Complexity Issues in Automated Synthesis of Failsafe Fault-Tolerance

Sandeep S. Kulkarni and Ali Ebnenasir

Abstract—We focus on the problem of synthesizing failsafe fault-tolerance where fault-tolerance is added to an existing (fault-intolerant) program. A failsafe fault-tolerant program satisfies its specification (including safety and liveness) in the absence of faults. However, in the presence of faults, it satisfies its safety specification. We present a somewhat unexpected result that, in general, the problem of synthesizing failsafe fault-tolerant *distributed* programs from their fault-intolerant version is NP-complete in the state space of the program. We also identify a class of specifications, *monotonic specifications*, and a class of programs, *monotonic programs*, for which the synthesis of failsafe fault-tolerance can be done in polynomial time (in program state space). As an illustration, we show that the monotonicity restrictions are met for commonly encountered problems, such as Byzantine agreement, distributed consensus, and atomic commitment. Furthermore, we evaluate the role of these restrictions in the complexity of synthesizing failsafe fault-tolerance. Specifically, we prove that if only one of these conditions is satisfied, the synthesis of failsafe fault-tolerance is still NP-complete. Finally, we demonstrate the application of monotonicity property in enhancing the fault-tolerance of (distributed) nonmasking fault-tolerant programs to masking.

Index Terms—Fault-tolerance, automatic addition of fault-tolerance, formal methods, program synthesis, distributed programs.

1 Introduction

We focus on the automated synthesis of failsafe fault-tolerant programs, i.e., programs that satisfy their safety specification if faults occur. We begin with a fault-intolerant program and systematically add fault-tolerance to it. The resulting program, thus, guarantees that if no faults occur then the specification is satisfied. However, if faults do occur, then at least the safety specification is satisfied.

There are several advantages of such automation. For one, the synthesized program is correct by construction and, hence, there is no need for its correctness proof. Second, since we begin with an existing fault-intolerant program, the derived fault-tolerant program reuses it. Third, in this approach, the concerns of the functionality of a program and its fault-tolerance are separated. This separation is known to help [14] in simplifying the reuse of the techniques used in the manual addition of fault-tolerance. We expect that the same advantage will apply in the automated addition of fault-tolerance.

The main difficulty in automating the addition of fault-tolerance, however, is the complexity involved in this process. In [16], Kulkarni and Arora showed that the problem of adding masking fault-tolerance (where both safety and liveness are satisfied in the presence of faults) to distributed programs is NP-complete (in program state space). We find that there exist three possible options to deal with this complexity: 1) developing heuristics under which the synthesis algorithm takes polynomial time in program state space, 2) considering a weaker form of fault-tolerance such as failsafe, where only safety is satisfied in the presence of faults, or nonmasking, where the nonmasking program recovers to state from where it satisfies its specification, however, safety may be violated during recovery, or 3) identifying a class of specifications and programs for which the addition of fault-tolerance can be performed in polynomial time (in program state space).

Kulkarni et al. [17] focused on the first approach and presented heuristics that are applicable to several problems including Byzantine agreement. When the heuristics are applicable, the algorithm in [17] obtains a masking fault-tolerant program in polynomial time in the state space of the fault-intolerant program. However, it fails to find a fault-tolerant program (even if one exists) if the heuristics are not applicable.

By adding failsafe fault-tolerance in an automated fashion, we *potentially* simplify, and partly automate, the design of masking fault-tolerant programs. More specifically, the algorithm that automates the addition of failsafe fault-tolerance and the stepwise method for designing masking fault-tolerance [14] can be combined to partially automate the design of masking fault-tolerant programs. The algorithm in [14] shows how we can design a masking fault-tolerant program by first designing a failsafe (respectively, nonmasking) fault-tolerant program and then adding nonmasking (respectively, failsafe) fault-tolerance to it. Thus, given an algorithm that automates the addition of fault-tolerance, one could begin with a fault-intolerant program, automate the addition of failsafe fault-tolerance, and then manually add nonmasking fault-tolerance. Likewise, one could begin with a nonmasking fault-tolerant program and obtain a masking fault-tolerant program by automating the addition of failsafe fault-tolerance. Thus,

- *The authors are with the Software Engineering and Network Systems Laboratory, Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824.
E-mail: {sandeep, ebnenasi}@cse.msu.edu.*

*Manuscript received 14 May 2004; revised 25 May 2005; accepted 20 June 2005; published online 2 Sept. 2005.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-0070-0504.*

1545-5971/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society

one step in designing masking fault-tolerance could be automated and, hence, the designer will not need prove the correctness of that step.

In this paper, we focus on the other two approaches. Regarding the second approach, we focus our attention on the synthesis of failsafe fault-tolerance. In our investigation, we find a somewhat unexpected result that the design of distributed failsafe fault-tolerant programs is also NP-complete (in program state space). (In [16], it was conjectured that the addition of failsafe fault-tolerance to distributed programs would be easy as it was expected that satisfying safety alone if faults occur would be simple whereas adding recovery would be difficult.) To show this, we provide a reduction from 3-SAT to the problem of adding failsafe fault-tolerance.

To deal with the complexity involved in automating the addition of failsafe fault-tolerance, we follow the third approach considered above. Specifically, we identify the restrictions that can be imposed on specifications and fault-intolerant programs in order to ensure that failsafe fault-tolerance can be added in polynomial time (in program state space). Toward 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 (in program state space) if monotonicity restrictions on the program and the specification are met.

Using monotonicity property, we will be able to extend the scope of programs for which the addition of failsafe fault-tolerance can be done in polynomial time. More specifically, one can design heuristics that convert non-monotonic programs (respectively, specifications) to monotonic programs so that polynomial synthesis of failsafe fault-tolerance becomes possible [9].

As another important contribution of this paper, 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 to distributed programs will remain NP-complete. Also, we demonstrate that the class of monotonic specifications contains well-recognized problems [5], [7], [19], [13], [20] of distributed consensus, atomic commitment and Byzantine agreement. Finally, we show how the monotonicity property may be used to enhance the fault-tolerance of distributed nonmasking programs to masking.

