The Complexity of Adding Failsafe Fault-Tolerance[1]

Sandeep S. Kulkarni Ali Ebnenasir
Department of Computer Science and Engineering
Michigan State University
East Lansing MI 48824 USA

Abstract

In this paper, we focus our attention on the problem of automating the addition of 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. And, in the presence of faults, it satisfies its safety specification. We present a somewhat unexpected result that, in general, the problem of adding failsafe fault-tolerance in distributed programs is NP-hard. Towards this end, we reduce the 3-SAT problem to the problem of adding failsafe fault-tolerance. We also identify a class of specifications, monotonic specifications *and a class of programs,* monotonic programs*. Given a monotonic specification and a monotonic program, we show that failsafe fault-tolerance can be added in polynomial time. We note that the monotonicity restrictions are met for commonly encountered problems such as Byzantine agreement, distributed consensus, and atomic commitment. Finally, we argue that the restrictions on the specifications and programs are necessary to add failsafe fault-tolerance in polynomial time; we prove that if only one of these conditions is satisfied, the addition of failsafe fault-tolerance is still NP-hard.*

Keywords : Fault-tolerance, Formal methods, Program synthesis, Program transformation, Distributed programs

1 Introduction

We focus on the automation 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 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. Therefore, it would be possible to add fault-tolerance even to programs for which the entire specification is not available or where the existing program is the de-facto specification. Third, in this approach, the concerns of the functionality of a program and its fault-tolerance are separated. This separation is known to help [1] in simplifying the reuse of the techniques used in *manually* adding 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 [2], Kulkarni and Arora showed that the problem of adding masking fault-tolerance –where both safety and liveness are satisfied in the presence of faults– is NP-hard. We find that there are three possible options to deal with this complexity: (1) develop heuristics under which the synthesis algorithm takes polynomial time, (2) consider a weaker form of fault-tolerance such as failsafe –where only safety is satisfied in the presence of faults, or nonmasking -where safety may be violated temporarily if faults occur, or (3) identify a class of specifications and programs for which the addition of fault-tolerance can be performed in polynomial time.

The first approach was used in [3] where Kulkarni, Arora and Chippada presented heuristics that are applicable to several problems including byzantine agreement. In polynomial time, their algorithm finds a fault-tolerant program or it declares that a fault-tolerant program cannot be synthesized.

In this paper, we focus on the other two approaches. Regarding the second approach, we focus our attention on the design of failsafe fault-tolerance. By adding failsafe fault-tolerance in an automated fashion, we can 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 [1] can be combined to partially automate the design of masking fault-tolerant programs. The algorithm in [1] shows how a masking fault-tolerant program can be designed 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 failsafe fault-tolerance, we can automate one step of designing masking fault-tolerance.

In our investigation, we find that the design of distributed failsafe fault-tolerant programs is also NP-hard. (In [2], it was conjectured that the addition of failsafe fault-tolerance 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

[1] Email: sandeep@cse.msu.edu, ebnenasi@cse.msu.edu
Web: `http://www.cse.msu.edu/~{sandeep,ebnenasi}`
Tel: +1-517-355-2387
This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, and a grant from Michigan State University.

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. Towards this end, we identify a class of specifications, namely *monotonic specifications*, and a class of programs, namely *monotonic programs*. Given a monotonic specification and a monotonic program, we show that failsafe fault-tolerance can be added in polynomial time. Finally, we note that the class of monotonic specifications contains well-recognized [4–6] problems of distributed consensus, atomic commitment, and byzantine agreement.

We also argue that the restrictions imposed on the specification and the fault-intolerant program are necessary. More specifically, we show that if restrictions are imposed only on the specification (respectively, the fault-intolerant program), then the problem of adding failsafe fault-tolerance is still NP-hard.

Organization of the paper. This paper is organized as follows: In Section 2, we provide a few 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. In Section 5, we precisely define the notion of monotonic specifications and monotonic programs, and show their necessity and sufficiency for adding failsafe fault-tolerance in polynomial time. Finally, we make concluding remarks in Section 6.

