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 automatic transformation of a given fault-intolerant program into a fault-tolerant program. We show how such a transformation is achieved for three levels of fault-tolerance properties, failsafe, nonmasking and masking. For high atomicity model, where programs can read all program variables as well as write all program 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 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 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

¹Email: sandeep@cse.msu.edu, anish@cis.ohio-state.edu
Web: http://www.cse.msu.edu/~sandeep, http://www.cis.ohio-state.edu/~anish
Tel: +1-517-435-2387, 1-614-292-1836

This work was partially sponsored by NSF Grant MDA904-96-1-0111, NSF Grant NSF-CCR-9972368, NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF equipment grant EIA-0130724, an Ameritech Faculty Fellowship, a 2002-2003 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. When a program is initially designed, it may not be possible to anticipate all faults that it may be subject to. Hence, such transformation is often required during maintenance and evolution of existing programs. During such transformation, it is desirable to reuse the existing fault-intolerant program. Moreover, the reuse of the fault-intolerant program is mandatory if the fault-intolerant program is the de-facto specification of the problem at hand.

In our previous work [1–3], we have shown that such transformation is not only possible but provides new insights in the design of fault-tolerant programs. In particular, we have shown 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. Moreover, we have found that programs designed using fault-tolerance components are easier to understand [2, 3] than programs where functionality and fault-tolerance are designed hand-in-hand.

While our previous work on manual design has shown that transforming a fault-intolerant program into a fault-tolerant program is possible and desirable, we expect that an automated algorithm to solve the transformation problem will further help in adding fault-tolerance to existing programs. More specifically, such automated algorithm 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.

With this motivation, we develop automated algorithms where we transform a given fault-intolerant program into a fault-tolerant program in such a way that the transformation is done solely for the purpose of dealing with faults. More specifically, we focus on the transformation problem where we do not introduce new ways to satisfy the specification in the absence of faults.

Choice of the model of computation for the transformation problem. If we consider a very general model of computation, e.g., processes that communicate via messages with unbounded message queues, then the transformation problem becomes undecidable. Hence, we need to consider simpler models of computation where the complexity of the transformation problem is low. We consider two such models of computation, the high atomicity model and the low atomicity model. In the high atomicity model, the program can read and write all program variables in an atomic step. In other words, the high atomicity model allows the program to perform operations such as ‘test-and-set’ over program variables. The low atomicity model is more general in that it allows us to specify restrictions on what the program can read and write.

Both these models can be used to describe abstract versions of distributed programs. Thus, the automation algorithm for the transformation problem in these models can be used to obtain an abstract version of the fault-tolerant program. While the approach to translate an abstract version of a fault-intolerant program into its concrete implementation is outside the scope of this paper, techniques such as tolerance preserving refinements [4–8] can be used to obtain a concrete implementation of the fault-tolerant programs.

Levels of fault-tolerance for the transformation problem. The level of fault-tolerance identifies the extent to which the original specification is satisfied when faults occur. Although one could consider arbitrary levels of fault-tolerance, experience shows that for most programs, the fault-tolerance level falls into one of the three levels, fail-safe, nonmasking or masking. These levels are based on [9] where Alpern and Schneider have shown that a specification can be decomposed into a safety specification and a liveness specification. More specifically, these levels are based on whether the fault-tolerant program satisfies the safety specification, the liveness specification or both.

In the presence of faults, a fail-safe fault-tolerant program only satisfies its safety specification. Failsafe fault-tolerance is often used in safety-critical systems. For the second level, nonmasking, we observe that satisfying
the liveness specification alone is not very useful as the liveness specification does not impose any restrictions on finite computations of a program. Hence, in addition to satisfying the liveness specification, we require that safety violation be temporary. Thus, after the occurrence of faults, a nonmasking program recovers to states from where its specification (both safety and liveness) is satisfied. Nonmasking fault-tolerance is often used in several networking related applications (e.g., routing, spanning tree maintenance, etc.) where maintaining safety in the presence of faults is either too expensive or impossible. Finally, in the presence of faults, a masking program satisfies both the safety and the liveness specification. Masking fault-tolerance is the ideal fault-tolerance and it is used in database applications as well as in several problems (e.g., leader election, mutual exclusion, etc.) in distributed systems. (See Section 2 for precise definitions of these three levels of fault-tolerance.)

Contributions of the paper. 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).

For brevity and simplicity of presentation, we have limited ourselves to toy examples to illustrate the algorithms in this paper. The algorithms presented in this paper have been used for more complex examples [8,10,11] to add fault-tolerance to several faults, e.g., input corruption, failure of a process and byzantine faults. More specifically, our algorithm in [10] is obtained by adding heuristics to our low atomicity algorithm, $Add_{ft}$, presented in Section 6; it is the first automated synthesis algorithm where fault-tolerance is added to byzantine faults. A more detailed discussion of these examples and the synthesis tool used to develop them are discussed in Section 9.

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. Finally, we discuss related work in Section 8 and make concluding remarks in Section 9.

2 Preliminaries

In this section, for the sake of completeness, we recall essentially standard definitions of programs, problem specifications, faults, and fault-tolerance. Programs are specified in terms of their state space and their transitions. The definition of specifications is adapted from Alpern and Schneider [9]. And, the definition of faults and fault-tolerance is due to Arora and Gouda [12] and Arora and Kulkarni [1].

2.1 Program

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 $\{(s_0, s_1) : s_0, s_1 \in S_p\}$. A state predicate of $p$ ($\langle S_p, \delta_p \rangle$) is any subset of $S_p$. A state predicate $S$ is closed in the program $p$ (respectively, $\delta_p$) iff $(\forall (s_0, s_1) : (s_0, s_1) \in \delta_p : (s_0 \in S \Rightarrow s_1 \in S))$, i.e., if the transition originates in a state in $S$ then it must terminate in a state in $S$. A sequence of states, $\langle s_0, s_1, \ldots \rangle$, 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 $\langle s_0, s_1, \ldots \rangle$ is finite and terminates in state $s$ then there does not exist state $s$ such that $\langle s, s \rangle \in \delta_p$. 

2
The projection of program $p$ on state predicate $S$, denoted as $p|S$, is the program $\langle S_p, \{(s_0, s_1) : (s_0, s_1) \in \delta_p \land s_0, s_1 \in S\}\rangle$. 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.** We call $\delta_p$ as the transitions of $p$. When it is clear from context, we use $p$ and $\delta_p$ interchangeably. Also, we say that a state predicate $S$ is true in a state $s \in S$.

### 2.2 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 $\sigma$ 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.

Following Alpern and Schneider [9], we let the specification consist of a safety specification and a liveness specification. For our transformation algorithm, 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 transformation algorithm; we show that the fault-tolerant program satisfies the liveness specification iff the fault-intolerant program satisfies the liveness specification. Moreover, in the transformation problem, the initial fault-intolerant program satisfies its specification (including the liveness specification). Thus, the liveness specification need not be specified explicitly.

**Remark.** Since the specification is suffix closed and fusion closed, it is always possible to specify the safety specification as a set of bad transitions. This result was proved earlier in [3]. We repeat the proof for reader’s convenience in Appendix A.1. Also, we refer the reader to [3] 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 $\text{spec}$, we say that $p$ refines $\text{spec}$ 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 $\text{spec}$. If $p$ refines $\text{spec}$ from $S$ and $S \neq \{ \}$, we say that $S$ is an invariant of $p$ for $\text{spec}$.

For a finite sequence (of states) $\alpha$, we say that $\alpha$ maintains $\text{spec}$ iff there exists a sequence of states $\beta$ such that $\alpha \beta \in \text{spec}$. Similarly, we say that $\alpha$ violates $\text{spec}$ iff it is not the case that $\alpha$ maintains $\text{spec}$.

**Notation.** Let $\text{spec}$ be a specification. We use the term safety of $\text{spec}$ to mean the smallest safety specification that includes $\text{spec}$. 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 $\text{spec}$.

