability graph. Before the execution of the synthesis algorithm, the framework uses initial states and program (respectively, fault) transitions to generate the state-transition graph of the fault-intolerant program. Since this directed graph only includes those states of the state space that are *reachable* by program/fault transitions from initial states, we call it a *reachability graph* of the fault-intolerant program. (It also represents the fault-span of the fault-intolerant program.)

The reachability graph of the token ring program. For the token ring program presented in Section 8.1.1, the reachability graph is equal to its state space and includes 81 states. Let $\langle x_0, x_1, x_2, x_3 \rangle$ denote a state of the token ring program. Thus, starting

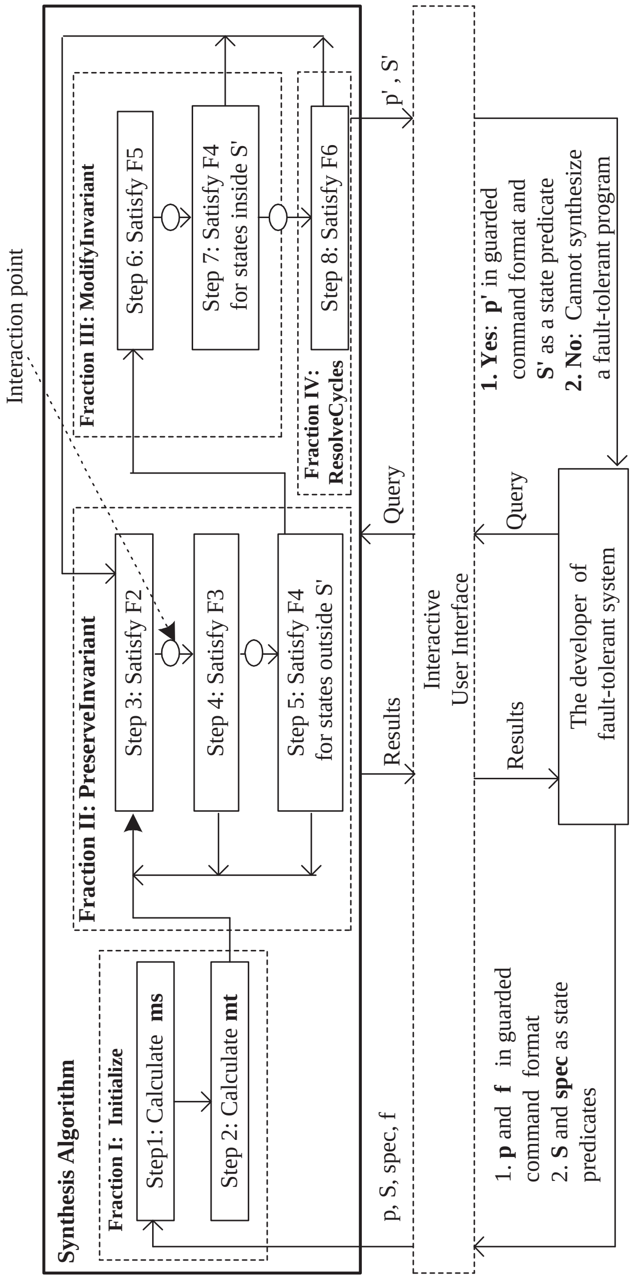
Synthesis Algorithm
Fraction I: Initialize
Step1: Calculate ms
Step 2: Calculate mt
Fraction II: PreserveInvariant
Step 3: Satisfy F2
Step 4: Satisfy F3
Step 5: Satisfy F4 for states outside S'
Fraction III: ModifyInvariant
Step 6: Satisfy F5
Step 7: Satisfy F4 for states inside S'
Fraction IV: ResolveCycles
Step 8: Satisfy F6
Interaction point
p, S, spec, f
Results
Query
p' , S'
Interactive User Interface
1. p and f in guarded command format
2. S and spec as state predicates
Results
Query
The developer of fault-tolerant system
1. Yes: p' in guarded command format and S' as a state predicate
2. No: Cannot synthesize a fault-tolerant program

Figure 8.1: The framework deterministic execution mechanism.

from the initial state $s_0 = \langle 0,0,0,0\rangle$, fault transitions may perturb the program to $s_1 = \langle -1,0,0,0\rangle$, where process P_0 is corrupted. From s_1, process P_1 copies the corrupted value and the fault-intolerant program reaches state $s_2 = \langle -1,-1,0,0\rangle$. As a result, starting from the given initial states, a combination of program and fault transitions can take the state of the program to any possible state in the whole state space.

Execution of fraction (I). After the expansion of the reachability graph, the framework executes every step of the synthesis algorithm (i.e., $F1$-$F6$ in Figure 2.4) on the reachability graph of the fault-intolerant program in order to derive a reachability graph of the fault-tolerant program. First, in fraction (I) (cf. Figure 8.1), the synthesis algorithm calculates the sets of ms states and mt transitions (in the reachability graph).

The token ring program in fraction (I). In the case of the token ring program, safety is violated when a process copies a corrupted value from its neighbor. Thus, fault transitions do not directly violate safety, and as a result, the set of ms states is empty. Also, since ms is empty, the set of mt transitions is equal to the set of program transitions that directly violate safety.

Execution of fraction (II). Then, the synthesis algorithm moves to fraction (II) where we attempt to identify a valid fault-span T' that (i) is closed in $p'[]f$; (ii) does not include any ms states or safety-violating transitions of mt, and (iii) does not include any deadlock states outside the invariant. While executing in fraction (II), we leave the invariant S' unchanged. This is due to the fact that the addition problem requires that the invariant of the fault-tolerant program is a subset of the invariant of the fault-intolerant program. Thus, states inside the invariant of the fault-intolerant program are important; removing them prematurely can cause the automated synthesis to fail.

Also, when we remove ms states (respectively, remove mt transitions) from T' in order to satisfy $F3$, the new fault-span will be a subset of initial T'. As a result, those transitions that start in the new fault-span and end in the part of T' that is not in the new fault-span violate the closure of the fault-span (i.e., $F2$) and must be removed. Hence, after satisfying $F3$, we may need to re-satisfy $F2$. A similar scenario can happen while resolving deadlock states (i.e., satisfying $F4$). Hence, fraction (II) is an iterative procedure. The execution continues in fraction (II) until an iteration does not cause any changes or until the number of iterations exceeds a predetermined bound.

The token ring program in fraction (II). For the token ring program, the framework removes (groups of) program transitions that violate safety of specification. For example, the transition that process P_1 takes from s_1 to s_2 violates the safety of specification. Hence, the synthesis algorithm removes (s_1, s_2) in fraction (II). As a result, $s_1 = \langle -1, 0, 0, 0 \rangle$ becomes a state without any outgoing transition; i.e., deadlock state.

The execution of fraction (II) does not create any deadlock states inside the invariant of the token ring program since ms is empty and no mt transition exists inside the invariant. Thus, in the first iteration, the synthesis algorithm only removes a set of transitions in the fault-span outside the invariant (i.e., mt transitions and the transitions that violate the closure of fault-span).

Execution of fraction (III). At the end of fraction (II), if the resulting program does not satisfy $F1$-$F6$, we modify the invariant S' in fraction (III) to ensure that the invariant S' is closed in the program p', i.e., $F5$ is satisfied. In fraction (III), we recalculate a valid invariant. In this fraction, the newly added transitions may violate the closure of the fault-span. Thus, when we exit fraction (III), the conditions $F2$-$F4$ may need to be re-satisfied. Hence, we jump to fraction (II) and attempt to re-satisfy $F2$-$F4$. Notice that in fraction (III), we satisfy $F4$ only for the invariant states; i.e., we ensure that there is no deadlock state inside the invariant whereas in fraction (II), we resolve deadlock states that are in the fault-span but outside the invariant.

The token ring program in fraction (III). As we mentioned earlier, the removal of mt transitions creates deadlock states outside the invariant of the token ring program. For example, state $s_1 = \langle -1, 0, 0, 0 \rangle$ became a deadlock state since the framework removed a transition to $s_2 = \langle -1, -1, 0, 0 \rangle$ taken by P_1. Now, in the fraction (III), the framework adds recovery transitions to the invariant by allowing a corrupted process to copy an uncorrupted value from its predecessor. Thus, from s_2, process P_0 can toggle the value of x_3 and correct itself by moving to state $s_3 = \langle 1, -1, 0, 0 \rangle$. Now, from s_3, process P_1 copies x_0 and takes the program to state $s_4 = \langle 1, 1, 0, 0 \rangle$, which is in the invariant. Note that since P_1 cannot read variables x_2 and x_3, the group of transitions associated with the transition (s_3, s_4), say g_{34}, includes 9 transitions. By definition, the values of x_3 and x_4 remain unchanged in each transition of g_{34}. Also, P_1 does not propagate a corrupted value by executing transition (s_3, s_4). Thus, no transition in g_{34} violates safety of specification.

Execution of fraction (IV). If the values of p', S', and T' satisfy formulae $F2$-$F5$ at the end of fraction (III) then we will ensure that p' will not stay outside its invariant forever. Toward this end, we move into fraction (IV) where we remove reachable non-progress cycles in $T' - S'$ (if any).

The token ring program in fraction (IV). As long as there exists an uncorrupted value, the token ring program can propagate that value along the ring and recover to the invariant. Since faults can perturb at most three processes, the existence of an uncorrupted process is always guaranteed. Also, no non-progress cycles exist outside the invariant of the token ring program. Thus, in this automatic execution scenario, our framework generates the fault-tolerant token ring program presented in Section 8.1.1 by adding safe recovery from deadlock states outside the invariant.

8.1.3 User Interactions

Although the framework can automatically synthesize a fault-tolerant program without user intervention, there are some situations where (i) user intervention can help to speed up the synthesis of fault-tolerant programs, or (ii) a fully automatic approach fails. In this subsection, we present the nature of the interactions that fault-tolerance developers can have with our framework.

Our framework permits developers to semi-automatically supervise the synthesis procedure. In such *supervised synthesis*, fault-tolerance developers interact with the framework and apply their insights during the synthesis. In order to achieve this goal, we have devised some *interaction points* (cf. Figure 8.1) where the developers can stop the synthesis algorithm and query it.

At each interaction point, the users can make the following kinds of *queries*: (i) apply a specific heuristic for a particular task; (ii) apply some heuristics in a particular order; (iii) view the incoming program (respectively, fault) transitions to a particular state; (iv) view the outgoing program (respectively, fault) transitions from a particular state; (v) check the membership of a particular state (respectively, transition) to a specific set of states (respectively, transition); e.g., check the membership of a given state s in the set of ms states, and finally (vi) view the intermediate representation of the program that is being synthesized. Since our goal is to focus on the technical details of the framework and its application in adding fault-tolerance, we omit the details about the user interface of the framework. We refer the reader to the tutorial about using this framework in the Appendix B.

While we expect that the queries included in this version will be sufficient for a large class of programs, we also provide an alternative for the cases where the heuristics fail and these queries are insufficient. Specifically, in such cases, the users of our framework need to determine what went wrong during synthesis. The answer to this question is very difficult without the help of automated techniques, especially for programs with large state space. To address this issue, developers of fault-tolerance can obtain the corresponding intermediate program in Promela modeling language [37]; this program can then be checked by the SPIN model checker to determine the exact scenario where the intermediate program does not provide the required fault-tolerance property. The counterexamples generated by SPIN enable the users to identify the appropriate heuristics that should be applied in subsequent steps of synthesis.

8.2 Framework Internals

The integration of new heuristics into our framework (respectively, modifying the internal representation of framework entities) requires some background knowledge about the design and the internal working of our framework. Hence, in this section, we present preliminary information that helps the users of the framework (especially the developers of heuristics) to understand the internal working of the framework. We use this information in Sections 8.3 and 8.4 to describe how the framework permits the addition of new heuristics and the ability to change the internal representation of its entities.

We organize this section as follows: In Section 8.2.1, we introduce the important classes (i.e., abstract data structures) used in the design of the framework and their relationship. Then, in Section 8.2.2, we identify three important design patterns that help to make the design of the framework extensible.

8.2.1 Class Modeling

The input to the synthesis algorithm consists of the following entities: program, process, fault, safety specification, invariant, and initial states. Hence, we create the following classes corresponding to each entity: Program, Process, Fault, SafetySpecification, Invariant, and InitialStates. Also, since we can generate the fault-span (i.e., reachability graph) of the fault-intolerant program using the initial states and program (respectively, fault) transitions, we regard the fault-span of the fault-intolerant program as an input entity. Thus, we model the fault-span of the fault-intolerant program using ReachabilityGraph (RG) class. The synthesis framework takes the input entities and then executes the synthesis algorithm in order to generate a fault-tolerant program, its invariant, and its fault-span. Thus, we model the output entities using the same category of classes Program, Invariant, and RG.

We depict the class diagram of the synthesis framework in Figure 8.2. This figure identifies the important classes and their relationship. For example, each Process is *composed of* one or more Action objects. (We annotate the composition relation by black diamonds attached to an arrowed line.) Every Process is *associated* with zero or more TransitionGroup objects that are created due to the read restrictions of that process. (We illustrate associations by solid lines.) Finally, we have derived some new classes from the original classes of our abstract design by *inheritance* relationship. (We annotate inheritance by a solid line attached to a triangle.) For example, we have an abstract class Transition from which we have inherited two concrete classes ProgramTransition and FaultTransition.

8.2.2 Design Patterns

In this section, we identify three important design patterns [47], Bridge, FactoryMethod, and Strategy, that we use in our framework. The advantage of using design patterns with respect to traditional abstract data types stems in the level of flexibility and reusability that these design patterns provide in the design and implementation of our framework.

We use the Bridge design pattern (cf. Figure 8.3) in order to achieve extensibility. The Bridge pattern is a structural design pattern [47] that allows us to separate the *design class hierarchy* from the *implementation class hierarchy*. This way, we can independently extend the design and the implementation of the framework

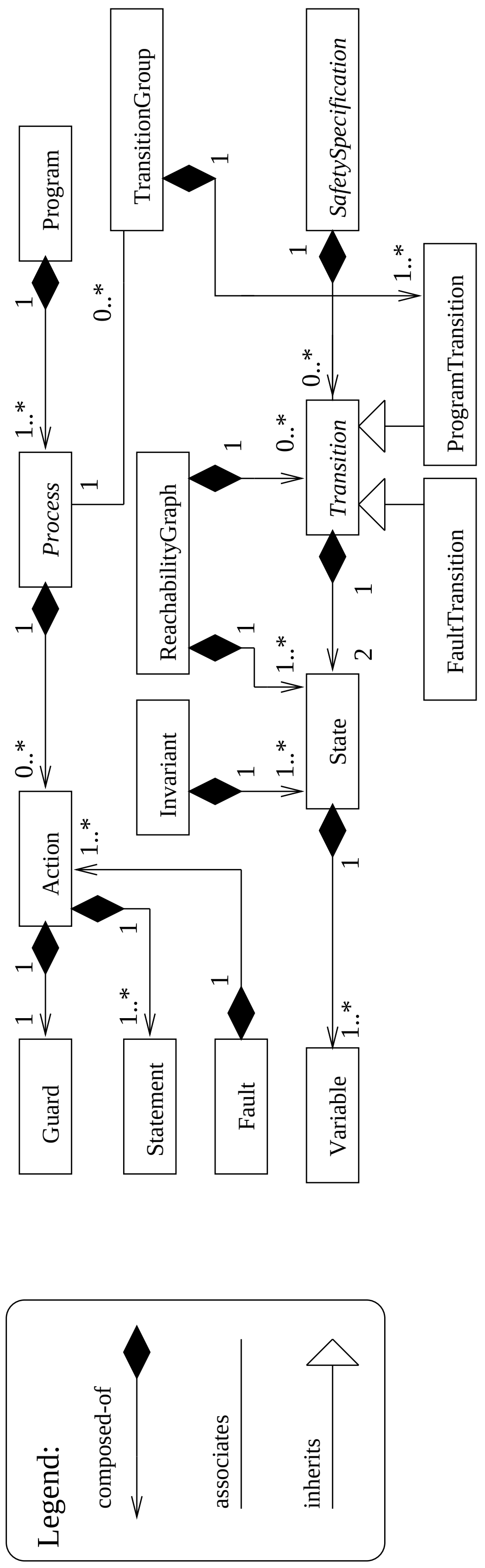

Figure 8.2: The class diagram of FTSyn.

by subclassing. For example, we can introduce different implementation hierarchies corresponding to the AbstractProgram class, where these implementation hierarchies implement a common interface Program_Implementor (cf. Figure 8.3).

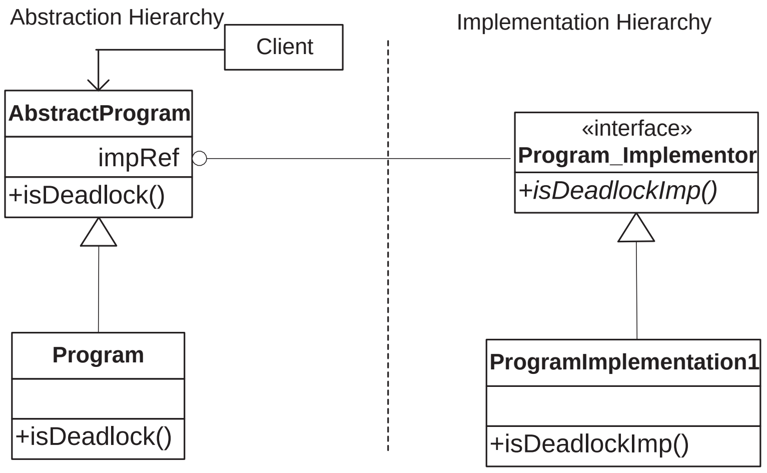

Figure 8.3: The Bridge design patterns.

Another requirement for the developers of fault-tolerance is the ability to apply a specific heuristic at a particular stage of synthesis. Hence, the framework has to dynamically instantiate different classes that represent different heuristics at run-time. In order to achieve this goal, we use the FactoryMethod design pattern (cf. Figure 8.4). The FactoryMethod pattern is a creational pattern [47] that facilitates the dynamic instantiation of objects at run-time. Hence, if one adds a new heuristic in the form of a new class, which is extended from the abstract design of the framework, then the users of the framework can activate the newly added heuristic at run-time.

As we mentioned in the Introduction, the developers of heuristics should be able to easily integrate new heuristics into the framework. We presented the contribution of the Bridge and the FactoryMethod patterns respectively in achieving extensibility and dynamic instantiation of heuristics at run-time. Yet another issue is the design of different versions of a heuristic. In the case where there are different algorithms for a specific step of the synthesis algorithm, we need to implement different versions of

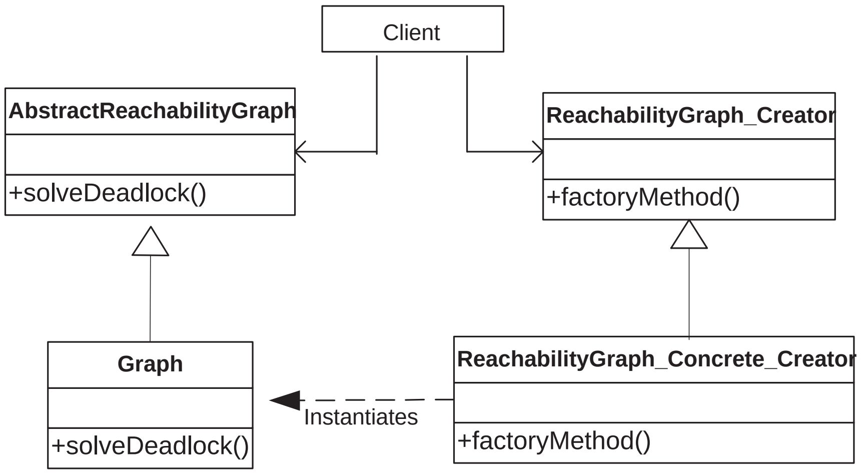

Figure 8.4: The FactoryMethod design patterns.

a particular class (respectively, method). For example, in resolving deadlock states, we may have different heuristics for dealing with a deadlock state. Hence, we need to have different versions of the solveDeadlock method of the RG class (cf. Figure 8.5).

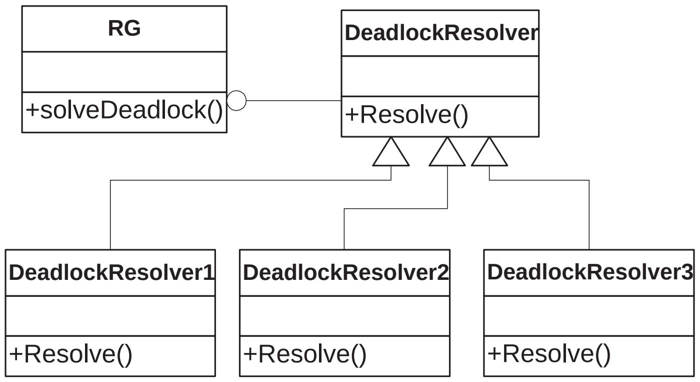

Figure 8.5: Integrating the deadlock resolution heuristics using Strategy pattern.

We use the Strategy pattern [47] to provide a flexible solution to the above-mentioned problem. In particular, we design a DeadlockResolver class for deadlock resolution (cf. Figure 8.5). This class has a method called Resolve, where we implement our deadlock resolution heuristic. Then, we apply the Strategy pattern to DeadlockResolver so that the developers of heuristics can extend new classes from the DeadlockResolver class and integrate their own heuristic in the Resolve method (cf. Figure 8.5). Finally, in the solveDeadlock method of the RG class, we use the FactoryMethod design pattern in order to dynamically instantiate different subclasses of the DeadlockResolver class at run-time.

8.3 Integrating New Heuristics

In this section, we address the problem of adding new heuristics into our framework (i.e., the second goal mentioned in the Introduction). Specifically, we show how one can integrate a new heuristic into our framework so that the added heuristic will be available to the developers of fault-tolerance during synthesis. Since a new heuristic will be integrated into a new class or into a method of an existing class, the problem of *adding new heuristics to the framework* reduces to the problem of adding new classes (respectively, methods) to the framework.