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 [7]. The definition of faults and fault-tolerance is adapted from Arora and Gouda [8] and Kulkarni [1]. The issues of modeling distributed programs is adapted from [2]. A similar modeling of distributed programs in read/write atomicity was independently identified by Attie and Emerson [9].

Most definitions in this section are straightforward, and are included to make the paper self-contained. A reader who is familiar with this area can skip this section if necessary. We ask the reader to carefully look how distribution is modeled (cf. Section 2.2), how safety specification is specified (cf. second paragraph in Section 2.3) and the definition of failsafe fault-tolerance (cf. fourth paragraph in Section 2.4).

2.1 Program

A program p is a set of finite variables and a set of finite processes. Each variable is associated with a finite domain of values. Let $v_1, v_2, .., v_n$ be variables of p, and let $D_1, D_2, .., D_n$ 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 is of the form: $\langle l_1, l_2, .., l_n \rangle$ where $\forall i : 1 \leq i \leq n : l_i \in D_i$. The state space of p, S_p, is the set of all possible states of p.

A process, say j, in p is associated with a set of program variables, say r_j, it can read and a set of variables, say w_j, that it can write.[2] Also, process 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. Hence, unless we need to talk about the transitions of a particular process or the values of the particular variables, we simply let program p be the tuple $\langle S_p, \delta_p \rangle$ where S_p is a finite set of states and δ_p is a subset of $\{(s_0, s_1) \mid s_0, s_1 \in S_p\}$.

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 $(\forall (s_0, s_1) : (s_0, s_1) \in \delta_p : (s_0 \in S \Rightarrow s_1 \in S))$. A sequence of states, $\langle s_0, s_1, ... \rangle$, is a computation of p iff the following two conditions are satisfied: (1) $\forall j : j > 0 : (s_{j-1}, s_j) \in \delta_p$, and (2) if $\langle s_0, s_1, ... \rangle$ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in \delta_p$.

The projection of program p on state predicate S, denoted as $p|S$, is the program $\langle S_p, \{(s_0, s_1) : (s_0, s_1) \in \delta_p \ \wedge \ s_0, s_1 \in S\} \rangle$. I.e., $p|S$ consists of transitions of p that start in S and end in S. 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 context, we use p and δ_p interchangeably. Also, we say that a state predicate S is true in a state s iff $s \in S$.

2.2 Issues of Distribution

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. 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 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 j. In other words, if j can only write variables in w_j then j cannot use the transitions in $nw(j, w_j)$, where

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

Read restrictions. Given a single transition (s_0, s_1), it appears that all the variables must be read in order for that transition to be executed. 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, and let their domain be $\{0, 1\}$. 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, we would need to read both a and b. However, when these

[2] For this paper, we assume that $w_j \subseteq r_j$, i.e., j cannot *blindly* write any variable. A more general case is discussed in [2]; we omit it here as this simple case suffices for this paper.

two transitions are grouped, the value of b is irrelevant and, hence, we do not need to read it.

More generally, consider the case where r_j is the set of variables that j can read, w_j is the set of variables that j can write, and $w_j \subseteq r_j$. Now, process j can include the transition (s_0, s_1) iff it 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. We define these transitions as $group(j, r_j)(s_0, s_1)$ for the case $w_j \subseteq r_j$, where

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

The grouping of transitions caused by the inability to read is used in Section 4 to show that the problem of adding fault-tolerance is NP-hard.

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.

Following Alpern and Schneider [7], we let the specification consist of a safety specification and a liveness specification. For the problem of adding failsafe fault-tolerance, the safety specification is specified in terms of a set of bad transitions that should not occur in the program computation. I.e., for program p, its safety specification is a subset of $\{(s_0, s_1) : s_0, s_1 \in S_p\}$. The liveness specification is not specified in our algorithm; 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 adding fault-tolerance, the initial fault-intolerant program satisfies its specification (including the liveness specification). Thus, the liveness specification need not be specified explicitly.

Since the specification is suffix closed, it is always possible to specify the safety specification as a set of bad transitions. For reasons of space, we refer the reader to [1] for the proof of this claim. We also refer the reader to [1] where we show that it is possible to convert a set of state sequences that is not suffix closed and/or fusion closed into an equivalent set that is suffix closed and fusion closed.