### 2.3 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).

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 $p \parallel f$ 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 $p \parallel f$. Thus, at each state where an invariant $S$ of $p$ is true, an $f$-span $T$ of $p$ from $S$ is also true. Also, $T$, like $S$, is closed in $p$. Moreover, if any transition in $f$ is executed in a state where $T$ is true, then $T$ is also true in the resulting state. It follows that for all computations of $p$ that start at states where $S$ is true, $T$ is a boundary in the state space of $p$ up to which (but not beyond which) the state of $p$ may be perturbed by the occurrence of the transitions in $f$. 
Just as we defined the computation of $p$, we say that a sequence of states, $\langle s_0, s_1, \ldots \rangle$, is a **computation of $p$ in the presence of $f$** iff the following three conditions are satisfied: (1) $\forall j : j > 0 : (s_{j+1}, s_j) \in (\delta_p \cup f)$, (2) if $\langle s_0, s_1, \ldots \rangle$ is finite and terminates in state $s_t$ then there does not exist state $s$ such that $(s, s_t) \in \delta_p$, and (3) $\exists n \geq 0 : (\forall j > n : (s_{j+1}, s_j) \in \delta_p)$. 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.

### 2.4 Fault-Tolerance

Using the definitions above, we now define what it means for a program to be fault-tolerant. We define three levels of fault-tolerance; **failsafe, nonmasking and masking**. Irrespective of the level of tolerance, in the absence of faults, a program should refine its specification. And, the level of fault-tolerance characterizes the extent to which the program refines $\text{spec}$ in the presence of faults. A failsafe fault-tolerant program ensures that in the presence of faults, the safety of $\text{spec}$ is maintained. A nonmasking fault-tolerant program ensures that in the presence of faults, the program recovers to states from where $\text{spec}$ is satisfied. A masking fault-tolerant program ensures that in the presence of faults both these properties are satisfied. Thus, these three levels of fault-tolerance are defined as follows:

Program $p$ is **failsafe $f$-tolerant** to $\text{spec}$ from $S$ iff the following two conditions hold: (1) $p$ refines $\text{spec}$ from $S$, and (2) there exists $T$ such that $T$ is an $f$-span of $p$ from $S$ and $p \parallel f$ maintains $\text{spec}$ from $T$.

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

Program $p$ is **masking $f$-tolerant** to $\text{spec}$ from $S$ iff the following two conditions hold: (1) $p$ refines $\text{spec}$ from $S$, and (2) there exists $T$ such that $T$ is an $f$-span of $p$ from $S$, $p \parallel f$ maintains $\text{spec}$ from $T$, and for every computation of $p \parallel f$ that starts from a state in $T$ has a state in $S$.

**Notation.** Henceforth, whenever the program $p$ is clear from the context, we will omit it; thus, “$S$ is an invariant” abbreviates “$S$ is an invariant of $p$” and “$f$ is a fault” abbreviates “$f$ is a fault for $p$”. Also, whenever the specification $\text{spec}$ and the invariant $S$ are clear from the context, we omit them; thus, “$f$-tolerant” abbreviates “$f$-tolerant for $\text{spec}$ from $S$”, and so on.

### 3 Problem Statement

In this section, we formally specify the problem of transforming a fault-intolerant program, say $p$ into a fault-tolerant program, say $p'$. Towards this end, we first identify constraints so that $p'$ is obtained by adding only fault-tolerance from $p$. Then, we discuss the soundness and completeness issues in the context of the transformation problem. Subsequently, in Section 3.1, we discuss generalization of the transformation problem, additional properties that are desirable for a solution to the transformation problem and suggestions about inputs provided to 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, $\delta_p$, and its invariant, $S$. The specification, $\text{spec}$, 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 $\text{spec}$, 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., we should be able to prove that \( p' \) refines \( \text{spec} \) from \( S' \) by simply using that \( p \) refining \( \text{spec} \) from \( S \). To precisely state this requirement, 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 \( S \). However, we cannot prove that this computation is in \( \text{spec} \) by just using the fact that \( p \) refines \( \text{spec} \) from \( S \) as we have no information about computations of \( p \) that originate outside \( S \). 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 \( \text{spec} \) in the absence of faults. Once again, we cannot prove that the resulting computation is in \( \text{spec} \) by just using the fact that \( p \) refines \( \text{spec} \) from \( S \). Therefore, we require that \( p'|S' \subseteq p|S' \).

Using the above two requirements, we define the transformation problem as follows (this definition will be instantiated in the obvious way for the special cases failsafe \( f \)-tolerance, nonmasking \( f \)-tolerance and masking \( f \)-tolerance):

**The Transformation Problem**
Given \( p, S, \text{spec} \) and \( f \) such that \( p \) refines \( \text{spec} \) from \( S \).
Identify \( p' \) and \( S' \) such that:
\[
S' \subseteq S,
\]
\[
p'|S' \subseteq p|S', \text{ and}
\]
\[
p' \text{ is } f \text{-tolerant to } \text{spec} \text{ from } S'.
\]

To define soundness and completeness in the context of the transformation problem, we define the corresponding decision problem: (This definition will also be instantiated for failsafe \( f \)-tolerance, nonmasking \( f \)-tolerance and masking \( f \)-tolerance):

**The Decision Problem**
Given \( p, S, \text{spec} \) and \( f \) such that \( p \) refines \( \text{spec} \) from \( S \).
Does there exist \( p' \) and \( S' \) such that:
\[
S' \subseteq S,
\]
\[
p'|S' \subseteq p|S', \text{ and}
\]
\[
p' \text{ is } f \text{-tolerant to } \text{spec} \text{ from } S'.
\]

**Notations.** Given \( p, \text{spec}, S \) and \( f \) as input, we say that \( p' \) and \( S' \) solve the transformation problem for this 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 \( p' \) and \( S' \) solve the transformation problem.

**Soundness and completeness.** An algorithm for the transformation problem is sound iff for any given input, its output, namely \( p' \) and \( S' \), 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 \( S' \).

### 3.1 Comments on the Problem Statement

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. In other words, if the specification can be met in different ways then the fault-intolerant program should not prematurely eliminate some of those ways. Once again, this is due to the fact that in the transformation problem, we only add fault-tolerance; we do not introduce new ways to satisfy the specification in the absence of faults.

The above discussion suggests that the derived fault-tolerant program should also have the weakest possible invariant, fault-span, and maximal non-determinism. An algorithm that provides such guarantees about its
output has the potential for its use in multitolerance [2], where fault-tolerance to multiple faults is added in a stepwise manner. For example, if we need to add fault-tolerance to faults $f_1$ and $f_2$, then we could first use the synthesis algorithm to add fault-tolerance to $f_1$ and then use the output from the synthesis algorithm as an input to add fault-tolerance to $f_2$. In this case, it is desirable that the intermediate program (one that tolerates $f_1$) should have the weakest invariant and maximal non-determinism. With this intuition, we show that in our solutions in Section 4, the invariant (respectively, fault-span) of the derived program is the weakest possible invariant (respectively, fault-span) that solves the transformation problem. Also, the program generated by our algorithms provide maximal non-determinism inside the invariant.

Our problem statement also 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 a generalized 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. However, when new variables are introduced, we need to address how the faults affect those variables. Since the way in which the new variables are affected is fault-dependent, the synthesis algorithm must take the effect of faults on new variables as an input. With the knowledge of how faults affect the new variables, the synthesis algorithm needs to systematically introduce new variables. Although such a general transformation problem is outside of scope of this paper, we briefly discuss how our algorithms can be used to solve such a general transformation problem in Appendix A2.

4 Adding Fault-Tolerance in High Atomicity Model

In this section, we consider the transformation problem for programs in the high atomicity model. To clarify the concept of atomicity, we refine the notion of the state space of a program. More specifically, we let the program consist of a finite set of variables each with a finite domain. A program state is obtained by assigning a value (from its domain) to each variable. And, the state space of the program is all possible states that can be obtained by assigning different values to the variables.

With this refined notion of the state space, in the high atomicity model, a program transition can read all variables as well as write all variables in one atomic step. In other words, if the enumerated states of the program are $s_0, s_1, \ldots, s_{\text{max}}$ then the program transitions can be any subset of $\{(s_j, s_k) : 0 \leq j \leq \text{max}\}$. If restrictions are imposed on what a program can read and write, these restrictions are translated into the subsets of $\{(s_j, s_k) : 0 \leq j \leq \text{max}\}$ that may be used as transitions. We discuss this in the next section.