We have used the ability to add heuristics for adding several heuristics from [14, 31, 15]. Of these heuristics, we now present the integration of the three heuristics that we added for resolving deadlocks and discuss our experience in adding them.

First heuristic. Kulkarni, Arora, and Chippada [14] present a heuristic for deadlock resolution that includes two passes. In the first pass, their heuristic tries to add single-step recovery transitions from a given deadlock state, s_d, to the invariant. Due to distribution restrictions, when their heuristic adds a recovery transition, t_{rec} , it has to add the group, g_{rec} , of transitions that is associated with t_{rec}. Moreover, the addition of g_{rec} is not allowed if there exists a transition $(s_0, s_1) \in g_{rec}$ such that (i) $(s_0, s_1) \in mt$; (ii) $(s_0, s_1 \in S) \;\wedge\; (s_0, s_1) \notin p$; (iii) $(s_0 \in T') \;\wedge\; (s_1 \notin T')$, or (iv) $(s_0 \in S) \;\wedge\; (s_1 \notin S)$. If adding recovery from s_d is not possible, and s_d is directly reachable from the invariant by fault transitions then their heuristic does nothing in the first pass. Otherwise, their heuristic makes s_d unreachable.

In the second pass, if there still exists a deadlock state s_d that is directly reachable from the invariant by fault transitions then their heuristic makes s_d unreachable by removing the corresponding invariant state. At the end of deadlock resolution, if the invariant is empty then they declare that their heuristic could not synthesize a fault-tolerant program. We have integrated their heuristic into the framework using the DeadlockResolver1 class (cf. Figure 8.5) that inherits from the DeadlockResolver class.

Second heuristic. The first heuristic only adds single-step recovery to deadlock states. As a result, it fails in cases where single-step recovery is not possible. For example, the first heuristic fails in the case where recovery from a deadlock state, say s'_d, is possible via another deadlock state, say s_d, from where we have already added a recovery transition to the invariant. Hence, we develop a new heuristic for adding *multi-step* recovery to deadlock states for the cases where single-step recovery to the invariant is not possible.

Our new heuristic also consists of two passes. In the first pass, we conduct a fixpoint computation that searches through the deadlock states outside the invariant in the fault-span. In the first iteration of the fixpoint computation, we find all *deadlock* states from where single-step recovery to the invariant is possible. In the second iteration, we find all *deadlock* states from where single-step recovery is possible to recovery states explored in the first iteration. Continuing thus, we reach an iteration of the fixpoint computation where either no more deadlock states exist or no more recovery is possible. In the latter case, we choose to deal with the remaining deadlock states in the second pass. In the former case, at the end of the fixpoint computation, we will have a set of states, *RecoveryStates*, from where there exists a multi-step recovery path to the invariant. (Notice that adding a recovery transition in a distributed program requires the satisfaction of the grouping requirements described in the first heuristic.)

In the second pass, we try to remove s_d if s_d is directly reachable by fault transitions from the invariant and no recovery can be added to s_d. If the removal of s_d requires the removal of one or more invariant states then we remove those invariant states. During deadlock resolution, if the invariant becomes empty then we declare that the synthesis framework failed to synthesize a fault-tolerant program.

In order to integrate this new heuristic into our framework, we extended a new class DeadlockResolver2 (cf. Figure 8.5) from the abstract class DeadlockResolver and then implemented our new heuristic in its Resolve method.

Third heuristic. The strategy of the third heuristic is similar to that in the second heuristic, except that the domain of the fixpoint computation includes all the states outside the invariant in the fault-span (i.e., $(T' - S')$). In other words, the third heuristic is more general than the second heuristic. (Likewise, the second heuristic is more general than the first heuristic.) We have also used this heuristic for enhancing the fault-tolerance of nonmasking programs – where the program only guarantees recovery to the invariant in the presence of faults and not necessarily a safe recovery – to masking fault-tolerance [15]. The integration of the third heuristic was fairly simple. We integrated the third heuristic into a class DeadlockResolver3 (cf. Figure 8.5) extended from the abstract class DeadlockResolver.

The application of heuristics. The second heuristic suffices for the synthesis of the fault-tolerant token ring program presented in Subsection 8.1.1. However, in the synthesis of a version of the Byzantine agreement program containing four non-general processes, since the second heuristic failed, we applied the third heuristic (see Appendix B for this program).

The developers of fault-tolerance have the option to select one of the above heuristics during synthesis. Despite the generality of the third heuristic, it is not as efficient as the first two heuristics. Therefore, given a particular problem, the developers can either use their insight to choose the appropriate heuristic or they can rely on the framework to make that choice. The former choice provides more efficiency whereas the latter choice allows more automation.

8.4 Changing the Internal Representations

As we mentioned in the Introduction, it is difficult to determine a priori the internal representation that one should use for different entities, namely Program, Fault, Specification, and Invariant, involved in the synthesis of fault-tolerant programs. Thus, it is necessary to provide the ability to modify the internal representation of these entities while reusing the remaining parts of the framework. In fact, there are situations where one needs to use one internal representation while executing in one fraction of the framework, and a different internal representation for the same entity while executing in another fraction of the framework.

In this section, we argue that our framework enables such a change of internal representation for entities involved in our framework. Towards this end, we discuss our experience in changing the internal representation of SafetySpecification and Invariant in our framework. We find that the ability to modify the representation of entities in this fashion is especially useful for improving the efficiency of the framework as well as in simplifying the tasks involved in responding to user queries at interaction points. We discuss these applications next.

Improving the efficiency. The initial implementation of the SafetySpecification class consisted of a linked list whose elements would each represent a set of safety-violating transitions. The SafetySpecification class includes a method violates by which we verify whether a given transition t violates the safety specification or not. In order to verify the safety of t, we needed to traverse the linked list structure of SafetySpecification. The traversal of the SafetySpecification structure was very time-consuming, especially when the size of the state space would become large. Since during the synthesis of a fault-tolerant program we need to invoke the method violates in many places, the efficiency of this method significantly degrades the overall efficiency of the synthesis. Hence, we changed the data structure used for the internal representation of the SafetySpecification class.

We replaced the linked list structure of the SafetySpecification class with a dummy data structure. Now, for a given transition t, we first take the source and destination states of t (specified as s_t and d_t). In order to verify the safeness of t, we then substitute the values of the program variables at s_t and d_t into the state predicates that represent the safety *specification* (e.g., refer to Section 8.5 or Subsection 8.1.1). If the specification predicate holds for s_t and d_t then t violates safety. (Note that we represent safety specification as a set of transitions that the program is not allowed to execute.) We have applied the same approach for the Invariant class. Therefore, instead of traversing a huge linked list data structure, we check only a predicate in order to find out the safeness of a transition or the membership of a state to the invariant.

Reasoning about a query. As we discussed in this section, we have two different implementations for the SafetySpecification class based on the *linked list* and the *dummy* data structures. The latter data structure helps to improve the efficiency of the synthesis when we need to automatically synthesize a fault-tolerant program without user intervention. On the other hand, when users interact with our framework, they may need to know why a particular transition violates the safety specification. To answer this query, the framework uses the information stored in the linked list data structure in order to provide the required reasoning for the users. Thus, in such situations, the framework switches the implementation of the SafetySpecification class from a dummy to a linked list data structure to provide the required reasoning for the developers of fault-tolerance.

8.5 Example: Altitude Controller

In this section, we show how we used our framework to synthesize a simplified version of an altitude switch (ASW) used in aircraft altitude controller. We have adapted this example from [48] and the output program of our framework is the same as the fault-tolerant program that is manually designed in [48]. This example illustrates the applicability of our framework in automatic synthesis of practical applications.

The program of the altitude switch reads a set of input variables coming from two analog altitude sensors and a digital altitude sensor. Then, the ASW program activates an actuator when the altitude is less than a pre-determined threshold.

The fault-intolerant altitude switch (ASW). The ASW program monitors a set of input variables and generates an output. There exist five internal variables, a mode variable that determines the operating mode of the program, and four input variables that represent the state of the altitude sensors. The internal variables are as follows: (i) $AltBelow$ is equal to 1 if the altitude is below a specific threshold, otherwise, it is equal to 0; (ii) $ActuatorStatus$ is equal to 1 if the actuator is powered on, otherwise, it is equal to 0; (iii) $Init$ represents the system initialization when it is equal to 1; otherwise, it is equal to 0; (iv) $Inhibit$ is equal to 1 when the actuator power-on is inhibited; otherwise, it is equal to 0, and (v) $Reset$ is equal to 0 if the system is being reset.

The ASW program can be in three different modes: (i) the *Initialization* mode when the ASW system is initializing; (ii) the *Await-Actuator* mode if the system is waiting for the actuator to power on, and (iii) the *Standby* mode. We use an integer variable $Status$ with domain $\{-1, 0, 1, 2\}$ to show the system modes in the program where (i) $Status = -1$ if the system is in the initialization mode; (ii) $Status = 0$ if the system is in the Await-Actuator mode; (iii) $Status = 1$ if the system is in the Standby mode, and (iv) $Status = 2$ if the system is in a faulty state.

Moreover, we model the signals that come from the input (analog and digital) altitude sensors using the following variables: (i) $AltFail$ is equal to 1 when analog and digital altitude meters are failed; (ii) if the system remains in the Initialization mode more than 0.6 second then the variable $InitFailed$ will be set to 1. Otherwise, $InitFailed$ remains 0; (iii) if the condition $AltFail = 1$ remains true more than 2 seconds then the variable $AltFailOver$ will be equal to 1. Otherwise, $AltFailOver$ remains 0, and (iv) if the system remains in the Await-Actuator mode more than 2 seconds then the variable $AwaitOver$ will be equal to 1. Otherwise, $AwaitOver$ remains 0.

The output of the ASW program is identified based on the system mode. The ASW program has an output integer variable $WakeupActuator$ that is equal to 1 if the system is in the Await-Actuator mode and is equal to 0 otherwise. The domain of all variables except $Status$ is equal to $\{0, 1\}$.

The fault-intolerant program consists of only one process, called Controller. In the input of our framework, we specify the Controller process as follows:

```
process  Controller
begin

((Status == -1) && (Init == 1))   -> Status = 1; Init = 0;
|
((Status == 1) && (Reset == 0))   -> Status = -1; Reset = 1;
|
((Status == 1) && (AltBelow == 0) && (Inhibit == 0)
                     && (ActuatorStatus ==0))  -> Status = 0; AltBelow = 1;
|
((Status == 0) && (ActuatorStatus == 0)) -> Status = 1; ActuatorStatus = 1;
|
((Status == 0) && (Reset == 0))  -> Status = -1; Reset = 1;

read  AltBelow, ActuatorStatus, Init, Inhibit, Reset,
 AltFail, InitFailed, AltFailOver, AwaitOver, Status;

write WakeupActuator, AltBelow, ActuatorStatus,
    Init, Inhibit, Reset,  Status;
end
```

The program changes its mode from Initialization to Standby when the $Init$ variable is equal to 1. Also, the program goes to the Initialization mode when it is either in Standby or in Await-Actuator mode and the reset signal is received. If the program is in the Standby mode and the actuator power-on is not inhibited and the actuator is not powered on then the program goes to Await-Actuator mode. In the Await-Actuator mode, the program either (i) powers on the actuator and goes to the standby mode, or (ii) goes to the Initialization mode upon receiving the reset signal.

The read/write sections in the body of the Controller process identify its read/write restrictions on the program variables.

Faults. If the altitude sensors incur malfunction then the state of the program will be perturbed to a faulty state. We represent the fault actions as follows:

```
fault  Malfunction
begin

 (InitFailed == 1 )  -> InitFailed = 0; Status = 2;
 |
 (AltFailOver == 1 ) -> AltFailOver = 0; Status = 2;
 |
 (AwaitOver == 1 )   -> AwaitOver = 0; Status = 2;

end
```

Safety specification. The problem specification requires that the program does not change its mode from Standby to Await-Actuator if the altitude sensors are failed; i.e., $AltFail$ is equal to 1. Also, from the faulty state, the program can only go to the Initialization mode. Moreover, in the faulty state, the program can recover if it is not reset. In the input file, we represent the specification as a state predicate.

```

((AltFails == 1) && (Statuss == 1) && (Statusd == 0)) ||
((Statuss == 2) && ((Statusd == 1) || (Statusd == 0)))||
((Statuss == 2) && (Resets == 1))
```

As we described in Subsection 8.1.1, to distinguish the value of a variable (e.g., $AltFail$) at the source of a transition from its value at the destination, we append the variable names with suffixes 's' and 'd' (e.g., $AltFails$ and $AltFails$).

Invariant. The invariant of the program consists of the states where the program is not in the faulty state; i.e., $Status \neq 2$. We specify the invariant as follows:

```
invariant

(Status != 2)
```

Initial states. We specify the initial state as follows:

```
init

state
        WakeupActuator = 0;
        AltBelow = 1;
        ActuatorStatus = 0;
        Init = 1;
        Inhibit = 0;
        Reset = 0;
        AltFail = 0;
        InitFailed = 1;
        AwaitOver = 1;
        AltFailOver = 1;
        Status = -1;


```

Fault-tolerant program. The framework automatically generates the following fault-tolerant program. We present the actions of the Controller process as follows:

```
1   ((Status == -1) && (Init == 1))   -> Status = 1; Init = 0;
2   |
3   ((Status == 1) && (Reset == 0))   -> Status = -1; Reset = 1;
4   |
5   ((Status == 1) && (AltBelow == 0) && (Inhibit == 0)
6                  && (ActuatorStatus ==0) && ( AltFail == 0))
7                                                  -> Status = 0; AltBelow = 1;
8   |
9   ((Status == 0) && (ActuatorStatus == 0))  -> Status = 1; ActuatorStatus = 1;
10  |
11  ((Status == 0) && (Reset == 0))  -> Status = -1; Reset = 1;
12  |
13
14   (Status == 2) && (Reset == 0)   ->  Status = -1; Reset = 1;
```

The fault-tolerant program has a new recovery action (cf. Line 14), where it recovers to the initialization mode from faulty state (i.e., states where $Status = 2$ holds). Also, a new constraint has been added to the third action (cf. Lines 7-9) where the program is allowed to change its state to the Await-Actuator mode only when the input sensors are not corrupted; i.e., the condition $(AltFail = 0)$ holds.

8.6 Summary

In this chapter, we presented a framework for adding fault-tolerance to existing fault-intolerant programs. Our notion of *program* refers to the abstract structure of programs (cf. Chapter 2), represented in Dijkstra's guarded command language [22]. Thus, the input to our framework is an abstract structure of the fault-intolerant program. The framework synthesizes the abstract structure of the fault-tolerant program.

We showed that our framework is extensible in that it permits easy addition of new heuristics that help in reducing the complexity of adding fault-tolerance. The framework also allows one to partially change the internal representation of different entities used in the synthesis while reusing other entities. These abilities are especially useful for testing different heuristics as well as testing the effect (in terms of space, time, etc.) of different internal representations of entities involved in synthesis. Finally, since we have developed the framework in Java, it is platform-independent; we have used this framework on Windows/Solaris environment. We also find that the choice of this implementation makes our framework suitable for pedagogical purposes.

Using our framework, we have synthesized fault-tolerant programs for, among others, token ring, agreement in the presence of Byzantine faults, and agreement in the presence of Byzantine and failstop faults. Thus, these examples demonstrate that the framework can be applied for the cases where we have different types of faults (process restart, Byzantine and failstop), and for the cases where a program is subject to multiple simultaneous faults.

Chapter 9

Ongoing Research

In this chapter, we present ongoing research work, where we have developed preliminary results. Specifically, we focus on developing heuristics that can extend the scope of efficient synthesis by transforming non-monotonic programs (respectively, specifications) to monotonic. Such heuristics are especially beneficial where for a specific program the monotonicity property (defined in Section 4.3) holds, whereas no guarantees are provided for the monotonicity of its specification (or vice versa). Towards this end, we present a set of heuristics for transforming non-monotonic programs (respectively, specifications) to monotonic where we benefit from Theorem 4.11 and synthesize fault-tolerant distributed programs in polynomial time.

Moreover, in this chapter, we present a SAT-based synthesis approach where we use state-of-the-art SAT solvers to synthesize fault-tolerant distributed programs. In particular, we show how we reduce different sub-problems in the synthesis of fault-tolerant programs to the satisfiability problem. Afterwards, we show how we implement our SAT-based approach in the FTSyn framework (presented in Chapter 8).

We proceed as follows: In Section 9.1, we present our heuristics for transforming non-monotonic programs (respectively, specifications) to monotonic. Then, in Section 9.2, we present an algorithm for transforming non-monotonic specifications to monotonic. We demonstrate our transformation algorithms by an example in Section 9.3. Subsequently, in Section 9.4, we present our SAT-based synthesis method. We summarize this chapter in Section 9.5.

9.1 Program Transformation

In this section, our goal is to address the following question: *Given a fault-intolerant distributed program and its invariant that do not satisfy monotonicity requirements, how can one modify the program and its invariant such that monotonicity requirements are met while ensuring that the program satisfies its specification from the modified invariant*? To address this question, first, we formally define the problem of transforming programs to monotonic (failsafe-ready) programs in Subsection 9.1.1. Then, in Subsection 9.1.2, we present an algorithm for solving the transformation problem. Finally, in Subsection 9.1.3, we show the soundness of our transformation algorithm.

9.1.1 Problem Statement

Given a program p, a state predicate Y, and a Boolean variable x, if p is not positive (respectively, negative) monotonic on Y with respect to x then our goal is to identify a program p' and a state predicate Y' such that p' is positive (respectively, negative) monotonic on Y' with respect to x. We require p' not to add new computations to the set of computations of p during such transformation. Thus, Y' should be a subset of Y. Otherwise, if Y' includes a state s, where $s \notin Y$, then p' may create new computations from s, which is not desirable. Also, for the same reason, p' must not include new transitions during such transformation. Thus, we require that the set of transitions of p' on Y' is a subset of the set of transitions of p on Y' (i.e., $p'|Y' \subseteq p|Y'$). Hence, we state the problem of transforming non-monotonic programs as follows:

Problem 9.1.1 Transforming Non-Monotonic Programs to Monotonic

Given p, Y, *spec*, and x such that p satisfies *spec* from Y, and

p is not positive (respectively, negative) monotonic on Y with respect to x

Identify p' and Y' such that

$Y' \subseteq Y$,

$p'|Y' \subseteq p|Y'$, and

p' is positive (respectively, negative) monotonic on Y' with respect to x

p' satisfies *spec* from Y'. $\square$

Before we present our algorithms, we recall the definition of the monotonicity property from Section 4.3. Observe that in the definition of monotonicity, we implicitly refer to transitions (s_0, s_1) and (s'_0, s'_1) where the value of all variables except x is the same in s_0 and s'_0 (respectively, in s_1 and s'_1). Hence, we introduce the concept of *symmetric* transitions with respect to x as follows:

Definition 9.1.2. We say two transitions $t = (s_0, s_1)$ and $t' = (s'_0, s'_1)$ are *symmetric with respect to a Boolean variable* x (denoted $\bar{t} =_x t'$) iff the condition $((x(s_0) = x(s_1)) \wedge (x(s'_0) = x(s'_1)) \wedge (x(s_0) \neq x(s'_0)))$ holds and the value of all variables in s_0 and s'_0 (respectively, in s_1 and s'_1) are the same. $\square$

9.1.2 Transformation Algorithm

In this subsection, we present a sound algorithm to solve Problem 9.1.1. We use the Definition 9.1.2 in the design of our transformation algorithm (see Figure 9.1). The algorithm To_Positive_Monotonic_Programs is an iterative procedure that takes the set of groups of transitions of a distributed program, a state predicate Y, and a Boolean variable x and generates a distributed program p' and a state predicate Y' such that p' is positive monotonic on Y' with respect to x. Intuitively, our algorithm removes the program transitions that go against the monotonicity property. Removing such transitions may create deadlock states in program invariant. Hence, we recalculate another invariant to guarantee that no deadlock states exist in the new invariant. If our algorithm succeeds in finding such an invariant then we generate a monotonic (failsafe-ready) program. Otherwise, our algorithm declares failure in generating a monotonic program.

```
To_Positive_Monotonic_Program($p$: set of transitions, $x$: Boolean variable, $Y$: state predicate )
// $p$ is the union of a set of groups of transitions $g_0, \cdots, g_m$.
{
  Step 1: $p' := p$; $Y' := Y$;
  Step 2: repeat {
            Step 2-1:  $TR_{rem} := \{(s_0, s_1) : (x(s_0) = false) \wedge (x(s_1) = false) \wedge ((s_0, s_1) \in p'|Y') \wedge$
                                    $(\exists (s'_0, s'_1) : (s'_0, s'_1) =_x \overline{(s_0, s_1)} : (s'_0, s'_1) \notin p'|Y')\}$;
            Step 2-2:  if $(TR_{rem} = \emptyset)$ then
                          Step 2-2-1: $Y', p'$ := Recalculate_Invariant($p', Y'$);
                          Step 2-2-2: if $((Y' \neq \emptyset))$     return $p', Y'$;
                                      else declare failure in finding a monotonic program;
            Step 2-3:  $t := (s_0, s_1)$, where $(s_0, s_1) \in TR_{rem}$ and $s_0$ has the maximum outdegree;
            Step 2-4:  $p' := p' - \{(s_2, s_3) : (\exists g_i : g_i \in p' : t \in g_i \wedge (s_2, s_3) \in g_i)\}$
            Step 2-5:  $Y_1$ := RemoveDeadlocks($p', Y'$);
            Step 2-6:  $p_1$ := EnsureClosure($p', Y_1$);
            Step 2-5:  $p' := p_1$; $Y' := Y_1$;
  Step 3: } until $(Y' = \emptyset)$;
  Step 4:   declare failure in finding a monotonic program;
}
```

Figure 9.1: Transforming non-monotonic programs to positive monotonic.

After the initialization, in Step 2-1 (cf. Figure 9.1), we calculate the set of transitions that violate the definition of positive monotonicity. If there exist no such transitions (i.e., $TR_{rem} = \emptyset$) then we will verify (i) the non-existence of deadlock states in Y', and (ii) the closure of p' in Y'. When we reach Step 2-2-1, we recalculate a valid invariant for p' by invoking the function Recalculate_Invariant (cf. Figure 9.2). Obviously, if we reach Step 2-2-1 in the first iteration then that means the input program p and Y inherently satisfy the monotonicity requirements. Note that Steps 2-1 and 2-2 verify the monotonicity of the input program, and hence, we do not need develop a separate verification algorithm.