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 a sequence of states β such that $\alpha\beta \in spec$.

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*.

2.4 Faults

The faults that a program is subject to are systematically represented by transitions. A fault f for program $p (= \langle S_p, \delta_p \rangle)$ is a subset of the set $\{(s_0, s_1) : s_0, s_1 \in S_p\}$. 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) $S \Rightarrow T$ 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.

Just as we defined the computation of p, we say that a sequence of states, $\langle s_0, s_1, ... \rangle$, is a computation of p in the presence of f iff the following three conditions are satisfied: (1) $\forall j : j > 0 : (s_{j-1}, s_j) \in (\delta_p \cup f)$, (2) if $\langle s_0, s_1, ... \rangle$ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in \delta_p$, and (3) $\exists n : n \geq 0 : (\forall j : j > n : (s_{j-1}, s_j) \in \delta_p)$. 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.

Using the above definitions, we now define what it means for a program to be failsafe fault-tolerant. We say that p is failsafe f-tolerant (read as fault-tolerant) to *spec* from S iff the following two conditions hold:

- p satisfies *spec* from S, and
- there exists T such that T is an f-span of p from S and $p[]f$ maintains *spec* from T. □

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); we use such a self-loop whenever s_0 is an *accaptable 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", and so on.

3 Problem Statement

In this section, we formally state the problem of adding fault-tolerance. During automated addition of fault-tolerance, we begin with the fault-intolerant program, its invariant, faults and the safety specification that needs to be satisfied in the presence of faults. The goal is to *only* add failsafe fault-tolerance to develop a program that *reuses* the 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, 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 computations of p that start from a state in S satisfy the specification, $spec$. Since we have no knowledge about 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 1).

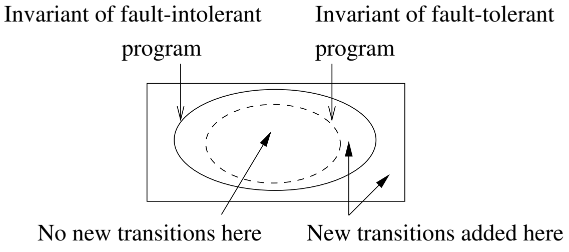
Invariant of fault-intolerant program
Invariant of fault-tolerant program
No new transitions here
New transitions added here

Figure 1. The relation between the invariant of a fault-intolerant and a fault-tolerant program

Likewise, to show that p' is correct in the absence of faults, we need to show that 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 1). Hence, we must not introduce new transitions in the absence of faults. Thus, we define the problem of adding failsafe fault-tolerance as follows (To obtain this problem statement, we have tailored the problem statement in [2] to deal with failsafe fault-tolerance.):

The Addition Problem

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'. □

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

4 NP-Completeness proof

In this section, we prove that the problem of adding failsafe fault-tolerance is NP-hard. Towards this end, we reduce 3-SAT to the problem of adding failsafe fault-tolerance. First, we present 3-SAT problem and then identify the mapping between 3-SAT and the decision problem in Section 3.

3-SAT problem.

Given is a set of literals, $x_1, x_2, ..., x_n$ and $x'_1, x'_2, ..., x'_n$, where x_i and x'_i are complements of each other, 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 $x_1, x_2, ..., x_n$ such that c is satisfiable?

Mapping 3-SAT to the problem of adding failsafe fault-tolerance. To map the given 3-SAT formula into an instance of the addition problem, we identify corresponding state to each literal and disjunction, the transitions, and the invariant of the fault-intolerant program. Subsequently, we identify the safety specification, and the value assignment to variables. Finally, we show that the 3-SAT formula is satisfiable iff failsafe fault-tolerance can be added to this instance of the addition problem.