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 use a running example of the parking lot problem, described first in Section 4.1.2. Also, we present soundness and completeness proofs for our algorithms.

In the algorithm for the failsafe and masking fault-tolerance, the calculation of the invariant of the fault-tolerant program involves computing the fixpoint of a formula that in turn includes other fixpoint calculations. We, therefore, present the algorithm as follows: Each statement in the program is either a fixpoint calculation or a simple statement that can be computed by examining the state space only once. Essentially, the simple statements are used as shorthand to simplify the fixpoint formulae. Finally, the invariant of the fault-tolerant program is computed as a fixpoint of a formula that includes the fixpoints computed earlier. Once the invariant is computed, the transitions of the fault-tolerant program can be computed in a straightforward manner. Since the invariant is a fixpoint formula, it can be efficiently calculated using techniques in the literature; in this paper, we simply show that it is possible to compute the required fixpoint in polynomial time. (In the case of masking fault-tolerance, the computation of the fault-span is also a fixpoint calculation.)
4.1 Automating the Addition of Failsafe Tolerance

As mentioned 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 \(f\), and (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 reached in the computations of \(p' \parallel f\), and (b) state \(s_0\) is never reached in any computation of \(p' \parallel f\). In the former subcase, the transition \((s_0, s_1)\) must be removed as the safety of \(spec\) can be violated if \(p' \parallel f\) 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 \(p' \parallel f\) 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 \((s_{-1}, s_0)\), is a fault transition then we must recursively remove all transitions of the form \((s, s_{-1})\) for each state \(s\).

Using the above two cases, our algorithm to obtain the failsafe fault-tolerant program is as follows. First, it identifies states from where execution of one or more fault transitions violates safety. This is done by a (smallest) fixpoint calculation where we begin with the empty set. Then, we add state \(s_0\) to this set if there exists a fault transition \((s_0, s_1)\) such that either (1) \((s_0, s_1)\) violates safety, or (2) \(s_1\) is added to this set earlier. Thus, the set, \(m_s\), of states from where faults alone can violate safety is the smallest fixpoint of the following equation:

\[
X = X \cup \{s_0 : (\exists s_1 : (s_0, s_1) \in f \land (s_1 \in X \lor (s_0, s_1) \text{ violates } spec)\}\}
\]

Then, we compute the transitions, \(mt\), that must be removed from \(p\). These transitions fall in two categories: (1) transitions that reach states in \(ms\), and (2) transitions that violate the safety of \(spec\).

If there exist states in the invariant such that execution of one or more fault transitions from those states violates the safety of \(spec\), then we recalculate the invariant by removing those states. The calculation of the invariant is a (largest) fixpoint calculation. As shown in Theorem 4.6, any invariant \(S'\) that solves the transformation problem must be a subset of \(S - ms\). Also, the same theorem shows that if \(p'\) solves the transformation problem then \(p' | S'\) must be a subset of \(p - mt\). Moreover, in Observation 4.1, we show that if \(p'\) and \(S'\) solve the transformation problem then no state in \(S'\) can be a deadlock state. Hence, the invariant \(S'\) equals \(ConstructInvariant(S - ms, p - mt)\), where \(ConstructInvariant(S, p)\) is the (largest) fixpoint of the equation:

\[
X = (X \cap S) - \{s_0 : (\forall s_1 : s_1 \in X : (s_0, s_1) \notin p)\}
\]

Finally, we compute the transitions of the fault-tolerant program by removing transitions of \(p - mt\) that start from a state in \(S'\) but reach a state outside \(S'\). Thus, 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)
{
    $ms := smallestfixpoint(X = X \cup \{(s_0, s_1) : (s_1 \in f \land (s_0, s_1) \text{ violates } spec)\})$
    $mt := \{(s_0, s_1) : ((s_0, s_1) \in ms) \lor (s_0, s_1) \text{ violates } spec\}$;
    $S' := \text{ConstructInvariant}(S - ms, p - mt)$;
    if ($S' = \emptyset$) declare no failsafe $f$-tolerant program $p'$ exists;
    else $p' := \text{ConstructTransitions}(p - mt, S')$
}

ConstructInvariant($S :$ state predicate, $p :$ transitions)
// Returns the largest subset of $S$ such that computations of $p$ within that subset are infinite
return largestfixpoint($X = (X \cap S) - \{(s_0) : (\forall s_1 : s_1 \in X : (s_0, s_1) \not\in p)\}$)

ConstructTransitions($p :$ transitions, $S :$ set of states)
{ return $p - \{(s_0, s_1) : s_0 \in S \land s_1 \not\in S\} \}$

### 4.1.1 Soundness and Completeness of Add_failsafe

Before presenting the soundness and completeness proofs of Add_failsafe, we summarize observations about programs and specifications. These observations will be used in the proofs of Add_nonmasking and Add_masking as well.

Recall 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 4.1** 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.

**Observation 4.2** If $p'$ is (nonmasking or masking) $f$-tolerant for $spec$ from $S'$ then all computations of $p' \parallel f$ that start from a state in $S'$ must be infinite.

Note 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 $p$. By considering such transitions, we allow the designer to distinguish between valid terminations and deadlock states.

We also make additional observations about the relation between $S'$ and $ms$ and $p'$ and $mt$ below. These observations are used in the verification of Add_failsafe and Add_masking.

**Observation 4.3** $S'$ \cap ms = \{

**Observation 4.4** Every computation of $p'$ that starts from a state in $S'$ is infinite.

**Theorem 4.5** 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. $S' \subseteq S$.

   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, $(p'|S') \subseteq (p|S')$. 

8
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 $spec$. Also, by construction, $S'$ is closed in $p'$. Hence, $p'$ refines $spec$ from $S'$.

We let the fault-span $T'$ to be the set of states reached in any computation of $p'\parallel f$ that starts from a state in $S'$. Consider a computation prefix $c$ of $p'/\parallel f$ that starts from a state in $T'$. From the definition of $T'$, there exists a computation prefix $c'$ of $p'/\parallel f$ such that $c$ is a suffix of $c'$ and $c'$ starts from a state in $S'$. We now show that each prefix of $c'$ maintains $spec$; in turn implies that each prefix of $c$ maintains $spec$.

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 $\langle s_{(n-1)}, s_n \rangle$ be the smallest such prefix. It follows that $\langle s_{(n-1)}, s_n \rangle$ violates the safety of $spec$ and, hence, $(s_{(n-1)}, s_n) \in mt$. Since $(s_{(n-1)}, s_n)$ is a transition of $p'/\parallel f$ and, by construction, $p'$ does not contain any transitions in $mt$; $(s_{(n-1)}, s_n)$ is a transition of $f$. If $(s_{(n-1)}, s_n)$ is a transition of $f$ then $s_{(n-1)} \in ms$. If $s_{(n-1)} \in ms$, it follows that $(s_{(n-2)}, s_{(n-1)}) \in mt$. By the same argument, $(s_{(n-2)}, s_{(n-1)})$ is a transition of $f$. Hence, $s_{(n-2)} \in ms$. Continuing thus, by induction, if $(s_0, s_1, ..., s_n)$ violates the safety of $spec$, $s_0 \in ms$, which is a contradiction since $s_0 \in S'$ (cf. Observation 4.3). Thus, each prefix of $c'$ maintains $spec$. By suffix closure, each prefix of $c$ also maintains $spec$. Thus, $p'/\parallel f$ maintains $spec$ from $T$.

Theorem 4.6 Algorithm $Add_{failsafe}$ is complete.

Proof. Let program $p''$ and predicate $S''$ solve the transformation problem. Clearly, $S'' \cap ms = \emptyset$; 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 from a state in $S''$ must be an infinite computation, if it were to be in $spec$ (cf. Observation 4.1). It follows that there exists a nonempty 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 nonempty 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.

The above proof also shows the maximality of the invariant and the transitions within them. Thus, we have

Corollary 4.7 Invariant output by $Add_{failsafe}$ is the largest invariant that solves the transformation problem.

Corollary 4.8 Let the output of $Add_{failsafe}$ be the fault-tolerant program $p'$ and its invariant $S'$. If $p''$ and $S'$ solve the transformation problem then $p''|S'' \subseteq p'|S'$.

