# Automating the Addition of Fault-Tolerance

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

Anish Arora
Department of Computer
and Information Science
Ohio State University
Columbus Ohio 43210 USA

**Abstract**

In this paper, we focus on automating the transformation of a given fault-intolerant program into a fault-tolerant program. We show how such a transformation can be done for three levels of fault-tolerance properties, failsafe, nonmasking and masking. For the high atomicity model where the program can read all the variables and write all the variables in one atomic step, we show that all three transformations can be performed in polynomial time in the state space of the fault-intolerant program. For the low atomicity model where restrictions are imposed on the ability of programs to read and write variables, we show that all three transformations can be performed in exponential time in the state space of the fault-intolerant program. We also show that the the problem of adding masking fault-tolerance is NP-hard and, hence, exponential complexity is inevitable unless $P = NP$.

**Keywords : Fault-tolerance, Addition of fault-tolerance, Concurrent programs, Low Atomicity, Program synthesis, Program transformation**

[1]Email: sandeep@cse.msu.edu, anish@cis.ohio-state.edu
Web: `http://www.cse.msu.edu/~sandeep`, `http://www.cis.ohio-state.edu/~anish`
Tel: +1-517-355-2387
Arora is currently on sabbatical leave at Microsoft Research. This work was partially sponsored by NSA Grant MDA904-96-1-0111, NSF Grant NSF-CCR-9972368, an Ameritech Faculty Fellowship, a grant from Microsoft Research, and a grant from Michigan State University.

# 1 Introduction

In this paper, we focus on automating the transformation of a fault-intolerant program into a fault-tolerant program. The motivations behind this work are multi-fold. The first motivation comes from the fact that the designer of a fault-tolerant program is often aware of a corresponding fault-intolerant program that is known to be correct in the absence of faults. Or, the designer may be able to develop a fault-intolerant program and its manual proof in a simple way. In these cases, it is expected that the designer will benefit from reusing that fault-intolerant program rather than starting from scratch. Moreover, the reuse of the fault-intolerant program will be virtually mandatory if the designer has only an incomplete specification.

The second motivation is that the use of such automated transformation will obviate the need for manually constructing the proof of correctness of the synthesized fault-tolerant program as the synthesized program will be correct by construction. This advantage is especially useful when designing concurrent and fault-tolerant programs as it is well-understood that manually constructing proofs of correctness for such programs is especially hard.

The third motivation stems from our previous work [1–3] that shows that a fault-tolerant program can be expressed as a composition of a fault-intolerant program and a set of 'fault-tolerance components'. The fault-intolerant program is responsible for ensuring that the fault-tolerant program works correctly in the absence of faults; it plays no role in dealing with fault-tolerance. The fault-tolerance components are responsible for ensuring that the fault-tolerant program deals with the faults in accordance to the level of tolerance desired; they play no role in ensuring that the program works correctly in the absence of faults. We have also found that the fault-tolerance components help in manually designing fault-tolerant programs [2, 3] as well as in manually constructing their proofs [3, 4]. Moreover, we have found that programs designed using fault-tolerance components are easier to understand and have a better structure [3] than programs designed from scratch.

The third motivation suggests that given a fault-intolerant program $p$, we should focus on transforming it to obtain a fault-tolerant program $p'$ such that the transformation is done solely for the purpose of dealing with faults according to the level of fault-tolerance desired. More specifically, it suggests that $p'$ should not introduce new ways to satisfy the specification in the absence of faults.

We study the problem of transforming a fault-intolerant program into a fault-tolerant program for three levels of fault-tolerance properties, namely, failsafe, nonmasking and masking. Intuitively, a failsafe fault-tolerant program only satisfies the safety of its specification, a nonmasking fault-tolerant program recovers to a state from where its subsequent computation is in the specification, and a masking fault-tolerant program satisfies the specification even in the presence of faults. (See Section 2 for precise definitions.)

For each of the three levels of fault-tolerance properties, we study the transformation problem in the context of two models; the high atomicity model and the low atomicity model. In the high atomicity model, the program can read and write all its variables in one atomic step. In the low atomicity model, the program consists of a set of processes, and the model specifies restrictions on the ability of processes to atomically read and write program variables. Thus, the transformation problem in the low atomicity model requires us to derive a fault-tolerant program that respects the restrictions imposed by the low atomicity model.

The main contributions of this paper are as follows: (1) For the high atomicity model, we present a sound and complete algorithm that solves the transformation problem. The complexity of our algorithm is polynomial in the state space of the fault-intolerant program (cf. Section 4). (2) For the low atomicity model, we present a sound and complete algorithm that solves the transformation problem. The complexity of our algorithm is exponential in the state space of the fault-intolerant program (cf. Section 6). (3) We also show that for the low atomicity model, the problem of transforming a fault-intolerant program into a masking fault-tolerant program is NP-hard. It follows that there is no sound and complete polynomial algorithm to solve the problem of adding masking fault-tolerance unless $P = NP$ (cf. Section 7).

**Organization of the paper.** This paper is organized as follows: We provide the definitions of programs, specifications, faults and fault-tolerance in Section 2. Using these definitions, we state the transformation problem in Section 3. In Section 4, we show how the transformation problem is solved in the high atomicity model. In Section 5, we show how to characterize the low atomicity model. Using this characterization, in Section 6, we show how the transformation problem is solved in the low atomicity model. In Section 7, we show that the problem of transforming a fault-intolerant program into a masking fault-tolerant program is NP-hard. In Section 8, we illustrate how our algorithm can be used to add fault-tolerance to a fault-intolerant program. Finally, we discuss related work in Section 9 and make concluding remarks in Section 10.

# 2 Programs, Specifications Faults, and Fault-Tolerance

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 [5]. And, the definition of faults and fault-tolerances is adapted from Arora and Gouda [6] and Arora and Kulkarni [1].

## 2.1 Program

**Definition.** A program $p$ is a tuple $\langle S_p, \delta_p \rangle$ where $S_p$ is a finite set of states, and $\delta_p$ is a subset of $\{(s_0, s_1) : s_0, s_1 \in S_p\}$. $\square$

**Definition (*State predicate*).** A state predicate of $p(= \langle S_p, \delta_p \rangle)$ is any subset of $S_p$. $\square$

*Notation.* A state predicate $S$ is true in state $s$ iff $s \in S$.

**Definition (*Projection*).** Let $p(= \langle S_p, \delta_p \rangle)$ be a program, and let $S$ be a state predicate of $p$. We define the projection of $p$ on $S$, denoted as $p|S$, as the program $\langle S_p, \{(s_0, s_1) : (s_0, s_1) \in \delta_p \ \wedge \ s_0, s_1 \in S\} \rangle$. $\square$

Note that $p|S$ consists of transitions of $p$ that start in $S$ and end in $S$.

**Definition (*Subset*).** Let $p(= \langle S_p, \delta_p \rangle)$ and $p'(= \langle S'_p, \delta'_p \rangle)$ be programs. We say $p' \subseteq p$ iff $S'_p = S_p$ and $\delta'_p \subseteq \delta_p$. $\square$

**Definition (*Closure*).** A state predicate $S$ is closed in a set of transitions $\delta_p$ iff
$(\forall (s_0, s_1) : (s_0, s_1) \in \delta_p : (s_0 \in S \Rightarrow s_1 \in S))$. $\square$

**Definition. (*Computation*).** A sequence of states, $\langle s_0, s_1, ... \rangle$, is a computation of $p(= \langle S_p, \delta_p \rangle)$ iff the following two conditions are satisfied:

- $\forall j : j > 0 : (s_{j-1}, s_j) \in \delta_p$,
- if $\langle s_0, s_1, ... \rangle$ is finite and terminates in state $s_l$ then there does not exist state $s$ such that $(s_l, s) \in \delta_p$.

$\square$

*Notation.* We call $\delta_p$ as the transitions of $p$. When it is clear from context, we use $p$ and $\delta_p$ interchangeably, e.g., we say that a state predicate $S$ is closed in $p(= \langle S_p, \delta_p \rangle)$ to mean that $S$ is closed in $\delta_p$.

## 2.2 Specification

**Definition.** 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 $\sigma$ is in that set then so are all the suffixes of $\sigma$. Fusion closure of the set means that if state sequences $\langle \alpha, x, \gamma \rangle$ and $\langle \beta, x, \delta \rangle$ are in that set then so are the state sequences $\langle \alpha, x, \delta \rangle$ and $\langle \beta, x, \gamma \rangle$, where $\alpha$ and $\beta$ are finite prefixes of state sequences, $\gamma$ and $\delta$ are suffixes of state sequences, and $x$ is a program state. $\square$

Following Alpern and Schneider [5], it can be shown that any specification is the intersection of some "safety" specification that is suffix closed and fusion closed and some "liveness" specification. Intuitively, the safety specification identifies a set of bad prefixes. A sequence is in the safety specification iff none of its prefixes are identified as bad prefixes. Intuitively, a liveness specification requires that any finite sequence be extensible in order to satisfy that liveness specification. Formally,

**Definition (*Safety*).** A safety specification is a set of state sequences that meets the following condition: for each state sequence $\sigma$ not in that set, there exists a prefix $\alpha$ of $\sigma$, such that for all state sequences $\beta$, $\alpha\beta$ is not in that set (where $\alpha\beta$ denotes the concatenation of $\alpha$ and $\beta$). □

**Definition (*Liveness*).** A liveness specification is a set of state sequences that meets the following condition: for each finite state sequence $\alpha$ there exists a state sequence $\beta$ such that $\alpha\beta$ is in that set. □

*Notation.* Let *spec* be a specification. We use the term 'safety of *spec*' to mean the smallest safety specification that includes *spec*.