The states of the fault-intolerant program. Corresponding to each literal x_i and its complement we introduce the following states (see Figure 2):

- $x_i, x'_i, a_i, y_i, y'_i, z_i, z'_i$

For each disjunction, $c_j = x_m \vee x'_k \vee x_l$, we introduce the following states:

- $c'_{jm}, d'_{jm}, c_{jk}, d_{jk}, c'_{jl}, d'_{jl}$

The transitions of the fault-intolerant program. Corresponding to each literal x_i and its complement x'_i, we introduce the following transitions (cf. Figure 2):

- $(a_{i-1}, x_i), (x_i, a_i), (y'_i, z'_i)$
- $(a_{i-1}, x'_i), (x'_i, a_i), (y_i, z_i)$

Corresponding to each $c_j = x_m \vee x'_k \vee x_l$, we introduce the following program transitions (Wlog, we assume that c_j does not include both x_i and x'_i):

- $(c'_{jm}, d'_{jm}), (c_{jk}, d_{jk}), (c'_{jl}, d'_{jl})$

In the reduction from the 3-SAT problem, the transition (c'_{jm}, d'_{jm}) is included iff x_m is false, and the transition (c_{jk}, d_{jk}) is included iff x_k is true. Thus, if c_j evaluates to true then at least one of the transitions introduced for c_j is not included. We choose the safety specification in such a way that it is violated iff all three transitions that correspond to any disjunction are included. Correspondingly, the truth value of x_i will be decided based upon whether the transition (a_{i-1}, x_i) is included or whether transition (a_{i-1}, x'_i) is included. By choosing the safety specification appropriately, we ensure that both these transitions are not included.

Fault transitions. For each literal x_i and its complement x'_i, we introduce the following fault transitions:

- $(x_i, y_i), (x'_i, y'_i)$

For each disjunction $c_j = (x_m \vee x'_k \vee x_l)$, 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 transitions that perturb 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:

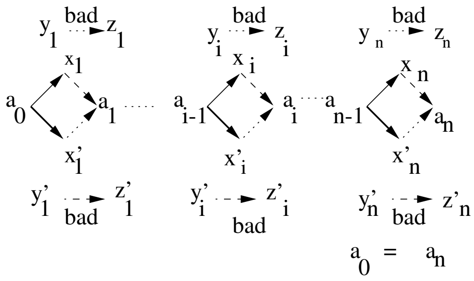

Figure 2. The transitions corresponding to the literals in the 3-SAT formula

- (a_i, c'_{jm}) and $(d'_{jm}, c_{jk}), (d_{jk}, c'_{jl})$

Note that the fault transition can perturb the program from state a_i only to the *first* state introduced for c_j.

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 literal x_i and its complement x'_i, the following two transitions violate the safety specification (cf. Figure 3).

- (y_i, z_i), (y'_i, z'_i)

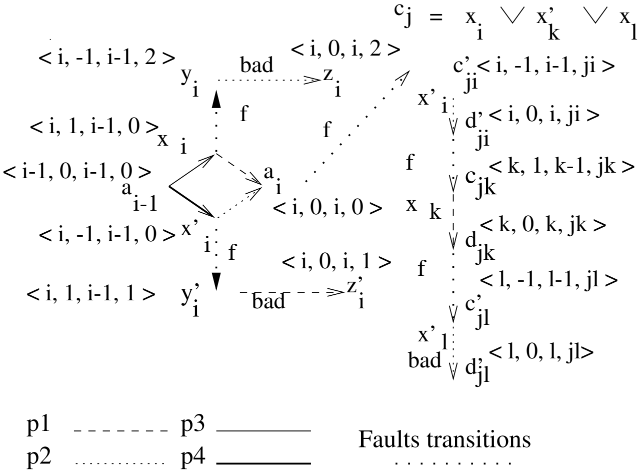

Figure 3. The structure of the fault-intolerant program for a literal x_i. 4-tuples represent the values of variables $< e, f, g, h >$ in each state.

For each disjunction $c_j = x_m \vee x'_k \vee x_l$, the following transition violates the safety specification (Note that only the *last* program transition added for c_j violates the specification).

- (c'_{jl}, d'_{jl})

Observe that from state a_i, the fault can perturb the program to the state c_{jm}. Now, if all three program transitions corresponding to c_j are included, the safety may be violated by the execution of program and fault actions (cf. Figure 3).