To recalculate the invariant, we develop an iterative procedure where we first use function RemoveDeadlocks to remove the existing deadlock states of p in a state predicate S (cf. Figure 9.2). The RemoveDeadlocks function returns the largest subset S_1 of S where there exist no deadlock states; i.e., the computations of p are infinite in S_1. After removing the deadlock states of S, there might exist transitions of p that start in S_1 and reach the removed states of S. Such transitions violate the closure of S_1. Using function EnsureClosure (cf. Figure 9.2), we remove (groups of) transitions that violate the closure of S_1. We repeat this procedure until there exist no more deadlock states or we remove all states of S. (We invoke the function HasDeadlocks that verifies if there exist deadlock states in a state predicate S of a program p.)

```
Recalculate_Invariant($p$ : set of transitions, $S$ : state predicate)
// $p$ is the union of a set of groups of transitions $g_0, \cdots, g_m$.
{
  $S' := S$; $p' := p$;
  repeat {
    $S_1 :=$ RemoveDeadlocks($p', S'$);
    $p_1 :=$ EnsureClosure($p', S_1$);
    $p' := p_1$; $S' := S_1$;
  } until ($\neg$ HasDeadlocks($p', S'$) $\vee$ $S' = \emptyset$ );
  return $S', p'$;
}

RemoveDeadlocks( $p$ : set of transitions, $S$ : state predicate)
// Returns the largest subset of $S$ such that computations of $p$ within that subset are infinite
    {   $S' := S$
        while ($\exists s_0 : s_0 \in S' : (\forall s_1 : s_1 \in S' : (s_0, s_1) \notin p)$)       $S' := S' - \{s_0\}$;
        return $S'$;     }

HasDeadlocks($p$ : set of transitions, $S$ : state predicate)
// Verify the existence of deadlock states in $S$.
    { if ($\exists s_0 : s_0 \in S : (\forall s_1 : s_1 \in S : (s_0, s_1) \notin p)$)       return true;
                                                                  return false;      }

EnsureClosure($p$ : set of transitions, $S$ : state predicate)
// $p$ is the union of a set of groups of transitions $g_0, \cdots, g_m$.
    { return $p - \{(s_0, s_1) : (\exists g_i : g_i \in p : ((s_0, s_1) \in g_i) \wedge$
                          $(\exists (s'_0, s'_1) : (s'_0, s'_1) \in g_i : (s'_0 \in S \wedge s'_1 \notin S)))\}$       }
```

Figure 9.2: Algorithms for removing deadlock states and ensuring the closure of the invariant.

In Step 2-3 (see Figure 9.1), we select one of the transitions of TR_{rem}, say t, whose source state has the maximum number of outgoing transitions (i.e., outdegree).

Afterwards, we remove the group of transitions associated with t (cf. Step 2-4). In this way, we reduce the chance of creating more deadlock states. Then, since the removal of transitions may create deadlock states, we invoke RemoveDeadlocks (in Step 2-5). Afterward, we use EnsureClosure to remove the transitions (and their associated groups) that violate the closure of Y_1. We continue the iterative procedure of the algorithm To_Positive_Monotonic_Program until in an iteration either (i) the state predicate Y' becomes empty (in Step 3 or in Step 2-2-2), or (ii) we find a positive monotonic program (in Step 2-2-2).

Likewise, we design an algorithm To_Negative_Monotonic_Programs for transforming distributed programs to negative monotonic programs. The only difference between such algorithm and To_Positive_Monotonic_Programs is in calculating the set of transitions TR_{rem} (see Step 2-1 in Figure 9.1), where we replace the condition $((x(s_0) = false) \wedge (x(s_1) = false))$ with $((x(s_0) = true) \wedge (x(s_1) = true))$.

9.1.3 Soundness

In this subsection, we show that the algorithm To_Positive_Monotonic_Programs (cf. Figure 9.1) is sound; i.e., the transformed program satisfies the requirements of Problem 9.1.1. Towards this end, we make the following observations:

Observation 9.1.3 The function RemoveDeadlocks returns a subset S' of a predicate S where the computations of program p in S' are infinite.

Proof. Since RemoveDeadlocks only *removes* states with no outgoing program transitions, it follows that S' does not have new states (i.e., $S' \subseteq S$). Also, every state that remains in S' has at least one outgoing transition in p. Otherwise, it would have been removed. Therefore, the computations of p are infinite in S'. □

Observation 9.1.4 The functions RemoveDeadlocks and EnsureClosure do not add any new transitions to the set of transitions of program p.

Proof. The proof follows by construction. □

Observation 9.1.5 The function Recalculate_Invariant does not add any new states (respectively, transitions) to the invariant (respectively, the set of transitions) of program p.

Proof. The proof follows from Observations 9.1.3 and 9.1.4. $\square$

Theorem 9.1.6 The algorithm To_Positive_Monotonic_Programs is sound.

Proof. We show that the program generated by To_Positive_Monotonic_Program satisfies the requirements of Problem 9.1.1.

- $Y' \subseteq Y$**.** The algorithm To_Positive_Monotonic_Program calculates state predicate Y' by invoking Recalculate_Invariant (in Step 2-2-1) and RemoveDeadlocks (in Step 2-5). Hence, using Observations 9.1.3 - 9.1.5, it follows that $Y' \subseteq Y$.

- $p'|Y' \subseteq p|Y'$**.** The algorithm To_Positive_Monotonic_Program modifies the transitions of the input program p in Steps 2-2-1, 2-4, and 2-6. Based on observations 9.1.4 and 9.1.5, Steps 2-2-1 and 2-6 do not add any new transitions to the set of transitions $p|Y'$. Also, by construction, Step 2-6 does not add new transitions to $p|Y'$ as well. Thus, it follows that $p'|Y' \subseteq p|Y'$.

- p' **is positive monotonic on** Y' **with respect to** x**.** Since the set of transitions TR_{rem} identifies transitions of $p|Y$ that violate the definition of positive monotonicity of p, and in the final iteration of the algorithm To_Positive_Monotonic_Program the set of transitions TR_{rem} becomes empty, it follows that when the algorithm To_Positive_Monotonic_Program terminates there exist no transitions in $p'|Y'$ that violate the positive monotonicity of p' on Y'. As a result, the program p' returned by To_Positive_Monotonic_Program is positive monotonic on Y' with respect to x.

- p' **satisfies** *spec* **from** Y'**.** Based on Observation 9.1.3, Y' is a subset of Y where the computations of p are infinite. Also, using the requirements $Y' \subseteq Y$ and $p'|Y' \subseteq p|Y'$, it follows that the computations of p' in Y' are a subset of computations of p in Y'. Since starting in Y every computation of p is in *spec*, it follows that starting in Y' every computation of p' is in *spec*. Also, by construction, Y' is closed in p'. Thus, p' satisfies *spec* from Y'.

Based on the above discussion, it follows that To_Positive_Monotonic_Program is sound. □

Theorem 9.1.7 The complexity of algorithm To_Positive_Monotonic_Programs is polynomial in the state space of the input program.

Proof. The maximum number of iterations of the while loop in the body of RemoveDeadlocks function (cf. Figure 9.2) is in the order of $|S|$. Also, for program p, since $S \subseteq S_p$, it follows that the worst-case complexity of RemoveDeadlocks is $\mathcal{O}(|S_p|)$. A similar reasoning shows that the worst-case complexity of HasDeadlocks is $\mathcal{O}(|S_p|)$.

Also, the number of groups of transitions of p is polynomial in $|S_p|$ since in a distributed program each transition is associated with a group of transitions, and the number of transitions included in each process is in the order of $|S_p|^2$. Moreover, by construction, the size of each group is in the order of $|S_p|$ as well. As a result, the worst-case complexity of the EnsureClosure (cf. Figure 9.2) will be polynomial in $|S_p|$. Based on the above discussion, the complexity of Recalculate_Invariant will be polynomial in $|S_p|$ since the loop inside this function can iterate at most $|S_p|$ times.

Now, in the To_Positive_Monotonic_Programs algorithm, the maximum number of iterations of the main loop cannot exceed $|Y|$, where the algorithm removes all states in Y and declares failure in Step 4. Also, each step of the algorithm has a polynomial-time complexity based on the above discussion. Therefore, the complexity of To_Positive_Monotonic_Programs is polynomial in the state space of the input program. □

9.2 Specification Transformation

In this section, our goal is to address the following question: *How can safety specifications be strengthened to meet the monotonicity requirements?* To address this question, in Subsection 9.2.1, we present a formal definition for the problem of transforming non-monotonic specifications to monotonic. Then, in Subsection 9.2.2, we present a sound algorithm for solving the transformation problem.

9.2.1 Problem Statement

Given a safety specification $spec_{sf}$, a state predicate Y, and a Boolean variable x, if $spec_{sf}$ is not positive (respectively, negative) monotonic on Y with respect to x then our goal is to derive a specification $spec'_{sf}$ that is positive (respectively, negative) monotonic on Y with respect to x. In such derivation, we require that if a transition t satisfies $spec'_{sf}$ then t will satisfy $spec_{sf}$ as well. As a result, $spec'_{sf}$ will be a strengthened version of $spec_{sf}$. Hence, we state the problem of transforming non-monotonic specifications to monotonic as follows:

Problem 9.2.1 Transforming Non-Monotonic Specifications to Monotonic

Given Y, $spec_{sf}$, and x such that $spec_{sf}$ is not positive (respectively, negative) monotonic on Y with respect to x

Identify $spec'_{sf}$ such that

$spec_{sf} \subseteq spec'_{sf}$

$spec'_{sf}$ is positive (respectively, negative) monotonic on Y' with respect to x $\square$

Note that we represent safety specifications $spec_{sf}$ and $spec'_{sf}$ as two sets of bad transitions in the state space that must not occur in program computations (cf. Section 2). Thus, the condition $spec_{sf} \subseteq spec'_{sf}$ states that $spec'_{sf}$ is a restricted version of $spec_{sf}$ by adding more transitions to $spec_{sf}$; i.e., strengthening $spec_{sf}$.

9.2.2 Transformation Algorithm

To address the transformation Problem 9.2.1 for positive monotonicity, we present an algorithm that takes a safety specification $spec_{sf}$, a state predicate Y, and a Boolean variable x, and generates a safety specification $spec'_{sf}$ such that $spec'_{sf}$ is positive monotonic on Y with respect to x.

```
To_Positive_Monotonic_Specification($spec_{sf}$: safety specification, $Y$: state predicate,
                                                                    $x$: Boolean variable)
{
  Step 1: $TR_{add} := \{(s_0, s_1) : (x(s_0) = false) \wedge (x(s_1) = false) \wedge$
                    $(s_0 \in Y) \wedge (s_1 \in Y) \wedge ((s_0, s_1) \notin spec_{sf}) \wedge$
                    $(\exists (s'_0, s'_1) : (s'_0, s'_1) =_x \overline{(s_0, s_1)} : (s'_0, s'_1) \in spec_{sf})\}$;
  Step 2: return $spec_{sf} \cup TR_{add}$;
}
```

Figure 9.3: Transforming non-monotonic specifications to monotonic.

In Step 1, the algorithm To_Positive_Monotonic_Specification calculates the set of transitions that violate the definition of positive monotonicity of specification. Then, the algorithm strengthens the specification $spec_{sf}$ by adding the set of *good* transitions TR_{add} to the existing set of bad transitions (specified by $spec_{sf}$) in order to construct a new safety specification $spec'_{sf}$. The new specification $spec'_{sf}$ is represented by a new set of bad transitions $spec_{sf} \cup TR_{add}$. Since the specification returned by To_Positive_Monotonic_Specification is a strengthened version of the original specification $spec_{sf}$, the soundness of the above algorithm follows accordingly. (In the case of negative monotonic specifications, we present a similar algorithm by replacing the condition $((x(s_0) = false) \wedge (x(s_1) = false))$ with $((x(s_0) = true) \wedge (x(s_1) = true))$ in Step 1 in Figure 9.3.)

Theorem 9.2.2 The algorithm To_Positive_Monotonic_Specification is sound. $\square$

Theorem 9.2.3 The complexity of algorithm To_Positive_Monotonic_Specification is polynomial in the size of Y. $\square$

Comment on strengthening the specification. Strengthening the specification does not destroy the fault-safe property of the specification. Specifically, the transformation of a specification to a monotonic specification adds new transitions to the set of bad transitions that must not occur in program computations. Since such new transitions are program transitions, no fault transition will be included as a safety-violating transition. As a result, the fault-safe property of the specification will be preserved during the transformation.

Also, since we add new transitions to the specification during transformation, there may exist program transitions in the invariant that do not violate the original specification but violate the strengthened monotonic specification. Such transitions must not occur in the computations of the transformed program, otherwise the program will violate the safety of the strengthened specification. In the next section, in the context of an example, we illustrate how we identify and remove such transitions from the invariant and then recalculate a new invariant.

9.3 Example: Distributed Control System

In this section, we present an example where we use our transformation algorithms for efficient addition of failsafe fault-tolerance. Specifically, we first present a distributed controlling program that is subject to input faults; i.e., the faults that perturb the input sensors of the program. Then, we transform the specification of the controlling program to a positive monotonic specification. Since the program is negative monotonic, efficient (i.e., polynomial-time) addition of failsafe fault-tolerance to it becomes possible.

The fault-intolerant process-control program (PC). The program PC consists of three processes P_1, P_2, and P_3 connected by a loosely-coupled network. The processes P_1 and P_2 respectively control the speeds of two electro motors M_1 and M_2 located in the same environment but in distant places. The motors M_1 and M_2 provide the driving force of a conveyer belt that can move in two different directions: left-to-right and right-to-left. The conveyer belt carries fragile objects that are loaded when the belt is stationary. Once the objects are loaded, the conveyer belt moves with an increasing speed up to a maximum speed. Then, the belt stops so that the already loaded objects can be unloaded and new objects are loaded.

The speed of the conveyer belt depends on the speeds of M_1 and M_2. The speeds of M_1 and M_2 should be *synchronous*; i.e., the speed of M_1 is equal to the speed of M_2 or is at most one unit more than the speed of M_2. When the two electro motors reach their maximum speed, process P_3 resets their speed to 0 and the whole process repeats. It is required that the temperature of the environment where electro motors function should not exceed a pre-determined threshold.

The program PC has four integer variables x, y, z, and w. The variable x (respectively, z) is a counter that contains the speed of M_1 (respectively, M_2). The domain of x (respectively, z) is equal to $\{0, \cdots, c\}$, where c is an integer constant. The variable y is used to represent the movement direction of the conveyer belt. Specifically, if the direction of the conveyer belt is from left to right then the value of y alternates between 1 and 0. In the case where the conveyer belt moves from right to left, the value of y alternates between -1 and 0. Moreover, the value of y is equal to 0 if $x = z$. Otherwise, y could be 1 or -1. As a result, the domain of y is equal to $\{-1, 0, 1\}$. The variable w represents the temperature of the environment, which could be in three different levels of normal, alarming, and critical that are respectively represented by three values 0, 1, and 2.

Let $\langle x, y, z, w\rangle$ denote the global state of the distributed program. The initial state of the program is $\langle 0, 0, 0, 0\rangle$, where process P_1 starts to speed up (i.e., increment its counter). Process P_1 is responsible to increment x and process P_2 increments z. When both counters reach the maximum value c (i.e., $(x = c) \wedge (z = c)$) the counting operation will be restarted by process P_3.

Read/write restrictions. Process P_1 is allowed to read x, y, and z and it can only write x and y. Process P_2 can read x, y, and z, but it is only allowed to write y and z. Process P_3 is allowed to read all program variables, however, it can only write x and z. Note that P_1 and P_2 cannot read w due to distribution restrictions.

Program actions. We present the action of process P_1 as follows:

$$PC_1 : \quad (x = z) \wedge (x < c) \qquad \longrightarrow \quad x := x + 1; y := 1 | -1;$$

When M_1 and M_2 have the same speed (i.e., $x = z$), P_1 increments the value of x (i.e., the speed of M_1). The action PC_1 indeed represents two actions depending on the direction of the belt (i.e., the value of y). The action of process P_2 is as follows:

$$PC_2 : \quad (x = z + 1) \qquad \longrightarrow \quad y := 0; z := z + 1;$$

Process P_2 increments the value of z (i.e., the speed of M_2) and resets y to zero since z has become equal to x. Finally, the transitions of P_3 are represented by the following action:

$$PC_3 : \quad (x = c) \wedge (z = c) \qquad \longrightarrow \quad x := 0; z := 0;$$

If both counters have reached the maximum value c (i.e., M_1 and M_2 have reached their maximum speed) then P_3 resets their values to 0.

Safety specification. For application-specific purposes, the safety specification stipulates that in the case where the belt is moving in the right-to-left direction and the temperature level is in a critical level (i.e., $w = 2$), the speed of M_2 must remain less than the speed of M_1; i.e., speed of the belt must not be increased in critical temperature. We represent the safety specification of PC with $spec_{PC}$, where

$$spec_{PC} = \quad \{(s_0, s_1) : (y(s_0) = -1) \wedge (x(s_1) = z(s_1)) \wedge (w(s_1) = 2)\}$$

Invariant. The temperature should be in the normal level in ordinary working conditions. Hence, we represent the invariant of the program by the state predicate S_{PC}, where

$$S_{PC} = \quad \{s : (w(s) = 0) \wedge ((x(s) = z(s)) \vee (x(s) = z(s) + 1))\}$$

M_1 and M_2 are synchronized in the invariant; i.e., $((x = z) \vee (x = z + 1))$.

Faults. Faults may change the value of the temperature sensor to 1 or 2 when the speed of M_1 is ahead of M_2. We represent the fault transitions by the following action:

$$F: \quad (x = z + 1) \qquad \longrightarrow \quad w := 1|2;$$

Fault-span. We represent the fault-span of program PC by the following state predicate:

$$\begin{aligned} T_{PC} = \quad & \{s : ((x(s) = z(s)) \vee (x(s) = z(s) + 1)) \ \wedge ((x(s) = z(s)) \Rightarrow (y(s) = 0)) \vee \\ & \qquad ((x(s) = z(s) + 1) \Rightarrow ((y(s) = 1) \vee (y(s) = -1))) \ \} \end{aligned}$$

Note that the value of w could vary in its domain $\{0, 1, 2\}$.

Negative monotonicity of program PC. Since w is not a Boolean variable, we apply the definition of program monotonicity on the program PC by partitioning the domain of w to zero and non-zero values. We consider the Boolean value *true* corresponding to non-zero values of w and the Boolean value $false$ corresponding to $(w = 0)$. Since there exists no transition in $PC|S_{PC}$ where the value of w is non-zero, it follows that the definition of negative monotonicity holds for the program PC. Thus, the program PC is negative monotonic on S_{PC} with respect to w.

Positive monotonicity of $spec_{PC}$. Now, we investigate the positive monotonicity of $spec_{PC}$ on S_{PC} with respect to w. First, we identify the set of transitions (s_0, s_1) that satisfy the following conditions: (i) $s_0, s_1 \in S_{PC}$; (ii) $(w(s_0) = 0) \wedge (w(s_1) = 0)$; (iii) (s_0, s_1) does not violate $spec_{PC}$, and (iv) there exists transition (s'_0, s'_1) that is grouped with (s_0, s_1) due to inability of reading w, where (s'_0, s'_1) violates $spec_{PC}$ and $(w(s'_0) \neq 0) \wedge (w(s'_1) \neq 0)$. Thus, for the specification $spec_{PC}$, the algorithm To_Positive_Monotonic_Specification calculates the set of transitions TR_{add} (cf. Figure 9.3) as follows:

$$TR_{add} = \{(s_0, s_1) : (x(s_0) = z(s_0) + 1) \wedge (y(s_0) = -1) \wedge (w(s_0) = 0) \wedge$$
$$(x(s_1) = z(s_1)) \wedge (y(s_1) = 0) \wedge (w(s_1) = 0)\}$$

The set of transitions TR_{add} includes those transitions of $PC|S_{PC}$ in which the values of x and z become equal in their destination state. Although the transitions of TR_{add} do not violate $spec_{PC}$ by themselves, they are grouped with *unsafe* transitions that reach a state where the condition $((w = 2) \wedge (x = z))$ holds. Hence, we strengthen the safety specification by including the set of transitions TR_{add} in the set of transitions that violate safety. As a result, the new safety specification $spec'_{PC} = spec_{PC} \cup TR_{add}$ satisfies the definition of positive monotonicity for $spec'_{PC}$ on S_{PC} with respect to w.

Recalculating the invariant of the program PC. After strengthening $spec_{PC}$, the program transitions in $TR_{add} \cap (PC|S_{PC})$ violate $spec'_{PC}$ although they do not violate $spec_{PC}$. When we remove the set of transitions $TR_{add} \cap (PC|S_{PC})$, we create the following deadlock states in the invariant S_{PC}.

$$Deadlocks = \{s : (x(s) = z(s) + 1) \wedge (y(s) = -1) \wedge (w(s) = 0)\}$$

We invoke the algorithm Recalculate_Invariant (cf. Figure 9.2) to recalculate a new invariant S'_{PC} where the computations of PC are infinite in S'_{PC}. In the first iteration of the algorithm Recalculate_Invariant, we remove the states in *Deadlocks* from the invariant S_{PC}. Since the removal of the above deadlock states does not introduce new deadlock states, we calculate the new invariant S'_{PC}, where

$$S'_{PC} = \{s : (((x(s) = z(s)) \wedge (y(s) = 0)) \vee$$
$$((x(s) = z(s) + 1) \wedge (y(s) = 1))) \wedge (w(s) = 0)\}$$

The action of the process P_1 in the new invariant is as follows:

$PC'_1 : \quad (x = z) \wedge (x < c) \quad \longrightarrow \quad x := x + 1; y := 1;$

Note that the above action only assigns 1 to y; i.e., all transitions corresponding to the action that assigns -1 to y have been removed during synthesis. Now, we represent the transitions of the process P_2 by the following action:

$$PC_2' : \quad (y = 1) \wedge (x = z + 1) \qquad \longrightarrow \quad y := 0; z := z + 1;$$

The action of process P_3 remains as is. Since program PC' is negative monotonic on S'_{PC} with respect to w and its new specification $spec'_{PC}$ is positive monotonic on S'_{PC} with respect to w, failsafe fault-tolerance can be added to PC' in polynomial time (using Theorem 4.11). In fact, in this case, the program PC' is failsafe F-tolerant to $spec'_{PC}$ from S'_{PC}.