Remark. Adding failsafe fault-tolerance only requires the safety of the distributed consensus and atomic commit in the presence of crash faults; i.e., if the processes reach an agreed decision, then their decision is valid. As a result, the termination condition (the liveness) of these problems may not be met in the presence of faults; i.e., the processes may never reach an agreement. For this reason, the results of this paper do not contradict with the impossibility result of [10].

Failsafe fault-tolerance to crash faults only requires the validity of the consensus, i.e., if all processes reach an agreement on a vote then that vote is valid in that it is the vote of all nonfaulty processes. Since failsafe fault-tolerance does not require the liveness condition of the consensus problem in the presence of faults, the impossibility result of [10] becomes irrelevant in the context of failsafe fault-tolerance.

Organization of the paper. This paper is organized as follows: In Section 2, we provide the basic concepts such as programs, computations, specifications, faults and fault-tolerance. In Section 3, we state the problem of adding failsafe fault-tolerance. In Section 4, we prove the NP-completeness of the problem of adding failsafe fault-tolerance to distributed programs. In Section 5, we precisely define the notion of monotonic specifications and monotonic programs, and identify their role in the complexity of synthesizing failsafe fault-tolerance. We give examples of monotonic specifications and monotonic programs in Section 6. Then, in Section 7, we illustrate how we use the monotonicity property for enhancing the fault-tolerance of nonmasking distributed programs to masking fault-tolerance in polynomial time (in program state space). Finally, in Section 8, we make concluding remarks and identify future research directions.

2 Preliminaries

In this section, we give formal definitions of programs, problem specifications, faults, and fault-tolerance. The programs are specified in terms of their state space and their transitions. The definition of specifications is adapted from Alpern and Schneider [1]. The definition of faults and fault-tolerance is adapted from Arora and Gouda [2] and Arora and Kulkarni [3]. The issues of modeling distributed programs is adapted from Kulkarni and Arora [16], and Attie and Emerson [4]. To illustrate our modeling approach, we use the Byzantine Generals problem [19] as a running example throughout this section.

2.1 Program

A program p is specified by a finite set of variables, say $V = \{v_1, .., v_u\}$, and a finite set of processes, say $P = \{P_1, \cdots, P_n\}$, where u and n are positive integers. Each variable is associated with a finite domain of values. Let $v_1, v_2, .., v_u$ be variables of p, and let $D_1, D_2, .., D_u$ 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: $\langle l_1, l_2, .., l_u \rangle$ where $\forall i : 1 \leq i \leq u : l_i \in D_i$. The state space of p, S_p, is the set of all possible states of p.

A process, say P_j $(0 \leq j \leq n)$, in p is associated with a set of program variables, say r_j, that P_j can read and a set of variables, say w_j, that P_j can write. We assume that $w_j \subseteq r_j$, i.e., P_j cannot *blindly* write any variable. Also, process P_j consists of a set of transitions δ_j; each transition is of the form (s_0, s_1) where $s_0, s_1 \in S_p$. We address the effect of read/write restrictions on δ_j in Section 2.2. The transitions of p, δ_p, is the union of the transitions of its processes.

In this paper, in most situations, we are interested in the state space of p and all its transitions. For the examples considered in this paper, it is straightforward to determine which transition belongs to which process. Hence, unless we need to expose the transitions of a particular process or the values of particular variables, we simply represent a program p by the tuple $\langle S_p, \delta_p \rangle$. A state predicate of p is any subset of S_p. A state predicate S is closed in the program p (respectively, δ_p) iff (if and only if) the following condition holds: $\forall s_0, s_1 :: (((s_0, s_1) \in \delta_p) \wedge (s_0 \in S)) \Rightarrow (s_1 \in S)$.

A sequence of states, $\sigma = \langle s_0, s_1, \ldots \rangle$, is a computation of p iff the following two conditions are satisfied: 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 $(s_l, s) \in \delta_p$. A sequence of states, $\langle s_0, s_1, \ldots, s_k \rangle$, is a computation prefix of p iff $\forall j : 0 < j \leq k : (s_{j-1}, s_j) \in \delta_p$, where k is a positive integer.

The projection of program p on a state predicate S, denoted as $p|S$, is the set of transitions

$$\{(s_0, s_1) : (s_0, s_1) \in \delta_p \ \wedge \ s_0, s_1 \in S\};$$

i.e., $p|S$ consists of transitions of p that start in S and end in S. Given two programs, p and p', we say $p' \subseteq p$ iff $S'_p = S_p$ and $\delta'_p \subseteq \delta_p$.

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

Byzantine generals example. We consider the canonical version of the Byzantine generals problem [19] where there are 4 distributed processes P_g, P_j, P_k, and P_l such that P_g is the general and P_j, P_k, and P_l are the nongenerals. (An identical explanation is applicable if we consider arbitrary number of nongenerals.) In the Byzantine generals program, the general sends its decision to nongenerals and subsequently nongenerals output their decisions. Thus, 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 if that process has finalized (output) its decision. The program variables and their domains are as follows:

$d.g : \{0,1\}$
$d.j, d.k, d.l : \{0, 1, \perp\}$
// $\perp$ denotes uninitialized decision
$b.g, b.j, b.k, b.l : \{true, false\}$
// $b.j = true$ iff P_j is Byzantine
$f.j, f.k, f.l : \{0,1\}$
// $f.j = 1$ iff P_j has finalized its decision

The fault-intolerant Byzantine generals program, *IB*. We use Dijkstra's guarded commands [6] as a shorthand for representing the set of program transitions. A guarded command (action) is of the form $grd \rightarrow st$, where grd is a state predicate and st is a statement that updates the program variables. The guarded command $grd \rightarrow st$ includes all program transitions $\{(s_0, s_1) : grd$ holds at s_0 and the atomic execution of st at s_0 takes the program to state $s_1\}$. We represent the actions of the nongeneral process P_j as follows (the actions of other nongenerals are similar):

$$IB_1 : d.j = \perp \ \wedge \ f.j = 0 \longrightarrow d.j := d.g,$$
$$IB_2 : d.j \neq \perp \ \wedge \ f.j = 0 \longrightarrow f.j := 1.$$

A nongeneral process that has not yet decided copies the decision of the general. When a nongeneral process decides, it can finalize its decision.