Theorem 4.9 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 fixpoint calculation for the invariant can also be done in polynomial time; we begin with $S - ms$ and in each iteration remove one or more deadlocked states. In each iteration, at least one deadlocked state is removed. Thus, the number of iterations to compute the invariant is at most $|S - ms|$. (4) The complexity of ConstructTransitions is clearly polynomial.

Remark. We do not explicitly identify the fault-span in $Add_{failsafe}$. If the fault-tolerant program output by $Add_{failsafe}$ is $p'$ and its invariant is $S'$, we can compute the fault-span by identifying states reached in the computations of $p'/\parallel f$ starting from a state in $S'$. However, this is not the weakest possible
fault-span. The weakest possible fault-span is true − ms, i.e., it includes all states except those from where faults alone violate safety. Also, we can obtain maximal transitions within the fault-span if we add to p′ all transitions that (1) begin in a state outside S′ and (2) are not in mt. We leave it to the reader to check that the program obtained by adding these transitions solves the transformation problem.

4.1.2 Case Study : Parking Lot Problem

![Parking Lot Problem Diagram]

To illustrate the algorithms presented in this paper, we use the parking lot problem, presented next (cf. Figure 1). The parking lot contains three spots for cars (marked a, b and c). There are two gates to enter the parking lot. Immediately after each gate, there is a space where the customer can drop the car off (marked l and r). If the gate is open, the customer may leave the car in this spot. The car will then be moved to one of the spots (a, b or c). We assume that if the car is being moved from l or r to a, b or c, no car can enter during this move. At the exit, there is one door. A car parked in spots a, b or c can leave through this door. However, only one car can leave the parking lot at a time. All doors are one-way, i.e., cars in spots l and r cannot leave and a car cannot enter in spots a, b or c.

Thus, in the parking lot problem, there are three possible events: (1) a car could be dropped off at the left gate and (possibly) moved to spots a, b or c, (2) a car could be dropped off at the right gate and (possibly) moved to spots a, b or c, or (3) a car could exit. For simplicity, we assume that each event is atomic. Also, we assume that in each spot there can be at most one car. Now, we describe the state space of the fault-intolerant program, its invariant, its safety specification, and its transitions. The fault-intolerant program, invariant, specification and faults identified in this section are used as input to the transformation problem. We use the same input for Add fail-safe (this section), Add nonmasking (in Section 4.2.2), and Add masking (in Section 4.3.2).

Variables and the state space of fault-intolerant program, IP. To model the parking lot, we maintain three variables; x, y and z. The variable x denotes the number of cars that can be let in through the left gate, y denotes the number of cars that can be let in through the right gate, and z denotes the number of cars in the lot. Hence, the left (respectively, right) gate is open when x (respectively, y) is positive. The domain of x and y is {0, 1, 2, 3}. The domain of z is {0, 1, 2, 3, 4, 5}. A state of IP is obtained by assigning each variable a value from its domain. The state space of IP, thus, contains 4x4x6 (=144) states.

Safety specification, sfIP. The safety specification requires that any car in the parking lot should be able to leave. Clearly, a car in spots a, b or c can leave. However, if spots a, b and c are occupied and there is a car in spot l (respectively, r) then that car cannot leave. Hence, it is required that the value of z is at most three. Also, to model the assumption that only one event (entry/exit) can occur at a time, it is required that the value of x, y and z can change at most by 1. Thus, the safety specification rules out the following transitions:

\[ sf_{IP} = \{(s_0, s_1) : z(s_0) > 3 \lor z(s_1) > 3 \lor |x(s_1) - x(s_0)| > 1 \lor |y(s_1) - y(s_0)| > 1 \lor |z(s_1) - z(s_0)| > 1\} \]

Note that the above safety specification allows x, y and z to change simultaneously. To model the requirement that if a car enters the left gate then another car cannot enter the right gate at the same time, we could strengthen the above safety specification to include a predicate of the form: ‘if the value of z is changed then
the value of $y$ cannot change.' However, we have chosen the above specification to simplify the presentation, especially while adding masking fault-tolerance.

**Transitions $\delta_{IP}$.** For brevity, we write program transitions in terms of guarded commands [13]. A guarded command is of the form $g \rightarrow st$, where $g$ is a state predicate and $st$ is statement that updates the variables in the program. The guarded command $g \rightarrow st$ corresponds to the set of transitions $\{(s_0, s_1) : g \text{ is true in state } s_0 \text{ and } s_1 \text{ is obtained by ‘executing’ } st \text{ in state } s_0\}$. The fault-intolerant program $IP$ contains four actions: The first two actions let a car enter, and the last two actions let a car exit. Upon exit, the value $x$ (or $y$) is non-deterministically increased so that a new car can enter. For simplicity, we do not model the actions corresponding to the movement of the car inside the parking lot. Thus, the transitions of the fault-intolerant program, $\delta_{IP}$, are captured by the following actions.

$$IP1 :: \quad x > 0 \land z < 5 \quad \rightarrow \quad x, z := x - 1, z + 1$$
$$IP2 :: \quad y > 0 \land z < 5 \quad \rightarrow \quad y, z := y - 1, z + 1$$
$$IP3 :: \quad y < 3 \land z > 0 \quad \rightarrow \quad y, z := y + 1, z - 1$$
$$IP4 :: \quad x < 3 \land z > 0 \quad \rightarrow \quad x, z := x + 1, z - 1$$

**Invariant, $S_{IP}$.** We let the invariant of the fault-intolerant program, $S_{IP}$, be $x + y + z = 3$.

Once again, this is not the only possible invariant; $x + y + z \leq 3$ is another possibility. We have chosen $S_{IP}$ as it describes several facets of the algorithm while keeping the presentation simple.

**Faults.** For our case study, we consider two faults. The first fault action, $F1$, allows a car to sneak in. We will use this fault to demonstrate an example where failsafe and masking fault-tolerance cannot be designed. The second fault action, $F2$, allows a car to sneak out. This fault action will be used to demonstrate an example where fault-tolerance can be added. Thus, the fault actions are as follows:

$$F1 :: \quad z < 5 \quad \rightarrow \quad z := z + 1$$
$$F2 :: \quad z > 0 \quad \rightarrow \quad z := z - 1$$

Now, given the transitions $\delta_{IP}$, faults $F1/F2$ and the specification $sf_{IP}$, we need to develop a fault-tolerant algorithm by transforming $IP$.

**Adding failsafe fault-tolerance to $F1$.** First, we compute the set $ms$, the set of states from where faults alone can violate safety. Towards this end, consider states $s_1, s_2, s_3$ and $s_4$ where the value of $x$ and $y$ is 0 and the value of $z$ in state $s_j$ is $j$. Observe that $(s_3, s_4)$ is a transition of fault $F1$ that violates safety. It follows that $s_3$ should be included in $ms$. Also, $(s_2, s_4)$ is a transition of $F1$ and, hence, $s_2$ is included in $ms$. Continuing thus, state $s_1$ will also be included in $ms$. Following this discussion, it is easy to see that starting from an arbitrary state, fault $F1$ can cause the value of $z$ to be greater than 3. Thus, $ms$ includes all states. And, since $ms$ contains all possible states, it follows that $mt$ contains all possible transitions.

Now, the first argument to ConstructInvariant, $S_{IP} - ms$, is the empty set. Hence, ConstructInvariant will return the empty set. Subsequently, $Add\_failsafe \; \delta_{IP}$ will declare that failsafe fault-tolerance cannot be added. (This is an expected result; if the fault can continue to increase the value of $z$, it will not be possible to guarantee that $z$ will be at most 3.)

**Adding failsafe fault-tolerance to $F2$.** Once again, we compute $ms$ for $F2$. Observe that if the value of $z$ in a state is 3 or less, execution of $F2$ cannot violate safety. However, if the value of $z$ in a state is 4 or 5, execution of $F2$ violates safety. Hence, $ms$ equals the set $\{s_0 : z(s_0) > 3\}$. Thus, $S_{IP} - ms$ equals $S_{IP}$.

The set $mt$ includes the set $sf_{IP}$ and the set of transitions that reach $ms$. It follows that $mt$ equals $sf_{IP}$. Using the values of $ms$ and $mt$, ConstructInvariant is called with parameters $S_{IP}$ and $\delta_{IP} - mt$. We leave it to the reader to verify that ConstructInvariant returns the set $S_{IP}$.