9.4 SAT-based Synthesis of Fault-Tolerance

In this section, we investigate the use of automated reasoning techniques in the synthesis of fault-tolerant distributed programs. There exist several heuristics-based approaches [14] (also see Chapter 5) for polynomial-time synthesis of fault-tolerant distributed programs. Each heuristic identifies a deterministic order for the *verification* of synthesis requirements, where synthesis requirements are conditions that have to be met by program states and transitions during synthesis so that the synthesized fault-tolerant program is correct by construction. As a result, the efficiency of synthesis is directly affected by the efficiency of verifying such synthesis requirements. Thus, it is desirable to benefit from the existing automated reasoning tools to efficiently verify synthesis requirements. Specifically, in this section, we focus our attention on using state-of-the-art SAT solvers during synthesis where we express different synthesis requirements in terms of the satisfiability problem and use existing SAT solvers to efficiently verify those requirements.

We organize this section as follows: First, in Subsection 9.4.1, we give an overview of our SAT-based approach for the synthesis of fault-tolerant distributed programs. In Subsection 9.4.2, we show how we formulate each synthesis requirement as an instance of the satisfiability problem. In Subsection 9.4.3, we discuss the implementation of our SAT-based synthesis method in the FTSyn framework.

9.4.1 Synthesis Method

In this subsection, we present a general overview of our SAT-based synthesis method. Specifically, in Subsection 9.4.1.1, we state the problem of reducing synthesis requirements to the satisfiability problem. Subsequently, in Subsection 9.4.1.2, we provide a strategy for using SAT solvers during synthesis for the verification of the synthesis requirements.

9.4.1.1 Synthesis Requirements Verification

The non-deterministic synthesis algorithm presented in Section 2.8 identifies six requirements that must be verified during the synthesis of a fault-tolerant program from its fault-intolerant version. For reader's convenience, we repeat the Add_ft algorithms in Figure 9.4:

```
Add_ft($p, f$ : set of transitions, $S$ : state predicate, $spec$ : specification,
                                        $g_0, g_1, ..., g_{max}$ : groups of transitions)
{
    $ms := \{s_0 : \exists s_1, s_2, ...s_n : (\forall j : 0 \leq j < n : (s_j, s_{(j+1)}) \in f) \wedge (s_{(n-1)}, s_n)$ violates $spec$ };
    $mt := \{(s_0, s_1) : ((s_1 \in ms) \vee (s_0, s_1)$ violates $spec$) };

    Guess $S', T'$, and $p' := \bigcup$ ($g_i$ : $g_i$ is chosen to be included in the fault-tolerant program);
    Verify the following
        ($F1$) $p'|S' \subseteq p|S'$;
        ($F2$) $S' \subseteq T'$; $T'$ is closed in $p'[]f$;          // $T'$ is a fault-span of $p'$.
        ($F3$) $T' \cap ms = \{\}$; $(p'|T') \cap mt = \{\}$;  // Safety cannot be violated from states in $T'$.
        ($F4$) $(\forall s_0 : s_0 \in T' : (\exists s_1 :: (s_0, s_1) \in p'))$;  // $T'$ does not have deadlocks.
        ($F5$) $S' \neq \{\}$; $S' \subseteq S$; $S'$ is closed in $p'$;      // $S'$ is an invariant of $p'$.
        ($F6$) $p'|(T' - S')$ is acyclic;                     // $p'$ cannot stay in $(T' - S')$ forever.
}
```

Figure 9.4: The non-deterministic algorithm for adding fault-tolerance to distributed programs.

Each one of the conditions $F1$-$F6$ identifies a property $\mathcal{P}$ of the states (respectively, transitions) of the synthesized program. If a program p' (consisting of processes $\{P_0, \cdots, P_n\}$) satisfies all these requirements then that program is fault-tolerant.

Given a process P_j ($0 \leq j \leq n$) that consists of a set of groups of transitions $g_0, \cdots, g_m$ ($0 \leq m$) and a property $\mathcal{P}$, we say P_j has the property $\mathcal{P}$ iff each group of transitions $g_0, \cdots, g_m$ has the property $\mathcal{P}$. Also, a group of transition g_i ($0 \leq i \leq m$) has the property $\mathcal{P}$ iff each transition of g_i has the property $\mathcal{P}$.

Also, given a state predicate X that consists of a set of program states, we say X has the property $\mathcal{P}$ iff each state $s \in X$ has the property $\mathcal{P}$. Hence, we present the verification problem as follows:

The verification problem

Given a group of transitions g (respectively, a state predicate X), and a property $\mathcal{P}$:

Does g (respectively, X) have the property $\mathcal{P}$? $\square$

9.4.1.2 Using SAT Solvers

The verification of the conditions $F1$-$F6$ requires an exhaustive enumeration of the states (respectively, transitions) of the program being synthesized, and as a result, such verification is not efficient for programs with large state space. In this subsection, we present a SAT-based solution for efficient verification of the synthesis requirements.

To verify the synthesis requirements, we transform the problem of verifying a property $\mathcal{P}$ for a group of transitions g (respectively, a state predicate X) to a Boolean formula whose satisfiability can be verified by SAT solvers. Specifically, we define a function BF that takes a group of transitions g (respectively, a state predicate X) and a property $\mathcal{P}$ and generates a Boolean formula. Such transformation can be done in polynomial time in the state space of the program (cf. Section 9.4.2).

Now, given the function BF, we design a *verification sub-layer* that provides verification abilities for the synthesis algorithm (cf. Figure 9.5). Specifically, every time the synthesis algorithm needs to verify a property $\mathcal{P}$ of a group of transitions g (respectively, a state predicate X), it queries the verification sub-layer with $\mathcal{P}$ and g

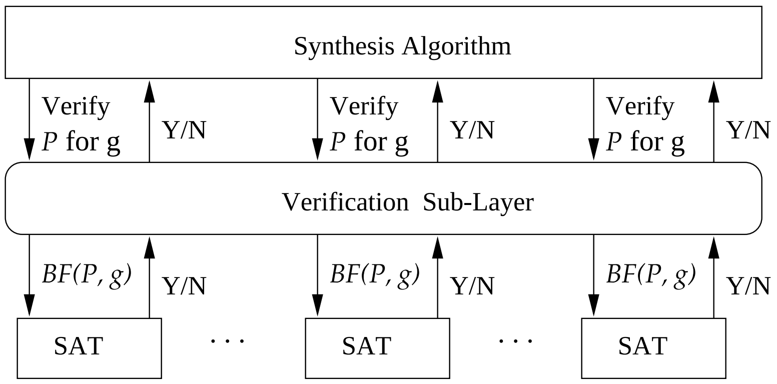

Figure 9.5: Using SAT solvers for the synthesis of fault-tolerant programs.

(respectively, X). The verification sub-layer transforms the request of the synthesis algorithm to a Boolean formula $BF(\mathcal{P}, g)$ and delivers it to the SAT solver. After the SAT solver provides the result of the satisfiability of $BF(\mathcal{P}, g)$, the verification sub-layer forwards this result to the synthesis algorithm. The verification sub-layer has the potential to create multiple instances of the SAT solver to verify $\mathcal{P}$ for a set of groups of transitions in parallel.

9.4.2 Representing Synthesis Requirements as Boolean Formulas

In this section, we show how we formulate the verification of a synthesis requirement in terms of a Boolean formula. First, in Subsection 9.4.2.1, we focus on representing the basic entities of our formal model (i.e., states, transitions, state predicates, and transitions predicates) in terms of Boolean formulas. Then, in Subsection 9.4.2.2, we use the representation of states and transitions to formulate synthesis requirements $F1$-$F6$ (shown in Figure 9.4) in terms of Boolean formulas.

9.4.2.1 Formulating State Transition Graphs

In this subsection, we show how we formulate states, state predicates, transitions, and transition predicates in terms of Boolean formulas. In the next subsection, we use the transformations presented in this subsection to formulate the synthesis requirements.

Representing a state. We recall the definition of a state from Chapter 2, where we define a state as a value assignment to program variables. Formally, a state s of a typical program p with the set of variables $\{v_0, \cdots, v_q\}$ has the form: $\langle l_0, l_1, .., l_q \rangle$ where $\forall i : 0 \leq i \leq q : l_i \in D_i$, D_i is the domain of v_i, and q is a positive integer. Thus, to represent a state s as a Boolean formula, we introduce the transformation $SBF : S_p \rightarrow \mathcal{B}$, where $\mathcal{B}$ is the set of Boolean formulas over program variables.

$SBF(s) = \wedge_{i=0}^{i=q}(v_i = l_i)$, where $l_i \in D_i$

The SBF transformation generates a unique Boolean formula corresponding to each state $s \in S_p$; i.e., SBF is a one-to-one function. However, the formula $SBF(s)$ is specified in terms of equalities over program variables; i.e., $(v_i = l_i)$. To generate a formula that consists of Boolean variables, we have to transform each term $(v_i = l_i)$ in $SBF(s)$ into a formula that only consists of Boolean variables. Towards this end, we introduce $\lceil log(|D_i|) \rceil$ Boolean variables corresponding to each program variable v_i, where $|D_i|$ represents the size of the domain of v_i. In other words, if the domain of v_i includes $|D_i|$ distinct values then we will need $\lceil log(|D_i|) \rceil$ Boolean variables to encode each value assignment to v_i by a unique binary code with length $\lceil log(|D_i|) \rceil$. Therefore, the maximum size of $SBF(s)$ is equal to $(q + 1) \cdot \lceil log(K) \rceil$, where K is the size of the domain of a variable v_j $(0 \leq j \leq n)$ that has the largest domain.

Representing a state predicate. By definition, a state predicate is the union of a set of states in the state space of p (i.e., S_p). Thus, to represent a state predicate $X \subseteq S_p$, we use the function SBF to define a function $SPBF : Pow(S_p) \rightarrow \mathcal{B}$ as follows:

$SPBF(X) = \vee_{\forall s :: s \in X} \; SBF(s)$

The transformation $SPBF$ takes the disjunction of all the Boolean formulas corresponding to all states in X. The resulting formula will be a formula $c_0 \vee c_1 \vee \cdots c_{|X|}$ in disjunctive normal form where each conjunction c_j $(0 \leq j \leq |X|)$ represents a state.

Representing a transition. To represent a transition $(s_0, s_1) \in S_p \times S_p$, we use the SBF function and define the function $TBF : S_p \times S_p \rightarrow \mathcal{B}$, where

$TBF((s_0, s_1)) = SBF(s_0) \ \wedge \ SBF(s_1)$

We represent a transitions (s_0, s_1) as a conjunction of the Boolean formula that represents its source state s_0 and the Boolean formula that represents its destination state s_1. One could argue that TBF should be defined as $SBF(s_0) \ \Rightarrow \ SBF(s_1)$. This way, $TBF((s_0, s_1))$ holds for all transitions terminating at s_1 and the Boolean formula $SBF(s_0) \ \Rightarrow \ SBF(s_1)$ represents more than a single transition. Hence, to represent an individual transitions (s_0, s_1), we use the conjunction of $SBF(s_0)$ and $SBF(s_1)$.

Representing a transition predicate. We use an approach similar to the one we used for defining state predicates. In other words, a transitions predicate $\Delta_p \in S_p \times S_p$ is the union of a set of transitions in the state space S_p. Hence, we define function $TPBF : Pow(S_p \times S_p) \rightarrow \mathcal{B}$ to represent a set of transitions Δ_p, where

$TPBF(\Delta_p) = \vee_{\forall (s_0, s_1)::(s_0, s_1) \in \Delta_p} TBF((s_0, s_1))$

Note that we use transition predicates to model the set of program transitions, a group of transitions, and the safety specification. For example, if $spec_{sf}$ represents the safety specification of a program p then $TPBF(spec_{sf})$ generates a Boolean formula corresponding to $spec_{sf}$.

9.4.2.2 Formulating Synthesis Requirements

In this subsection, we show how we formulate the requirements $F1$-$F6$ of the non-deterministic algorithm presented in Subsection 9.4.1. Towards this end, we use the functions presented in the previous subsection.

We observe that the condition $F1 \equiv (p'|S' \subseteq p|S')$ verifies whether the set of transitions $p'|S'$ is a subset of the set of transitions $p|S'$. Since $p'|S'$ and $p|S'$ are transition predicates, we use $TPBF$ to generate the Boolean formulas corresponding to $p'|S'$ and $p|S'$. Hence, to verify $F1$ we verify the satisfiability of $TPBF(p'|S') \Rightarrow TPBF(p|S')$.

Likewise, for the requirements $F2 \equiv (S' \Rightarrow T')$ and $F5 \equiv (S' \Rightarrow S)$, we respectively verify the satisfiability of $SPBF(S') \Rightarrow SPBF(T')$ and $SPBF(S') \Rightarrow SPBF(S)$. To verify the closure of the state predicate S' in the set of transitions of p' (cf. $F5$ in Figure 9.4), we verify the satisfiability of $CLBF(S', p')$, where

$$CLBF(S', p') = \wedge_{\forall (s_0, s_1)::(s_0, s_1) \in p'} \\ (SBF(s_0) \Rightarrow SPBF(S')) \;\Rightarrow\; (SBF(s_1) \Rightarrow SPBF(S'))$$

To verify $F3$, we simply verify the satisfiability of $SPBF(T') \wedge SPBF(ms)$ and $TPBF(p'|T') \wedge TPBF(mt)$. If these two formulas are not satisfiable then $F3$ is satisfied.

The requirements $F5$ stipulates that there exists no cycles in the set of transitions of $p'|(T' - S')$. As a result, we have to formulate the cycle detection problem in terms of a Boolean formula. To achieve this goal, we adopt the techniques used in the existing approaches for symbolic cycle detection [49, 50, 51] where one generates a Boolean formula whose satisfiability shows the existence of a non-progress cycle in $p'|(T' - S')$. Towards this end, we define a transformation $\mathit{Reach}(s, \Delta_p)$ from $S_p \times \mathit{Pow}(S_p \times S_p)$ to the set of Boolean formulas $\mathcal{B}$, where $\mathit{Pow}(S_p \times S_p)$ is the power set of $(S_p \times S_p)$, and

$$\mathit{Reach}(s, \Delta_p) = SPBF(R) \text{ , where} \\ R = \{s' : s' \text{ is reachable from } s \text{ by transitions of } \Delta_p\}$$

Using function *Reach*, we can construct a Boolean formula that represents the set of states reachable from a particular state $s \in S_p$. Now, to verify if s is in a cycle, we only need to verify the satisfiability of $\mathit{Cycle}(s, \Delta_p)$, where

$$\mathit{Cycle}(s, \Delta_p) \equiv (SBF(s) \Rightarrow \mathit{Reach}(s, \Delta_p))$$

If $Cycle(s, \Delta_p)$ is satisfiable then s is in a cycle in the graph constructed by the set of transitions Δ_p. In the case where $Reach(s, \Delta_p) \equiv false$ then it follows that s is a deadlock state in the state transition graph of Δ_p. Thus, using the *invalidity* of $Reach(s, \Delta_p)$, we conclude that s is a deadlock state (i.e., verifying $F4$ in Figure 9.4).

9.4.3 Implementing SAT-based Synthesis

In this subsection, we present an overview of our implementation strategy where we implement our SAT-based synthesis method in the FTSyn framework presented in Chapter 8. Towards this end, we only focus on the part of implementation that is related to the verification of the requirement $F3$ (cf. Figure 9.4) since the implementation approach for verifying other synthesis requirements is similar.

Given a program p, its groups of transitions $g_0, \cdots, g_m$ and its safety specification $spec_{sf}$, our goal is to identify the groups of transitions whose transitions do not violate $spec_{sf}$; i.e., safe groups. In the initial implementation of FTSyn, we exhaustively verify the safety of the transitions of a group $g_i \in p$ $(0 \leq i \leq m)$. The exhaustive verification is inefficient for the cases where the size of a group is very large. Hence, we expect that our SAT-based approach provides a better performance in verifying the safety of the transition groups.

In the rest of this section, we proceed as follows: First, we present the necessary transformation for formulating the safety verification problem. Then, we introduce different layers of our implementation in FTSyn for solving the safety verification problem.

Safety verification problem. For the program p, we say a group g_i of transitions is safe iff no transition $(s_0, s_1) \in g_i$ violates $spec_{sf}$. Since we represent $spec_{sf}$ as a set of transitions that must not occur in program computations, we say g_i is safe iff the set of transitions of g_i does not intersect with $spec_{sf}$. Formally, we use the transformation $Safe(g_i)$ to represent the safety of g_i, where

$Safe(g_i, spec_{sf}) = TPBF(g_i) \ \wedge \ TPBF(spec_{sf})$

To verify the safety of g_i, we verify the satisfiability of $Safe(g_i, spec_{sf})$. If $Safe(g_i, spec_{sf})$ is satisfiable then it follows that the group g_i intersects $spec_{sf}$; i.e., g_i includes a transition that violates safety. Thus, $Safe(g_i, spec_{sf})$ is satisfiable iff g_i is not safe.

The layers of SAT-based safety verification. To solve the safety verification problem in FTSyn, we implement the following three layers *Boolean formula generation*, *CNF formula generation*, and *native method invocation*. In the first layer, we use a Java API package provided by Alloy analyzer [52] of MIT to formulate the safety verification problem in terms of a Boolean formula. Then, in the CNF formula generation layer, we transform the Boolean formula to Conjunctive Normal Form (CNF) as the existing SAT solvers only accept formulas in CNF format. We use the SAT solver zChaff [53] since zChaff is one of the most efficient SAT solvers at the time of implementing our SAT-based approach. Towards this end, we implement a Java native method where we invoke zChaff to verify the satisfiability of the calculated CNF formulas. The CNF formula is satisfiable iff the group of transitions whose safety is being verified is not safe. Now, we discuss the implementation of each layer.

- *Boolean formula generation.* To generate the Boolean formulas, we first introduce a set of Boolean variables by which we encode the value assignment to program variables. For example, if a program p has an integer variables x with the domain $\{-1, 0, 1\}$ then we use two Boolean variables a_1 and a_2 to represent the terms $(x = -1)$, $(x = 0)$, and $(x = 1)$ respectively by the following Boolean formulas: $(a_1 \wedge a_2)$, $(\neg a_1 \wedge a_2)$, and $(a_1 \wedge \neg a_2)$, where $\neg a_j$ is the complement of a_j $(1 \leq j \leq 2)$. Hence, we represent a state predicate $(x = 0) \vee (x = 1)$ by the Boolean formula $(\neg a_1 \wedge a_2) \vee (a_1 \wedge \neg a_2)$. Note that since the domain of x contains only three values, the term $(\neg a_1 \wedge \neg a_2)$ will never be used in the transformation of state predicates to Boolean formulas.

In the generation of a Boolean formula corresponding to a transition, say (s_0, s_1), the value of a specific variable may be different in s_0 and s_1. Thus, using a set of Boolean variables (e.g., a_1 and a_2 in the above example) for the representation of the source and the destination states may result in the generation of contradictory Boolean formulas. To illustrate this problem, consider the above-mentioned example where we use two Boolean variables a_1 and a_2 to represent value assignments to an integer variable x. Suppose that we need to generate the Boolean formula corresponding to a transition (s_0, s_1) where the value of x at s_0 is -1 (denoted $x(s_0) = -1$) and the program changes the value of x to 0 during the transition (s_0, s_1) (i.e., $x(s_1) = 0$). Now, to formulate (s_0, s_1) using Boolean variables a_1 and a_2, the resulting formula would be equal to $(a_1 \wedge a_2) \wedge (\neg a_1 \wedge a_2)$, which is a logical contradiction. Hence, we need to distinguish the value assignment to variables at the source and the destination of program transitions.

To distinguish the value assignment to a specific variable in a transition, we introduce two separate sets of Boolean variables for representing the value of that variable at the source and at the destination state. For example, we introduce two new Boolean variables b_1 and b_2 to represent the value assignment to variable x in the destination of transitions. Thus, the transition (s_0, s_1), where $x(s_0) = -1$ and $x(s_1) = 0$, will be formulated as $(a_1 \wedge a_2) \wedge (\neg b_1 \wedge b_2)$.

- *CNF formula generation.* Using the approach presented above, we transform the safety specification and each group of transitions to a Boolean formula in terms of variables introduced for encoding the value assignments to program variables. Since zChaff requires the input formula in DIMACS CNF format [54], we have to transform the generated Boolean formulas to CNF format. Towards this end, we use an API provided in the Alloy analyzer [52] and integrate it in FTSyn. Using this API, we transform the generated Boolean formulas to

CNF format, which can be directly delivered to the SAT solver zChaff. For example, in DIMACS format, the formula $(a_1 \vee \neg a_2 \vee a_3) \wedge (\neg a_1 \vee a_2 \vee \neg a_3)$ will be represented as follows:

$$\begin{array}{lll} p\ \ cnf\ \ 3\ \ 2 & & \\ 1 & -2 & 3 \\ -1 & 2 & -3 \end{array}$$

The first line identifies that a CNF formula with 3 variables and two clauses is being specified. Each clause (i.e., disjunction) must be specified on a separate line. Also, the variables and their complements are distinguished by a minus sign.

- *Native method invocation.* In FTSyn, after we automatically generate a CNF formula corresponding to $TPBF(spec_{sf}) \wedge TPBF(g_i)$, we invoke a native method where we query zChaff with the generated CNF formula. The source code of zChaff is available for educational purposes. Hence, we have generated a Dynamic Link Library so that we invoke zChaff from Java environment when we instantiate an instance of our framework FTSyn. Therefore, for every group of transitions g_i, we invoke zChaff once to verify the safety of g_i.