2.2 Distribution Issues

Now, 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), it is straightforward to determine the variables that need to be changed in order to modify the state from s_0 to s_1. Specifically, if $x(s_0)$ denotes the value of x in state s_0 and $x(s_1)$ denotes the value of x in state s_1 then we say that (s_0, s_1) writes the value of x iff $x(s_0) \neq x(s_1)$. Thus, the write restrictions amount to ensuring that the transitions of a process only modify those variables that it can write.

More specifically, if process P_j can only write the variables in w_j and the value of a variable other than that in w_j is changed in the transition (s_0, s_1) then that transition cannot be used in obtaining the transitions of P_j. In other words, if P_j can only write variables in w_j, then P_j cannot use the transitions in $nw(w_j)$, where

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

Read restrictions. 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, and let their domain be $\{0,1\}$. Suppose that we have a process P_j 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 in the set of transitions of P_j iff the transition from $\langle a = 0, b = 1 \rangle$ to $\langle a = 1, b = 1 \rangle$ is also included in the set of transitions of P_j. If we were to include only one of these transitions, both a and b must be read. However, when these two transitions are grouped, the value of b is irrelevant and, hence, it need not be read.

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 $w_j \subseteq r_j$. 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

$$\begin{aligned} group(r_j)(s_0, s_1) = \{(s'_0, s'_1) : (\forall x : x \in r_j : x(s_0) = x(s'_0) \\ \wedge x(s_1) = x(s'_1)) \wedge (\forall x : x \notin r_j : x(s'_0) \\ = x(s'_1) \wedge \ x(s_0) = x(s_1))\}. \end{aligned}$$

In Section 4, we use the grouping of transitions, caused by the inability to read, to show that the problem of synthesizing failsafe fault-tolerance is NP-complete.

The read/write restrictions in the *IB* program. Each nongeneral non-Byzantine process P_j is allowed to only 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, $w_j \subseteq r_j$.

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. We refer the reader to [14], [12] where it is shown that it is possible to refine the specification, $spec$, of a program, p, that is not suffix-closed and/or fusion-closed into a specification $spec'$ by adding history variables where $spec'$ is both suffix and fusion-closed. In such refinement, p satisfies $spec$ from an initial state s_i iff the program p' satisfies $spec'$ from s'_i, where p' (respectively, s'_i) is the program (respectively, initial state) obtained after adding history variables to p.

Following Alpern and Schneider [1], we rewrite a specification as a conjunction of a safety specification and a liveness specification. Since the specification is suffix-closed, it is possible to represent the safety specification of a program as a set of bad transitions that the program is not allowed to execute. For reasons of space, we refer the reader to [14] (see p. 26, Lemma 3.6 of [14]) for the proof of this claim. Thus, for program p, its safety specification is a subset of $S_p \times S_p$. We do not explicitly specify the liveness specification as we show that the fault-tolerant program satisfies the liveness specification (in the absence of faults) iff the fault-intolerant program satisfies the liveness specification. Moreover, in the problem of synthesizing fault-tolerance, the initial fault-intolerant program satisfies its specification (including the liveness specification). Thus, the liveness specification need not be specified explicitly.

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 where S is true 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 an infinite sequence of states β such that $\alpha\beta \in spec$. We say that p maintains (does not violate) $spec$ from S iff 1) S is closed in p and 2) every computation prefix of p that starts in a state in S maintains $spec$. Note that the definition of *maintains* focuses on finite sequences of states, whereas the definition of *satisfies* concentrates on infinite sequences of states.

Also, 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. However, p may deadlock if it starts in a state that is not in S. Also, note that p is allowed to contain a self-loop of the form (s_0, s_0); i.e., it is permissible for p to reach s_0 and remain there forever.

Notation. Let $spec$ be a specification. We use the term safety of $spec$ to mean the smallest safety specification that includes $spec$. Also, whenever the specification is clear from the context, we will omit it; thus, S is an invariant of p abbreviates S is an invariant of p for $spec$.

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

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

Notation. In this section, we use process variables p and q to represent nongeneral processes in the quantifications.

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

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

Invariant of *IB*. In the absence of faults, no process is Byzantine. Also, a nongeneral is either undecided or has copied the decision of the general. If a nongeneral process has finalized then it must have decided. Thus, the invariant of program IB is S_{IB}, where

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

2.4 Faults

We systematically represent the faults that a program is subject to by a set of transitions. Thus, a class of fault f for program p 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. 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) $S \subseteq T$ (equivalently, $S \Rightarrow T$) and 2) T is closed in $p[\!]f$. Thus, at each state where an invariant S of p is true, an f-span T of p from S is also true. Also, T, similar to S, is closed in p. Moreover, if any transition in f is executed in a state where T is true, then T is also true in the resulting state. It follows 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.

As we defined a computation of p, we say that a sequence of states, $\sigma = \langle s_0, s_1, \ldots \rangle$, is a computation of p in the presence of f iff the following three conditions are satisfied: 1) if σ is infinite, then $\forall j : j > 0 : (s_{j-1}, s_j) \in (\delta_p \cup f)$; 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) $\exists n : n \geq 0 : (\forall j : j > n : (s_{j-1}, s_j) \in \delta_p)$. 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, it is not required that the fault transition 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. Although we assume that the number of fault occurrences is finite, our model permits unbounded occurrences of faults [15].

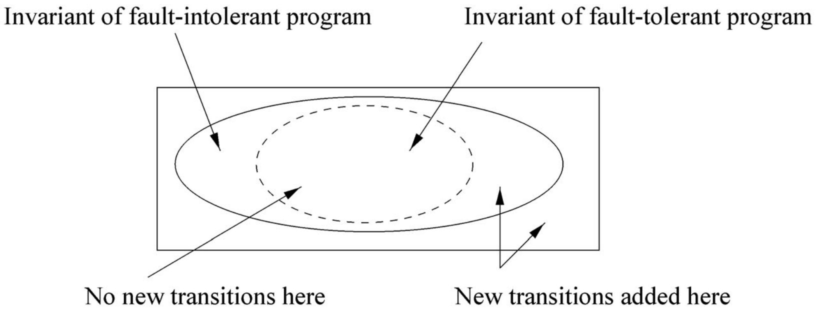

Invariant of fault-intolerant program
Invariant of fault-tolerant program
No new transitions here
New transitions added here

Fig. 1. The relation between the invariant of a fault-intolerant program p and a fault-tolerant program p'.