Note that the synthesis algorithm must be provided with a specification that is described in finite space. To simplify further presentation, however, we have defined specifications to contain infinite sequences of states. A concise representation of these infinite sequences is given in Section 2.6.

## 2.3 Program Correctness with respect to a Specification

Let *spec* be a specification.

**Definition (*Refines*).** $p$ refines *spec* from $S$ iff the following two conditions hold:

- $S$ is closed in $p$, and
- Every computation of $p$ that starts in a state where $S$ is true is in *spec*. □

**Definition (*Maintains*).** Let $\alpha$ be a finite sequence of states. The prefix $\alpha$ maintains *spec* iff there exists a sequence of states $\beta$ such that $\alpha\beta \in$ *spec*. □

**Definition (*Violates*).** Let $\alpha$ be a finite sequence of states. The prefix $\alpha$ violates *spec* iff it is not the case that $\alpha$ maintains *spec*. □

*Notation.* We say that $p$ maintains *spec* from $S$ iff $S$ is closed in $p$ and every computation prefix of $p$ that starts in a state in $S$ maintains *spec*. We say that $p$ violates *spec* from $S$ iff it is not the case that $p$ refines *spec* from $S$.

**Definition (*Invariant*).** $S$ is an invariant of $p$ for *spec* iff $S \neq \{\}$ and $p$ refines *spec* from $S$. □

*Notation.* Henceforth, 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. We emphasize that such representation is possible notwithstanding the type of the faults (be they stuck-at, crash, fail-stop, omission, timing, performance, or Byzantine), the nature of the faults (be they permanent, transient, or intermittent), or the ability of the program to observe the effects of the faults (be they detectable or undetectable).

**Definition (*Fault*).** A fault for $p(= \langle S_p, \delta_p \rangle)$ is a subset of $\{(s_0, s_1) : s_0, s_1 \in S_p\}$. □

For the rest of the section, let *spec* be a specification, $T$ be a state predicate, $S$ an invariant of $p$, and $f$ a fault for $p$.

**Definition (*Computation in the presence of faults*).** A sequence of states, $\langle s_0, s_1, ... \rangle$, is a computation of $p(= \langle S_p, \delta_p \rangle)$ in the presence of $f$ iff the following three conditions are satisfied:

- $\forall j : j > 0 : (s_{j-1}, s_j) \in (\delta_p \cup f)$,
- 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
- $\exists n : n \geq 0 : (\forall j : j > n : (s_{j-1}, s_j) \in \delta_p)$. □

*Notation.* For brevity, we use '$p[]f$' to mean '$p$ in the presence of $f$'. More specifically, a sequence is a computation of '$p[]f$' iff it is a computation of '$p$ in the presence of $f$'. And, the transitions of $p[]f$ are obtained by taking the union of the transitions of $p$ and the transitions of $f$.

**Definition (*Fault-span*).** $T$ is an $f$-span of $p$ from $S$ iff $S \Rightarrow T$ and $T$ is closed in $p[]f$. $\square$

Thus, at each state where an invariant $S$ of $p$ is true, and an $f$-span $T$ of $p$ from $S$ is also true. Also, $T$, like $S$, is also closed in $p$. Moreover, if any action in $f$ is executed in a state where $T$ is true, the resulting state is also one where $T$ is true. It follows that for all computations of $p$ that start at states where $S$ 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 actions in $f$.

*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 $p$".

## 2.5 Fault-Tolerance

In the absence of faults, a program should refine its specification. In the presence of faults, however, it may refine a weaker version of the specification as determined by the level of tolerance provided. With this notion, we define three levels of fault-tolerance below.

**Definition (*failsafe $f$-tolerant for spec from $S$*).** $p$ is failsafe $f$-tolerant to *spec* from $S$ iff the following two conditions hold:

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

**Definition (*nonmasking $f$-tolerant for spec from $S$*).** $p$ is nonmasking $f$-tolerant to *spec* from $S$ iff the following two conditions hold:

- $p$ refines *spec* from $S$, and
- 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 $S$. $\square$

**Definition (*masking $f$-tolerant for spec from $S$*).** $p$ is masking $f$-tolerant to *spec* from $S$ iff the following two conditions hold:

- $p$ refines *spec* from $S$, and
- 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 $S$. $\square$

The level of fault-tolerance characterizes the extent to which the program refines *spec* in the presence of faults. A failsafe fault-tolerant program ensures that in the presence of faults only the safety of *spec* is maintained. A nonmasking fault-tolerant program ensures that in the presence of faults, the program recovers to states from where *spec* is satisfied. A masking fault-tolerant program ensures that in the presence of faults both these properties are satisfied and, hence, every computation of $p[]f$ is in *spec* (see [3] for proof).

*Notation.* In the sequel, whenever the specification *spec* and the invariant $S$ are clear from the context, we omit them; thus, "masking $f$-tolerant" abbreviates "masking $f$-tolerant for *spec* from $S$", and so on.

## 2.6 Observations on Programs and Specifications

In this section, we summarize observations about our programs and specifications. Subsequently, we present the form in which specifications are given to the synthesis algorithm.

Note that a specification, say *spec*, is a set of infinite sequences of states. If $p$ refines *spec* from $S$ then all computations of $p$ that start from a state in $S$ are in *spec* and, hence, all computations of $p$ that start from a state in $S$ must be infinite. Using the same argument, we make the following two observations.

**Observation 2.1** If $p'$ is (failsafe, nonmasking or masking) $f$-tolerant for *spec* from $S'$ then all computations of $p'$ that start from a state in $S'$ must be infinite. $\square$

**Observation 2.2** If $p'$ is (nonmasking or masking) $f$-tolerant for *spec* from $S'$ then all computations of $p'[]f$ that start from a state in $S'$ must be infinite. $\square$

Observe that we do not disallow fixed-point computations; we simply require that if $s_0$ is a fixed-point of $p$ then the transition $(s_0, s_0)$ should be included in the transitions of $p$.

**Concise Representation for Specifications.** Recall that a safety specification identifies a set of bad prefixes that should not occur in program computations. For fusion closed and suffix closed specifications, we can focus on only prefixes of length 2. In other words, if we have a prefix $\langle \alpha, s_0 \rangle$ that maintains *spec* then we can determine whether an extended prefix $\langle \alpha, s_0, s_1 \rangle$ maintains *spec* by focusing on the transition $(s_0, s_1)$, and ignoring $\alpha$. Formally we state this in Lemma 2.3 as follows (cf. [3] for proof.):

**Lemma 2.3.** Let $\alpha$ be finite sequence of states, and let *spec* be a specification.

If $\quad \langle \alpha, s_0 \rangle$ maintains *spec*
Then $\langle \alpha, s_0, s_1 \rangle$ maintains *spec* iff $\langle s_0, s_1 \rangle$ maintains *spec*. $\square$

From Lemma 2.3, it follows that the safety specification can be concisely represented by the set of 'bad transitions'. For simplicity, we assume that for a given *spec* and a state space $S_p$, the set of bad transitions corresponding to the minimal safety specification that includes *spec* are given. If this is not the case and *spec* is given in terms of a temporal logic formula, the set of bad transitions can be computed in polynomial time by considering all transitions $(s_0, s_1)$, where $s_0, s_1 \in S_p$.

Our proof that a fault-tolerant program refines the liveness specification solely depends on the fact that the fault-intolerant program refines the liveness specification. Therefore, our algorithm can transform a fault-intolerant program into a fault-tolerant program even if the liveness specification is unavailable.

# 3 Problem Statement

In this section, we formally specify the problem of deriving a fault-tolerant program from a fault-intolerant program. We first intuitively characterize what it means for a fault-tolerant program $p'$ to be derived from a fault-intolerant program $p$. We use this characterization to precisely state the transformation problem. Finally, we also discuss the soundness and completeness issues in the context of the transformation problem.

As described in Section 2, the fault-intolerant program $p$ is specified in terms of its state space $S_p$, its transitions, $\delta_p$, its invariant, $S$. The specification, *spec*, provides a set of bad prefixes that should not occur in program computations. The faults, $f$, are specified in terms of state transitions. Similarly, the fault-tolerant program $p'$ is specified in terms of its state space $S_p$, its state transitions, say $\delta'_p$, its invariant $S'$, its specification *spec*, and the type of fault-tolerance it provides.

Now, we consider what it means for a fault-tolerant program $p'$ to be derived from $p$. As mentioned in the introduction, our derivation is based on the premise that $p'$ is obtained by adding fault-tolerance alone to $p$, i.e., $p'$ does not introduce new ways of refining *spec* when no faults have occurred. To precisely state our concept of derivation we consider the relation between (1) the invariants $S$ and $S'$, and (2) the transitions $\delta_p$ and $\delta'_p$.