Variables and Their Values. The fault-intolerant program has 4 variables: *e, f, g,* and *h*. The domains of these variables are respectively $\{0, .., n\}$, $\{-1, 0, 1\}$, $\{0, .., n\}$, and $\{0, .., M + n + 1\}$. For the states introduced for the literals x_i and x'_i, the value assignments are as follows:

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

For the states introduced for the disjunction c_j, the variable values are as follows (Recall that state c'_{ji} (respectively, c_{ji}) is introduced if c_j contains x_i (respectively, x'_i).):

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	$j+i+1$

Processes and read/write restrictions. The fault-intolerant program consists of four processes, P_1, P_2, P_3, and P_4. The read/write restrictions on these processes are as follows: i) 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*, ii) 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*.

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 (cf. Figure 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. □

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. □

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 (or P_2). Moreover, this group also includes the transition (c_{ji}, d_{ji}) for each c_j that includes x'_i. □

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 x_i. □

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

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

For i, $1 \leq i \leq n$, the set of transitions for each process is the union of the transitions mentioned above. Now that we have identified the fault-intolerant program, say p, as the union of these four processes, we show that 3-SAT has a satisfying truth value assignment if and only if there exists a failsafe fault-tolerant program derived from p. Towards this end, we prove the following two lemmas:

Lemma 4.7 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 earlier.

Proof. Since the 3-SAT formula is satisfiable, there exists an assignment of truth values to the literals x_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:

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

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

- For each literal x_i, $1 \leq i \leq n$, if x_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). And, for each disjunction c_j that includes x'_i, we have to include the transition (c_{ji}, d_{ji}). As an illustration, we have shown the partial mapping when $x_1 = true$, $x_2 = false$, and $x_3 = true$ in Figure 4.

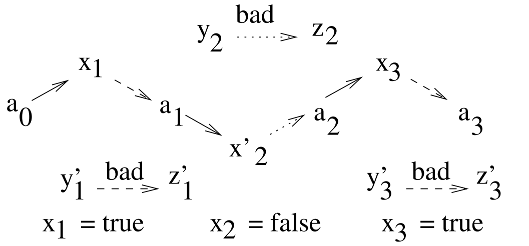

Figure 4. The partial structure of the fault-tolerant program

- For each literal x_i, $1 \leq i \leq n$, if x_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). For each disjunction c_j that includes x_i, we have to include the transition (c'_{ji}, d'_{ji}).

Now, we show that the safety violating transitions are not executed by p', even if faults occur.

- *Safety violating transitions related to x_i* . If x_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, the p' cannot reach state y'_i. It follows that p' cannot execute the transition (y'_i, z'_i). By the same argument, the transition (y_i, z_i) is also not executed in p'.
- *Safety violating transitions related to disjunction c_j*. Since the 3-SAT formula is satisfiable, every disjunction in the formula is true. Let $c_j = x_m \vee x'_k \vee x_l$. Wlog, let x_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 addition problem. □

Lemma 4.8 If there exists a failsafe fault-tolerant program that solves the instance of the addition problem identified earlier 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 earlier in this section. 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 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, one of the transitions (a_{i-1}, x_i) or (a_{i-1}, x'_i) should be in p'. If p' includes (a_{i-1}, x_i) we set $x_i = true$ in the 3-SAT formula. If p' contains the transition (a_{i-1}, x'_i), we set $x_i = false$. Now, we show that this value assignment is consistent and each c_j is true.

- *Each literal gets a unique truth assignment.* Suppose that there exists a literal x_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 (4.1–4.4), the transitions $(a_{i-1}, x_i), (x_i, a_i)$, (y'_i, z'_i) and the transitions $(a_{i-1}, x'_i), (x'_i, a_i)$, (y_i, z_i) must 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 $c_j = x_i \vee x'_k \vee x_l$, which is not true. Therefore, $x_i = false$, $x_k = true$, and $x_l = false$. Based on the grouping discussed earlier, the transitions $(c'_{ji}, d'_{ji}), (c_{jk}, d_{jk}), (c'_{jl}, d'_{jl})$ 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. □

5 Monotonic Specifications and Programs: Necessity and Sufficiency

Since the addition of failsafe fault-tolerance is NP-hard, as discussed in the Introduction, we focus on this question: *What restrictions can be imposed on the 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, a group of transitions g may include a transition within the invariant of the fault-intolerant program and a transition that violates safety, together. To add failsafe fault-tolernace we have to determine whether g should be included. This issue is one of the reasons behind the complexity of adding safety.