Using the implementation of our SAT-based approach, we have synthesized the token ring program presented in Chapter 6. Since we invoke zChaff from Java environment, the current implementation of our SAT-based approach suffers from the performance of the Java Native Interface. Nonetheless, our implementation provides a platform for SAT-based synthesis of fault-tolerant (distributed) programs and the efficiency of this platform can be improved as the software technology improves.

9.5 Summary

In this chapter, we presented two directions of research in progress. Specifically, we discussed the development of heuristics that can transform non-monotonic programs (respectively, specifications) to monotonic. Since adding failsafe fault-tolerance to distributed programs that satisfy the monotonicity requirements can be done in polynomial time (cf. Chapter 4), such heuristics extend the scope of programs that can reap the benefits of efficient synthesis.

Also, we presented a technique for using SAT solvers in the synthesis of fault-tolerant distributed programs from their fault-intolerant version. We reduce the synthesis requirements to the satisfiability problem and then invoke SAT solvers to solve those problems. This way, we benefit from the efficiency of the state-of-the-art SAT solvers during the synthesis of fault-tolerant distributed programs. Currently, we have created a centralized implementation of our approach, however, we plan to extend this work for the cases where we deploy our synthesis algorithm on a distributed platform. Also, we plan to investigate the applicability of other decision procedures [55] in the synthesis of fault-tolerant distributed programs.

Chapter 10

Conclusion and Future Work

In this chapter, we discuss related work, make concluding remarks, and provide some insight for future research work. Specifically, in Section 10.1, we compare our synthesis approach to the existing approaches in the literature. Then, in Section 10.2, we present the contributions of this dissertation. In Section 10.3, we demonstrate the impacts of the synthesis approach presented in this dissertation. Finally, in Section 10.4, we present open problems and future research directions.

10.1 Discussion

In this section, we discuss issues related to the approach presented in this dissertation. Specifically, we compare our synthesis method with the existing synthesis approaches in the literature. Towards this end, we address some questions raised regarding our synthesis method and the framework FTSyn that we have developed for the synthesis of fault-tolerant (distributed) programs.

How does the synthesis method presented in this dissertation differ from model-theoretic synthesis approach?

The synthesis method in model-theoretic approach [2, 56, 3, 57, 4] is based on a decision procedure for the satisfiability proof of the specification. Although such synthesis methods may have slight differences with respect to the input specification language and the program model that they synthesize, the general approach is based on the satisfiability proof of the specification. This makes it difficult to provide reuse in the synthesis of programs; i.e., any changes in the specification require the synthesis to be restarted from scratch. By contrast, since the input to our synthesis method is the set of transitions of a fault-intolerant program, our approach has the potential to reuse those transitions in the synthesis of the fault-tolerant version of the input program.

Nevertheless, similar to the above-mentioned methods that generate the synchronization skeleton (i.e., abstract structure) of programs, we also generate the abstract structure of programs. Synthesizing the abstract structure of programs allows us to (i) focus on concurrency issues in the synthesis of fault-tolerant distributed programs instead of their functional properties, and (ii) provide the potential of translating the abstract structure of the synthesized program to multiple programming languages unlike approaches that focus on the synthesis of programs in a specific programming language [58].

Model-theoretic approaches model distribution by atomic read/write actions [4] where in an atomic action a process performs either a read or a write operation. Kulkarni and Arora [1] present a more general way for modeling distribution restrictions where a process is allowed to read/write only a subset of program variables. Since we have adapted Kulkarni and Arora's approach for modeling distribution, our synthesis algorithms benefit from the generality of their modeling.

In addition to the above-mentioned issues, the only implementation of model-theoretic synthesis approaches that we are aware of is an implementation of Emerson and Clarke's method for the synthesis of mutual exclusion protocol [59]. On the other hand, we have implemented an extensible framework (cf. Chapter 8) where developers of fault-tolerance synthesize fault-tolerant distributed programs. Our framework is not problem-dependent and developers of fault-tolerance can use our framework for the synthesis of a variety of programs [60]. Also, due to the incompleteness of the heuristics integrated in our framework, we have chosen to design our framework for change so that if the existing heuristics fail to synthesize a program then developers can integrate their new heuristics in the framework without an expensive overhead.

How does the synthesis method presented in this dissertation differ from automata-theoretic approach where one synthesizes reactive distributed programs [5, 6, 7] that interact with a non-deterministic environment?

The automata-theoretic approach is a specification-based synthesis method where one synthesizes a program from its tree automaton specification. Also, automata-theoretic approaches are mostly used for the synthesis of reactive systems that interact with a non-deterministic environment [5, 61, 6] whereas in the case of our synthesis problem, we have complete information about the behavior of the environment (i.e., faults) with which the program interacts.

Since our approach supports incremental synthesis of multitolerant programs (cf. Chapter 7), it has the potential to incrementally add desired fault-tolerance properties to programs once a new behavior of the environment (i.e., a new class of faults) is discovered. This way, we decompose the problem of synthesizing reactive programs into simpler problems. As a result, we do not encounter the complexity of synthesizing a reactive distributed program [6, 7] that interacts with a hostile environment.

How does the synthesis method presented in this dissertation differ from synthesizing proof-carrying (certified) code?

In the synthesis of proof-carrying code, the synthesis method takes the input specification and generates the code of the program annotated by its proof of correctness [62, 63]. Also, the synthesis method generates a proof checker that is delivered to the program user. Then, using the proof checker, users verify the correctness of the synthesized program to gain high assurance in safety-critical systems. Also, in the synthesis of certified code, there exists an option for adding domain-specific knowledge in order to derive more efficient programs. However, such approaches mostly focus on safety properties of programs whereas our focus is to add all levels of fault-tolerance to programs.

How does the synthesis method presented in this dissertation differ from synthesizing controllers in control theory?

Synthesizing discrete-event controllers in control theory is indeed an automata-theoretic approach. Our approach has several advantages with respect to existing approaches for the synthesis of controllers. First, the general-case complexity of synthesizing controllers is PSPACE-complete [64, 65, 66, 67, 68] in the size of the uncontrolled automaton, whereas our problem is NP-complete. Second, our model of distribution is general enough to capture different modeling cases in distributed computing whereas in Control theory each controller performs its controlling task individually and there exists limitations on the communication between controllers. Finally, our approach is incremental in that we reuse the computations of the fault-intolerant program for the synthesis of its fault-tolerant version. Such reuse of computations is expected to be helpful in the cases where the state space is large.

How does the synthesis method presented in this dissertation differ from synthesizing strategies for two-player games?

Regarding two-player games, most of the approaches in the literature [5, 61, 69, 70] for the synthesis of winning strategies are focused on the cases where the program is interacting with an adversary via input/output variables. Such model restricts us to the cases where faults can only affect a subset of program variables, whereas in our model faults can perturb the state of the program to any state. Although the authors of [71] address this shortcoming of two-player games, the language chosen for expressing the winning strategy is Propositional Linear Temporal Logic (PLTL) [72]. Since fault-tolerance properties are existential properties, PLTL does not have the expressiveness power to capture such properties.

Does the fault model used in this dissertation enable us to capture different types of faults?

Yes. The notion of state perturbation is general enough to model different types of faults (namely, stuck-at, crash, fail-stop, omission, timing, or Byzantine) with different natures (intermittent, transient, and permanent faults). As an illustration of the generality of the notion of state perturbation, we have modeled (i) Byzantine faults (cf. Subsections 4.4.1 and 5.3.1); (ii) fail-stop faults (cf. Subsection 4.4.2); (iii) input-corruption faults (cf. Subsection 5.2.1), and (iv) the process-restart faults that affect the token ring program synthesized in Chapter 6. State-perturbation model has also been used in designing fault-tolerance to (i) omission faults (e.g., [17]), and (ii) transient faults and improper initialization (e.g., [19]).

How does FTSyn scale as the state space of programs increase?

In this dissertation, we showed that using FTSyn, we synthesize fault-tolerant programs that tolerate different types of faults and are simultaneously subject to multiple faults. The largest state space among the programs that we have synthesized belongs to an agreement program (see Appendix B for this program) that is simultaneously perturbed by Byzantine and fail-stop faults (1.3 million states) [73, 60]. Also, in Section 8.5, we synthesized a simplified version of an altitude switch used in the altitude controller of an aircraft. Although the state space of 1.3 million is much smaller than the state space of many practical applications, we argue that our synthesis framework has the potential in adding fault-tolerance to real-world applications. Towards this end, we discuss the following three points:

1. We argue that model checkers were also faced with similar problems with which

our framework faces regarding the state space explosion. Researchers were using early versions of model checkers for checking small protocols and verifying the correctness of operating system kernels [74, 75] despite a state space limit of about 500,000 states on an average workstation (in the early 90s) [74]. The state space handled by our framework is comparable to that reported by early model checkers. We expect that by incorporating the recent optimizations developed for model checking, it will be possible to increase the state space for which fault-tolerance can be added using our framework.

2. We have not currently included these optimizing techniques in the current version of the synthesis framework as the goal of the framework is to study the effectiveness of different heuristics, different internal representation of programs, faults, and the ability to add fault-tolerance to different types of faults. There exist several possible optimizations that can be applied to the framework to reduce the synthesis time. However, these optimizations are orthogonal to the issues at hand. For example, the techniques that are used to determine if a given group of transitions violates safety or if a given group of transitions is appropriate for adding recovery equally affect the above-mentioned goals. (One can either take advantage of the SAT-based approach (presented in Section 9.4) to check the safety of a group of transitions, or exhaustively check every transition of a given group of transitions.) While the design of the framework permits one to use these techniques, these techniques are orthogonal to the issue of adding heuristics that focuses on (i) *which* recovery transitions should be added, and (ii) *how* one should deal with safety-violating transitions. In other words, it is expected that the relative improvement of these optimizations will have the same effect on different heuristics.

10.2 Contributions

The contributions of this dissertation are two-fold: *theoretical* and *practical*. Regarding theoretical contributions, we showed that the problem of synthesizing failsafe fault-tolerant distributed programs from their fault-intolerant version is NP-complete. This result was counterintuitive in the sense that Kulkarni and Arora [1] had already conjectured that adding failsafe fault-tolerance to distributed programs would be polynomial. Subsequently, in Section 4.3, we identified sufficient conditions for polynomial-time synthesis of failsafe fault-tolerant distributed programs. Specifically, we identified monotonic programs and specifications where the addition of failsafe fault-tolerance to distributed programs can be done in polynomial time. We showed that if only programs (respectively, specifications) are monotonic then the synthesis of failsafe fault-tolerant distributed programs will remain NP-complete.

Another theoretical contribution of this dissertation is the enhancement synthesis algorithms presented in Chapter 5. We showed that one approach for reducing the complexity of synthesis is to reuse the computational structure of the fault-intolerant programs in the synthesis of their fault-tolerant version. In particular, we formalized the problem of enhancing the fault-tolerance of nonmasking fault-tolerant programs to masking fault-tolerance. Also, we presented a sound and complete algorithm for enhancing the fault-tolerance of programs in the high atomicity model – where processes can atomically read/write program variables. Then, we designed a sound algorithm for the enhancement of the fault-tolerance of nonmasking distributed programs.

The enhancement technique allows us to partially automate the design of masking fault-tolerant programs and reap the benefits of automation. Specifically, in the synthesis of masking fault-tolerant programs, if automatic synthesis of the fault-tolerant program fails due to the large state space of the fault-intolerant program then one can manually design a nonmasking program and then automatically enhance the level of fault-tolerance to masking using the enhancement algorithms of Chapter 5.

We used monotonicity property to extend the scope of programs and specifications that can reap the benefits of efficient automation. Specifically, we developed heuristics (cf. Sections 9.1 and 9.2) for the transformation of non-monotonic programs (respectively, specification) to monotonic where Theorem 4.11 can be applied for efficient addition of failsafe fault-tolerance to distributed programs. In other words, given a monotonic program (respectively, specification) and a non-monotonic specification (respectively, program), we design heuristics that transform a non-monotonic specification (respectively, program) to a monotonic specification (respectively, program) so that failsafe fault-tolerance can be added in polynomial time. To show the advantage of developing such heuristics, we enhanced the fault-tolerance of a nonmasking distributed program using our heuristics (cf. Section 9.3).

We also presented a synthesis method for automatic addition of pre-synthesized fault-tolerance components to fault-intolerant programs (cf. Chapter 6). Our method enables us to identify commonly encountered *patterns* in the synthesis of fault-tolerant distributed programs, and reuse those patterns in the synthesis of different programs. In other words, to reuse the effort put in the synthesis of one program for the synthesis of another program, we introduced the notion of pre-synthesized fault-tolerance components.

Moreover, we presented algorithms for automatic specification of pre-synthesized components during synthesis where we extract a specified component from a library of pre-synthesized components. Afterwards, in Chapter 6, we presented an algorithm for ensuring the interference-freedom between the program being synthesized and the fault-tolerance components being added to that program. Finally, we designed an algorithm for automatic addition of a pre-synthesized component to a fault-intolerant program. Since the existing algorithms for the synthesis of fault-tolerant distributed programs are not complete (i.e., the algorithms may fail to synthesize a fault-tolerant program from a given fault-intolerant program although there exists a fault-tolerant program), usage of pre-synthesized components allows us to reduce the chance of failure in the synthesis of fault-tolerant distributed programs. Furthermore, we have added pre-synthesized fault-tolerance components with different topologies (e.g., linear and hierarchical) to different programs (cf. Chapter 6). These examples, illustrate the applicability of pre-synthesized fault-tolerance components in the synthesis of a variety of fault-tolerant distributed programs with different topologies.

Using pre-synthesized fault-tolerance components, we also extended the problem of adding fault-tolerance to the case where new variables can be introduced while synthesizing fault-tolerant programs. By contrast, previous algorithms required that the state space of the fault-tolerant program is the same as that of the fault-intolerant program. Moreover, our synthesis method controls the way new variables are introduced; new variables are determined based on the added components. Hence, the synthesis method of Chapter 6 controls the way in which the state space is expanded.

Also, in this dissertation, we investigated the problem of synthesizing multitolerant programs from their fault-intolerant versions (cf. Chapter 7). Specifically, we formally defined what multitolerance means where a multitolerant program provides (i) the specified level of fault-tolerance if a fault from any single class of faults occurs, and (ii) the minimal level of fault-tolerance if faults from multiple classes occur. Then, we showed that, in general, the problem of adding multitolerance to high atomicity programs is NP-complete in the state space of the fault-intolerant program. Subsequently, we presented sound and complete synthesis algorithms for special cases of adding multitolerance where one incrementally adds failsafe (respectively, nonmasking) fault-tolerance to one class of faults and masking fault-tolerance to another fault-class.

Regarding the practical contributions of this dissertation, we developed the synthesis framework FTSyn (presented in Chapter 8) for developers of fault-tolerance where they can synthesize fault-tolerant programs. FTSyn integrates existing algorithms and heuristics for the synthesis of fault-tolerant distributed programs and allows developers to automatically synthesize fault-tolerant programs from their fault-intolerant version. Also, FTSyn is extensible in the sense that developers of heuristics can easily integrate new heuristics into the framework.

Moreover, FTSyn is changeable in the sense that developers can easily change its implementation, without changing the design of FTSyn. The changeability of FTSyn is important since changing the implementation of FTSyn may help to increase the efficiency of the synthesis. Thus, any changes in the implementation should be simple and cheap. Furthermore, we have integrated a SAT-based synthesis approach in FTSyn where we use efficient SAT solvers in the synthesis of fault-tolerant distributed programs (cf. Section 9.4).

10.3 Impact

In this section, we discuss the impacts of this dissertation in research and education. Regarding research, this dissertation has significant impacts on the development of fault-tolerant and dependable distributed programs as the extensible and changeable design of our software framework will help to develop a rich integrated framework of heuristics for the development of fault-tolerant distributed programs.

Moreover, the approach presented in this dissertation for the synthesis of fault-tolerant programs can be extended for the synthesis of reactive programs [5]. Towards this end, we have designed a hybrid synthesis method that benefits from specification-based approaches [76, 2, 77, 56, 78, 79, 80, 57, 4, 5, 6, 71, 81, 7] and the synthesis approach presented in this dissertation. Specifically, we have developed an incremental synthesis method [82] for automatic addition of liveness properties to finite-state concurrent programs. In particular, in [82], we present a sound and complete algorithm for adding Leads-to [30] properties to programs. The incremental approach of [82] has the potential to reuse the efforts put in the synthesis of a program for the synthesis of its improved version.

Furthermore, the synthesis algorithm in [82] can be integrated with model checkers to provide automated assistance beyond generating counterexamples; i.e., in the cases where a model fails to satisfy a property, our synthesis algorithm automatically (i) identifies the fixability of the model, and (ii) fixes the model if it is fixable. Hence, we believe the synthesis method presented in this dissertation has the potential to provide a practical methodology for the synthesis of reactive programs.

Regarding educational impact, we note that using our framework provides the opportunity to experience non-trivial concepts regarding distributed and fault-tolerant systems. We have used the synthesis framework in the graduate distributed system class as well as in a seminar on fault-tolerance.

In the class on distributed systems, the students find that the interactive nature of the framework is extremely useful in understanding several concepts about fault-tolerant programs. In this class, the students focused on re-synthesizing a fault-tolerant program for which the framework had been used successfully. In this case, the students began with the fault-intolerant program. First, they used the automated approach to obtain the fault-tolerant program. Subsequently, they focused on interactive synthesis of the same fault-tolerant program. During this interactive synthesis, they applied different heuristics and observed the intermediate program. They explored the state transition diagram of the intermediate program and used the framework to understand why the intermediate program was not fault-tolerant. This allowed them to experience the non-deterministic execution of different processes of the program. Moreover, they could observe individual states and transitions in the global state transition diagram and could experience the effect of distribution restrictions on the complexity of the synthesis of fault-tolerant distributed programs.

10.4 Future Work

In this section, we present open theoretical problems in the synthesis of fault-tolerant distributed programs. Also, we discuss future extensions and modification to the FT-Syn framework presented in Chapter 8. First, we discuss open theoretical problems:

- *Identify the polynomial boundary of synthesizing nonmasking fault-tolerant distributed programs.*

  As we identified sufficient conditions for the synthesis of failsafe fault-tolerant distributed programs in Chapter 4, we would like to at least identify the sufficient conditions for polynomial synthesis of nonmasking fault-tolerant programs. Although we do not have a proof for the NP-hardness of the problem of synthesizing nonmasking fault-tolerant distributed programs from their fault-intolerant version, we already know that this problem is in NP (cf. Section 2.8). To the best of our knowledge, no polynomial-time algorithm has yet been presented for the synthesis of nonmasking distributed programs. Thus, finding properties of programs that identify sufficient conditions for polynomial-time synthesis of nonmasking distributed programs remains an open problem.

- *Develop nonmasking programs that satisfy the monotonicity requirements.*

  Since the worst-case complexity of enhancing the fault-tolerance of nonmasking fault-tolerant distributed programs to masking is exponential (cf. Chapter 5), we would like to use the notion of monotonicity in order to identify nonmasking programs whose level of fault-tolerance can be enhanced to masking in polynomial time. Thus, it is desirable to develop a methodology for the design of nonmasking programs that satisfy the requirements of program monotonicity. Such design methodology provides a framework for partial automation in the design of masking programs where one *manually* develops a nonmasking monotonic program and then applies Theorem 4.11 to automatically enhance the level of fault-tolerance to masking.

- *Identify the necessary and sufficient conditions for simultaneous addition of multiple pre-synthesized components.*

  In Chapter 6, we showed how we add a pre-synthesized corrector to the program being synthesized in order to resolve a deadlock state from which existing heuristics fail to add recovery. Also, we ensured that the execution of the pre-synthesized component does not interfere with the execution of the program. Now, since there exist many situations where we need to simultaneously add such correctors to the program being synthesized, we plan to identify necessary and sufficient conditions for an interference-free addition of multiple pre-synthesized components to a program.

- *Develop a platform for providing automated assistance in model checking beyond generating counterexamples.*

  Although model checkers provide user-friendly counterexamples in cases where a model fails to satisfy a desired property, it is difficult to *manually* fix a failed model so that it satisfies a desired property while preserving its existing properties. We have developed a synthesis algorithm [82] that has the potential to provide such automated assistance for developers when the model checking of the program at hand fails. Using the synthesis algorithm of [82], we automatically (i) identify whether or not a model is *fixable* to satisfy a particular property in addition to its existing properties, and (ii) fix the model if it is fixable so that it satisfies a new property in addition to its existing properties. However, currently, the synthesis algorithm in [82] can only be used for linear computation model where program properties are specified in Linear Temporal Logic [72]. To develop a platform for *automatic model correction*, it is desirable to (i) integrate the algorithm [82] in one of the existing model checkers (e.g., SPIN [36]) to investigate the practicality of the algorithm of [82], and (ii) extend the results of [82] for the case where the program computation is non-linear (e.g., tree-like computation) and program properties are specified in Computation Tree Logic (CTL) [72].

Now, we discuss issues related to the extensions and improvements of the synthesis framework FTSyn presented in this dissertation.

- *Use model checkers in the synthesis of fault-tolerant programs in order to reduce the complexity of synthesis.*

  As mentioned in Chapter 8, the FTSyn framework has the ability to interact with developers of fault-tolerance. If the synthesis of a fault-tolerant program fails then developers can ask FTSyn to generate an intermediate version of the program being synthesized in order to identify what went wrong during synthesis. FTSyn generates the intermediate program in Promela [37] modeling language. Thus, developers can benefit from the SPIN model checker and verify the fault-tolerance properties. The SPIN model checker returns counterexamples that are enlightening for developers in that they can identify what heuristic should be applied next in synthesis. Currently, the users of FTSyn should perform this verification manually. We plan to develop an automated approach for the communication between FTSyn and model checkers. Such communication has an important impact on reducing the complexity of synthesis as model checkers can provide behavioral information about the program at hand. The synthesis algorithm uses this behavioral information to make more intelligent decisions during synthesis.