The transitions of the failsafe fault-tolerant program are obtained by calling ConstructTransitions. ConstructTransitions removes transitions of $IP$ that originate in $S_{IP}$ but reach a state outside $S_{IP}$. Since $S_{IP}$
is an invariant of $IP$, no such transitions exist. Thus, the transitions of the failsafe fault-tolerant program, $\delta_{FP}$ are as follows:

$$\delta_{FP} = \{(s_0, s_1) : (s_0, s_1) \in \delta_{IP} \land z(s_0) \leq 3 \land z(s_1) \leq 3\}$$

From the above discussion, the invariant of the failsafe fault-tolerant program is $S_{IP}$ and its transitions are those transitions of $IP$ where the value of $z$ in the initial and final state is 3 or less. In the presence of faults, the failsafe fault-tolerant program may reach a state where $x+y+z$ is less than 3. In this case, the failsafe fault-tolerant program can deadlock if it reaches a state where $x$, $y$ and $z$ are 0.

**Remark.** Note that the execution of $F2$ from states in $S_{IP}$ did not violate the safety specification. Hence, the invariant of the fault-tolerant program remained unchanged. If we had considered the fault where $z$ could be increased only if the initial value of $z$ was 3 then in that case the invariant of the fault-tolerant program would have been $S_{IP} \land (z < 3)$.

### 4.2 Automating the Addition of 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:

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

### 4.2.1 Soundness and Completeness of Add_nonmasking

**Theorem 4.10** Algorithm $Add_{\text{nonmasking}}$ 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. □

**Corollary 4.11** Invariant output by $Add_{\text{nonmasking}}$ is the largest invariant that solves the transformation problem.

**Corollary 4.12** Let the output of $Add_{\text{nonmasking}}$ be the fault-tolerant program $p'$ and its invariant $S'$. If $p''$ and $S'$ solve the transformation problem then $p''|S' \leq p'|S'$.

**Theorem 4.13** Algorithm $Add_{\text{nonmasking}}$ is in $P$. □

### 4.2.2 Case Study: Parking Lot Problem (continued)

We consider the problem of adding nonmasking fault-tolerance to $F1$ and $F2$ identified in Section 4.1.2. Applying $Add_{\text{nonmasking}}$ to program $IP$ with invariant $S_{IP}$, the invariant and the transitions of the nonmasking fault-tolerant program, say $OP$ are as follows:

$$S_{OP} = S_{IP}$$

$$\delta_{OP} = \{(s_0, s_1) : (s_0 \in S_{IP} \land (s_0, s_1) \in \delta_{IP}) \lor (s_0 \notin S_{IP} \land s_1 \in S_{IP})\}$$

Thus, in states in $S_{IP}$, the nonmasking program has the same transitions as $IP$. And, from states outside $S_{IP}$, $OP$ simply recovers to any state where $S_{IP}$ is true. Note that some of these recovery transitions violate
safety; for example, a transition that changes the value of \( x \) (respectively, \( y \) or \( z \)) by more than one is included in these transitions. In the derivation of a masking fault-tolerant version of \( IP \), we need to ensure that such transitions are not executed. In other words, it is necessary that the recovery to the invariant should occur without these transitions.

4.3 Automating the Addition of 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 4.6, our first estimate of \( S' \) is \( S_1 \) where \( S_1 = \text{ConstructInvariant}(S - ms, p - mt) \). And, we estimate \( T' \) to be \( T_1 \) where \( T_1 = \text{true} - ms \), i.e., \( T_1 \) includes all states except those in \( ms \).

The computation of the invariant and the fault-span for the fault-tolerant program is also a (largest) fixpoint calculation. However, due to the nesting involved in this fixpoint calculation, for simplicity, we present it operationally as a loop (lines 5-13) where we strengthen \( S_1 \) and \( T_1 \) while ensuring that if some \( S' \) solves the transformation problem then \( S' \Rightarrow S_1 \). This loop contains three key steps (lines 7, 8 and 9). Line 7 is a simple statement that computes the transitions, \( p_1 \), that may be used if \( S_1 \) is the invariant and \( T_1 \) is the fault-span. Lines 8 and 9 are fixpoint computations for the fault-span and the invariant respectively. The value of \( p_1 \) is used to compute these fixpoints. These steps are as follows:

1. To compute the transitions, \( p_1 \), first, we include transitions that start from a state in \( S_1 \). By the second condition of the transformation problem, these transitions are \( p|S_1 \). Then, we consider transitions that start from a state in \( T_1 - S_1 \). By the closure of the fault-span, these transitions include those that reach a state in \( T_1 \). Finally, we remove the transitions \( mt \) from this set.

2. We recompute the fault-span on line 8. This computation itself is a (largest) fixpoint computation. For this fixpoint calculation, we first remove states in \( T_1 \) from where it is not possible to reach a state in \( S_1 \) using the transitions of the program identified in step 7. Now, if there are states, say \( s_0 \) and \( s_1 \), such that \( s_0 \) is in the fault-span, \( s_1 \) is outside the fault-span and \((s_0, s_1)\) is a fault transition then \( s_0 \) must be removed from the fault-span. Thus, we strengthen \( T_1 \) to \( \text{ConstructFaultSpan}(T_1 - \{s : S_1 \text{ is not reachable from } s \text{ in } p_1 \}, f) \), where \( \text{ConstructFaultSpan}(T, f) \) is the (largest) fixpoint of the following equation:

\[
X = (X \cap T) - \{s_0 :: (\exists s_1 :: (s_0, s_1) \in f \land s_1 \notin X)\}
\]

3. Since \( S_1 \) must be a subset of \( T_1 \), we recompute (line 9) the invariant as \( \text{ConstructInvariant}(S_1 \land T_1, p_1) \) where \( \text{ConstructInvariant} \) itself is a (largest) fixpoint calculation.

We continue the loop on lines 5-13 until we achieve the largest fixpoint for \( S_1 \). After the largest fixpoint is found, we compute the program transitions by assigning ranks to all states in \( T_1 \) and removing cycles where all states are outside \( S_1 \). Thus, our algorithm for adding masking fault-tolerance is as follows:
Add\_masking(p, f : transitions, S : state predicate, spec : specification)
{} 
ms := smallest fixedpoint(X = X \cup \{s_0 :: (\exists s_1 : (s_0, s_1) \in f \land (s_1 \in X \lor (s_0, s_1) \text{ violates } spec)\}) (1)
mt := \{(s_0, s_1) : ((s_1 \in ms) \lor (s_0, s_1) \text{ violates } spec)\}; (2)
S_1 := \text{ConstructInvariant}(S - ms, p - mt); (3)
T_1 := \text{true} - ms; (4)

repeat
T_2, S_2 := T_1, S_1;
p_1 := p|S_1 \cup \{(s_0, s_1) : s_0 \notin S_1 \land s_0 \in T_1 \land s_1 \in T_1\} - mt; (6)
T_1 := \text{ConstructFaultSpan}(T_1 - (s : S_1 \text{ is not reachable from } s \text{ in } p_1, f)); (8)
S_1 := \text{ConstructInvariant}(S_1 \land T_1, p_1); (9)
if (S_1 = \{\} \lor T_1 = \{\})
decide no masking f-tolerant program p' exists; (10)
exit (12)
until (T_1 = T_2 \land S_1 = S_2); (13)

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

ConstructFaultSpan(T : state predicate, f : transitions)
// Returns the largest subset of T that is closed in f.
{}
return largest fixedpoint(X = (X \cap T) - \{s_0 : (\exists s_1 : (s_0, s_1) \in f \land s_1 \notin X)\})

4.3.1 Soundness and Completeness of Add\_masking

Similar to the derivation of fail-safe fault-tolerance, we make the following observation about the output of Add\_masking.

Observation 4.14. T' \cap ms = {} \hspace{1cm} \Box

Lemma 4.15. p' \parallel f maintains spec from T'.