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, and hence, p' will satisfy its specification from S. 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 first define a class of specifications, *monotonic specifications*, and a class of programs, *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. Then, we introduce the concept of *fault-safe specifications*. Subsequently, we argue that the monotonicity restrictions imposed on specifications and programs are sufficient and necessary for adding failsafe fault-tolerance in polynomial time.

Consider the case where process j cannot read the value of a boolean variable x. The definition of (positive) monotonicity captures the case where j can safely assume that x is false. Thus, we define monotonic specification as follows:

Definition. A specification *spec* is *positive monotonic* with respect to a boolean variable x iff the following condition is satisfied:

$\forall s_0, s_1, s_0', s_1' ::$
$(x(s_0) = false \ \wedge x(s_1) = false \ \wedge \ x(s_0') = true$
$\wedge \, x(s_1') = true$
$\wedge$ the value of all other variables in s_0 and s_0' are the same
$\wedge$ the value of all other variables in s_1 and s_1' are the same
$\wedge$ (s_0, s_1) does not violate *spec*)
$\Rightarrow$ (s_0', s_1') does not violate *spec*

Likewise, we define monotonicity for programs by considering transitions within the invariant, and define monotonic programs as follows:

Definition. A program p with the invariant S is positive monotonic with respect to a boolean variable x iff the following condition is satisfied.

$\forall s_0, s_1, s_0', s_1' ::$
$(x(s_0) = false \ \wedge x(s_1) = false \ \wedge \ x(s_0') = true$
$\wedge \, x(s_1') = true$
$\wedge$ the value of all other variables in s_0 and s_0' are the same
$\wedge$ the value of all other variables in s_1 and s_1' are the same
$\wedge$ $(s_0, s_1) \in p|S$)
$\Rightarrow$ $(s_0', s_1') \in p|S$

Negative monotonicity and monotonicity with respect to non-boolean variables. By swapping the word $false$ and $true$ in the above definition, we can define negative monotonicity. Also, although we defined monotonicity with respect to boolean variables, it can be extended to deal with non-boolean variables. One approach is to partition the domain of the non-boolean variable x into two parts. If $x = true$ in the above definition, x belongs to the first part of its domain and if $x = false$ then x belongs to the second part of its domain. We use this definition later in the section while discussing the necessity of the monotonic programs and specifications.

Fault-safe specifications. In a *fault-safe* specifcation *spec*, if a fault transition (s_0, s_1) violates *spec* then all transitions that reach state s_0 violate *spec*. One interpretation of this definition is that the first transition that causes safety to be violated is a program transition.

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

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

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. In fact, given a specification *spec* and a 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 5.1 and Corollary 5.2 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 (the following theorem only applies for a fault-intolerant program p in which no processes can blindly write a variable):

Theorem 5.1 Given is a fault-intolerant program p, its invariant S, faults f and an f-safe specification *spec*,

If
$(\forall j, x : j$ is a process in p and j cannot read x :
$spec$ is positive monotonic with respect to x
$\wedge$ The program consisting of the transitions of j is negative monotonic with respect to x)

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

We generalize Theorem 5.1 as follows:

Corollary 5.2 Given is a fault-intolerant program p, its invariant S, faults f and an f-safe specification *spec*,

If
$\forall j, x : j$ is a process in p and j cannot read x :
($spec$ is positive monotonic with respect to x
$\wedge$ The program consisting of the transitions of j is negative monotonic with respect to x)
$\vee$
($spec$ is negative monotonic with respect to x
$\wedge$ The program consisting of the transitions of j is positive monotonic with respect to x)