Byzantine faults. The Byzantine faults, f_B, can affect at most one process. A Byzantine process permanently remains Byzantine, and can change its decision arbitrarily. To model the effect of *permanent* faults—faults that permanently perturb the state of a program (e.g., Byzantine), we add new variables to a fault-intolerant program before synthesizing its fault-tolerant version. For example, we have added a local Boolean variable b to each process of the fault-intolerant program IB to represent whether or not that process is Byzantine. The following guarded command illustrates how we model the way a process becomes Byzantine.

$$F_1 : \neg b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l \quad \longrightarrow \quad b.j := true.$$

After a process becomes Byzantine (e.g., P_j), it may arbitrarily change the value of its decision and finalization variables.

$$F_2 : \; b.j \qquad\qquad \longrightarrow \quad d.j, f.j := 0|1, 0|1.$$

Although for the case of Byzantine faults we add new variables to model the effect of faults, there exist cases (e.g., transient faults) where we do not need add new variables.

2.5 Fault-Tolerance

In this section, we define what it means for a program to be failsafe/nonmasking/masking fault-tolerant. We have adapted the following definitions from [16].

Failsafe fault-tolerance. We say that p is failsafe f-tolerant (read as fault-tolerant) to $spec$ 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.

Nonmasking fault-tolerance. We say that p is nonmasking f-tolerant to $spec$ 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.

Masking fault-tolerance. We say that p is masking f-tolerant to $spec$ 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.

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 to $spec$ from S," and so on.

3 Problem Statement

In this section, we focus on the definition of synthesizing failsafe fault-tolerant programs from their fault-intolerant version. First, in Section 3.1, we formally state the synthesis problem. Then, in Section 3.2, we reiterate the result showed in [16] that the problem of synthesizing failsafe fault-tolerance is in NP. We reuse this result in Section 4 to show the NP-Completeness of synthesizing failsafe fault-tolerance.

3.1 Problem of Synthesizing Failsafe Fault-Tolerance

In this section, 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. Fig. 1).

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 the computations of p that start in a state in S (cf. Fig. 1). Hence, we must not introduce new transitions in the absence of faults. Thus, we define the problem (recalled from [16]) 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

```
Add_Failsafe_FT($f$ : transitions, $S$ : state predicate, $spec$ : specification,
                                   $g_0, g_1, ..., g_{max}$ : groups of transitions)
// groups of transitions $g_0, g_1, ..., g_{max}$ represent program $p$
{
    $ms := \{s_0 : \exists s_1, s_2, ...s_n : (\forall j : 0 \leq j < n : (s_j, s_{(j+1)}) \in f) \;\wedge\; (s_{(n-1)}, s_n)$ 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)  $S' \neq \{\}$; $S' \subseteq S$; $S'$ is closed in $p'$;      // $S'$ is a non-empty subset of $S$;
        (F2)  $S' \Rightarrow T'$; $T'$ is closed in $p'[]f$;              // $T'$ is a valid fault-span;
        (F3)  $(\forall s_0 : s_0 \in S' : (\exists s_1 :: (s_0, s_1) \in p'))$;  // Infinite computations guaranteed from $S'$;
        (F4)  $p'|S' \subseteq p|S'$;                             // No new transitions are added in $p'|S'$;
        (F5)  $T' \cap ms = \{\}$; $(p'|T') \cap mt = \{\}$;          // Safety is not violated from $T'$;
}
```

Fig. 2. The nondeterministic polynomial algorithm for adding failsafe fault-tolerance.

$S' \subseteq S$,
$p'|S' \subseteq p|S'$, and
p' is *failsafe* fault-tolerant to $spec$ from S'.

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.

Synthesizing failsafe fault-tolerant Byzantine generals. The problem of synthesizing a failsafe fault-tolerant version of the Byzantine generals program (presented in Section 2) amounts to finding a new invariant S'_{IB}, and calculating the set of transitions of a program FSB such that 1) $S'_{IB} \subseteq S_{IB}$, 2) $(FSB|S'_{IB}) \subseteq (IB|S'_{IB})$, and 3) FSB is failsafe f_B-tolerant to t_{sf} from S'_{IB}. In Section 6, we illustrate how we synthesize such a program.

3.2 Nondeterministic Algorithm

In this section, we reiterate the result presented in [16] that the problem of synthesizing failsafe fault-tolerant distributed programs is in NP. Since in Section 4 we use this result to show that indeed this problem is NP-complete, we represent a nondeterministic polynomial algorithm (Fig. 2) adapted from [16].

The algorithm Add_Failsafe_FT from [16] 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. Then, Add_Failsafe_FT calculates the set of ms states from where safety can be violated by the execution of fault transitions alone. Also, it computes the set of transitions mt that violate safety or reach a state in ms. Afterward, it nondeterministically guesses the fault-tolerant program, p', its invariant, S' and its fault-span, T'. Subsequently, it verifies that the synthesized (guessed) failsafe fault-tolerant program satisfies the three conditions of the synthesis problem (cf. Section 3.1).

Theorem 1. *The problem of synthesizing failsafe fault-tolerant distributed programs from their fault-intolerant version is in NP [16].*

4 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 in program state space. Toward this end, we reduce the 3-SAT problem to the problem of synthesizing failsafe fault-tolerance. In Section 4.1, we present the mapping of the given 3-SAT formula into an instance of the synthesis problem. Afterward, in Section 4.2, 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 recall the 3-SAT problem [11].

Proof The 3-SAT Problem. Given is a set of propositional variables, $b_1, b_2, \ldots, b_n$, and a Boolean formula $c = c_1 \wedge c_2 \wedge \ldots \wedge c_M$, where each c_j is a disjunction of exactly three literals.

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

4.1 Mapping 3-SAT to 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 and its complement $\neg b_i$, we introduce the following states (see Fig. 3): x_i, x'_i, a_i, y_i, y'_i, z_i, and z'_i.