- *Develop a distributed synthesis platform.*

  Currently, the implementation of FTSyn is centralized. To extend the scope of synthesis for real-world applications, we adopt two directions: developing a scalable parallel synthesis algorithm, and extending FTSyn for deployment on a distributed platform. In the first direction, we plan to conduct a survey on the existing approaches [83, 84, 85, 86, 87] for parallel and distributed model checking, where one distributes the reachability graph of the model at hand on a network. Towards this end, we note that the synthesis problem differs from model checking problem in that during synthesis we modify the program model to satisfy specific synthesis requirements, whereas model checkers only verify the program model without performing any modification. We conjecture that the scalable synthesis will be in a higher complexity class than the scalable model checking, thus making the development of a scalable synthesis algorithm more challenging. In the second direction, we plan to simultaneously implement the achievements in the design of the scalable synthesis algorithm in FTSyn. As a result, we can experience the applicability of our theoretical results in the context of distributed FTSyn.

- *Develop an on-the-fly synthesis method.*

  In the synthesis of a fault-tolerant program, FTSyn initially expands the reachability graph of the fault-intolerant program using program and fault transitions. For real-world applications, the size of the reachability graph is very large, and as a result of the space complexity of synthesis, FTSyn may fail to synthesize a fault-tolerant program. To remedy this problem, we plan to develop a space-efficient synthesis algorithm where FTSyn partially generates the reachability graph of the program. Towards this end, we benefit from existing techniques [88] in the model checking literature for providing space efficiency. Such space-efficient techniques are orthogonal to the development of a distributed synthesis algorithm in that we deploy the space-efficient synthesis algorithm on each node of the scalable synthesis platform discussed above.

Appendices

Appendix A: Programs Synthesized Using Pre-Synthesized Components

In this appendix, we present the programs that we have synthesized using pre-synthesized components. Specifically, we first present an Alternating Bit Protocol (in Section A.1) that is nonmasking fault-tolerant to message loss faults. Then, in Section A.2, we present an intermediate diffusing computation program synthesized by our synthesis framework, FTSyn. Subsequently, in Section A.3, we present the synthesized diffusing computation program after we have added pre-synthesized components to refine one of the high atomicity recovery actions in the intermediate program. Finally, in Section A.4, we present a refined version of the synthesized diffusing computation program in the syntax of the Promela modeling language [37] where we have verified the synthesized program in the SPIN model checker to gain more confidence in the implementation of FTSyn.

A.1 The Promela Model of the Alternating Bit Protocol

In this section, we present the Promela model of the alternating bit protocol (ABP) synthesized by adding linear pre-synthesized components to the fault-intolerant ABP program presented in Section 6.5.

```

#define inv
( (((rr != 1) && (cr == -1))  || (br == bs)) &&
  (((rs != 1) && (cs == -1))  || (br != bs)) &&
  ((cs == -1) || (cs == bs)) &&
  ((cs != -1) || (cr != -1) || ((rr + rs) == 1)) &&
  ((cs == -1) || (cr != -1) || ((rr + rs) == 0)) &&
  ((cs != -1) || (cr == -1) || ((rr + rs) == 0)))

#define fs ((cs == -1) || (cs == bs)) &&
           (((cs != -1) && (cr != -1)) ||
           (((rr + rs) == 1) || ((rr + rs) == 0)))

/* The property to be verified   [](fs -> <> inv)   */

#define Zs (rs == 0) && (bs ==1) && (cs == -1)   /* LCs */
#define Zr (rr == 0) && (br ==1) && (cr == -1)   /* LCr */
#define ZPs (rs == 0) && (bs ==0) && (cs == -1)  /* LC’s */
#define ZPr (rr == 0) && (br ==0) && (cr == -1)  /* LC’r */

#define Xs Zs && ZPr
#define XPs ZPs && Zr
#define Xr Zs && Zr
#define XPr ZPs && ZPr

```

```
bool rs = 1;
bool rr = 0;
bool bs = 1;
bool br = 0;

bool ypr;   /*  y'r  */
bool ys;
bool yr;
bool yps;        /*  y's  */

bool us;
bool ur;
bool ups;        /*  u's  */
bool upr;        /*  u'r  */

int cs = -1;
int cr = -1;

proctype  sender() {
do
:: atomic { ((rs == 1)) -> rs = 0;  cs = bs ;
                              us =0; ups =0; }
:: atomic { (cr != -1) ->  rs = 1; cr = -1;
             bs = (bs+1)%2 ;  us =0; ups =0; }

::  atomic { Zs && !ys && ypr -> ys = 1; }
::  atomic { ys -> cs = 1; ys=0; }

:: atomic {  ZPs && !yps && yr -> yps = 1; }
:: atomic {  yps -> cs = 0; yps=0; }

::  atomic { Zs && !us -> us = 1; }
::  atomic { ZPs && !ups -> ups = 1; }
od;
}

proctype receiver() {
do
:: atomic {  ( cs != -1) -> cs = -1;  rr = 1;
              br = (br+1)%2  ;  yr =0;  ypr =0; }
:: atomic {  ( rr == 1) -> rr = 0; cr = br ;
                                yr =0;  ypr =0; }

:: atomic {  ZPr && !ypr -> ypr = 1; }
:: atomic {  Zr && !yr -> yr = 1; }

:: atomic { Zr && !ur && us -> ur = 1; }
:: atomic {  ur -> cr = 1;  ur=0; }

:: atomic { ZPr && !upr && ups -> upr = 1; }
:: atomic { upr -> cr = 0; upr=0; }
od;
}

proctype MessageLossFaults() {
if
:: ((cs != -1)) -> cs = -1;
:: ((cr != -1)) -> cr = -1;
:: skip;
fi;
}

init{
run sender(); run receiver();  run MessageLossFaults();
}
```

A.2 The Synthesized Intermediate Diffusing Computation Program

In this section, we present the intermediate program that we have synthesized using FTSyn. This program includes the actions of the high atomicity processes added for the purpose of adding recovery. FTSyn represents the synthesized program in a syntax close to the syntax of the Promela modeling language [37]. The semantic of the output program is based on the Dijkstra's guarded commands, where each guarded command $grd \rightarrow st$ represents a set of transitions $\{(s_0, s_1) :\ grd$ holds at s_0 and the atomic execution of st at s_0 takes the state of the program to $s_1\}$. In the following program, ci, pi, and sni respectively represent the color, the parent, and the session number of process P_i. Also, cpi and $snpi$ respectively represent the color and the session number of the parent of P_i $(0 \leq i \leq 3)$.

```
----------  The actions of Process P0 ----------
(c0 == 1) &&
((p0 == 0) && (sn0 == 1))   -> c0 := 0; sn0 := 0;

(c0 == 1) &&
((p0 == 0) && (sn0 == 0))   -> c0 := 0; sn0 := 1;

(c0 == 1) &&
( ((c1 == 0) && (c2 == 0) && (sn0 == 1) && (sn1 == 0) &&
               (sn2 == 0) && ((p0 == 1) || (p0 == 2)) )  ||
  ((c2 == 0) && (sn0 == 1) && (sn2 == 0) && (p0 == 2) )  ||
  ((c1 == 0) && (sn0 == 0) && (sn1 == 1) && (p0 == 1) )  ||
  ((c2 == 0) && (sn0 == 0) && (sn2 == 1) && (p0 == 2) )  ||
  ((c1 == 0) && (sn0 == 1) && (sn1 == 0) && (p0 == 1) )  ||
  ((c1 == 0) && (c2 == 0) && (sn0 == 0) && (sn1 == 1) &&
  (sn2 == 1) && ((p0 == 1) || (p0 == 2)))  )
                                   -> c0 := cp0; sn0 := snp0;
```

```

(c0 == 0) &&
( ((c1 == 1) && (c2 == 1) && (sn0 == 0) && (sn1 == 0) && (sn2 == 0))  ||
  ((c1 == 1) && (c2 == 1) && (sn0 == 1) && (sn1 == 1) && (sn2 == 1)) )
                                                            -> c0 := 1;

----------  The actions of Process P1 ----------

(c1 == 1) &&
( ((cp1 == 0) && (sn1 == 0) && (snp1 == 1))  ||
  ((cp1 == 0) && (sn1 == 1) && (snp1 == 0))  )
                      -> c1 := cp1; sn1 := snp1;

(c1 == 0) && ((sn1 == 1) || (sn1 == 0)) -> c1 := 1;

----------  The actions of Process P2 ----------

(c2 == 1) &&
( ((cp2 == 0) && (sn2 == 0) && (snp2 == 1))  ||
  ((cp2 == 0) && (sn2 == 1) && (snp2 == 0))  )
                          -> c2 := cp2; sn2 := snp2;

(c2 == 0) &&
( ((sn2 == 0) && (c3 == 1) && (sn3 == 0) && (p3 == 2))  ||
  ((sn2 == 1) && (c3 == 1) && (sn3 == 1) && (p3 == 2)) )
                                                  -> c2 := 1;

 ----------  The actions of Process P3 ----------

(c3 == 1) &&
( ((cp3 == 0) && (sn3 == 0) && (snp3 == 1))  ||
  ((cp3 == 0) && (sn3 == 1) && (snp3 == 0)) )
                       -> c3 := cp3; sn3 := snp3;
```

```

(c3 == 0) &&
  ((sn3 == 0) || (sn3 == 1))   -> c3 := 1;


----------  The actions of the high atomicity Process 0 ----------

(c0 == 1) &&
( ((c1 == 1) && (c2 == 1) && (c3 == 1) && (sn0 == 1) && (sn3 == 0) &&
    ((p0 == 2) || (p0 == 1)) && (p1 == 0) && (p2 == 0) && (p3 == 2) )  ||
  ((c1 == 1) && (c2 == 1) && (c3 == 1) && (sn0 == 1) && (sn1 == 0) &&
    ((p0 == 2) || (p0 == 1)) && (p1 == 0) && (p2 == 0) && (p3 == 2) )  ||
  ((c1 == 1) && (c2 == 1) && (c3 == 1) && (sn0 == 1) && (sn2 == 0) &&
    ((p0 == 2) || (p0 == 1))  && (p1 == 0) && (p2 == 0) && (p3 == 2))  )
                                                            -> sn.0 := 0;

(c0 == 1) &&
( ((c1 == 1) && (c2 == 1) && (c3 == 1) && (sn0 == 0) && (sn1 == 0) &&
   (sn2 == 0) && (sn3 == 0) && ((p0 == 2) || (p0 == 1))  && (p1 == 0) &&
                                               (p2 == 0) && (p3 == 2) )  ||
  ((c1 == 1) && (c2 == 1) && (c3 == 1) && (sn0 == 1) && (sn1 == 1) &&
   (sn2 == 1) && (sn3 == 1) && ((p0 == 1) || (p0 == 2))  && (p1 == 0) &&
                                   (p2 == 0) && (p3 == 2))  )   -> p0 := 0;

(c0 == 1) &&
( ((c1 == 1) && (c2 == 1) && (c3 == 1) && (sn0 == 0) && (sn1 == 0) &&
   (sn2 == 1) && (p0 == 2) && (p1 == 0) && (p2 == 0) && (p3 == 2))  ||
  ((c1 == 1) && (c2 == 1) && (c3 == 1) && (sn0 == 0) && (sn1 == 1) &&
   (sn2 == 0) && (p0 == 2) && (p1 == 0) && (p2 == 0) && (p3 == 2))  ||
  ((c1 == 1) && (c2 == 1) && (c3 == 1) && (sn0 == 0) && (sn2 == 0) &&
   (sn3 == 1) && (p0 == 2) && (p1 == 0) && (p2 == 0) && (p3 == 2))  ||
  ((c1 == 1) && (c2 == 1) && (c3 == 1) && (sn0 == 0) && (sn1 == 1) &&
   (sn3 == 0) && (p0 == 2) && (p1 == 0) && (p2 == 0) && (p3 == 2))  ||
  ((c1 == 1) && (c2 == 1) && (c3 == 1) && (sn0 == 0) && (sn2 == 1) &&
   (sn3 == 0) && (p0 == 2) && (p1 == 0) && (p2 == 0) && (p3 == 2))  ||
  ((c1 == 1) && (c2 == 1) && (c3 == 1) && (sn0 == 0) && (sn1 == 0) &&
   (sn3 == 1) && (p0 == 2) && (p1 == 0) && (p2 == 0) && (p3 == 2))  )
                                                       -> p0 := 1;

(c0 == 1) &&
((c1 == 1) && (c2 == 1) && (c3 == 1) && (sn0 == 0) && (sn1 == 1) &&
 (sn2 == 1) && (sn3 == 1) && ((p0 == 2) || (p0 == 1)) && (p1 == 0) &&
   (p2 == 0) && (p3 == 2)) )  -> c0 := 1; sn0 :=1; p0 := 0;

----------  The actions of the high atomicity Process 1 ----------

(c0 == 1) &&
( (c1 == 1) && (c2 == 1) && (c3 == 1) && (sn0 == 0) && (sn1 == 1) &&
  (sn2 == 0) && (sn3 == 0) && (p0 == 1) && (p1 == 0) && (p2 == 0) &&
                                          (p3 == 2) )   -> sn1 := 0;

(c0 == 1) &&
( ((c1 == 1) && (c2 == 1) && (c3 == 1) && (sn0 == 0) && (sn1 == 0) &&
   (sn2 == 1) && (p0 == 1) && (p1 == 0) && (p2 == 0) && (p3 == 2))  ||
  ((c1 == 1) && (c2 == 1) && (c3 == 1) && (sn0 == 0) && (sn1 == 0) &&
   (sn3 == 1) && (p0 == 1) && (p1 == 0) && (p2 == 0) && (p3 == 2))  )
                                                       -> sn1 := 1;


----------  The actions of the high atomicity Process 2 ----------

 (c0 == 1) &&
((c1 == 1) && (c2 == 1) && (c3 == 1) && (sn0 == 0) && (sn1 == 1) &&
 (sn2 == 0) && (sn3 == 1) && (p0 == 1) && (p1 == 0) && (p2 == 0) &&
                                        (p3 == 2)) )   -> sn2 := 1;

```

```
---------  The actions of the high atomicity Process 3 ----------

(c0 == 1) &&
((c1 == 1) && (c2 == 1) && (c3 == 1) && (sn0 == 0) && (sn1 == 1) &&
 (sn2 == 1) && (sn3 == 0) && (p0 == 1) && (p1 == 0) && (p2 == 0) &&
                                              (p3 == 2))  -> sn3 := 1;
```

A.3 The Actions of the Synthesized Diffusing Computation Program

In this section, we present the actions of processes P_2 and P_3 in the DC program (from Section 6.6.2). These actions construct the actions of the synthesized program. We presented the actions of P_0 in Section 6.6.2.

$$\begin{array}{lll}
DC_{31}: \ (c_3 = 1) \ \wedge \ (par_3 = 3) & \longrightarrow & c_3 := 0;\ sn_3 = \neg sn_3; \\
 & & y_3 := false;\ y_2 := false; \\
 & & \text{if } ((sn_3 = 1) \wedge (y'_3 = true)) \\
 & & \text{then } y'_3 := false;\ y'_2 := false
\end{array}$$

$$\begin{array}{lll}
DC_{32}: \ (c_3 = 1) \ \wedge \ (c_{par_3} = 0) \ \wedge \ (sn_3 \not\equiv sn_{par_3}) & \longrightarrow & c_3 := c_{par_3};\ sn_3 = sn_{par_3}; \\
 & & \text{if } ((c_3 = 0) \wedge (y_3 = true)) \\
 & & \text{then } y_3 := false;\ y_2 := false; \\
 & & \text{if } ((sn_3 = 1) \wedge (y'_3 = true)) \\
 & & \text{then } y'_3 := false;\ y'_2 := false;
\end{array}$$

$$\begin{array}{lll}
DC_{33}: \ (c_3 = 0) \ \wedge \ (\forall k :: p_k = 3 \Rightarrow (c_k = 1 \wedge sn_3 \equiv sn_k)) & \longrightarrow & c_3 := 1; \\
D_{31}: \ (c_3 = 1) \wedge (c_2 = 1) \wedge (y_3 = false) & \longrightarrow & y_3 := true; \\
D'_{31}: \ (sn_3 = 0) \wedge (c_2 = 1) \wedge (y'_3 = false) & \longrightarrow & y'_3 := true;
\end{array}$$

Note that, in action DC_{31}, our synthesis method has added new statements to the statements of the first action in the fault-intolerant DC program. These new statements falsify the witness predicates of the detectors. For example, when c_3 becomes 0 the state predicate LC_3 no longer holds. Thus, the witness predicate y_3 must be falsified to ensure the interference-freedom of the program and the pres-synthesized detectors. Now, we present the actions of process P_2 composed with the detectors d_2 and d'_2.

```
$DC_{21}: (c_2 = 1) \wedge (par_2 = 2)$
        $\longrightarrow c_2 := 0; sn_2 = \neg sn_2;$
            $y_2 := 0; y_0 := 0;$
            if $((y'_3 = false) \wedge (sn_2 = 1)$
            $\wedge((y'_2 = true) \vee (y'_0 = true)))$
            then $y'_2 := false; y'_0 := false;$
$DC_{22}: (c_2 = 1) \wedge (c_{par_2} = 0) \wedge (sn_2 \not\equiv sn_{par_2})$
        $\longrightarrow c_2 := c_{par_2}; sn_2 = sn_{par_2};$
            if $((c_2 = 1) \vee (y_3 = false))$
            $\wedge((y_2 = true) \vee (y_0 = true)))$
            then $y_2 := false; y_0 := false;$
            if $((sn_2 = 1) \vee (y'_3 = false))$
            $\wedge((y'_2 = true) \vee (y'_0 = true)))$
            then $y'_2 := false; y'_0 := false;$

$DC_{23}: (c_2 = 0) \wedge (\forall k :: p_k = 2 \Rightarrow (c_k = 1 \wedge sn_2 \equiv sn_k))$
        $\longrightarrow c_2 := 1;$
            if $(y_3 = false)) \wedge$
              $((y_2 = true) \vee (y_0 = true)))$
            then $y_2 := false; y_0 := false;$
            if $(y'_3 = false)) \wedge$
              $((y'_2 = true) \vee (y'_0 = true)))$
            then $y'_2 := false; y'_0 := false;$
$D_{21}: (y_3 = true) \wedge (c_2 = 1) \wedge (sn_0 = 1) \wedge (c_0 = 1) \wedge((par_0 = 2) \vee (par_0 = 1)) \wedge (y_2 = false)$
        $\longrightarrow y_2 := true;$
$D'_{21}: (y'_3 = true) \wedge (c_2 = 1) \wedge (sn_0 = 1) \wedge (c_0 = 1) \wedge((par_0 = 2) \vee (par_0 = 1)) \wedge (y'_2 = false)$
        $\longrightarrow y'_2 := true;$
```