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

□

Necessity of Monotonicity. We consider the following question: *Is monotonicity of specifications/ programs necessary to obtain polynomial time synthesis of failsafe fault-tolerance*? We argue that the answer to this question is affirmative. More specifically, we observe that if only monotonicity of the fault-intolerant program (respectively, specification) were available, the addition of failsafe fault-tolerance is NP-hard. To see this, we recall the reduction of 3-SAT to the problem of adding failsafe fault-tolerance. In that proof, we mapped the 3-SAT problem to a fault-intolerant program p, its invariant S, faults f and specification $spec$. We make the following observations about them: i) $spec$ is f-safe as no fault transition violates $spec$ ($spec$ is violated if some program action is executed after the fault action), ii) $spec$ is negative monotonic with respect to h, iii) p is negative monotonic with respect to h. Thus, if the fault-intolerant program is negative monotonic (with respect to the appropriate variables) and no condition is imposed on the specification, the problem of adding failsafe fault-tolerance is NP-hard. By symmetry, if the specification is positive monotonic (with respect to the appropriate variables) and no condition is imposed on the fault-intolerant program, the problem still remains NP-hard.

6 Concluding Remarks and Future Directions

In this paper, we focused on the problem of adding failsafe fault-tolerance to an existing fault-intolerant 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, that the problem of adding failsafe fault-tolerance is NP-hard. In 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 easily reap the benefits of automation and the problems for which heuristics need to be developed in order to benefit from automation. This paper helps to make this boundary more precise than [2] in three ways. For one, the proof in [2] 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 [2] 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.

The third –and the most important– step in identifying the boundary is addressed in Section 5 where we identified a class of specifications and a class of programs for which failsafe fault-tolerance can be added in polynomial time.

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 both necessary and sufficient.

It follows that the problem of adding fault-tolerance remains NP-hard if one begins with a negative monotonic program and an arbitrary specification. Likewise, by symmetry, polynomial time algorithm cannot be synthesized (unless $P = NP$) if the specification is positive monotonic but the fault-intolerant program is not negative monotonic.

The synthesis approach in this paper differs from that in [9–13] where one begins with a specification and obtains a fault-tolerant program. When a fault-tolerant program can be designed in an automated fashion, we expect that it will be easier to add fault-tolerance if we begin with a fault-intolerant program than if we begin with just the specification. Also, our approach allows one to reuse a given fault-intolerant program and, hence, it provides the potential that it can preserve properties such as efficiency that are difficult to model in an automated synthesis procedure.

Our work suggests several future directions. For one, given a fault-intolerant program and its invariant that do not satisfy monotonicity requirements, how can we modify the invariant such that monotonicity requirements are met while ensuring that the program satisfies the specification from the new invariant. Thus, a heuristic based on the principle of modifying the given invariant may be used to add failsafe fault-tolerance in polynomial time. We are investigating the conditions under which this heuristic will be applicable.

We are also developing heuristics to deal with the case where the given specification does not satisfy the monotonicity requirements. In this case, we are addressing the question: How can the given specification be strengthened to satisfy monotonicity requirements?

References

[1] S. S. Kulkarni. *Component-based design of fault-tolerance*. PhD thesis, Ohio State University, 1999.

[2] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 2000.

[3] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of byzantine agreement. *Symposium on Reliable Distributed Systems*, 2001.

[4] M. Barborak, A. Dahbura, and M. Malek. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171–220, 1993.

[5] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 1982.

[6] L. Gong, P. Lincoln, and J. Rushby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. *IEEE Computer Society Press*, pages 139-157, September 1995.

[7] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[8] 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.

[9] P. Attie and A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM TOPLAS*, 23(2), March 2001.

[10] A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC)*, 1998.

[11] O. Kupferman and M. Vardi. Synthesis with incomplete information. *ICTL*, 1997.

[12] A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. *In Proceedings of the 16th ACM Symposium on Principles of Programming Languages (POPL 1989)*, pages 179-190, 1989.

[13] A. Anuchitanukul and Z. Manna. Reliability and synthesis of reactive modules. *International Conference on Computer-Aided Verification*, pages 156–169, 1994.