Proof. This proof is similar to that of Theorem 4.5 (part 3). By construction, T' is closed in p' \parallel f. Let c be a computation of p' \parallel f that starts from a state in T'. If c violates the safety of spec, there exists a prefix, say \langle s_0, s_1, \ldots, s_n \rangle, that violates the safety of spec. Wlog, let \langle s_0, s_1, \ldots, 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 ms. 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 (s_0, s_1, \ldots, s_n) violates the safety of spec, s_0 \in ms, which is not possible since s_0 \in T' (cf. Observation 4.14). Thus, p' \parallel f maintains spec from T'. \hspace{1cm} \Box

Lemma 4.16. 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, \ldots \rangle, that starts from a state in T'. By construction of p' (line 16), we have (\forall j : \text{Rank}(s_j) > 0 \Rightarrow \text{Rank}(s_{j+1}) < \text{Rank}(s_j)). It follows there exists a state s_{n_1}, s_n \in S', in \langle s_0, s_1, \ldots \rangle. \hspace{1cm} \Box

Corollary 4.17 Every computation of p' \parallel f that starts from a state in T' contains a state in S'.

\hspace{1cm} \Box

14
Theorem 4.18 Algorithm Add\_masking is sound.

Proof.

1. $S' \subseteq S$. The proof of this part is identical to case 1 in Theorem 4.5.

2. $p | S' \subseteq p | S'$. The proof of this part is identical to case 2 in Theorem 4.5.

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 4.15, it follows that $p' \ || f$ maintains $spec$ from $T'$. From Lemma 4.16, it follows that every computation of $p' \ || f$ that starts from a state in $T'$ has a state in $S'$. Also, by construction, $T'$ is closed in $p' \ || f$. Combining these results, $p'$ is masking $f$-tolerant to $spec$ from $S'$.

Theorem 4.19 Algorithm Add\_masking is complete.

Proof. Let program $p''$ and predicate $S''$ solve the 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 4.6, $S'' \subseteq ConstructInvariant(S - ms, p - mt)$. 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 can 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 $T_1$ such that $p''$ and $S_1$ solve the transformation problem. For a transition $(s_0, s_1)$ of $p''$, we consider three cases depending upon whether $s_0 \in S_1$, $s_0 \in (T_1 - S_1)$, or $s_0 \notin T_1$.

- 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 $s_0$. Hence, wlog, we assume that $p''$ does not contain any transitions that originate in a state outside $T_1$.

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, $T'' \not\subseteq (T_1 - \{s : S_1 \text{ is not reachable from } s \text{ in } p_1\})$. Moreover, if there exists a fault transition $(s_0, s_1)$ such that $s_0 \in (T_1 - \{s : S_1 \text{ is not reachable from } s \text{ in } p_1\})$ and $s_1 \notin (T_1 - \{s : S_1 \text{ is not reachable from } s \text{ in } p_1\})$, $s_0$ also cannot be in $T''$. It follows that $T''$ is a subset of 'the largest subset of $(T_1 - \{s : S_1 \text{ is not reachable from } s \text{ in } p_1\})$ 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 9 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 $S'' \subseteq (S_1 \cap T_1)$. Also, all computations from $S''$ are infinite (cf. observation 4.1) and ConstructInvariant($S_1 \cap (T_1, p_1$) returns the largest subset of $S_1 \cap T_1$ from where all computations are infinite, $S'' \subseteq ConstructInvariant(S_1 \cap T_1, p_1$). 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\_m\_masking is complete. 

The above proof also shows the maximality of the invariant and the transitions within it. Thus, we have

**Corollary 4.20** Invariant output by Add\_m\_masking is the largest invariant that solves the transformation problem.

**Corollary 4.21** Let the output of Add\_m\_masking be the fault-tolerant program \( p' \) and its invariant \( S' \). If \( p'' \) and \( S \) solve the transformation problem then \( p'' \mid S' \subseteq p' \mid S \).

**Theorem 4.22** Algorithm Add\_m\_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 4.9, each statement in Add\_m\_masking is in \( P \). Thus, the algorithm Add\_m\_masking is in \( P \).

**Modification for stepwise synthesis.** As discussed earlier, the invariant and the fault-span computed by an algorithm that solves the transformation problem should be maximal if the output of the transformation algorithm is to be used as an input when fault-tolerance is added in a stepwise fashion. Corollaries 4.20 and 4.21 show the maximality of the invariant and the transitions inside it. Regarding the fault-span, these conditions are satisfied by the value of \( T_1 \) and \( p_1 \) at the end of Add\_m\_masking. However, to ensure that the fault-tolerant program does not remain in \( T_1 - S_1 \) forever, we removed certain transitions from \( p_1 \). This removal may be premature if the output of Add\_m\_masking is to be used as an input to add fault-tolerance to another fault. Hence, we suggest that for such stepwise construction, \( p_1 \) should be used as the output of Add\_m\_masking; removing cycles that stay entirely outside the invariant should be done only after fault-tolerance is added to all faults at hand.

### 4.3.2 Case Study: Parking Lot Problem (continued)

**Adding masking fault-tolerance to \( F_1 \).** As in the case of failsafe fault-tolerance, we begin with computation of \( ms, mt \) and the first guess at the invariant of the fault-tolerant program. As discussed in Section 4.1.2, the invariant \( S_1 \) computed on line 3 is the empty set and, hence, Add\_m\_masking will declare that masking fault-tolerance cannot be added. Once again this is expected; if a fault could increase the value of \( z \) arbitrarily, we cannot ensure that \( z \) is bounded by 3.

**Adding masking fault-tolerance to \( F_2 \).** As in the case of failsafe fault-tolerance, the invariant \( S_1 \) on line 3 will be equal to \( S_1 \). Also, since \( ms \) equals the set \( \{ z > 3 \} \), the value of \( T_1 \) on line 4 is \( z \leq 3 \).

Now, we consider the first iteration of the loop on lines 5-13. On line 7, we compute \( p_1 \) as follows. First, we include a transition that originates in \( S_1 \) (i.e., where the value of \( x + y + z \) equals 3) iff it is in \( \delta_{TP} \). Then, we add a transition that originates in \( T_1 - S_{TP} \) (i.e., where the value \( z \) is less than 3 but the sum of \( x, y \) and \( z \) is not 3) iff it reaches a state in \( T_1 \). Then, we remove the transitions in \( mt \) (i.e., transitions where the value of \( x \) (or \( y \) or \( z \) ) changes by more than 1 and transitions where the value of \( z \) is more than three).

Now, observe that from each state in \( T_1 \), it is possible to reach a state in \( S_{TP} \) by using the transitions in \( p_1 \). (From any state in \( T_1 \), we can systematically increase or decrease \( x, y \) and \( z \) by 1 so that \( x + y + z \) equals 3.) Hence, the value of \( T_2 \) is the same as \( T_1 \). Subsequently, on line 9, \( S_2 \) is also the same as \( S_1 \). Thus, the loop on line 13 terminates.

After the loop terminates, we consider the transitions in \( p_1 \) and decide the rank of each state in \( T_1 \). Consider the state where \( x = 0, y = 0 \), and \( z = 0 \). From this state, in one transition, we can reach a state where \( x = 1, y = 1 \) and \( z = 1 \), and the resulting state is in \( S_1 \). Hence, the rank of the state where \( x = 0, y = 0 \) and \( z = 0 \) is 1. Likewise, the rank of the states where the sum of \( x, y \) and \( z \) is in the set \( \{ 0, 1, 2, 3, 4, 5 \} \) is 1.

For states where \( x + y + z \) equals 6, if all three variables are non-zero, in one transition a state in \( S_1 \) is reached (by decreasing each \( x, y \) and \( z \) ). Hence, the rank of these states is 1. Now, consider a state, say \( s \), where the value of one variable, say \( x \), is 0. If \( x + y + z \) equals 6 then the values of \( y \) and \( z \) must be 3. From \( s \), it is
not possible to reach $S_i$ by using one transition. However, by using two transitions, it is possible to reach a state in $S_i$. Hence, the rank of a state, where $x+y+z$ equals 6 and the value of $x$ (respectively, $y$ or $z$) is 0, is 2.

Finally, the rank of the states where $x+y+z$ equals 7, 8 or 9, is 2.

Now, to obtain the transitions of the masking fault-tolerant program (line 16), we use the transitions of $p_1$ where the rank decreases by 1. Thus, the transitions of the masking fault-tolerant program, $p'$ are as shown in Figure 2.