A.4 The Promela Model of the Synthesized Diffusing Computation Program

In this section, we present the Promela model of the synthesized diffusing computation program where we verify the nonmasking fault-tolerance property of the synthesized program. Although the synthesized program is correct by construction, we have conducted this formal verification in order to gain more confidence in the implementation of FTSyn.

```
#define inv
 (((
   (((c[0] == c[p0]) && (c[4] == c[p0+4])) || ((c[0] ==1) && (c[p0] == 0)))&&
   (((c[1] == c[p1]) && (c[5] == c[p1+4])) || ((c[1] ==1) && (c[p1] == 0)))&&
   (((c[2] == c[p2]) && (c[6] == c[p2+4])) || ((c[2] ==1) && (c[p2] == 0)))&&
   (((c[3] == c[p3]) && (c[7] == c[p3+4])) || ((c[3] ==1) && (c[p3] == 0)))
)) &&
      ((p0 ==0) && (p1 == 0) && (p2 == 0) && (p3 == 2)) )

#define safety0 (!Z0 || X0)
#define safety0p (!Z0p || X0p)

#define safety2 (!Z2 || X2)
#define safety2p (!Z2p || X2p)

#define safety3 (!Z3 || X3)
#define safety3p (!Z3p || X3p)

#define X0 (c[3] == 1) && (c[1] == 1) && (c[2] == 1) && (c[0] == 1) &&
                   ( c[4] == 1 ) && ((p0 == 2 ) || (p0 == 1))
#define Z0 (y0 == 1)

#define X0p (c[7] == 0) && (c[1] == 1)  && (c[2] == 1) && (c[0] == 1) &&
                     ( c[4] == 1 ) && ((p0 == 2 ) || (p0 == 1))
#define Z0p (y0p == 1)


#define X2 (c[3] == 1) && (c[2] == 1) && (c[4] == 1) && (c[0] == 1)  &&
                   ((p0 == 2 ) || (p0 == 1))
#define Z2 (y2 ==1)

#define X2p (c[7] == 0) && (c[2] == 1) && (c[4] == 1) && (c[0] == 1)  &&
                    ((p0 == 2 ) || (p0 == 1))
#define Z2p (y2p == 1)

#define X3 (c[3] == 1) && (c[2] == 1)
#define Z3 (y3 ==1)

#define X3p (c[7] == 0) && (c[2] == 1)
#define Z3p (y3p ==1)

/*  Properties to be verified
 [] safety
 [] (!inv -> <> inv)
 [] (<> inv)
*/

bool c[8];
bool y3 =0, y2=0, y3p=0, y2p=0, y0 =0, y0p =0;

/* The cells of this array respectively represent
             c0, c1, c2, c3, sn0, sn1, sn2, sn3
                // c0  ---> c[0]
                // c1  ---> c[1]
                // c2  ---> c[2]
                // c3  ---> c[3]
                // sn0 ---> c[4]
                // sn1 ---> c[5]
                // sn2 ---> c[6]
                // sn3 ---> c[7]
*/

int p0 = 0;
int p1 = 0;
int p2 = 0;
int p3 = 2;

proctype  P0() {
do
:: atomic{ ((c[0] ==1) && (p0 == 0) ) ->  c[0] = 0; c[4] = !c[4];
                 y0 = 0; y0p =0;  }

:: atomic{ ((c[0] == 1) && (c[p0] == 0) && (c[4] != c[p0+4])) ->
  { c[0] = c[p0]; c[4] = c[p0+4];
        if ::  (c[0] == 0) && (y0 ==1)  -> y0 = 0; y0p =0;
        :: else skip;
        fi;       }
}

:: atomic{ ((c[0] == 0) && ((p1 != 0) || ((c[1] == 1) &&
       (c[4] == c[5] ))) && ((p2 != 0) || ((c[2] == 1) &&
                                (c[4] == c[6])) ) )  -> { c[0] = 1;
                             if ::  (y2 == 0) && (y0 ==1) ->  y0 =0;
                                :: else skip;
                             fi;
                             if ::  (y2p == 0) && (y0p ==1)->  y0p =0;
                                :: else skip;
                             fi;
                              }
```

```
}
/* component-based actions of P0 */

 :: atomic { ( ( y0 == 1 )   &&
( ( y0p == 1 ) ||( c[5] == 0 ) ||( c[6] == 0 ))  )  -> c[4] = 0;
                                y0 =0;  y0p = 0; y2 =0;  y2p = 0; }

:: atomic { (y2 == 1) && ( c[1] == 1 )  && (c[2] == 1)  &&
                             (c[0] == 1) && ( c[4] == 1 ) &&
               ((p0 == 2 ) || (p0 == 1)) && (y0 == 0) -> y0 = 1; }

:: atomic { (y2p == 1) && ( c[1] == 1 )  && (c[2] == 1) &&
                             (c[0] == 1) && ( c[4] == 1 ) &&
             ((p0 == 2 ) || (p0 == 1)) && (y0p == 0) -> y0p = 1; }
od;
}

proctype P1() {
do
:: atomic { ((c[1] ==1)  && (p1 == 1) ) -> c[1] = 0; c[5] = !c[5]; }
:: atomic { ((c[1] == 1) && (c[p1] == 0) &&  (c[5] != c[p1+4]) )
                                    -> c[1] = c[p1]; c[5] = c[p1+4]; }
:: atomic { (c[1] == 0)   -> c[1] = 1;  }
od;
}

proctype P2() {
do
:: atomic{ ((c[2] ==1) && (p2 == 2) ) -> { c[2] = 0; c[6]= !c[6];
                                       y2 =0;  y0 =0; y3 =0; y3p =0;
      if ::  ((y3p == 0) && (c[6] == 1)) && ((y2p ==1)|| (y0p ==1))
                                        -> y2p =0;  y0p =0;  y3p =0;
         :: else skip;
      fi;
      }
}

:: atomic { ((c[2] == 1) && (c[p2] == 0) &&  (c[6] != c[p2+4]))
                                -> { c[2] = c[p2]; c[6] = c[p2+4];
                 if ::  ((c[2] == 0) || (y3 == 0))  &&
                                 ((y2 ==1) || (y0 ==1) || (y3 ==1))
                                            -> y2 =0;  y0 =0; y3 =0;
                    :: else skip;
                 fi;
                 if ::  ((y3p == 0) || (c[7] == 1) || (c[3] == 0) ||
                 (c[2] == 0)) && ((y2p ==1)|| (y0p ==1)|| (y3p ==1))
                                         -> y2p =0;  y0p =0;  y3p =0;
                    :: else skip;
                 fi;
                 }
}

:: atomic { ((c[2] == 0) && ((p3 != 2) ||
                 ((c[3] == 1) && (c[7] == c[6]))))  -> { c[2] = 1;
   if :: (y3 == 0) && ((y2 ==1)||(y0 ==1)) -> y2 =0;  y0 =0; y3 =0;
      :: else skip;
   fi;
   if :: (y3p == 0)&&((y2p ==1)||(y0p ==1))-> y2p =0; y0p =0; y3p =0;
      :: else skip;
   fi;
   }
}

:: atomic { (y3 == 1) && (c[2] == 1) && (c[4] == 1) &&
              (c[0] == 1)  && ((p0 == 2 ) || (p0 == 1)) &&
                                               (y2 == 0) -> y2 = 1; }
```

```

:: atomic { (y3p == 1) && (c[2] == 1) && (c[4] == 1) &&
             (c[0] == 1)  && ((p0 == 2 ) || (p0 == 1)) &&
                                             (y2p == 0) -> y2p = 1; }
od;
}

proctype P3() {
do
:: atomic { ((c[3] ==1) && (p3 == 3) ) -> { c[3] = 0; c[7] = !c[7];
y3 = 0; y2 = 0;

if :: ((c[7] == 1) || (c[2] ==0)) && (y3p ==1) ->  y3p =0; y2p =0;
:: else skip;
fi;
  }
}



:: atomic { ((c[3] == 1) && (c[p3] == 0) &&  (c[7] != c[p3+4]))
                                -> { c[3] = c[p3]; c[7] = c[p3+4];
 if :: ((c[3] == 0) || (c[2] ==0)) && (y3 ==1)  -> y3 = 0; y2 =0;
    :: else skip;
 fi;
 if :: ((c[7] == 1)  || (c[2] ==0)) && (y3p ==1)-> y3p =0; y2p =0;
 :: else skip;
 fi;
 }
}

:: atomic { (c[3] == 0)  -> { c[3] = 1;
 if :: ((c[7] == 1)  || (c[2] ==0)) && (y3p ==1)-> y3p =0; y2p =0;
```

```
   :: else skip;
 fi;
 }
}

:: atomic { (c[3] == 1) && (c[2] == 1) &&
             ((c[6] ==0) || (c[7] ==0)) && (y3 == 0) -> y3 = 1; }

:: atomic { (c[7] == 0) && (c[2] == 1) && (y3p == 0)-> y3p = 1; }
od;
}




proctype Pseudo0() {
do
/*   This high atomicity recovery action has been refined by adding
             the pre-synthesized components. Thus, we comment it out.
:: atomic {  ( c[0] == 1) &&
( ( ( c[1] == 1 ) && (  c[2] == 1 ) && (  c[3] == 1 ) && ( c[4] == 1 ) &&
( ( p0 == 2 ) || ( p0 == 1 )  )  )  &&
 (( c[7] == 0 ) || ( c[5] == 0 ) || ( c[6] == 0 ))  ) -> c[4] = 0; }
*/
:: atomic{ ((c[0] == 1) &&  (c[1] == 1) && (c[2] == 1) &&
             (c[3] == 1) && ((p0 == 2) || (p0 == 1))  ) &&
(
((c[4] == 0) && (c[5] == 0) && (c[6] == 0) && (c[7] == 0)) ||
((c[4] == 1) && (c[5] == 1) && (c[6] == 1) && (c[7] == 1))
)
 -> p0 = 0;  }

:: atomic {
```

```
(c[0] == 1) && (c[1] == 1 ) && (c[2] == 1) && (c[3] == 1) &&
(c[4] == 0) && (p0 == 2) &&
(
((c[4] == 0) && (c[5] == 0) && (c[6] == 1)) ||
((c[4] == 0) && (c[5] == 1) && (c[6] == 0)) ||
((c[4] == 0) && (c[5] == 0) && (c[7] == 1)) ||
((c[4] == 0) && (c[5] == 1) && (c[7] == 0)) ||
((c[4] == 0) && (c[6] == 1) && (c[7] == 0)) ||
((c[4] == 0) && (c[5] == 0) && (c[7] == 1))
)  -> p0 = 1; }

:: atomic {
(c[0] == 1) &&
      ((c[1] == 1) && (c[2] == 1) && (c[3] == 1) &&
        (c[4] == 0)&& (c[5] == 1) && (c[6] == 1) &&
          (c[7] == 1) && ((p0 == 2) || (p0 == 1)) )
                      -> c[0] =1; c[4] = 1; p0 = 0;
 }
od;
}

proctype Pseudo1() {
do
:: atomic {
(c[0] == 1) &&
       ((c[1] == 1) && (c[2] == 1) && (c[3] == 1) &&
        (c[4] == 0) && (c[5] == 1) && (c[6] == 0) &&
            (c[7] == 0) && (p0 == 1) && (p1 == 0) &&
                 (p2 == 0) && (p3 == 2))  -> c[5] =0;
 }

:: atomic{(c[0] == 1) && (c[1] == 1) && (c[2] == 1) &&
            (c[3] == 1)&& (c[4] == 0) && (c[5] == 0) &&
              (p0 == 1) && ((c[6] == 1) || (c[7] == 1))
                                          -> c[5] = 1; }
od;
}

proctype Pseudo2() {
do
:: atomic {(c[0] == 1) && (c[1] == 1) && (c[2] == 1) &&
             (c[3] == 1) && (c[4] == 0) && (c[5] == 1) &&
                   (c[6] == 0) && (c[7] == 1) && (p0 == 1)
                                            -> { c[6] = 1;
}
}
 od;
}

proctype Pseudo3() {
do
:: atomic { (c[0] == 1) && (c[1] == 1) && (c[2] == 1) &&
              (c[3] == 1) && (c[4] == 0) && (c[5] == 1) &&
                (c[6] == 1) && (c[7] == 0) && (p0 == 1) &&
                    (p1 == 0) && (p2 == 0) && (p3 == 2)
                                               ->  c[7] = 1;
 }
od;
}

proctype Faults() {
if
:: atomic { (true) -> c[0] = 0;  }
:: atomic { (true) -> c[0] = 1;  }
:: atomic { (true) -> c[1] = 0; }
:: atomic { (true) -> c[1] = 1;  }

:: atomic { (true) -> c[2] = 0;  }
:: atomic { (true) -> c[2] = 1;  }
:: atomic { (true) -> c[3] = 0; }
:: atomic { (true) -> c[3] = 1;  }

:: atomic { (true) -> c[4] = 0;  }
:: atomic { (true) -> c[4] = 1;  }
:: atomic { (true) -> c[5] = 0; }
:: atomic { (true) -> c[5] = 1;  }

:: atomic { (true) -> c[6] = 0;  }
:: atomic { (true) -> c[6] = 1;  }
:: atomic { (true) -> c[7] = 0; }
:: atomic { (true) -> c[7] = 1;  }

:: atomic{ (true) ->  p0 = 0;   }
:: atomic{ (true) ->  p0 = 1;  }
:: atomic{ (true) ->  p0 = 2; }
fi;

}

init{
run Faults();
run P0(); run P1(); run P2(); run P3();
run Pseudo0();  run Pseudo1();
run Pseudo2();
run Pseudo3();
}
```

Appendix B: Agreement in the Presence of Byzantine and Failstop Faults

In this section, we present a comprehensive example of adding fault-tolerance to a fault-intolerant program using our framework. Specifically, we show how developers of fault-tolerance can interact with our framework, FTSyn, in order to add masking fault-tolerance to an agreement program. This example may be thought of as a brief version of the user manual for our framework. A more detailed user manual including the source code is available at [73].

The fault-intolerant program consists of a general process and four non-general processes that are perturbed by Byzantine and fail-stop faults. The user should specify the input fault-intolerant program, its variables, its invariant, its specification, and the faults in a text file. The input file of the agreement program is as follows:

```
program Byzantine-Failstop
 var
bool bi;
bool bj;
bool bk;
bool bl;
bool bg;

```

```
  int dg=0,    domain   0 .. 1;
  int  di,    domain   -1 .. 1;
                 // (di == -1) means process $i$ has not yet decided.
  int  dj,    domain   -1 .. 1;
  int  dk,    domain   -1 .. 1;
  int  dl,    domain   -1 .. 1;

  bool  fi;
  bool  fj;
  bool  fk;
  bool  fl;

  bool  upi;
  bool  upj;
  bool  upk;
  bool  upl;

// The structure of process i.
   process   i
   begin
 ((di == -1) && (fi == 0) && (upi == 0)) ->  di = dg ;
|
((di != -1) && (fi == 0) && (upi == 0)) ->  fi = 1 ;

  read di, dj, dk, dl, dg, fi, upi, bi;
  write di, fi;
  end

// The structure of process j.
   process  j
   begin
((dj == -1) && (fj == 0) && (upj == 0)) -> dj = dg;
|
((dj != -1) && (fj == 0) && (upj == 0)) -> fj = 1;

  read di, dj, dk, dl, dg, fj, upj, bj;
  write dj, fj;
  end

// The structure of process k.
   process  k
   begin
((dk == -1) && (fk == 0) && (upk == 0)) -> dk = dg;
|
((dk != -1) && (fk == 0) && (upk == 0)) -> fk = 1;

  read di, dj, dk, dl, dg, fk, upk, bk;
  write dk, fk;
  end

// The structure of process l.
   process  l
   begin
((dl == -1) && (fl == 0) && (upl == 0)) -> dl = dg;
|
((dl != -1) && (fl == 0) && (upl == 0)) -> fl = 1;

  read di, dj, dk, dl, dg, fl, upl, bl;
  write dl, fl;
  end

// Faults are represented as a process.

  fault  FailstopAndByzantine
   begin
((upi == 1)&&(upj == 1)&&(upk == 1)&&(upl == 1))
```

```
                  -> upi = 0, upj = 0, upk = 0, upl = 0 ,
|
((bi == 0)&&(bj == 0)&&(bk == 0)&&(bl == 0)&&(bg == 0))
                  -> bi = 1, bj = 1, bk = 1, bl = 1, bg = 1,
|
((bi == 1)) -> di = 1 , di =0 ,
|
((bj == 1)) -> dj = 1 , dj =0 ,
|
((bk == 1)) -> dk = 1 , dk =0 ,
|
((bl == 1)) -> dl = 1 , dl =0 ,
|
((bg == 1)) -> dg = 1 , dg =0 ,
  end

// The invariant of the program.
  invariant
( (
  ((bg==0) &&
        (((bi == 1) && (bj == 0)&& (bk == 0)&& (bl == 0)) ||
         ((bj == 1) && (bi == 0)&& (bk == 0)&& (bl == 0)) ||
         ((bk == 1) && (bj == 0)&& (bi == 0)&& (bl == 0)) ||
         ((bl == 1) && (bj == 0)&& (bk == 0)&& (bi == 0)) ||
         ((bi == 0) && (bj == 0)&& (bk == 0)&& (bl == 0)) ) &&
         ((bi==1)||(di==-1)||(di==dg))&&
         ((bj==1)||(dj==-1)||(dj==dg))&&
         ((bk==1)||(dk==-1)||(dk==dg))&&
         ((bl==1)||(dl==-1)||(dl==dg))&&
         ((bi==1)||(fi==0)||(di!=-1) )&&
         ((bj==1)||(fj==0)||(dj!=-1) )&&
         ((bk==1)||(fk==0)||(dk!=-1) )&&
         ((bl==1)||(fl==0)||(dl!=-1) ) ) ||

  ((bg==1)&& (bi==0)&&(bj==0)&&(bk==0)&&(bl==0)&& (
     ((((upi == 1) && (upj == 1)&& (upk == 1)&& (upl == 1)))  &&
                      ((di==dj)&&(dj==dk)&&(dk==dl)&&(di!=-1)) ) ||
     ((((upi == 1) && (upj == 1)&& (upk == 1)&& (upl == 0)))  &&
                      ((di==dj)&&(dj==dk)&&(di!=-1)) ) ||
     ((((upi == 1) && (upj == 1)&& (upk == 0)&& (upl == 1)))  &&
                      ((di==dj)&&(dj==dl)&&(di!=-1)) ) ||
     ((((upi == 1) && (upj == 0)&& (upk == 1)&& (upl == 1)))  &&
                      ((di==dk)&&(dk==dl)&&(di!=-1)) ) ||
     ((((upi == 0) && (upj == 1)&& (upk == 1)&& (upl == 1)))  &&
                      ((dj==dk)&&(dk==dl)&&(dj!=-1)) )
         ))
          )
         &&
         (
         ((upi == 0) && (upj == 1) && (upk ==1) && (upl == 1)) ||
         ((upi == 1) && (upj == 0) && (upk ==1) && (upl == 1)) ||
         ((upi == 1) && (upj == 1) && (upk ==0) && (upl == 1)) ||
         ((upi == 1) && (upj == 1) && (upk ==1) && (upl == 0)) ||
         ((upi == 1) && (upj == 1) && (upk ==1) && (upl == 1))   ))

// The specification of the program is specified in three parts starting
// with  specification keyword.

  specification

// The  destination part identifies a set of states that every
// transition reaching them violates safety.

  destination
(
( (bid == 0) && (bjd == 0) && (upid == 1) && (upjd == 1) && (did != -1) &&
                 (djd != -1) && (did != djd) && (fid == 1)  && (fjd == 1)) ||
( (bid == 0) && (bkd == 0) && (upid == 1) && (upkd == 1) && (did != -1) &&
                 (dkd != -1) && (did != dkd) && (fid == 1) && (fkd == 1)) ||
( (bid == 0) && (bld == 0) && (upid == 1) && (upld == 1) && (did != -1) &&
                 (dld != -1) && (did != dld) && (fid == 1) && (fld == 1)) ||
( (bjd == 0) && (bkd == 0) && (upkd == 1) && (upjd == 1) && (djd != -1) &&
                 (dkd != -1) && (djd != dkd) && (fjd == 1) && (fkd == 1)) ||
( (bjd == 0) && (bld == 0) && (upld == 1) && (upjd == 1) && (djd != -1) &&
                 (dld != -1) && (djd != dld) && (fjd == 1) && (fld == 1)) ||
( (bkd == 0) && (bld == 0) && (upkd == 1) && (upld == 1) && (dkd != -1) &&
                 (dld != -1) && (dkd != dld) && (fkd == 1) && (fld == 1)) ||

((bgd == 0) && (bid == 0)  && (did != -1) && (did != dgd) && (fid == 1)) ||
((bgd == 0) && (bjd == 0)  && (djd != -1) && (djd != dgd) && (fjd == 1)) ||
((bgd == 0) && (bkd == 0)  && (dkd != -1) && (dkd != dgd) && (fkd == 1)) ||
((bgd == 0) && (bld == 0)  && (dld != -1) && (dld != dgd) && (fld == 1))
)

// The  relation part identifies a set of transitions that violate safety.

  relation
((((bis == 0)&& (bid == 0) &&  (fis == 1) && (dis != did))) ||
             (((bjs == 0) && (bjd == 0) && (fjs == 1) && (djs != djd)))||
             (((bks == 0) && (bkd == 0) && (fks == 1) && (dks != dkd)))||
             (((bls == 0) && (bld == 0) && (fls == 1) && (dls != dld)))||
             (((bis == 0) && (bid == 0) && (fis == 1) && (fid == 0)))||
             (((bjs == 0) && (bjd == 0) && (fjs == 1) && (fjd == 0)))||
             (((bks == 0) && (bkd == 0) && (fks == 1) && (fkd == 0)))||
             (((bls == 0) && (bld == 0) && (fls == 1) && (fld == 0))))

// The  init section is used for specifying the initial states.

  init
```

```
174
175 // Each initial state is specified using the  state keyword.
176
177    state
178 bi = 0; bj = 0; bk = 0; bl = 0; bg = 0; dg = 0;
179 di = -1; dj = -1; dk = -1; bl = -1;
180 fi = 0; fj = 0; fk = 0; fl = 0; upi = 1; upj = 1;
181 upk = 1; upl = 1;
182
183
184    state
185 bi = 0; bj = 0; bk = 0; bl = 0; bg = 0; dg = 1;
186 di = -1; dj = -1; dk = -1; bl = -1;
187 fi = 0; fj = 0; fk = 0; fl = 0; upi = 1; upj = 1;
188 upk = 1; upl = 1;
189
```

B.1 The Description of the Input File

The fault-intolerant agreement program consists of four non-general processes P_i, P_j, P_k, P_l and a general P_g. Each non-general process has four variables d, f, b, and *up*. Variable *di* represents the decision of a non-general process P_i, fi denotes whether P_i has finalized its decision, *bi* denotes whether P_i is Byzantine or not, and *upi* states whether P_i has failed or not. Process P_g also has variables *dg* and *bg*. We assume that the process P_g never fails. Thus, the variables of the agreement program are as shown in the *var* section (cf. Lines 2-24).

Transitions of the fault-intolerant program. If process P_i has not copied a value from the general and P_i has not failed (i.e., upi = 1) then P_i copies the decision of the general (first action in the body of process P_i (cf. Line 29)). If P_i has copied a decision and as a result *di* is different from -1 then P_i can finalize its decision if it has not failed (second action in the body of process P_i (cf. Line 31)). Other non-general processes (P_j, P_k, and P_l) have a similar structure as shown in the input file (cf. Lines 37-68).

Read/Write restrictions. Each non-general process P_i is allowed to read $\{di, dj, dk, dl, dg, fi, upi, bi\}$. Thus, P_i can read the d values of other processes and all its variables. The set of variables that P_i can write is $\{di, fi\}$. Read/write restrictions of each process are specified in its body after the program actions (using *read* and *write* keywords (e.g., Lines 33-34)).

Faults. A Byzantine fault transition can cause a process to become Byzantine if no process is initially Byzantine. A Byzantine process can arbitrarily change its decision (i.e., the value of d). Moreover, the program is subject to fail-stop faults such that at most one of the non-general processes can be failed, and as a result, it will stop executing any action. The developers of fault-tolerance should specify the faults similar to an independent process that can perturb program variables (cf. Lines 72-89).

Invariant. The developers of fault-tolerance our framework should represent the invariant of the program as a state predicate. In particular, the invariant is a Boolean function (over program variables) that takes a state s and identifies whether s is an invariant state or not.

In the agreement program, the bg variable partitions the invariant into two parts: the set of states where P_g is non-Byzantine (cf. Line 94), and the set of states where P_g is Byzantine (cf. Line 109). When P_g is non-Byzantine, at most one of the non-generals could be Byzantine (cf. Lines 95-107). Also, for every non-general process P_i that is non-Byzantine (i) P_i has not yet decided or it has copied the value of dg (cf. Lines 100-103), and (ii) P_i has not yet finalized or P_i has decided (cf. Lines 104-107). When P_g becomes Byzantine, all the non-general processes are non-Byzantine and all the processes that have not failed agree on the same decision (cf. Lines 109-119).

The invariant of the agreement program stipulates the above conditions on the states where at most one non-general process has failed (cf. Lines 124-128).

Safety specification. The safety specification requires that if P_g is Byzantine, all the non-general non-Byzantine processes that have not failed should finalize with the same decision (*agreement*). If P_g is not Byzantine, then the decision of every finalized non-general non-Byzantine process should be the same as dg (*validity*). Thus, safety is violated if the program executes a transition that satisfies at least one of the conditions specified in the *specification* section of the input file (cf. Lines 133-169).