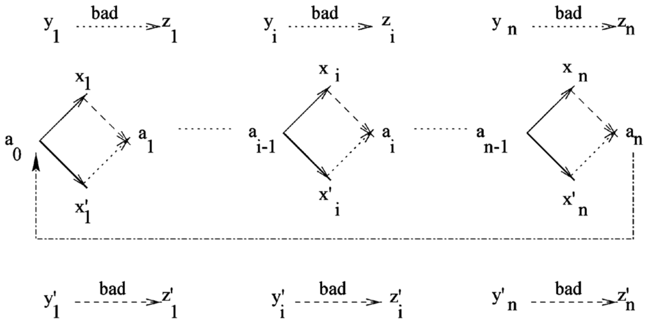


Fig. 3. The transitions corresponding to the propositional variables in the 3-SAT formula.

For each disjunction, $c_j = b_m \vee \neg b_k \vee b_l$ (cf. Fig. 4), 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 and its complement $\neg b_i$, we introduce the following transitions (cf. Fig. 3): (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).

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. Fig. 4): (c'_{jm}, d'_{jm}), (c_{jk}, d_{jk}), and (c'_{jl}, d'_{jl}).

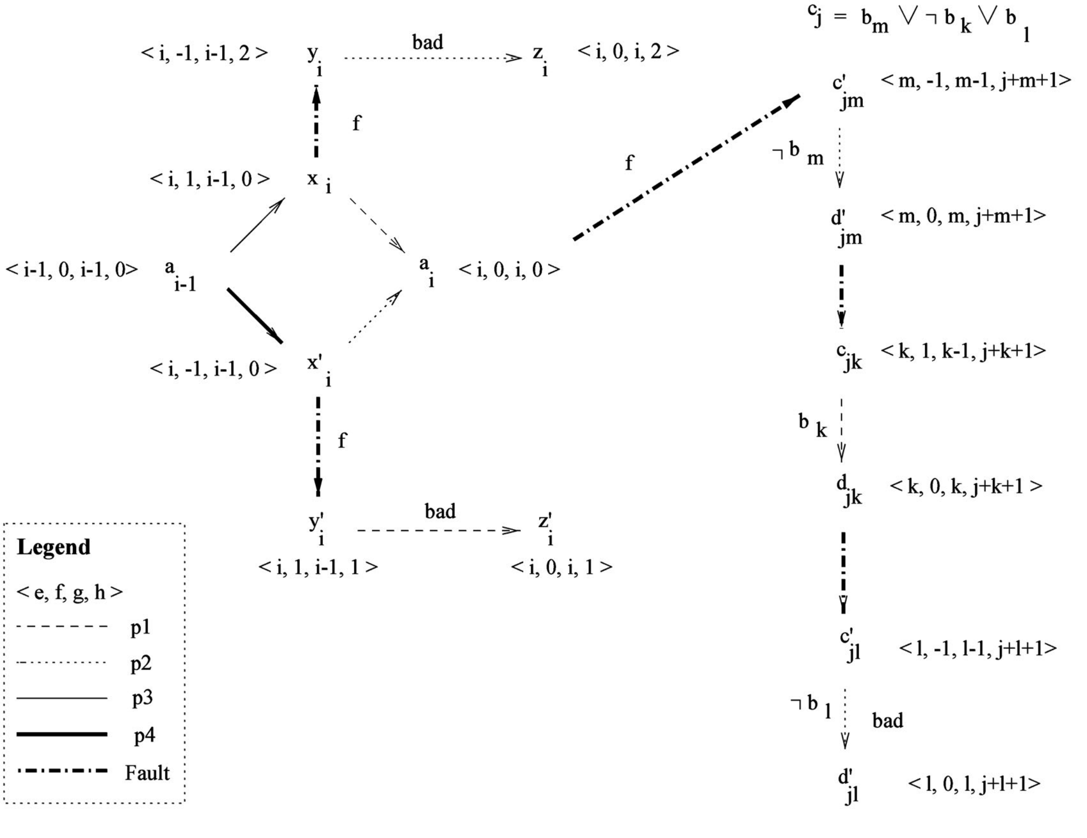
Fault transitions. We introduce the following fault transitions: From state x_i, the fault-intolerant program can reach y_i by the execution of faults. From state x'_i, the faults can perturb the program to state y'_i. Thus, for each propositional variable b_i and its complement $\neg b_i$, we introduce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

In addition, for each 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 transitions 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:

$$\{x_1, \cdots, x_n\} \cup \{x'_1, \cdots, x'_n\} \cup \{a_0, \cdots, a_n\}.$$

Safety specification of the fault-intolerant program. For each propositional variable b_i and its complement $\neg b_i$, 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. Fig. 4). 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. Now, 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. Fig. 4).



$c_j = b_m \vee \neg b_k \vee b_l$
< i, -1, i-1, 2 > y_i bad z_i < i, 0, i, 2 >
f
< i, 1, i-1, 0 > x_i
< i-1, 0, i-1, 0> a_{i-1}
a_i < i, 0, i, 0 >
< i, -1, i-1, 0 > x'_i
f
y'_i bad z'_i
< i, 1, i-1, 1 >
< i, 0, i, 1 >
f
c'_{jm} < m, -1, m-1, j+m+1>
$\neg b_m$
d'_{jm} < m, 0, m, j+m+1>
c_{jk} < k, 1, k-1, j+k+1>
b_k
d_{jk} < k, 0, k, j+k+1 >
c'_{jl} < l, -1, l-1, j+l+1>
$\neg b_l$ bad
d'_{jl} < l, 0, l, j+l+1>
Legend
< e, f, g, h >
p1
p2
p3
p4
Fault

Fig. 4. The structure of the fault-intolerant program for a propositional variable b_i and a disjunction $c_j = b_m \vee \neg b_k \vee b_l$.

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	$j+i+1$

Fig. 5. The value assignment to variables.

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 four 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. We have shown the value assignments in Fig. 5.

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 Fig. 4.

Observation 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.

Observation 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.

Observation 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 $\neg b_i$.

Observation 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.

Observation 5. (a_{i-1}, x_i) is grouped in P_3.

Observation 6. (a_{i-1}, x'_i) is grouped in P_4.

Observation 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).