### Figure 2: Transitions of the Masking Fault-tolerant Program to Parking Lot Problem

**Remark.** Note that all states in $T_1$ are not reached in the presence of $F2$. However, as argued in Section 3, one of the goals of our algorithms is to compute the largest possible invariant and largest possible fault-span. This is desirable if the program output by our synthesis procedure is to be used as an input to add fault-tolerance to another fault. Also, the parking lot program designed in this section is masking fault-tolerant to the fault that increments/decrements the value of $x$, $y$ and $z$ such that the value of these variables is at 3 or less.

## 5 Characterizing the Low Atomicity Model

The algorithms in Section 4 use an abstract notion of a *program state*. Towards characterizing the low atomicity model, we refine this notion. We let the program contain a (finite) set of variables with their respective (finite) domains. A *state* of a program is obtained by assigning each variable a value from its domain. Since the algorithms in Section 4 assume that the fault-tolerant program can contain a transition $(s_0, s_1)$ for any two states $s_0, s_1$, they implicitly assume that all variables can be read and written in one atomic step. In this section, 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.

To model the read-write restrictions, we introduce the concept of processes. More specifically, for each restriction on how variables can be read or written, we introduce one process. In Section 5.1, we identify how two commonly used models namely, Read/Write model and Shared Memory Interleaving model, can be characterized as low atomicity models. Then, in Section 5.2, we describe how we model restrictions on the ability to write. Section 5.3 describes how the read restrictions are captured. Finally, in Section 5.4, we show how to combine the read and write restrictions.
5.1 Examples of Low Atomicity Models

Read/Write model. In the read/write model (that is used for automated synthesis in [14,15]), the
program consists of nodes and an adjacency relation that specifies their neighborhood relation. Each node
contains its public variables and local (private) variables that are used to track the variables of the neighbors.
Each node is allowed to either (1) atomically read the state of one of its neighbors and write its local variables
(which cannot be read by other processes), or (2) atomically read all its local copies and update its variables
(including the variables that can be read by its neighbors). For the read/write model, we introduce the
processes as follows:

- If \( k \) is a neighbor of \( j \) then we introduce a process \( \langle j, k \rangle \). This process can read the public variables of
  \( k \) and the local variables of \( j \), and it can write the local variables at \( j \). A similar process \( \langle k, j \rangle \) is also
  introduced if the neighborhood relation is symmetric.

- For each node \( j \), we introduce a process. This process can read and write all variables of \( j \).

Shared memory interleaving model. Once again, in this model the program consists of nodes and an
adjacency relation that specifies their neighborhood relation. A node can read the variables of its neighbors
and write its own variables. Hence, we introduce a process for each node.

5.2 Modeling Write Restrictions

Towards characterizing the restriction on the ability to read and write, we define the following notation.

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

Let \( w_j \) denote the set of variables that \( j \) is allowed to write. Now, 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 \( nwrite(j, w_j) \) that \( j \) cannot use in the synthesis
algorithm, where

\[
nwrite(j, w_j) = \{ (s_0, s_1) : (\exists x : x \notin w_j : x(s_0) \neq x(s_1)) \}
\]

In the synthesis of (failsafe and masking) fault-tolerant programs, we needed to ensure that transitions that
violate the safety specification are not used. For a given process, the write restrictions simply prohibit the use
of some additional transitions. Hence, for each transition, we need to keep track of the process responsible
for executing it. Except for this modification, the write restriction does not create any difficulties during
synthesis.

5.3 Modeling Read Restrictions

Given a transition \( (s_0, s_1) \), we can easily determine the variables that need to be written. However, since
each state specifies the values of all variables, on the surface, the transition \( (s_0, s_1) \) needs to read all the
variables. To understand the modeling of read-restrictions, we consider a simple example first.

Consider a program consisting of variables \( a \) and \( b \) and let their domain be \( \{0,1\} \). Also, consider a process
that cannot read the variable \( a \). Now, consider the transition from the state \( \langle a = 0, b = 0 \rangle \) to the state
\( \langle a=0, b=1 \rangle \). This transition can be thought of as 'if \( a \) is 0 and \( b \) is 0 then set \( b \) to 1'. Clearly, in this case,
the process must read \( a \). However, if we also include the transition from the state \( \langle a=1, b=0 \rangle \) to the state
\( \langle a=1, b=1 \rangle \) then these two transitions can be thought of as 'if \( b \) is 0 then set \( b \) to 1'. In other words, the
inability to read causes the transitions \( (\langle a = 0, b = 0 \rangle, \langle a = 0, b = 1 \rangle) \) and \( (\langle a = 1, b = 0 \rangle, \langle a = 1, b = 1 \rangle) \) to be
grouped. And, we need to choose all transitions in this group or omit all of them.
More generally, if \( r_j \) denotes the set of variables that \( j \) is allowed to read and \( w_j \) denotes the set of variables that \( j \) is allowed to write, we determine the transitions that need to be grouped. 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 \( w_j \subseteq r_j \), \( s'_0 \) and \( s'_1 \) must agree on the values of variables that are not in \( r_j \).

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 \( \text{group}(j, r_j)(s_0, s_1) \), for the case where \( w_j \subseteq r_j \), where

\[
\text{group}(j, r_j)(s_0, s_1) = \{(s'_0, s'_1) : (\forall x : x \in r_j : x(s_0) = x(s'_0) \land x(s_1) = x(s'_1)) \land \\
(\forall x : x \notin r_j : x(s'_1) = x(s'_0) \land x(s_1) = x(s_0))\}
\]

**Blind writes.** 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 \( \text{chan}_j \) denote the sequence of messages on channel \( \text{chan} \) which is an outgoing channel from process \( j \). When \( j \) sends a message, it does so by appending it to \( \text{chan}_j \) and, thus, \( j \) writes \( \text{chan}_j \). However, \( j \) cannot read what messages are still pending on channel \( \text{chan} \), i.e., \( j \) cannot read \( \text{chan}_j \).

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

More generally, consider the case where \( j \) can write multiple variables, say \( x_1, x_2, \ldots \), 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 \( \text{group}(j, f, r_j)(s_0, s_1) \) must also be transitions of \( j \), where

\[
\text{group}(j, f, r_j)(s_0, s_1) = \{(s'_0, s'_1) : (\forall x : x \in r_j : x(s_0) = x(s'_0) \land x(s_1) = x(s'_1)) \land \\
(\forall x : x \not\in r_j : x(s'_1) = x(f(s'_0)) \land x(s_1) = x(f(s_0)))\}
\]

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 \( p \mid S \). Also, the read/write restrictions apply for the program transitions only. Faults are not restricted in any way, i.e., a fault transition can read and write all the variables in one atomic step.

### 5.4 Combining Read Restrictions and Write Restrictions

As 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 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 the 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 related to the low atomicity model that make the transformation more complex. Then, we present our algorithm in Section 6.2. Finally, in Section 6.3, we describe examples where the algorithm in Section 6.2 can be applied; we state one example, triple modular redundancy, in detail and point out other examples.

### 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 $p'$ that starts from a state in $S'$. We now discuss two issues that make the transformation in the low atomicity model difficult.

- 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 $p' \parallel f$ 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 $p' \parallel f$ can reach $s_0$ from where safety of $spec$ will be violated if $p'$ executes the transition $(s_0, s_1)$. Conversely, 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.

### 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 each 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 $p' \parallel f$.