- If $S'$ contains states that are not in $S$ then, in the absence of faults, $p'$ will include computations that start outside $S$. Since $p'$ refines *spec* from $S'$, it would imply that $p'$ is using a new way to refine *spec* in the absence of faults (since $p$ refines *spec* only from $S$). Therefore, we require that $S' \subseteq S$ (equivalently $S' \Rightarrow S$).
- Regarding the transitions of $p$ and $p'$, we focus only on the transitions of $p'|S'$ and $p|S'$. If $p'|S'$ contains a transition that is not in $p|S'$, $p'$ can use this transition in order to refine *spec* in the absence of faults. Since this was not permitted in $p$, we require that $p'|S' \subseteq p|S'$.

Using the above two requirements, we define the problem of transformation as follows (This definition will be instantiated for failsafe $f$-tolerance, nonmasking $f$-tolerance and masking $f$-tolerance):

**The Transformation Problem**
Given $p$, $S$, *spec* and $f$ such that $p$ refines *spec* from $S$
Identify $p'$ and $S'$ such that
$S' \Rightarrow S$,
$p'|S' \subseteq p|S'$, and
$p'$ is $f$-tolerant to *spec* from $S'$. $\square$

We also define the corresponding decision problem as follows:(This definition will also be instantiated for failsafe $f$-tolerance, nonmasking $f$-tolerance and masking $f$-tolerance):

**The Decision Problem**
Given $p$, $S$, *spec* and $f$ such that $p$ refines *spec* from $S$
Does there exist $p'$ and $S'$ such that
$S' \Rightarrow S$,
$p'|S' \subseteq p|S'$, and
$p'$ is $f$-tolerant to *spec* from $S'$? $\square$

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

**Soundness and completeness.** An algorithm for the transformation problem is sound iff for any given input, its output, namely program $p'$ and the state predicate $S'$, solves the transformation problem. An algorithm for the transformation problem is complete iff for any given input if the answer to the decision problem is affirmative then the algorithm always finds program $p'$ and state predicate $S'$.

*Remark.* Our notion of derivation suggests that the fault-intolerant program we start with should be maximal, i.e., we should consider a program which has the weakest invariant and maximal non-determinism. Also, the above transformation problem requires that the state space of the fault-tolerant program is the same as that of the fault-intolerant program. Thus, it suggests that the fault-intolerant program should be such that it contains all the variables needed for the fault-tolerant program. We can similarly define the transformation problem where the fault-tolerant program can introduce new states by introducing new variables that were not present in the given fault-intolerant program. To solve such a general version of the transformation problem, we will need to systematically introduce new variables. Although such a general transformation problem is out of scope of this paper, we would like to note that our algorithms can be modified to systematically introduce such new variables. Under such a modification, our algorithms will have the following properties: (1) if a fault-tolerant program exists the complexity of deriving a fault-tolerant program depends on the size of the derived fault-tolerant program, and (2) if a fault-tolerant program does not exist the algorithm is nonterminating.

# 4 Adding Fault-Tolerance in High Atomicity Model

In this section, we consider the transformation problem for programs in the high atomicity model, where a program transition can read any number of variables as well as update any number of variables in one atomic step. In other words, if the enumerated states of the program are $s_0, s_1, ...s_{max}$ then the program transitions can be any subset of $\{(s_j, s_k) : 0 \leq j \leq max\}$.

We first describe, in Section 4.1, our algorithm for adding failsafe fault-tolerance. Then, in Section 4.2, we present our algorithm for adding nonmasking fault-tolerance. Finally, in Section 4.3, we present our algorithm for adding masking fault-tolerance. In each of these sections, we present a soundness and completeness proof for our algorithm.

## 4.1 Problem of Designing Failsafe Tolerance

As shown in Section 2, the safety specification identifies a set of bad transitions that should not occur in program computations. Given a bad transition $(s_0, s_1)$, we consider two cases: (1) $(s_0, s_1)$ is not a transition of $f$, (2) $(s_0, s_1)$ is a transition of $f$.

For case (1), we claim that $(s_0, s_1)$ can be removed while obtaining $p'$. To see this consider two subcases: (a) state $s_0$ is ever reached in the computation of $p'[]f$, and (b) state $s_0$ is never reached in the computation of $p'[]f$. In the former subcase, the transition $(s_0, s_1)$ must be removed as the safety of *spec* can be violated if $p'[]f$ ever reaches state $s_0$ and executes the transition $(s_0, s_1)$. In the latter subcase, the transition $(s_0, s_1)$ is irrelevant and, hence, can be removed.

For case (2), we cannot remove the transition $(s_0, s_1)$ as it would mean removing a fault transition. Therefore, we must ensure that $p'[]f$ never reaches the state $s_0$. In other words, for all states $s$, the transition $(s, s_0)$ must be removed in obtaining $p'$. Also, if any of these removed transitions, say $(s'_0, s_0)$, is a fault transition then we must recursively remove all transitions of the form $(s, s'_0)$ for each state $s$.

Using the above two cases, our algorithm to obtain the failsafe fault-tolerant program is as follows: it first identifies states from where execution of one or more fault transitions violates safety. Then, it removes transitions of $p$ that reach these states as well as transitions of $p$ that violate the safety of *spec*. If there exist states in the invariant such that execution of one or more fault actions from those states violates the safety of *spec*, then we recalculate the invariant by removing those states. Our algorithm is as follows (As mentioned in Section 2, we use a program and its transitions interchangeably.):

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

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

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

In order to prove the soundness of our algorithm, we first make the following two observations.

**Observation 4.1** $S' \cap ms = \{\}$ □

**Observation 4.2** Every computation of $p'$ that starts from a state in $S'$ is infinite. □

**Theorem 4.3** The algorithm *Add_failsafe* is sound.

**Proof.** To show the soundness of our algorithm, we need to show that the three conditions of the *transformation problem* are satisfied.

1. $S' \Rightarrow S$.
   By the construction of $S'$, $S'$ is obtained by removing zero or more states in $S$. Therefore this condition is trivially satisfied.
2. $p'|S' \subseteq p|S'$.
   From the construction of $p'$, the transitions of $p'$ are a subset of the transitions of $p$. Therefore, $(p'|S') \subseteq (p|S')$.

3. $p'$ is failsafe $f$-tolerant to *spec* from $S'$.

   Consider a computation $c$ of $p'$ that starts from a state in $S'$: From 1, $c$ starts from a state in $S$, and from 2, $c$ is a computation of $p$. It follows that $c \in spec$. Thus, every computation of $p'$ that starts from a state in $S'$ is in *spec*, i.e., $p'$ refines *spec* from $S'$.

   We let the fault-span $T'$ to be the set of states reached in any computation of $p'[]f$ that starts from a state in $S'$. Consider a computation prefix $c$ of $p'[]f$ that starts from a state in $T'$. From the definition of $T'$ there exists a computation prefix $c'$ of $p'[]f$ such that $c$ is a suffix of $c'$ and $c'$ starts from a state in $S'$. If $c'$ violates the safety of *spec* then there exists a prefix of $c'$, say $\langle s_0, s_1, ..., s_n \rangle$, such that $\langle s_0, s_1, ..., s_n \rangle$ violates the safety of *spec*. Wlog, let $\langle s_0, s_1, ..., s_n \rangle$ be the smallest such prefix. It follows that $(s_{(n-1)}, s_n)$ violates the safety of *spec* and, hence, $(s_{(n-1)}, s_n) \in mt$. By construction, $p'$ does not contain any transitions in $mt$. Thus, $(s_{(n-1)}, s_n)$ is a transition of $f$. If $(s_{(n-1)}, s_n)$ is a transition of $f$ then $s_{(n-1)} \in ms$ and $(s_{(n-2)}, s_{(n-1)}) \in mt$ and, hence, $(s_{(n-2)}, s_{(n-1)})$ is a transition of $f$. By induction, if $\langle s_0, s_1, ..., s_n \rangle$ violates the safety of *spec*, $s_0 \in ms$, which is contradiction since $s_0 \in S'$ (cf. Observation 4.1). Thus, each prefix of $c'$ maintains *spec*. By suffix closure, each prefix of $c$ also maintains *spec*. Thus, $p'[]f$ maintains *spec* from $T$. $\square$

**Theorem 4.4** The algorithm *Add_failsafe* is complete.

**Proof.** Let program $p''$ and predicate $S''$ solve transformation problem. Clearly, $S'' \cap ms = \{\}$; if $s_0 \in (S'' \cap ms)$ then the execution of faults alone from $s_0$ can violate the safety of *spec*. It follows that $S'' \subseteq (S - ms)$. Moreover, $p''|S''$ cannot include any transitions in $mt$; if $p''|S''$ contains a transition in $mt$ then the execution of this transition followed by zero or more fault transitions can violate the safety of *spec*. Thus, $p''|S'' \subseteq (p - mt)$. Finally, every computation of $p''$ that starts in a state in $S''$ must be an infinite computation, if it were to be in *spec* (cf. Observation 2.1). It follows that there exists a subset of $S$ (namely $S''$) such that all computations of $p - mt$ within that subset are infinite.