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

4.2 The Reduction of 3-SAT

In this section, 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 4.1. Toward this end, we prove the following lemmas:

Lemma 1. *If the given 3-SAT formula is satisfiable, then there exists a failsafe fault-tolerant program that solves the instance of the synthesis problem identified in Section 4.1*

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:

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

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 3, as we include (x_i, a_i), we have to include (y'_i, z'_i). Also, based on Observation 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, 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 disjunction. 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 $S' \subseteq S$, $p' \mid S' \subseteq p \mid S'$, 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 2. *If there exists a failsafe fault-tolerant program that solves the instance of the synthesis problem identified in Section 4.1, 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 4.1. 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 $0 \leq i \leq n$, every a_i must be included in S'. For the same reason, since P_5 cannot execute from a_{i-1} (cf. Observation 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 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 1 and 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 2 and 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}), and (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 disjunction in the 3-SAT formula is true. □

Theorem 2. *The problem of synthesizing failsafe fault-tolerant distributed programs from their fault-intolerant version is NP-complete in program state space.*

5 Monotonic Specifications and Programs

Since the synthesis of failsafe fault-tolerance is NP-complete, as discussed in the Introduction, 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, 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 $p|S$.

We proceed as follows: In Section 5.1, we define the class of *monotonic specifications*, and the class of *monotonic programs*. 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 Section 5.2, we show the role of monotonicity restrictions imposed on specifications and programs in adding failsafe fault-tolerance. When these restrictions are satisfied, we show that the transitions in $p|S$ and the transitions grouped with them form the failsafe fault-tolerant program.

5.1 Sufficiency of Monotonicity

In this section, we identify sufficient conditions for polynomial-time synthesis of failsafe 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:*

$$\begin{aligned}
&\forall s_0, s_1, s'_0, s'_1 ::\\
&\quad x(s_0) = \mathit{false} \wedge x(s_1) = \mathit{false} \wedge x(s'_0) = \mathit{true} \wedge x(s'_1) = \mathit{true}\\
&\quad \wedge \text{ the value of all other variables in } s_0 \text{ and } s'_0 \text{ are the same}\\
&\quad \wedge \text{ the value of all other variables in } s_1 \text{ and } s'_1 \text{ are the same}\\
&\quad \wedge (s_0, s_1) \text{ does not violate } \mathit{spec} \wedge s_0 \in Y \wedge s_1 \in Y\\
&\quad \Rightarrow\\
&\qquad (s'_0, s'_1) \text{ does not violate } \mathit{spec}.
\end{aligned}$$

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.*

$$\begin{aligned}
&\forall s_0, s_1, s'_0, s'_1 ::\\
&\quad x(s_0) = \mathit{false} \wedge x(s_1) = \mathit{false} \wedge x(s'_0) = \mathit{true} \wedge x(s'_1) = \mathit{true}\\
&\quad \wedge \text{ the value of all other variables in } s_0 \text{ and } s'_0 \text{ are the same}\\
&\quad \wedge \text{ the value of all other variables in } s_1 \text{ and } s'_1 \text{ are the same}\\
&\quad \wedge (s_0, s_1) \in p|Y\\
&\quad \Rightarrow\\
&\quad (s'_0, s'_1) \in p|Y.
\end{aligned}$$

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 nonBoolean variables. One approach is to replace $x = \mathit{false}$ with $x = 0$ and $x = \mathit{true}$ with $x \neq 0$ in the above definition. In this case, the estimate for x is 0. We use this definition later in Section 5.2, where we identify the role of monotonicity in the complexity of synthesis.

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

$$\begin{aligned}
&\forall s_0, s_1 :: ((s_0, s_1) \in f \wedge (s_0, s_1) \text{ violates } \mathit{spec}) \Rightarrow\\
&(\forall s_{-1} :: (s_{-1}, s_0) \text{ violates } \mathit{spec}).
\end{aligned}$$

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 6). 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:

$$\begin{aligned}
&\{(s_0, s_1) : (s_0, s_1) \text{ violates } \mathit{spec}\\
&\vee (\exists s_2 :: (s_1, s_2) \in f \wedge (s_1, s_2) \text{ violates } \mathit{spec})\}.
\end{aligned}$$

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 3 and Corollary 1 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 3. *Given is a fault-intolerant program p, its invariant S, faults f and an f-$safe$ specification spec,*
If

$$\begin{aligned}
&\forall P_j, x : P_j \text{ is a process in } p, x \text{ is a Boolean variable such that}\\
&\qquad P_j \text{ cannot read } x : \mathit{spec} \text{ is positive monotonic on } S\\
&\qquad \text{with respect to } x\\
&\quad \wedge \text{ The program consisting of the transitions of } P_j\\
&\qquad \text{is negative monotonic on } S \text{ with respect to } x.
\end{aligned}$$

Then

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

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 3 as follows:

Corollary 1. *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 in the state space of p.

5.2 The Role of Monotonicity in the Complexity of Synthesis

In Section 5.1, 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 5 and 6, we show that if only one of these conditions is satisfied then the problem remains NP-complete.

Observation 5. Given is a fault-intolerant program p, its invariant S, faults f and an f-safe specification *spec*. If the monotonicity restrictions (from Corollary 1) are satisfied for p and no restrictions are imposed on the monotonicity of *spec* on S, 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 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 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 1 holds for the program and no assumption is made about the monotonicity of the specification. Therefore, based on Lemmas 1 and 2, the proof follows. □

Furthermore, the specification obtained by mapping the 3-SAT problem in Section 4 is negative monotonic with respect to h. Hence, similar to Observation 5, we have:

Observation 6. Given is a fault-intolerant program p, its invariant S, faults f, and an f-safe specification *spec*. If the monotonicity restrictions (from Corollary 1) 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 5. □

Based on the above discussion, it follows that the monotonicity of both programs and specifications is necessary in the proof of Theorem 3.

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 [9], 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.

6 Examples of Monotonic Specifications

In this section, we present an example of the application of the monotonicity theorem (Theorem 3) for adding failsafe fault-tolerance to the Byzantine generals problem presented in Section 2. Specifically, we show the monotonicity property of the Byzantine generals program and its specification with respect to appropriate variables. We have also shown in [18] that the consensus and atomic commit programs and their specifications meet the monotonicity requirements. Thus, failsafe fault-tolerance to fail-stop faults can be added to these programs in polynomial time. Now, we make the following observations about the monotonicity of the specification and the program of the Byzantine generals problem.