The *specification* section is divided into two parts: *destination* and *relation* parts. Intuitively, in the *destination* part (cf. Lines 138-158), we write a state predicate that identifies a set of states $s_{destination}$, where if a transition t reaches $s_{destination}$ then t violates safety. In the *relation* part (cf. Lines 162-169), we specify a condition that identifies a set of transitions that should not be executed by the program. Note, that we have added a suffix "d" (respectively, suffix "s") to the variable names in the specification section that stands for *destination* (respectively, *source*). Since the relation condition specifies a set of transitions t_{spec} using their source and destination states, we need to distinguish between the value of a specific variable x in the source state of t_{spec} (i.e., xs means the value of x in the source state of t_{spec}) and in the destination state of t_{spec} (i.e., xd means the value of x in the destination state of t_{spec}).

In the case that the program specification does not stipulate any destination condition on safety-violating transitions, we leave the destination section empty with the keyword *noDestination* . We use similar keyword *noRelation* for the case where we do not have relation conditions in the specification.

Initial states. The keyword *init* (cf. Line 173) identifies the section of the input file where the user has to specify some initial states. These initial states should belong to the invariant. For each initial state, the user should use the reserved word *state* (cf.

Line 177). In the *state* section (cf. Lines 177-181 and 185-188), the user should assign some values to the program variables that belong to their corresponding domain.

B.2 The Output of the Framework

In this section, we present the output of the synthesis framework. In particular, we present the actions of other non-general processes. Observe that the structures of the non-generals are not symmetric.

In the rest of this section, we describe the structure of each non-general process that is subject to Byzantine and fail-stop faults. Note that each non-general process can take an action if and only if it has not yet finalized and also has not failed due to fail-stop faults.

The description of process Pi. Process P_i of the fault-tolerant agreement program consists of 5 actions. We describe each action as a separate item.

1. If process P_i has not yet decided then it performs one of the following actions: either P_i copies the decision of the general, or if at least two other non-generals have decided on the same value then P_i copies their decision.

```
(di == -1) && (
   ((dk == 0)&&(dl == 0)&&(fi == 0)&&(upi == 1)) ||
   ((dg == 0)&&(fi == 0)&&(upi == 1)) ||
   ((dj == 0)&&(dl == 0)&&(fi == 0)&&(upi == 1)) ||
   ((dj == 0)&&(dk == 0)&&(fi == 0)&&(upi == 1)) ) ->  set_di_val0

(di == -1) && (
   ((dk == 1)&&(dl == 1)&&(fi == 0)&&(upi == 1)) ||
   ((dg == 1)&&(fi == 0)&&(upi == 1)) ||
   ((dj == 1)&&(dl == 1)&&(fi == 0)&&(upi == 1)) ||
   ((dj == 1)&&(dk == 1)&&(fi == 0)&&(upi == 1)) ) ->  set_di_val1
```

2. If process P_i has copied 1, and at least one of the following conditions holds then process P_i changes its decision to 0: (i) P_k and P_l have decided on 0 and P_j has decided; (ii) P_j and P_l have decided on 0, or (iii) P_j and P_k have decided on 0 and P_l has decided.

```
(di == 1) && (
   (((dj ==0 )||(dj == 1))&&(dk == 0)&&(dl == 0)&&(fi == 0)&&(upi == 1)) ||
   ((dj == 0)&&(dl == 0)&&(fi == 0)&&(upi == 1)) ||
   ((dj ==0 )&&(dk == 0)&&((dl == 0)&&(dl == 1))&&(fi == 0)&&(upi == 1)) )
                                                              -> set_di_val0
```

3. If process P_i has copied 0, and at least one of the following conditions holds then process P_i changes its decision to 1: (i) P_j and P_k have decided on 1; (ii) P_l and P_g have decided on 1; (iii) P_j and P_l have decided on 1, or (iv) P_k and P_l have decided on 1.

```
(di == 0) && (
   ((dj == 1 )&&(dk == 1)&&(fi == 0)&&(upi == 1)) ||
   ((dl == 1 )&&(dg == 1)&&(fi == 0)&&(upi == 1)) ||
   ((dj == 1 )&&(dl == 1)&&(fi == 0)&&(upi == 1)) ||
   ((dk == 1 )&&(dl == 1)&&(fi == 0)&&(upi == 1)) ) -> set_di_val1
```

4. Process P_i finalizes with decision 0 if at least one of the following conditions holds. (i) P_j has decided on 0 or P_j has not yet decided, and P_k has decided on 0, and P_l has decided on 0 or P_l has not yet decided; (ii) P_j has decided on 0 or P_j has not yet decided, and P_k has decided on 0 or P_k has not yet decided, and P_l has decided on 0; (iii) P_j has decided on 0, and P_k has decided on 0 or P_k has not yet decided, and P_l has decided on 0 or P_l has not yet decided.

```
(di == 0) && (
   (((dj == 0)||(dj == -1))&&(dk == 0)&&((dl == 0)||(dl == -1))&&
                                              (fi == 0)&&(upi == 1)) ||
    (((dj == 0)||(dj == -1))&&(dl == 0)&&((dk == 0)||(dk == -1))&&
                                            (fi == 0)&&(upi == 1)) ||
    ((dj == 0)&&((dk == 0)||(dk == -1))&&((dl == 0)||(dl == -1))&&
                                            (fi == 0)&&(upi == 1)) )
                                                         -> set_fi_val1
```

5. Process P_i finalizes with decision 1 if at least one of the following conditions holds. (i) P_j has decided on 1, and P_k has decided on 1 or P_k has not yet decided, and P_l has decided on 1 or P_l has not yet decided; (ii) P_j has decided on 1 or P_j has not yet decided, and P_l has decided on 1 or P_l has not yet decided, and P_k has decided on 1; (iii) P_j has decided on 1 or P_j has not yet decided, and P_k has decided on 1 or P_k has not yet decided, and P_l has decided on 1.

```
(di == 1) && (
    ((dj == 1)&&((dk == 1)||(dk == -1))&&((dl == 1)||(dl == -1))&&
                                            (fi == 0)&&(upi == 1)) ||
    (((dj == 1)||(dj == -1))&&(dk == 1)&&((dl == 1)||(dl == -1))&&
                                            (fi == 0)&&(upi == 1)) ||
    (((dj == 1)||(dj == -1))&&((dk == 1)||(dk == -1))&&(dl == 1)&&
                                            (fi == 0)&&(upi == 1)) )
                                                         -> set_fi_val1
```

The description of process P_j. The actions of process P_j in the fault-tolerant agreement program are as follows:

1. If process P_j has not yet decided then it performs one of the following actions: P_j either copies the decision of the general, or if at least two other non-generals have decided on the same value then P_j copies their decision.

2. If process P_j has copied 1, and at least one of the following conditions holds then process P_j changes its decision to 0: (i) P_i and P_l have decided on 0; (ii) P_k and P_l have decided on 0, or (iii) P_i and P_k have decided on 0.

3. If process P_j has copied 0, and at least one of the following conditions holds then process P_j changes its decision to 1: (i) P_i and P_k have decided on 1; (ii) P_i and P_l have decided on 1, or (iii) P_k and P_l have decided on 1.

4. Process P_j finalizes with decision 0 if at least one of the following conditions holds: (i) P_i has decided on 0 or P_i has not yet decided, and P_k has decided on 0 or P_k has not yet decided, and P_l has decided on 0; (ii) P_i has decided on 0, and P_k has decided on 0 or P_k has not yet decided, and P_l has decided on 0 or P_l has not yet decided; (iii) P_i has decided on 0 or P_i has not yet decided, and P_k has decided on 0, and P_l has decided on 0 or P_l has not yet decided.

5. Process P_j finalizes with decision 1 if at least one of the following conditions holds: (i) P_i has decided on 1 or P_i has not yet decided, and P_k has decided on 1 or P_k has not yet decided, and P_l has decided on 1; (ii) P_i has decided on 1 or P_i has not yet decided, and P_l has decided on 1 or P_l has not yet decided, and P_k has decided on 1; (iii) i has decided on 1, and k has decided on 1 or k has not yet decided, and l has decided on 1 or l has not yet decided.

The description of process P_k. The actions of process P_k in the fault-tolerant agreement program are as follows:

1. If process P_k has not yet decided then it performs one of the following actions: P_k either copies the decision of the general, or if at least two other non-generals have decided on the same value then P_k copies their decision.

2. If process P_k has copied 1, and at least one of the following conditions holds then process P_k changes its decision to 0: (i) P_l and P_g have decided on 0; (ii) P_i and P_j have decided on 0; (iii) P_j and P_l have decided on 0; (iv) P_i and P_l have decided on 0; (v) P_j and P_g have decided on 0, or (vi) P_i and P_g have decided on 0.

3. If process P_k has copied 0, and at least one of the following conditions holds then process P_k changes its decision to 1: (i) P_j and P_g have decided on 1; (ii) P_l and P_g have decided on 1; (iii) P_i and P_j have decided on 1; (iv) P_j and P_l have decided on 1, or (v) P_i and P_l have decided on 1.

4. Process P_k finalizes with decision 0 if at least one of the following conditions holds: (i) P_i has decided on 0, and P_j has decided on 0 or P_j has not yet decided, and P_l has decided on 0 or P_l has not yet decided; (ii) P_i has decided on 0 or P_i has not yet decided, and P_j has decided on 0, and P_l has decided on 0 or P_l has not yet decided; (iii) P_i has decided on 0 or P_i has not yet decided, and P_l has decided on 0, and P_j has decided on 0 or P_j has not yet decided.

5. Process P_k finalizes with decision 1 if at least one of the following conditions holds: (i) P_i has decided on 1, and P_j has decided on 1 or P_j has not yet decided, and P_l has decided on 1 or P_l has not yet decided; (ii) P_i has decided on 1 or P_i has not yet decided, and P_j has decided on 1 or P_j has not yet decided, and P_l has decided on 1; (iii) P_i has decided on 1 or P_i has not yet decided, and P_l has decided on 1 or P_l has not yet decided, and P_j has decided on 1.

The description of process P_l. The actions of process P_l in the fault-tolerant agreement program are as follows:

1. If process P_l has not yet decided then it performs one of the following actions: P_l either copies the decision of the general, or if at least two other non-generals have decided on the same value then P_l copies their decision.

2. If process P_l has copied 1, and at least one of the following conditions holds then process P_l changes its decision to 0: (i) P_i and P_g have decided on 0; (ii) P_j and P_k have decided on 0; (iii) P_i and P_j have decided on 0; (iv) P_i and P_k have decided on 0.

3. If process P_l has copied 0, and at least one of the following conditions holds then process P_l changes its decision to 1: (i) P_i and P_j have decided on 1; (ii) P_i and P_k have decided on 1; (iii) P_j and P_k have decided on 1.

4. Process P_l finalizes with decision 0 if at least one of the following conditions holds: (i) P_i has decided on 0, and P_j has decided on 0 or P_j has not yet decided, and P_k has decided on 0 or P_k has not yet decided; (ii) P_i has decided on 0 or P_i has not yet decided, and P_j has decided on 0 or P_j has not yet decided, and P_k has decided on 0; (iii) P_i has decided on 0 or P_i has not yet decided, and P_j has decided on 0, and P_k has decided on 0 or P_k has not yet decided.

5. Process P_l finalizes with decision 1 if at least one of the following conditions holds: (i) P_i has decided on 1, and P_j has decided on 1 or P_j has not yet decided, and P_k has decided on 1 or P_k has not yet decided; (ii) P_i has decided on 1 or P_i has not yet decided, and P_j has decided on 1 or P_j has not yet decided, and P_k has decided on 1; (iii) P_i has decided on 1 or P_i has not yet decided, and P_k has decided on 1 or P_k has not yet decided, and P_j has decided on 1.

Bibliography

Bibliography

[1] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, page 82, 2000.

[2] 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.

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

[4] P. Attie and A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM TOPLAS (a preliminary version of this paper appeared in PODC96)*, 23(2), March 2001.

[5] A. Pnueli and R. Rosner. On the synthesis of a reactive module. *In Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 179–190, 1989.

[6] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesis. *In Proc. of 31st IEEE Symposium on Foundation of Computer Science*, pages 746–757, 1990.

[7] O. Kupferman and M.Y. Vardi. Synthesizing distributed systems. In *Proc. 16th IEEE Symp. on Logic in Computer Science*, July 2001.

[8] Ali Ebnenasir. Automatic synthesis of distributed programs: A survey. `http://www.cse.msu.edu/~ebnenasi/survey.pdf`, 2002.

[9] Felix C. Gartner and Arshad Jhumka. Automating the addition of fault-tolerance: Beyond fusion-closed specifications. *Formal Modeling and Analysis of Timed Systems - Formal Techniques in Real-Time and Fault Tolerant System (FORMATS-FTRTFT 2004), Grenoble, France, September 22-24*, 2004.

[10] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

[11] V. Hadzilacos E. Anagnostou. Tolerating transient and permanent failures. *Proceedings of the 7th International Workshop on Distributed Algorithms. Les Diablerets, Switzerland*, pages 174–188, September 1993.

[12] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 3.1 – 3.15, 1995.

[13] S. Tsang and E. Magill. Detecting feature interactions in the intelligent network. *Feature Interactions in Telecommunications Systems II, IOS Press*, pages 236 – 248, 1994.

[14] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of byzantine agreement. *Symposium on Reliable Distributed Systems*, page 130, 2001.

[15] S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. *In Proceedings of International Conference on Distributed Computing Systems*, page 441, 2003.

[16] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[17] 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.

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

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

[20] A. Arora and S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *Revised for* IEEE Transactions on Software Engineering, 1995. A preliminary version appears in the Proceedings of the Fourteenth Symposium on Reliable Distributed Systems, Bad Neuenahr, 174–185, 1995.

[21] G. Varghese. *Self-stabilization by local checking and correction.* PhD thesis, MIT/LCS/TR-583, 1993.

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

[23] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 7 October 1985.

[24] M. Barborak, A. Dahbura, and M. Malek. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171–220, 1993.

[25] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. In *European Dependable Computing Conference*, pages 71–87, 1999.

[26] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382 – 401, July 1982.

[27] L. Gong, P. Lincoln, and J. Rushby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. *In Proceedings Dependable Computing for Critical Applications-5, Champaign, IL*, pages 139–157, September 1995.

[28] M. Singhal and N. Shivaratri. *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems.* McGraw-Hill Publishing Company, 1994.

[29] Ali Ebnenasir and Sandeep S. Kulkarni. Efficient synthesis of failsafe fault-tolerant distributed programs. Technical Report MSU-CSE-05-13, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, April 2005.

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

[31] S. S. Kulkarni and A. Ebnenasir. The complexity of adding failsafe fault-tolerance. *In Proceedings of International Conference on Distributed Computing Systems*, page 337, 2002.

[32] A. Arora and S. S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Transactions on Software Engineering*, pages 435–450, June 1998. A preliminary version appears in the Proceedings of the Fourteenth Symposium on Reliable Distributed Systems, Bad Neuenahr, 1995, pages 174–185.

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

[34] A. Arora and S. S. Kulkarni. Component based design of multi-tolerant systems. *IEEE Transactions on Software Engineering*, 24:63–78, January 1998.

[35] A. Moormann Zaremski and J.M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering Methods (A preliminary version appeared in Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1995)*, 6(4):333 – 369, 1997.

[36] G. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 1997.

[37] Spin language reference. `http://spinroot.com/spin/Man/promela.html`.

[38] Anish Arora, Mohamed G. Gouda, and George Varghese. Constraint satisfaction as a basis for designing nonmasking fault-tolerant systems. *Journal of High Speed Networks*, 5(3):293–306, 1996.

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

[40] A.I. Tomlinson and V.K. Garg. Detecting relational global predicates in distributed systems. *In proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, California.*, pages 21–31, May 1993.

[41] Neeraj Mittal. *Techniques for Analyzing Distributed Computations.* PhD thesis, The University of Texas at Austin, 2002.

[42] Klaus Havelund and Tom Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), April 2000.

[43] Gerard J. Holzmann. From code to models. *In Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD'01)*, pages 3–10, 2001.

[44] M.G. Gouda and T. McGuire. Correctness preserving transformations for network protocol compilers. *Prepared for the Workshop on New Visions for Software Design and Productivity: Research and Applications*, 2001.

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

[46] M. Demirbas and A. Arora. Convergence refinement. *International Conference on Distributed Computing Systems*, pages 589 – 596, 2002.

[47] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1995.

[48] R. Bharadwaj and C. Heitmeyer. Developing high assurance avionics systems with the SCR requirements method. *In Proceedings of the 19th Digital Avionics Systems Conference, Philadelphia, PA*, October 2000.

[49] R. Hardin, R. Kurshan, S. Shukla, and M. Vardi. A new heuristic for bad cycle detection using bdds. *Computer Aided Verification (CAV'97). LNCS Springer-Verlag*, 1254:268 – 278, 1997.

[50] R. Bloem, H.N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in n log n symbolic steps. *In Proc. FMCAD, LNCS Springer-Verlag*, 1954:37–54, 2000.

[51] K. Fisler, R. Fraer, G. Kamhi, Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? *In Proc. Tools and Algorithms for Construction and Analysis of Systems, LNCS*, 2031:420–434, 2001.

[52] The alloy analyzer. `http://alloy.mit.edu`.

[53] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. *39th Design Automation Conference, Las Vegas*, 2001.

[54] Satisfiability suggested format dimacs, may. `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/satformat.tex`, 1993.

[55] Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and N. Shankar. ICS: integrated canonizer and solver. *Proceedings of the 13th Conference on Computer-Aided Verification (CAV'01), volume 2102 of Lecture Notes in Computer Science. Springer-Verlag*, 2001.

[56] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, 1984.

[57] P. Attie. Synthesis of large concurrent programs via pairwise composition. *CONCUR'99: 10th International Conference on Concurrency Theory*, 1999.

[58] Xinghua Deng, Matthew B. Dwyer, John Hatcliff, and Masaaki Mizuno. Invariant-based specification, synthesis and verification of synchronization in concurrent programs. *Proceedings of the 24th International Conference on Software Engineering*, May 2002.

[59] Y.Inaba. An implementation of synthesizing synchronization skeletons using temporal logic specifications. *Master Thesis, The University of Texas at Austin*, 1984.

[60] Sandeep S. Kulkarni and Ali Ebnenasir. A framework for automatic synthesis of fault-tolerance. Technical Report MSU-CSE-03-16, Computer Science and Engineering, Michigan State University, East Lansing MI 48824, Michigan, July 2003.

[61] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. *In Proceeding of 16th International Colloqium on Automata, Languages, and Programming*, Lec. Notes in Computer Science 372, Springer-Verlag:652–671, 1989.

[62] A.W. Appel and A.P. Felty. A semantic model of types and machine instructions for proof-carrying code. *In Proceedings of the 27th ACM Symposium of Principles of Programming Languages, ACM Press*, pages 243–253, 2001.

[63] Bernd Fisher, Johann Schumann, and Mike Whalen. Synthesizing certified code. *In Proceedings Formal Methods Europe(FME'02). Copenhagen, Denmark. LNAI, Springer*, 2002.

[64] P.J. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[65] S. Lafortune and F. Lin. On tolerable and desirable behaviors in supervisory control of discrete event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 1(1):61–92, 1992.

[66] Feng Lin and W. Murray Wonham. Decentralized control and coordination of discrete-event systems with partial observation. *IEEE Transactions On Automatic Control*, 35(12), December 1990.

[67] Karen Rudie and W.M. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions On Automatic Control*, 37(11):1692–1708, 1992.

[68] Kurt Ryan Rohloff. Computations on distributed discrete-event systems. *Ph.D. thesis, University of Michigan*, 2004.

[69] Wolfgang Thomas. On the synthesis of strategies in infinite games. *STACS*, pages 1–13, 1995.

[70] R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1993.

[71] A. Anuchitanukul and Z. Manna. Realizability and synthesis of reactive modules. *International Conference on Computer-Aided Verification*, pages 156–169, 1994.

[72] E.A. Emerson. *Handbook of Theoretical Computer Science: Chapter 16, Temporal and Modal Logic.* Elsevier Science Publishers B. V., 1990.

[73] FTSyn: A framework for automatic synthesis of fault-tolerance. `http://www.cse.msu.edu/~ebnenasi/research/tools/ftsyn.htm`.

[74] Audun Jusang. Security protocol verification using spin. *The First SPIN Workshop*, 1995.

[75] Gregory Duval and Jacques Julliand. Modeling and verification of rubis microkernel with spin. *The First SPIN Workshop*, 1995.

[76] M.S. Laventhal. *Synthesis of Synchronization Code for Data Abstraction.* PhD thesis, MIT, 1978.

[77] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[78] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specification. *In Proceeding of 16th International Colloqium on Automata, Languages, and Programming, volume 372 of LNCS*, pages 1–17, 1989.

[79] H. Wong-Toi and D. Dill. Synthesizing processes and schedulers from temporal logic specifications. *Computer-Aided Verification (Proceeding of CAV90 Workshop*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science Vol. 3, 1991.

[80] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. *In Hybrid System II, LNCS 999*, pages 1 – 20, 1995.

[81] M. Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. *Computer Aided Verfification, volume 939 of LNCS,*, pages 267–278, 1995.

[82] Ali Ebnenasir and Sandeep Kulkarni. Automatic addition of liveness. Technical Report MSU-CSE-04-22, Department of Computer Science, Michigan State University, East Lansing, Michigan, June 2004.

[83] Jiri Barnat, Lubos Brim, and Jitka Stříbrná. Distributed LTL model-checking in SPIN. *Lecture Notes in Computer Science*, 2057:200–216, 2001.

[84] U.Stern and D. L. Dill. Parallelizing the murphi verifier. *Proceedings of Computer Aided Verification (CAV '97)*, 1254 of LNCS:256–267, 1997.

[85] S. Ben-David, T. Heyman, and O. Grumberg. Scalable distributed on-the-fly symbolic model checking. *In third International Conference on Formal methods in Computer-Aided Design (FMCAD'00), Austin, Texas*, 2000.

[86] S. Aggarwal, R. Alonso, and C. Courcoubetis. Distributed reachability analysis for protocol verification environments. *in Discrete Event Systems: Models and Applications, IIASA Conference*, pages 40–56, 1987.

[87] H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model checking. *In Proc. SPIN Workshop on Model Checking of Software*, 2057 of LNCS:215+, 2001.

[88] Gerard Holzmann. State compression in spin:recursive indexing and compression training runs. *The Third SPIN Workshop*, 1997.