Our algorithm declares that no solution for the transformation problem exists only when there is no subset of $S - ms$ such that all the computations of $p - mt$ within that subset are infinite. It follows that our algorithm declares that no failsafe fault-tolerant program exists only if the answer to the corresponding decision problem is false. $\square$

**Theorem 4.5** The algorithm *Add_failsafe* is in $P$.

**Proof.** To prove this, we consider the complexity of each statement in *Add_failsafe*. (1) Calculating $ms$ is in $P$ as we can use the following algorithm: For each fault transition $(s_0, s_1)$ such that $(s_0, s_1)$ violates safety of *spec*, include $s_0$ in $ms$. Now, in each iteration, check if there exists a fault transition $(s_0, s_1)$ such that $s_0 \notin ms$ and $s_1 \in ms$. If such a transition exists add $s_0$ to $ms$. Since the size of $ms$ increases by at least one in each iteration, the number of iterations is polynomial in the state space, namely $S_p$. (2) Calculating $mt$ is in $P$ as we need to check each transition only once. (3) The while loop in ConstructInvariant can execute only $|S_p|$ number of times. (4) The complexity of ConstructTransitions is clearly polynomial. $\square$

## 4.2 Problem of Designing Nonmasking Tolerance

To design a nonmasking $f$-tolerant program $p'$, we ensure that if $p$ is perturbed by faults in $f$ then it eventually recovers to a state in $S$. To obtain the nonmasking $f$-tolerant program, for each state $s_0$, $s_0 \notin S$, we add a transition $(s_0, s_1)$ such that $s_1 \in S$. Our algorithm for nonmasking $f$-tolerant program is as follows:

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

**Theorem 4.6** The algorithm *Add_nonmasking* is sound and complete.

**Proof.** By construction, $p'$ and $S'$ satisfy the conditions of the transformation problem. Thus, the algorithm is sound. Also, the algorithm always finds a solution to the transformation problem. Thus, the algorithm is complete. $\Box$

**Theorem 4.7** The algorithm *Add_nonmasking* is in $P$. $\Box$

## 4.3 Problem of Designing Masking Tolerance

To design a masking $f$-tolerant program $p'$, we proceed to identify the weakest invariant $S'$ (which is stronger than $S$) and the weakest fault-span $T'$. As argued in Theorem 4.4, our first estimate of $S'$ is $S_1$ where $S_1$=ConstructInvariant$(S-ms, p-mt)$. And, we estimate $T'$ to be $T_1$ where $T_1 = true - ms$, i.e., $T_1$ includes all states except those in $ms$.

We continue to strengthen our $S_1$ and $T_1$ while ensuring that if some $S'$ solves the transformation problem then $S' \Rightarrow S_1$. We first identify and remove states in $T_1$ from where it is not possible to reach a state in $S_1$ without violating the safety of *spec*. We then find the largest subset of the remaining states that is closed in $f$. This represents the new estimate for fault-span. Since $S_1$ must be a subset of $T_1$, we recalculate $S_1$ to be the largest subset of $S_1 \wedge T_1$ such that all the computations from that subset are infinite. We continue this process until we reach a fixpoint. Our algorithm for adding masking fault-tolerance is as follows:

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

$\quad$ repeat
$\qquad T_2, S_2 := T_1, S_1$;
$\qquad p_1 := p|S_1 \ \cup \{(s_0, s_1) : s_0 \notin S_1 \ \wedge \ s_0 \in T_1 \ \wedge \ s_1 \in T_1\} - mt$;
$\qquad T_1 :=$ ConstructFaultSpan$(T_1 - \{s : S_1$ is not reachable from $s$ in $p_1 \}, f)$;
$\qquad S_1 :=$ ConstructInvariant$(S_1 \wedge T_1, p_1)$;
$\qquad$ if $(S_1 = \{\} \ \vee \ T_1 = \{\})$
$\qquad\quad$ `declare no masking` $f$`-tolerant program` $p'$ `exists`;
$\qquad\quad$ exit
$\quad$ until $(T_1 = T_2 \ \wedge \ S_1 = S_2)$;

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

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

Similar to the derivation of failsafe fault-tolerance, we make the following observation:

**Observation 4.8.** $T' \cap ms = \{\}$ $\square$

**Lemma 4.9.** $p'[]f$ maintains *spec* from $T'$.

**Proof.** This proof is similar to that of Theorem 4.3 (part 3). By construction, $T'$ is closed in $p'[]f$. Let $c$ be a computation of $p'[]f$ that starts in a state in $T'$. If $c$ violates the safety of *spec*, there exists a prefix, say $\langle s_0, s_1, ..., s_n\rangle$, that violates the safety of *spec*. Wlog, let $\langle s_0, s_1, ..., s_n\rangle$ be the smallest such prefix. It follows that $(s_{(n-1)}, s_n)$ violates the safety of *spec* and, hence, $(s_{(n-1)}, s_n) \in mt$. By construction, $p'$ does not contain any transition in $mt$. Thus, $(s_{(n-1)}, s_n)$ is a transition of $f$. If $(s_{(n-1)}, s_n)$ is a transition of $f$ then $s_{(n-1)} \in ms$ and $(s_{(n-2)}, s_{(n-1)}) \in mt$ and, hence, $(s_{(n-2)}, s_{(n-1)})$ is a transition of $f$. By induction, if $\langle s_0, s_1, ..., s_n\rangle$ violates the safety of *spec*, $s_0 \in ms$, which is not possible since $s_0 \in T'$ (cf. Observation 4.8). Thus, $p'[]f$ maintains *spec* from $T'$.

**Lemma 4.10.** Every computation of $p'$ that starts from a state in $T'$ contains a state in $S'$.

**Proof.** By construction, when the repeat-until loop (on line 5-13) terminates, there exists a path from each state in $T_1$ to a state in $S_1$. Therefore, for each state $s$, $s \in T$, $Rank(s)$ is finite. And, $Rank(s) = 0$ iff $s \in S_1$. Now, consider a computation of $p'$, $\langle s_0, s_1, ...,\rangle$ that starts from a state in $T'$. By construction of $p_1$ (line 7), we have $(\forall j :: Rank(s_j) > 0 \Rightarrow Rank(s_{(j+1)}) < Rank(s_j))$. It follows there exists a state $s_n$, $s_n \in S'$, in $\langle s_0, s_1, ...,\rangle$.

**Corollary 4.11** Every computation of $p'[]f$ that starts from a state in $T'$ contains a state in $S'$. $\square$

**Theorem 4.12** The algorithm *Add_masking* is sound.

**Proof.**

1. $S' \Rightarrow S$. The proof of this part is identical to case 1 in Theorem 4.3.
2. $p'|S' \subseteq p|S'$. The proof of this part is identical to case 2 in Theorem 4.3.
3. $p'$ is masking $f$-tolerant to *spec* from $S'$.

   Consider a computation $c$ of $p'$ that starts from a state in $S'$: From 1, $c$ starts from a state in $S$, and from 2, $c$ is a computation of $p$. It follows that $c \in spec$. Thus, every computation of $p'$ that starts from a state in $S'$ is in *spec*, i.e., $p'$ refines *spec* from $S'$.

   From Lemma 4.9, it follows that $p'[]f$ maintains *spec* from $T'$. From Lemma 4.10, it follows that every computation of $p'[]f$ that starts from a state in $T'$ has a state in $S'$. Combining these two results, $p'$ is masking $f$-tolerant to *spec* from $S'$.

**Theorem 4.13** The algorithm *Add_masking* is complete.

**Proof.** Let program $p''$ and predicate $S''$ solve transformation problem. Also, let $T''$ be the fault-span of $p''$ from $S''$ that is used to prove masking fault-tolerance of $p''$. We first show that our algorithm ensures that $S'' \subseteq S_1$ and $T'' \subseteq T_1$. We consider three cases: initial assignments on line 3 and 4, assignment of $T_1$ on line 8, and assignment of $S_1$ on line 9.

- Assignments on line 3 and 4 ensure that $S'' \subseteq S_1$ and $T'' \subseteq T_1$ become true.

  Note that the first estimate for $S_1$ is ConstructInvariant$(S - ms, p - mt)$. And, as shown in Theorem 4.4, $S'' \subseteq$ ConstructInvariant$(S - ms, p - mt)$. Also, the first estimate for $T_1$ is $(true - ms)$. Clearly $T'' \subseteq T_1$; if $T''$ contains a state in $ms$ then the execution of the faults from that state may violate safety of *spec*.
- Assignment to $T_1$ on line 8 ensures that $T'' \subseteq T_1$ continues to remain true.

  Consider any fault-tolerant program, say $p'''$, with invariant $S_1$ and fault-span $T_1$. For a transition $(s_0, s_1)$ of $p'''$ we consider three cases depending upon whether $s_0 \in S_1$, $s_0 \in (T_1 - S_1)$, and $s_0 \notin T_1$.

  - If $s_0 \in S_1$, it must be the case that $(s_0, s_1) \in p|S_1$. (since $p'''$ is derived from $p$.)
  - If $s_0 \in (T_1 - S_1)$, it must be the case that $s_1 \in T_1$. (since $T_1$ is closed in $p'''$.)
  - If $s_0 \notin T_1$, $(s_0, s_1)$ is irrelevant since $p'''$ can never reach state $s_0$.