Observation 7. The specification of Byzantine generals is positive monotonic on S_{IB} 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 P_j, where validity and agreement are not violated even though P_k is not Byzantine. Let (s'_0, s'_1) be the corresponding transition where P_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. □

Likewise, we can make the following observations:

1. the specification of Byzantine generals problem is negative monotonic on S_{IB} with respect to $f.k$ (respectively, $f.j$ and $f.l$);
2. the program IB_j, consisting of the transitions of P_j, with invariant S_{IB} is negative monotonic on S_{IB} with respect to $b.k$ (respectively, $b.j$ and $b.l$);
3. the program IB_j, consisting of the transitions of P_j, with invariant S_{IB} is positive monotonic on S_{IB} with respect to $f.k$ (respectively, $f.j$ and $f.l$), and
4. the specification of Byzantine generals problem is f_B-safe.

Theorem 4. *Failsafe fault-tolerant Byzantine generals program can be obtained in polynomial time in the state space of the IB program.*

To obtain the failsafe fault-tolerant program, based on Theorem 3, 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 nongeneral process P_j in the fault-tolerant program are as follows:

$$FSB_1 : d.j{=}\bot \wedge f.j = 0 \quad \longrightarrow \quad d.j := d.g$$
$$FSB_2 : (d.j = 0) \wedge ((d.k{\neq}1) \wedge (d.l{\neq}1)) \wedge f.j{=}0 \quad \longrightarrow \quad f.j{:=}1$$
$$FSB_3 : (d.j = 1) \wedge ((d.k{\neq}0) \wedge (d.l{\neq}0)) \wedge f.j{=}0 \quad \longrightarrow \quad f.j{:=}1.$$

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

7 Extension: 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). Toward this end, first, we state the enhancement problem in Section 7.1. Then, in Section 7.2, we present a theorem that identifies the sufficient conditions for enhancing the fault-tolerance of nonmasking programs in polynomial time. Finally, in Section 7.3, we present an example to illustrate the application of the theorem presented in Section 7.2.

7.1 The Enhancement Problem

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 fault-tolerant program, p, its invariant, S, faults, f, and specification, $spec$. Given p, S, and f, we 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 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' needs to satisfy both its safety and liveness specification in the absence and in the presence of faults.

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 is 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. Toward 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, this is the second requirement of the enhancement problem. We state the enhancement problem 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 to $spec$ from T'.

7.2 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 in program state space. Specifically, 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 anonmasking 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 nonprogress cycles in $(T - S)$. When we remove a nonprogress cycle, we may create more deadlock states. Thus, 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.*

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

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. To achieve this goal, 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.*

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.

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.*

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. *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

$$\forall P_j, x : P_j \text{ is a process in } p, x \text{ is a Boolean variable such that}$$
$$P_j \text{ cannot read } x : spec \text{ is independent of } x \text{ on } T$$
$$\wedge \text{ The program consisting of the transitions of } P_j$$
$$\text{is independent of } x \text{ on } S.$$

Then,

$$\text{A masking fault-tolerant program } p' \text{ can be derived from}$$
$$p \text{ in polynomial time in the state space of } p.$$

Proof. Let (s_0, s_1) be a transition of process P_j. We consider two cases 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 in 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 at least a safe transition (s_0, s_2) (respectively, (s'_0, s'_2)) that guarantees 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.

Based on the above discussion, we have shown that p' satisfies *spec* from S, every computation prefix of $p'[]f$ maintains *spec*, and starting from every state in T, every computation of $p[]f$ will reach a state in S. Therefore, p' is masking f-tolerant to *spec* from S. □

7.3 Example

In this section, we present an example for enhancing the fault-tolerance of nonmasking distributed programs to masking using the monotonicity property. Toward 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.

Nonmasking program. The nonmasking program p represents an even counter. The program p consists of two processes, namely, P_0 and P_1. Process 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. 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.

$$\begin{array}{lll} (x_1 = 1) \wedge (x_0 = 0) & \longrightarrow & x_1 := 0 \\ x_1 = 0 & \longrightarrow & x_1 := 1. \end{array}$$

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 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}. Let (s_0, s_1) and (s'_0, s'_1) be two arbitrary transitions of P_0 that are grouped due to inability of P_0 to read x_1. First, since there is no transition (s_0, s_1) in $p|S_{ctr}$, where $(x_1(s_0) = 1)$ and $(x_1(s_1) = 1)$, p is negative monotonic on S_{ctr} with respect to x_1. A similar argument shows that p is positive monotonic on S_{ctr} with respect to x_1. Therefore, p is independent of x_1 on S_{ctr}.

$spec_{ctr}$ is independent of x_1 on T_{ctr}. Let (s_0, s_1) and (s'_0, s'_1) be two arbitrary transitions of P_0 that are grouped due to inability of P_0 to read x_1. Consider the case where $(x_1(s_0) = 0)$ and $(x_1(s_1) = 0)$, and (s_0, s_1) does not violate safety. Since $(x_1(s'_0) = 1)$ and $(x_1(s'_1) = 1)$, the transition (s'_0, s'_1) preserves safety as well (because the value of x_1 does not change during this transition). Hence, $spec_{ctr}$ is positive monotonic on T_{ctr} with respect to x_1. A similar argument shows that $spec_{ctr}$ is negative monotonic on T_{ctr} with respect to x_1. Therefore, $spec_{ctr}$ 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, we can derive a masking fault-tolerant version of p in polynomial time. In the synthesized masking program, the action of P_0 remains as is, and the actions of P_1 are as follows:

$$\begin{array}{lll} (x_1 = 1) \wedge (x_0 = 0) & \longrightarrow & x_1 := 0, \\ (x_1 = 0) \wedge (x_0 = 0) & \longrightarrow & x_1 := 1. \end{array}$$

Note that the second action of P_1 can only be executed when the program is in its invariant; i.e., $(x_0 = 0)$.

8 Concluding Remarks and Future Directions

In this paper, we focused on the problem of synthesizing failsafe fault-tolerant distributed programs from their fault-intolerant version. We showed, in Section 4, a counterintuitive result that the problem of synthesizing failsafe fault-tolerant distributed programs from their fault-intolerant version is NP-complete in program state space. Toward this end, we reduced the 3-SAT problem to the problem of synthesizing failsafe fault-tolerance. Moreover, we identified monotonicity requirements (cf. Section 5) that are sufficient for polynomial-time synthesis of failsafe fault-tolerant distributed programs. Finally, 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 (in program state space).