From the above three cases, it follows that the transitions of any fault-tolerant program with invariant $S_1$ and fault-span $T_1$ must be a subset of $p_1$ (cf. line 7). Consider a state $s$ in $T_1$ such that $S_1$ is not reachable from $s$ in $p_1$. Clearly, $s$ cannot be in $T''$. Therefore, $T'' \subseteq (T_1 - \{s : S_1$ is not reachable from s in $p_1$ $\})$. Moreover, if there exists a fault transition $(s_0, s_1)$ such that $s_0 \in (T_1 - \{s : S_1$ is not reachable from s in $p_1$ $\})$ and $s_1 \notin (T_1 - \{s : S_1$ is not reachable from s in $p_1$ $\})$, $s_0$ also cannot be in $T''$. It follows that $T''$ is a subset of ' the largest subset of $(T_1 - \{s : S_1$ is not reachable from s in $p_1$ $\})$ that is closed in $f$ '. Thus, after the assignment of $T_1$ on line 8, $T'' \subseteq T_1$ continues to be true.

- Assignment to $S_1$ on line 8 ensures that $S'' \subseteq S_1$ continues to remain true.

  Before execution of line 9, we have $T'' \subseteq T_1$ and $S'' \subseteq S_1$. Moreover, $S'' \subseteq T''$. It follows that $S'' \subseteq (S_1 \wedge T_1)$. Also, since all computations from $S''$ are infinite (cf. observation 2.1) and ConstructInvariant$(S_1 \wedge T_1, p_1)$ returns the largest subset of $S_1 \wedge T_1$ from where all computations are infinite, $S'' \subseteq$ ConstructInvariant$(S_1 \wedge T_1, p_1)$. Thus, after the assignment of $S_1$ on line 9, $S'' \subseteq S_1$ continues to be true.

Our algorithm declares non-existence of a solution only when $S_1 = \{\}$ or $T_1 = \{\}$. From the above result, in this case, $S'' = \{\}$ or $T'' = \{\}$. This is a contradiction. Thus, the algorithm *Add_masking* is complete. □

**Theorem 4.14** The algorithm *Add_masking* is in $P$.

**Proof.** Note that the repeat-until loop can be executed only for a polynomial number of times as in each iteration either size of $S_1$ or size of $T_1$ decreases. Also, as in the proof of Theorem 4.5, each statement in *Add_masking* is in $P$. Thus, the algorithm *Add_masking* is in $P$. □

# 5 Characterizing the Low Atomicity Model

The synthesis algorithm in Section 4 assumes that the fault-tolerant program can contain a transition $(s_0, s_1)$ for any two states $s_0, s_1$. If we think of the program state to consist of variables and their corresponding values, the synthesis algorithm assumes that the program can read the values of all variables and write the values of all variables in an atomic step. In this section, we show how we describe the low atomicity model where it may not be possible to read all variables and write all variables in an atomic step. Using this description, in Section 6, we present a synthesis algorithm for adding fault-tolerance in the low atomicity model.

We assume that the program consists of processes; each process can atomically read a subset of the program variables and write (a possibly different) set of variables. The low atomicity model presents restrictions on how a process can atomically read and write variables. Some examples of low atomicity variables that are considered in the literature are as follows:

- Read/Write model : A process is allowed to either (1) atomically read the state of one of its neighbor and write its local copy (which cannot be read by other processes) or (2) atomically read all its local copies and update its state (including the variables that which can be read by other processes).
- Interleaving model : A process is allowed to atomically read the state of all its neighbors and update its state.

Towards characterizing the low atomicity model by identifying what it means for a process to read and write a variable, we define a notation and an assumption.

*Notation.* Let $x$ be a variable. $x(s_0)$ denotes the value of variable $x$ in state $s_0$.

*Assumption.* Let $r_j$ denote the set of variables $j$ is allowed to read and $w_j$ denote the set of variables that $j$ is allowed to write. For simplicity, we assume that $j$ can *atomically* read all variables in $r_j$ and write all variables in $w_j$.

Note that if the above assumption is not true, we split process $j$ into multiple processes that satisfy the above constraint. For example, in Read/Write model described above, we split $j$ into $nbr_j+1$ sub-processes where $nbr_j$ is the neighbors of $j$; one sub-process for each neighbor that copies the state of that neighbor, and one that uses these local copies and updates the state of $j$. We leave it to the reader to verify that this can always be done.

*Remark.* Note that the above restrictions are for the program actions only. Faults are not restricted in any way, i.e., a fault transition can read and write all the variables in one atomic step.

**Write-restrictions.** Now, we define what it means for a process, say $j$, to be able to write only a subset of variables. Once again, let $w_j$ denote the set of variables that $j$ is allowed to write. Consider a transition $(s_0, s_1)$ that is a transition of $j$. Since $j$ is not allowed to write variables that are outside $w_j$, the values of these variables must remain unchanged in this transition. In other words, if a model permits $j$ to write only variables in $w_j$ then it is equivalent to providing a set of transitions $write(j, w_j)$ that $j$ cannot use synthesis algorithm, where

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

**Read-restrictions.** Now, we define what it means for a process, say $j$, to read only a subset of variables. Once again, let $r_j$ denote the set of variables that $j$ is allowed to read, and let $w_j$ denote the set of variables that $j$ is allowed to write. Initially, we consider the case where $w_j \subseteq r_j$, i.e., $j$ can write a variable only if it can read it.

Let $(s_0, s_1)$ be some transition of process $j$ such that $s_0 \neq s_1$. Now, consider a state $s_0'$ such that the values of all variables in $r_j$ are identical to that in $s_0$. Since $j$ can only read variables in $r_j$, $j$ must have a transition that starts in state $s_0'$. Let this transition be $(s_0', s_1')$. The values of variables in $r_j$ in $s_1'$ must be the same as that in $s_1$. Moreover, since $w_j \subseteq r_j$, the values of variables that are not in $r_j$ must be the same as that in $s_0'$.

By considering all the states where the values of $r_j$ are the same, we will get a group of transitions such that if $(s_0, s_1)$ is a transition of $j$ then all the transitions in that group must also be transitions of $j$. We define these transitions as $group(j, r_j)(s_0, s_1)$, for the case where $w_j \subseteq r_j$, where

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

Now, we consider the case where $w_j \not\subseteq r_j$, i.e., $j$ writes variables without reading them. To motivate such cases, consider the following scenario: Let $chan_j$ denote the sequence of messages on channel $chan$ which is an outgoing channel from process $j$. When $j$ sends a message, it does so by appending it to $chan_j$ and, thus, $j$ writes $chan_j$. However, $j$ cannot read what messages are still pending on channel $chan$, i.e., $j$ cannot read $chan_j$.

Continuing with this scenario, we consider how $chan_j$ is updated. The new value of $chan_j$ depends upon the initial state of the program (including the initial value of $chan_j$). In other words, there exists a function $f_{chan_j}$ such that when $j$ executes in state $s_0$, $j$ assigns the value $f_{chan_j}(s_0)$ to $chan_j$.

More generally, consider the case where $j$ can write multiple variables, say $x_1, x_2, ...$, without being able to read any of them. In this case, the low atomicity model provides a function $f$ such that when $j$ executes in state $s_0$, $j$ assigns the value $x_i(f(s_0))$ to variable $x_i$. Using this function, we now define a group of transitions such that if transition $(s_0, s_1)$ is a transition of $j$ then all the transitions in $group(j, f, r_j)(s_0, s_1)$ must also be transitions of $j$, where

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

Note that if $j$ can use different functions to assign values to variables without reading them, there would be one such group for each function. Moreover, if $j$ cannot write some variable, say $x$, then we can capture the same by requiring $f$ to be such that $x(f(s_0)) = x(s_0)$.

*Remark.* The above grouping is done for the case where the transition was not a self-loop. Regarding the self-loop, there are no restrictions. We model this by introducing a group $(s_0, s_0)$ for each state $s_0$. Note, however, given a program $p$ with invariant $S$, the masking (respectively, nonmasking) fault-tolerant program $p'$ can contain a self-loop only if it is in $p|S$.

**Combining read-restrictions and write-restrictions.** As we described above, in the low atomicity model, the inability of a process to write certain variables is characterized in terms of transitions that should not be included in the transitions of that process. The inability of a process to read is characterized in terms of grouping of transitions; we need to choose zero or more such groups to obtain the transitions of that process. Thus, if a transition in some group violates the restrictions imposed by the inability to write, then that entire group must be excluded in the design of fault-tolerant program. It follows that after combining the read restrictions and the write-restrictions, we get another grouping of transitions; we need to choose zero or more such groups to obtain the transitions of that process and the transitions in each group satisfy the restrictions imposed by the inability to write certain variables.

To summarize, the low atomicity model provides restrictions on the ability of processes to read and write variables. Also, if the model permits a process to write variables without reading them, the model specifies a set of functions that the process can use to write those variables. Given the low atomicity model and the state space of the fault-intolerant program, we can compute the groups of transitions that characterize the low atomicity model in polynomial time in the size of the input. Thus, we have

**Observation 5.1** The groups of transitions corresponding to the given fault-intolerant program and the low atomicity model describing the processes (with the restriction on their ability to read and write) can be computed in polynomial time. □

Typically, the state space of the fault-intolerant program is much larger than the description of the low atomicity model. Therefore, the time required to compute the groups is polynomial in the state space of the fault-intolerant program.

For the rest of the paper, we assume that the groups corresponding to the low atomicity model are given. Thus, to design a fault-tolerant program $p'$ corresponds to choosing zero or more groups of transitions such that the program corresponding to the union of the chosen transitions solves the transformation problem.

# 6 Adding Fault-Tolerance in Low-Atomicity Model

In this section, we present our algorithm for adding fault-tolerance in the low atomicity model. We present a non-deterministic algorithm whose complexity is polynomial in the state space of the fault-intolerant program. It follows that the complexity of the corresponding (brute-force) deterministic algorithm is at most exponential. We first discuss in Section 6.1 two issues relating to the low atomicity that make the transformation more complex. Then, we present our algorithm in Section 6.2.

## 6.1 Problems in Adding Fault-Tolerance in the Low-Atomicity Model

For the following discussion, let $p$ be the fault-tolerant program, $S$ be its invariant, *spec* be the problem specification, and $f$ be the fault transitions. Also, let $p'$ be a derived fault-tolerant program, and $S'$ be its invariant. Moreover, let $T'$ be the set of states reached in any computation of $p'[]f$ that starts from a state in $S'$.

- This issue relates to the design of failsafe and masking fault-tolerant programs. In the context of failsafe and masking fault-tolerance, we ensure that $p'[]f$ maintains *spec*. Recalling the definition of $mt$ from Section 4, consider a transition $(s_0, s_1)$ that is in $mt$. As discussed Section 4, this transition can be dropped from consideration in deriving $p'$. In the proof, we considered two cases (1) $s_0 \in T'$ and (2) $s_0 \notin T'$. In case (1), $(s_0, s_1)$ cannot be in $p'$ as $p'[]f$ can reach $s_0$ from where safety of *spec* will be violated if $p'$ executes the $(s_0, s_1)$ transition. In case (2), the program obtained by dropping the transition $(s_0, s_1)$ from $p'$ also solves the transformation problem.

  In the context of the low atomicity model, we cannot always drop a transition $(s_0, s_1)$ such that $(s_0, s_1) \in mt$. This is due to the possibility that $(s_0, s_1)$ is grouped with the transition $(s_0', s_1')$ where the latter transition is required in order to refine *spec* from $S'$. In this situation, there may exist a fault-tolerant program that uses the transition $(s_0, s_1)$ by ensuring that state $s_0$ is never reached.

- This issue relates to the design of nonmasking and masking fault-tolerant programs. In the context of nonmasking and masking fault-tolerant programs, we ensure that every computation of $p'$ that starts from a state in $T'$ reaches a state in $S'$. In the context of high atomicity programs, we could consider all transitions that start in $T'$, and choose any subset of them such that there are no cycles in which all states are in the set $T'-S'$.

In the context of the low atomicity model, this problem is complicated due to the fact that we need to pick groups of transitions. Thus, there may be a group consisting of transitions $(s_0, s_1)$ and $(s'_0, s'_1)$ such that the former is used to reach a state in $S'$ but the latter forms a cycle with other selected transitions.

Observe that both these issues are applicable in the design of masking fault-tolerance. And, as shown in the next section, this problem is NP-hard. In designing failsafe fault-tolerance only the first issue applies and in designing nonmasking fault-tolerance only the second issue applies. We conjecture that the second issue is what makes the problem of adding masking fault-tolerance difficult and, hence, we expect the problem of adding nonmasking fault-tolerance to be NP-hard as well.

## 6.2 Non-Deterministic Algorithm

In our non-deterministic algorithm, we guess a solution, namely the fault-tolerant program $p'$, invariant $S'$ and the fault-span $T'$. Then, we verify that $p'$ is $f$-tolerant from $S'$, and $T'$ is a boundary upto which $p'$ can be perturbed in the presence of $f$. Moreover, we verify that $p'$ and $S'$ solve the transformation problem.

Towards verifying that $p'$ refines *spec* from $S'$, we verify that $S'$ is closed in $p'$, $p|S' \subseteq p|S$ and every computation of $p'$ that starts from a state in $S'$ is infinite. We verify the last property by verifying that every state in $S'$ has a successor. (Note that the successor must be in $S'$ as $S'$ is closed in $p'$.)

To verify that $T'$ is a boundary upto which $p'$ is perturbed, we verify that $S' \Rightarrow T'$ and that $T'$ is closed in $p'[]f$.

In case of adding failsafe or masking fault-tolerance, we ensure that execution of $p'|T'$ does not violate safety. Towards this end, we verify that $T' \cap ms = \{\}$ and $(p'|T') \cap mt = \{\}$. In case of nonmasking and masking fault-tolerance, we need to ensure that every computation that starts in $T'$ has a state in $S'$. We verify this by ensuring that every computation in $T$ is infinite and that there are no cycles in the computations of $p'|(T'-S')$. Our algorithm is as follows:

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

  Guess $S', T'$, and $p' := \bigcup(g_i : g_i$ is chosen to be included in the fault-tolerant program);
  Verify the following
    $S' \neq \{\}$; $S' \Rightarrow S$; $S' \Rightarrow T'$;
    $S'$ is closed in $p'$; $T'$ is closed in $p'[]f$;
    $p'|S' \subseteq p|S'$;
    $(\forall s_0 : s_0 \in S' : (\exists s_1 :: (s_0, s_1) \in p'))$;

    if (*type*$=$*failsafe* $\vee$ *type*$=$*masking*)
      $T' \cap ms = \{\}$; $(p'|T') \cap mt = \{\}$;

    if (*type*$=$*nonmasking* $\vee$ *type*$=$*masking*)
      $(\forall s_0 : s_0 \in T' : (\exists s_1 :: (s_0, s_1) \in p'))$; $p'|(T'-S')$ is acyclic
}

**Theorem 6.1** Algorithm *Add_ft* is sound and complete. □

**Theorem 6.2** The algorithm *Add_ft* is in NP.

**Proof.** This proof follows from Observation 5.1 and the fact that each statement in algorithm *Add_ft* is executed in polynomial time. □

# 7 NP-completeness of Adding Masking Fault-Tolerance in Low Atomicity

From Theorem 6.2, we know that the problem of adding masking fault-tolerance in the low atomicity model is in NP. In this section, we show that this problem is NP-hard. It follows that there is no sound and complete deterministic polynomial algorithm for this problem unless $P = NP$. Towards this end, we reduce 3-SAT to the problem of adding masking fault-tolerance in the low atomicity model. We first define the 3-SAT problem. Then, we show how to map a given 3-SAT problem into a problem of adding masking fault-tolerance. This mapping will identify the fault-intolerant program $p$, faults $f$, the invariant $S$, and *spec*. Finally, we show that the given 3-SAT formula is satisfiable iff the mapped decision problem has an affirmative answer.

**3-SAT problem.**

Given is a set of literals, $a_1, a_2, ..., a_n$ and $a'_1, a'_2, ..., a'_n$, where $a_i$ and $a'_i$ 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 $a_1, a_2, ..., a_n$ such that $c$ is satisfiable?

## 7.1 Mapping 3-SAT to Adding Masking Fault-Tolerance

Towards mapping the given 3-SAT problem into a problem of adding masking fault-tolerance, we first identify states and transitions that can be used in the problem of adding masking fault-tolerance. Then, we identify the variables needed and the values of those variables in each state. Finally, we present the read/write restrictions, the invariant, *spec*, and the fault transitions.

**States and transitions.** Given a formula with literals $a_1, a_2, ..., a_n$ and $a'_1, a'_2, ..., a'_n$ and a boolean formula $c_1 \wedge c_2 \wedge ... \wedge c_m$, we introduce one state, named $c_j$, for each disjunct $c_j$, $0 \leq j \leq m$. We also introduce six states, named $a_i, a'_i, b_i, b'_i, s_i$ and $s'_i$, for each pair of literals $a_i, a'_i$. For each disjunct $c_j$, we introduce three transitions, one for each literal in $c_j$. (For example, if $c_j$ is of the form $a_1 \vee a'_2 \vee a_4$, then the state $c_j$ has transitions going to states $a_1$, $a'_2$ and $a_4$.) The transitions between the six states introduced for each literal are as shown in Figure 1 (a).

**Fault-intolerant program, its invariant, faults and *spec*.** The invariant, $S$, of fault-intolerant program $p$ consists of $2n$ states, $s_1, s'_1, s_2, s'_2, ..., s_n, s'_n$. The fault-intolerant program consists of only self-loops at each of these states. The faults consist of the transitions $\{ (s_0, s_1) : s_0 \in S \wedge (\exists j : 1 \leq j \leq m : s_1 = c_j) \}$, i.e., the faults take the program from an invariant state to the states corresponding to each disjunct. The *spec* requires that all transitions other than the transitions described above (including the fault transitions) violate safety.

**Variables.** The fault-intolerant program consists of five variables, say $x, y, z, w, v$. The values of these variables in the states introduced above are as follows (cf. Figure 1 (b).):

- $c_j$ : $x(c_j)$ consists of a 6-tuple; the first, third and fifth part corresponds the literal (without considering complement or not) in $c_j$ and the second, fourth and sixth corresponds to whether the literal is complemented, e.g., if $c_j$ is of the form $a_1 \vee a'_2 \vee a_4$, $x(c_j)$ will be $\langle 1, 0, 2, 1, 4, 0 \rangle$. $y(c_j) = z(c_j) = 0, w(c_j) = v(c_j) = false$.
- $a_i$ : $x(a_i) = \langle i \rangle$, $y(a_i) = 0$, $z(a_i) = +0$, $w(a_i) = false$, $v(a_i) = true$.
- $a'_i$ : $x(a'_i) = \langle i \rangle$, $y(a'_i) = 1$, $z(a'_i) = -0$, $w(a'_i) = true$, $v(a'_i) = false$.