The results of this paper differ from [16] in three ways. For one, the proof in [16] is for masking fault-tolerance where both safety and liveness need to be satisfied. By contrast, the NP-completeness in this paper applies to the class where only safety is satisfied. Also, the proof in [16] relies on the ability of a process to *blindly* write some variables. By contrast, the proof in this paper does not rely on such an assumption. Furthermore, in this paper, we identified sufficient conditions (i.e., monotonicity of programs and specifications (cf. Section 5)) for polynomial-time synthesis of failsafe fault-tolerant distributed programs.

To extend the scope of synthesis, we have developed the extensible software framework Fault-Tolerance Synthesizer (FTSyn) [8] where developers can automatically add 1) fault-tolerance to distributed programs and 2) new heuristics for reducing the time complexity of synthesis. Using FTSyn, we have synthesized several programs (e.g., distributed token ring, Byzantine agreement, diffusing computation) among which a simplified version of an aircraft altitude switch [8]. We are currently investigating the application of state space reduction techniques (used in model checking) in dealing with the state space explosion problem in FTSyn. Such techniques will help FTSyn to deal with the space complexity of synthesis along with the integrated heuristics that reduce the time complexity of synthesis.

ACKNOWLEDGMENTS

A preliminary version of this paper has appeared in a conference proceeding [18]. This work was partially sponsored by US National Science Foundation CAREER Grant CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, and a grant from Michigan State University.

REFERENCES

[1] B. Alpern and F.B. Schneider, "Defining Liveness," *Information Processing Letters,* vol. 21, pp. 181-185, 1985.

[2] A. Arora and M.G. Gouda, "Closure and Convergence: A Foundation of Fault-Tolerant Computing," *IEEE Trans. Software Eng.,* vol. 19, no. 11, pp. 1015-1027, 1993.

[3] A. Arora and S.S. Kulkarni, "Detectors and Correctors: A Theory of Fault-Tolerance Components," *Proc. Int'l Conf. Distributed Computing Systems,* pp. 436-443, May 1998.

[4] P. Attie and A. Emerson, "Synthesis of Concurrent Programs for an Atomic Read/Write Model of Computation," *ACM Trans. Programming Languages and Systems,* vol. 23, no. 2, Mar. 2001.

[5] M. Barborak, A. Dahbura, and M. Malek, "The Consensus Problem in Fault-Tolerant Computing," *ACM Computing Surveys,* vol. 25, no. 2, pp. 171-220, 1993.

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

[7] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper, "Muteness Failure Detectors: Specification and Implementation," *Proc. European Dependable Computing Conf.,* pp. 71-87, 1999.

[8] A. Ebnenasir and S.S. Kulkarni, "FTSyn: A Framework for Automatic Synthesis of Fault-Tolerance," Technical Report MSU-CSE-03-16, Computer Science and Eng. Dept., Michigan State Univ., East Lansing, July 2003.

[9] A. Ebnenasir and S.S. Kulkarni, "Efficient Synthesis of Failsafe Fault-Tolerant Distributed Programs," Technical Report MSU-CSE-05-13, Computer Science and Eng. Dept., Michigan State Univ., East Lansing, Apr. 2005.

[10] M.J. Fischer, N.A. Lynch, and M.S. Peterson, "Impossibility of Distributed Consensus with One Faulty Processor," *J. ACM,* vol. 32, no. 2, pp. 373-382, 1985.

[11] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[12] F.C. Gärtner and A. Jhumka, "Automating the Addition of Failsafe Fault-Tolerance: Beyond Fusion-Closed Specifications," *Proc. Conf. Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT),* Sept. 2004.

[13] L. Gong, P. Lincoln, and J. Rushby, "Byzantine Agreement with Authentication: Observations and Applications in Tolerating Hybrid and Link Faults," *Proc. Dependable Computing for Critical Applications-5,* pp. 139-157, Sept. 1995.

[14] S.S. Kulkarni, "Component-Based Design of Fault-Tolerance," PhD thesis, Ohio State Univ., http://www.cse.msu.edu/~sandeep/dissertation/dissertation.ps, 1999.

[15] S.S. Kulkarni and A. Arora, "Compositional Design of Multitolerant Repetitive Byzantine Agreement," *Proc. 17th Int'l Conf. Foundations of Software Technology and Theoretical Computer Science,* pp. 169-183, Dec. 1997.

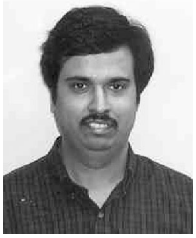
[16] S.S. Kulkarni and A. Arora, "Automating the Addition of Fault-Tolerance," *Proc. Sixth Int'l Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems,* pp. 82-93, 2000.

[17] S.S. Kulkarni, A. Arora, and A. Chippada, "Polynomial Time Synthesis of Byzantine Agreement," *Proc. 20th IEEE Symp. Reliable Distributed Systems,* pp. 130-140, 2001.

[18] S.S. Kulkarni and A. Ebnenasir, "The Complexity of Adding Failsafe Fault-Tolerance," *Proc. Int'l Conf. Distributed Computing Systems,* pp. 337-344, 2002.

[19] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. Programming Languages and Systems,* vol. 4, no. 3, pp. 382-401, July 1982.

[20] M. Singhal and N. Shivaratri, *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems.* McGraw-Hill, 1994.



Sandeep S. Kulkarni received the BTech degree in computer science and engineering from Indian Institute of Technology, Mumbai, India in 1993. He received the MS and PhD degrees in computer and information science from Ohio State University, Columbus, in 1994 and 1999, respectively. He has been working as an assistant professor at Michigan State University, East Lansing, since August 1999. He is a member of the Software Engineering and Network Systems (SENS) Laboratory. He is a recipient of the US National Science Foundation CAREER award. His research interests include fault-tolerance, distributed systems, group communication, security, self-stabilization, compositional design, and automated synthesis.

Ali Ebnenasir received the BE degree in computer engineering from the University of Isfahan, Iran, in 1994. He received the MS degree from Iran University of Science and Technology, Tehran, in 1998. He received the PhD degree in computer science and engineering from Michigan State University, in 2005. His research interests include fault-tolerance, distributed systems, security, compositional design, and automated synthesis.