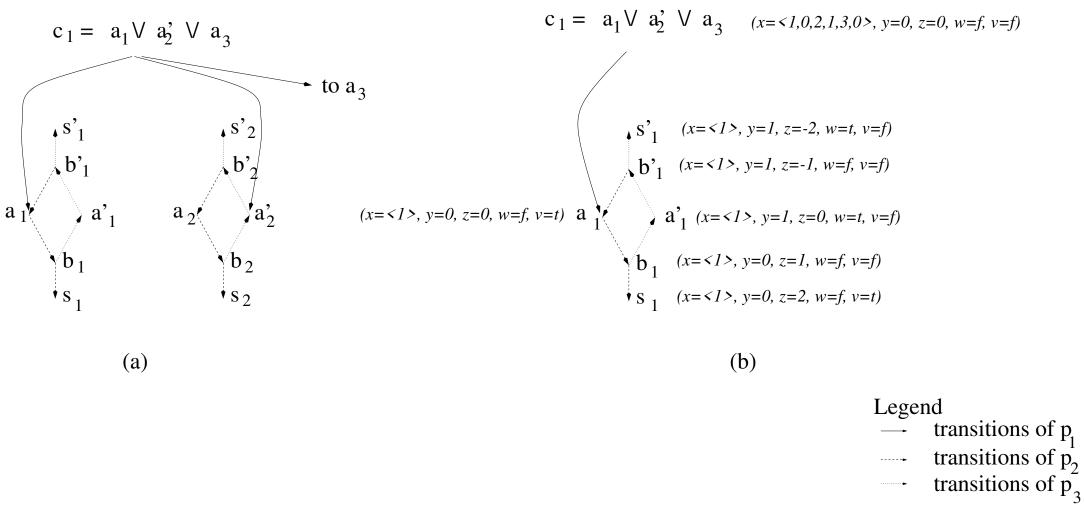


Figure 1: (a) : Permitted state transitions in the mapped program, (b) : Variable assignments

- $b_i$ : $x(b_i)=\langle i\rangle$, $y(b_i)=0$, $z(b_i)=+1$, $w(b_i)=false$, $v(b_i)=false$.
- $b'_i$ : $x(b'_i)=\langle i\rangle$, $y(b'_i)=1$, $z(b'_i)=-1$, $w(b'_i)=false$, $v(b'_i)=false$.
- $s_i$ : $x(s_i)=\langle i\rangle$, $y(s_i)=0$, $z(s_i)=+2$, $w(s_i)=false$, $v(s_i)=true$.
- $s'_i$ : $x(s'_i)=\langle i\rangle$, $y(s'_i)=1$, $z(s'_i)=-2$, $w(s'_i)=true$, $v(s'_i)=false$.

**Processes and read/write restrictions.** The low atomicity model consists of three processes, say $p_1, p_2, p_3$, and the read/write restrictions are as follows

- process $p_1$ can read all variables. Also, it can write all variables except $z$.
- process $p_2$ can read $x$ and $w$. It cannot write $x$ and $w$. Also, it can write (without being able to read) $y, z, v$ using the function $f$, where $y(f(s_0))=0, z(f(s_0))=max(z(s_0){+}1, 2), v(f(s_0))=((z\neq 2)\oplus(v(s_0))')$, i.e., it resets $y$ to zero, increments $z$, and negates $v$ if $z$ had not reached its maximum value.
- process $p_2$ can read $x$ and $v$. It cannot write $x$ and $v$. Also, it can write (without being able to read) $y, z, w$ using the function $f$, where $y(f(s_0)) = 1, z(f(s_0)) = min(z(s_0)-1, -2), w(f(s_0)) = ((z \neq -2) \oplus (w(s_0))')$, i.e., it sets $y$ to one, decrements $z$, and negates $w$ if $z$ had not reached its minimum value.

*Grouping of transitions.*

- Since process $p_1$ cannot write $z$, it cannot perform transitions between $a_i, a'_i, b_i, b'_i, s_i$ and $s'_i$. However, it can perform all transitions from $c_j$ to the literals included in $c_j$.
- For each $i$, process $p_2$ can perform only transitions $(b'_i, a_i), (a_i, b_i)$ and $(b_i, s_i)$. Moreover, since the values of $x$ and $w$ are the same in $b'_i, a_i, b_i$ and $s_i$, we need to form a group of these transitions, i.e., we must choose all three transitions if we choose either one.
- For each $i$, process $p_3$ can perform only transitions $(b_i, a'_i), (a'_i, b'_i)$ and $(b'_i, s'_i)$. Moreover, since the values of $x$ and $v$ are the same in $b_i, a'_i, b'_i$ and $s'_i$, we need to form a group of these transitions, i.e., we must choose all three transitions if we choose either one.

**Theorem 7.1** The given 3-SAT formula is satisfiable iff the answer to the mapped decision problem is affirmative.

**Proof.**

$\Rightarrow$ : There exists an assignment of truth values to the literals, $a_i$, $1 \le i \le n$, such that $c$ is true.

The invariant of the fault-tolerant program, $S'$, is identical to $S$, i.e., $S'$ contains $2n$ states $s_1, s'_1, s_2, s'_2, ..., s_n, s'_n$. We obtain the transitions of the fault-tolerant program $p'$ as follows:

- For each literal $a_i$, $1 \le i \le n$, if $a_i$ is assigned the truth value $true$, include the transitions $(b'_i, a_i), (a_i, b_i)$ and $(b_i, s_i)$. (Note that this is a grouping for process $p_2$.)
- For each literal $a_i$, $1 \le i \le n$, if $a_i$ is assigned the truth value $false$, include the transitions $(b_i, a'_i), (a'_i, b'_i)$ and $(b'_i, s'_i)$. (Note that this is a grouping for process $p_3$.)
- For each disjunct $c_j$, $1 \le j \le m$, include one transition from $c_j$ to one of the literals in $c_j$ which is assigned the truth value *true*. (Note that each of these transitions can be performed by process $p_1$.)

From the construction, it follows that if $p'$ is perturbed due to faults, it will eventually recover to states where $S'$ is *true*. Moreover, this recovery will maintain $SPEC$. Thus, $p'$ is masking $f$-tolerant from $S'$. Finally, the other two requirements of the decision problem are trivially satisfied as $S' = S$ and $p|S' = p'|S'$.

$\Leftarrow$ : Consider the case where a fault-tolerant $p'$ derived from the given fault-intolerant program exists.

Since the invariant $S'$ of $p'$ consists of at least one state in $S$ and the faults take the program from each state in $S$ to a state corresponding to $c_j$, $p'$ must include transitions so that $p'$ can recover from states corresponding to each $c_j$, $1 \le j \le m$. Since $c_j$ has exactly three possible outgoing transitions going to each of the literals included in it, $p'$ must choose one or more of those edges. If $p'$ includes an edge of the form $(c_j, a_i)$, then we set $a_i$ to *true* in the 3-SAT formula. If $p'$ includes an edge of the form $(c_j, a'_i)$ then we set $a_i$ to $false$ in the 3-SAT formula. It follows that each disjunct in the 3-SAT formula is truthified.

If some of the literals are not yet assigned a truth value, the satisfiability of the 3-SAT formula is independent of the truth values of those literals and, hence, without loss of generality, we assign the truth value *true* to those literals. Now, all that remains to show is that the truth values assigned to the literals are consistent, i.e., it is not the case that the literal $a_i$ is assigned both *true* and $false$. If $a_i$ is assigned the truth value *true* because of a transition of the form $(c_j, a_i)$ then $p'$ must include transition $(a_i, b_i)$. Moreover, due to the grouping constraints for $p'$, $p'$ must contain the transitions $(b'_i, a_i)$ and $(b_i, s_i)$. Similarly, if $a_i$ is assigned the truth value $false$, $p'$ must include transitions $(b_i, a'_i), (a'_i, b'_i)$ and $(b'_i, s'_i)$. Thus, if $a_i$ is assigned the truth value *true* and the truth value $false$ then $p'$ will include the cycle $(a_i, b_i), (b_i, a'_i), (a'_i, b'_i)$ and $(b'_i, a_i)$ and, hence, there exists a computation of $p'[]f$ which does not reach a state in $S'$. This is a contradiction since $p'$ is masking $f$-tolerant from $S'$. □

# 8 Examples

In this section, we show how the triple modular redundancy (TMR) can be designed using the algorithm *Add_ft* in Section 6. Since *Add_ft* is a non-deterministic algorithm, we will simply present a fault-intolerant program, the invariant of the fault-intolerant program, specification, faults, the fault-tolerant program and the invariant of the fault-tolerant program, and we will check that the conditions verified in algorithm *Add_ft* are indeed true.

For brevity, we write program transitions in terms of guarded commands [7]. A guarded command is of the form $g \longrightarrow st$ where $g$ is a state predicate and $st$ is statement that updates the variables in the program. The guarded command $g \longrightarrow st$ corresponds to the set of transitions $\{(s_0, s_1) : g$ is true in state $s_0$ and $s_1$ is obtained by 'executing' $st$ in state $s_0\}$. To describe the $TMR$ example, we first present the low atomicity model that describes the variables and the restrictions on reading and writing them. Then, we present the input to *Add_ft*, namely, the fault-intolerant program, its invariant, specification and faults. Subsequently, we present the output of $Add\_ft$, namely, the fault-intolerant program and its invariant. Finally, we check that the conditions verified in $Add\_ft$ are true.

**Model for $TMR$.** The triple modular redundancy program consists of five variables $in, x, y, z$ and $out$; $in$ is the 'input', $x, y, z$ are 'sensors' and $out$ is the 'output'. The domain of $in, x, y$ and $z$ is $\{0,1\}$. The domain of $out$ is $\{\perp, 0, 1\}$ ($out = \perp$ denotes that the value of $out$ is not yet assigned). The program consists of one process which can read $x, y, z, out$, and write $out$. The process cannot read $in$.

**Input to $Add\_ft$.** The fault-intolerant program simply assigns the value of $x$ to $out$ provided the value of $out$ is still not assigned. If $out$ is already assigned, the program simply stays in the same state. Thus, the fault-intolerant program consists of the actions: '$out = \perp \longrightarrow out := x$' and '$out \neq \perp \longrightarrow skip$'. The invariant of the fault-intolerant program consists of states where $out$ is not assigned and no sensors are corrupted or the states where the value of $out$ is equal to $in$. Thus, the invariant of the fault-intolerant program is $\{s_0 : (out(s_0) = \perp \ \wedge \ (x(s_0) = y(s_0) = z(s_0) = in(s_0))) \ \vee \ (out(s_0) = in(s_0))\}$. The faults perturb one sensor when all sensors are correct. Thus, fault actions are as follows: '$x = y = z = in \longrightarrow x := 1 - x$', '$x = y = z = in \longrightarrow y := 1 - y$', and '$x = y = z = in \longrightarrow z := 1 - y$'. The set of transitions that violate the safety of $spec$ are $\{(s_0, s_1) : out(s_1) \neq in(s_1) \ \wedge \ out(s_1) \neq \perp\}$. (Recall that we do not require the liveness specification explicitly.)

**Output of $Add\_ft$.** The fault-tolerant program consists of four actions: '$x = majority(x, y, z) \ \wedge \ out = \perp \longrightarrow out := x$', '$y = majority(x, y, z) \ \wedge \ out = \perp \longrightarrow out := y$', '$z = majority(x, y, z) \ \wedge \ out = \perp \longrightarrow out := z$', '$out \neq \perp \longrightarrow skip$'. The invariant of the fault-tolerant program is the same as that of the fault-intolerant program.

Observe that the transitions that the fault-tolerant program executes in the absence of faults (i.e., the transitions that originate in the invariant of the fault-tolerant program) are the same as those executed by the fault-intolerant program. Letting $T'$ to be the set of states reached in any computation of the fault-intolerant program in the presence of the faults, we leave it to the reader to check that the other conditions verified in $Add\_ft$ are true.

Regarding other examples, we would like to note that in [2] we have presented the derivation of a fault-tolerant token ring program from the corresponding fault-intolerant token ring program. And, in [8], we have presented the derivation of a fault-tolerant Byzantine agreement program [9] from the corresponding fault-intolerant program. In both these cases, the derived fault-tolerant programs satisfy the conditions verified in algorithm $Add\_ft$. It follows that the algorithm $Add\_ft$ can be used to derive fault-tolerant programs for these problems as well.

# 9 Related Work

Initial work on program synthesis such as [10] was designed to produce only high atomicity programs where reading all the variables and writing all the them in one atomic step was permitted. Since then several new algorithms [11–15] have been proposed which ensure that the synthesized programs have lower atomicity. For example, the algorithms in [11,13,16] produce programs that have Read/Write atomicity. The algorithm in [15] is designed to generate CSP programs for a special topology where one central process is connected to all other processes in a star network.

Previous work [10, 12, 15] on program synthesis has addressed the problem of synthesizing a program that satisfies a given specification. These synthesis algorithms are based on the decision procedure for testing temporal satisfiability proposed by Emerson and Clarke [10] and Manna and Wolper [15]. Previous work on synthesizing fault-tolerant programs includes [11, 12, 17, 18]. In [12, 18], the authors assume that the fault cannot affect the internal state of a process; the fault can interact only through input and output signals. By way of contrast [11] and this paper permits a more general model of faults, where faults can affect the all variables of a process. Also, our low atomicity model is more general than the Read/Write model used in [11, 16].

Unlike the previous work that starts from a temporal specification, our work deals with adding fault-tolerance to a given fault-intolerant program that is known to be correct with respect to its specification. For this reason, we believe that this approach will be especially useful if a fault-intolerant program is already known or if other constraints (such as unavailability of a complete specification of the given fault-intolerant program) require that we reuse the fault-intolerant program.

In terms of complexity, our low atomicity algorithm as well as that of the algorithms in [12,16] is exponential in the state space of the fault-tolerant program. Although the algorithm in [11] is polynomial in the state space of the fault-intolerant program, it is incomplete.

# 10 Conclusion and Future Work

In this paper, we focused on transforming a fault-intolerant program into a fault-tolerant program. We considered three levels of fault-tolerance, namely failsafe, nonmasking and masking. We showed that in the high atomicity model, where the program can read and write all the variables in one atomic step, all these transformations can be performed in polynomial time in the size of the fault-intolerant program. We also showed that in the low atomicity model, where the program consists of processes each of which can only read and write only a limited set of variables, all these transformations can be performed in exponential time in the size of the fault-intolerant program. Moreover, we demonstrated that in the low atomicity model the problem of adding masking fault-tolerance to a given fault-intolerant program is NP-hard. We showed how our algorithm can be applied to construct the triple modular redundancy algorithm. Also, we argued that our algorithm can be used to derive other fault-tolerant programs for Byzantine agreement and token ring circulation.

Unlike previous work that starts with a specification (typically in some temporal logic), we start with a fault-intolerant program that is known to be correct. For this reason, our algorithms only needed the safety specification that the program is supposed to satisfy in the presence of faults; the algorithms did not need the liveness specification.

We would like to note that our low atomicity model is more general than the Read/Write model considered elsewhere [11, 16] in the literature. More specifically, the Read/Write model is an instance of our low atomicity model (cf. Section 5). This generality is especially important if the primitives available to the designer permit a more general model, e.g., the interleaving model. More specifically, if a fault-tolerant program in the interleaving model exists although a fault-tolerant program in Read/Write model does not, our algorithm can still be used. We are not aware of previous work that deals with synthesizing fault-tolerant program for such a general model.

We have identified two issues in Section 6.1 that make the transformation problem difficult in the low atomicity model. Both these issues are applicable in the design of masking fault-tolerance, and the problem of adding masking fault-tolerance is NP-hard. However, the lower bound on the complexity of adding failsafe and nonmasking fault-tolerance is still an open question as only the first issue is applicable in the design of failsafe fault-tolerance and only the second issue is applicable in the nonmasking fault-tolerance. It is our conjecture that the problem of adding failsafe fault-tolerance in the low atomicity model is in P, and the problem of adding nonmasking fault-tolerance is NP-hard.

Regarding other extensions to our work, we will be focusing on reducing the complexity of our algorithms by identifying heuristics. These heuristics will be specially tailored according to the type of fault being considered. By focusing on only a limited types of faults, it may be possible to solve the transformation problem in polynomial time.

Another extension of this work is to precompute a set of fault-tolerance components which often occur in fault-tolerant programs and use these components directly in the synthesis algorithm. We expect that these precomputed fault-tolerance components will not only help in reducing the complexity of designing fault-tolerant programs but also permit the derived fault-tolerant program to be more efficient.

## References


[1] A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *International Conference on Distributed Computing Systems*, pages 436–443, May 1998. An extended version of this paper has been invited to *IEEE Transactions on Computers*.

[2] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

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

[4] S. S. Kulkarni, J. Rushby, and N. Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilization (WSS'99) Austin, Texas, USA*, pages 33–40, June 1999.

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

[6] 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.

[7] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[8] S. S. Kulkarni and A. Arora. Compositional design of multitolerant repetitive byzantine agreement. *Proceedings of the Seventeenth International Conference on Foundations of Software Technology and Theoretical Computer Science, Kharagpur, India*, pages 169–183, dec 1997.

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

[10] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synchronize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[11] 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.

[12] A. Pnueli and R. Rosner. On the synthesis of a reactive module. *ACM Symposium on Principles of Programming Languages*, pages 179–190, 1989.

[13] P. Attie and E. Emerson. Synthesis of concurrent systems for an atomic read/write model of computation. *ACM Symposium on Principles of Distributed Computing*, 1996.

[14] P. Attie and E. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Transactions on Programming Languages and Systems*, 20(1):51–115, 1998.

[15] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6:68–93, 1984.

[16] D. Dill and H. Wong-Toi. Synthesizing processes and schedulers from temporal specifications. *International Conference on Computer-Aided Verification*, 1990.

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

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