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 $T' \cap m_s = \{\}$ 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:

```plaintext
Algorithm AddAtt

AddAtt(p, f : transitions, S : state predicate, spec : specification,
        type : {failsafe, nonmasking, masking}, g0, g1, ..., gm.ax : groups of transitions)
{
  ms := smallest fixpoint(X = X \cup \{s_0 :: (\exists s_1 : (s_0, s_1) \in f \land (s_1 \in X \lor (s_0, s_1) violates spec))\})
  mt := \{(s_0, s_1) : ((s_1 \in ms) \lor (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' \parallel 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 \lor type = masking)
    T' \cap ms = \{\}; (p'|T') \cap mt = \{\};
  if (type = nonmasking \lor 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 AddAtt is sound and complete.

\[\square\]

21
**Theorem 6.2** Algorithm $Add_{All}$ is in NP.

**Proof.** This proof follows from Observation 5.1 and the fact that each statement in algorithm $Add_{All}$ is executed in polynomial time. □

### 6.3 Examples of Low Atomicity Programs

In this section, we show how the triple modular redundancy (TMR) can be designed using the algorithm $Add_{All}$. Since $Add_{All}$ 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 check that the conditions verified in algorithm $Add_{All}$ are indeed true. We also point out other examples where $Add_{All}$ can be used.

We write program transitions in terms of guarded commands [13]. Recall that a guarded command is of the form $g \rightarrow st$ where $g$ is a state predicate and $st$ is statement that updates the variables in the program. The guarded command $g \rightarrow st$ corresponds to the set of transitions $\{(s_0, s_1) : g \text{ true in state } s_0 \text{ and } s_1 \text{ is obtained by 'executing' } st \text{ in state } s_0\}$. Now, we describe the model of TMR, input to $Add_{All}$ and output of $Add_{All}$.

**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 $\{\emptyset, \top\}$ ($out = \top$ 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 $Add_{All}$.** 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:

- $out = \emptyset \rightarrow out := x$
- $out \neq \emptyset \rightarrow skip$

**Invariant of the fault-intolerant program.** 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

$$\{s_0 : (out(s_0) = \emptyset \land (x(s_0) = y(s_0) = z(s_0) = \text{in}(s_0))) \lor (out(s_0) = \text{in}(s_0))\}$$

**Faults.** The faults perturb one sensor when all sensors are correct. Thus, fault actions are as follows:

- $x = y = z = \text{in} \rightarrow x := 1 - x$
- $x = y = z = \text{in} \rightarrow y := 1 - y$
- $x = y = z = \text{in} \rightarrow z := 1 - z$

**Safety specification.** The set of transitions that violate the safety of spec are as follows (Recall that we do not require the liveness specification explicitly.):

$$\{(s_0, s_1) : out(s_1) \neq \text{in}(s_1) \land out(s_1) \neq \emptyset\}.$$  

**Output of $Add_{All}$.** The fault-tolerant program guessed by $Add_{All}$ consists of four actions:

- $x = \text{majority}(x, y, z) \land out = \emptyset \rightarrow out := x$
- $y = \text{majority}(x, y, z) \land out = \emptyset \rightarrow out := y$
- $z = \text{majority}(x, y, z) \land out = \emptyset \rightarrow out := z$
- $out \neq \emptyset \rightarrow skip$

22
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.

Other examples. For brevity, we chose to present a simple triple modular redundancy example to illustrate how \( Add_{ft} \) is used. We have used \( Add_{ft} \) for several other programs as well. These examples include token rings [2], byzantine agreement [10,16,17], consensus [11] and commit [11]. We have also presented a heuristic-based deterministic implementation of \( Add_{ft} \) in [10]. We have used this deterministic implementation in a synthesis tool that solves the transformation problem. This tool has been used to derive solutions for Byzantine agreement, token rings and majority 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 \( c = c_1 \land c_2 \land ... \land c_m \), 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 \( c_1 \land c_2 \land ... \land c_m \), we introduce one state, named \( c_j \), for each disjunct \( c_j \), \( 0 \leq j \leq m \). We also introduce six states, named \( a_1, a'_1, a_2, a'_2, ..., a_n, a'_n \), and \( b_i, s_i \) 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 \( a_i \lor a'_i \lor a_j \), then the state \( c_j \) has transitions going to states \( a_1, a'_2, a'_4 \).) The transitions between the six states introduced for each literal are as shown in Figure 3 (a).

Fault-intolerant program, its invariant, faults and \( spec \). The invariant, \( S \), of fault-intolerant program \( p \) consists of \( 2n \) states, \( s_1, s'_1, s_2, s'_2, ..., s_m, s'_m \). The fault-intolerant program consists of only self-loops at each of these states. The faults consist of the transitions \{ \( (s_0, s_1) : s_0 \in S \land (\exists j : 1 \leq j \leq m : s_1 = c_j) \} \), i.e., the faults take the program from an invariant state to the states corresponding to each disjunct. The specification, \( spec \), 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 \(x, y, z, w\) and \(v\). The values of these variables in the states introduced above are as follows (cf. Figure 3 (b)):

- \(c_j : x(c_j)\) consists of a 6-tuple; the first, third and fifth part corresponds to the literals (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 \(a_1 \lor a_2 \lor a_3\), \(x(c_j)\) will be \((1, 0, 2, 1, 4, 0)\). \(y(c_j) = z(c_j) = 0, w(c_j) = v(c_j) = false\).

- \(a_i : x(a_i) = (i), y(a_i) = 0, z(a_i) = 0, w(a_i) = false, v(a_i) = true\).

- \(a'_i : x(a'_i) = (i), y(a'_i) = 1, z(a'_i) = 0, w(a'_i) = true, v(a'_i) = false\).

- \(b_i : x(b_i) = (i), y(b_i) = 0, z(b_i) = 1, w(b_i) = false, v(b_i) = false\).

- \(b'_i : x(b'_i) = (i), y(b'_i) = 1, z(b'_i) = -1, w(b'_i) = false, v(b'_i) = false\).

- \(s_i : x(s_i) = (i), y(s_i) = 0, z(s_i) = 2, w(s_i) = false, v(s_i) = true\).

- \(s'_i : x(s'_i) = (i), y(s'_i) = 1, z(s'_i) = -2, w(s'_i) = true, v(s'_i) = false\).

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) \(y, z\) and \(v\) 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) \(y, z\) and \(w\) 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 $y$ 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 $y$ 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, $a_i, 1 \leq i \leq n$, such that $c$ is true.

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

- For each literal $a_i$, $1 \leq i \leq n$, 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 $a_i$, $1 \leq i \leq n$, 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 $c_j$, $1 \leq j \leq m$, 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 program $p'$ 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 $c_j, 1 \leq j \leq m$. 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 transitions. If $p'$ includes a transition of the form $(c_j, a_i)$, then we set $a_i$ to true in the 3-SAT formula. If $p'$ includes a transition 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 the corresponding positive (unprimed) 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. For $a_i$ 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 $S'$. 

\[\square\]
8 Related Work

Initial work on program synthesis such as [18] was designed to produce only high atomicity programs, where reading and writing of all variables in one atomic step was permitted. Since then several new algorithms [14, 15, 19–22] have been proposed which ensure that the synthesized programs have lower atomicity. For example, the algorithms in [14, 15, 20, 23] produce programs that have Read/Write atomicity. Thus, these algorithms assume that in an atomic step, a process can read a remote variable or write its own variable but not both. The algorithm in [22] focuses on distributed programs where there is one central process that communicates with all other processes in a system.

Previous work [14, 15, 18–22] 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 [18] and Manna and Wolper [22]. One of the major differences between previous synthesis algorithms and our synthesis algorithm is its input. We begin with an existing program whereas the algorithms in [14, 15, 18–22] begin with a specification is some temporal logic. For this reason, we believe that our algorithms 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.

Previous work on synthesizing fault-tolerant/distributed programs includes [15, 19, 24–26]. In [19, 25], 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, [15] and this paper permits a more general model of faults, where faults can affect all variables of a process.

One of the issues in the synthesis algorithms is the efficiency of the synthesized program. It is complex to model efficiency of the synthesized program during synthesis. Moreover, in [21], authors have pointed out that programs synthesized using temporal specification (in which efficiency issues are not modeled explicitly) are likely to be inefficient. By beginning with a fault-intolerant program, however, our synthesis algorithm has the potential to preserve the efficiency of the fault-intolerant program.

9 Conclusion and Future Work

In this paper, we considered the problem of adding fault-tolerance to a given fault-intolerant program. The input to the problem was a fault-intolerant program, its invariant, its safety specification and faults. The output of the problem was the fault-tolerant program and its invariant.

While solving the transformation problem, we considered three commonly encountered fault-tolerance levels and two models of computation. The three levels –failsafe, nonmasking and masking– were based on the extent to which the original specification is satisfied when faults occur. The models of computation –high atomicity model and the low atomicity model– were based on the extent to which processes in a program can read and write program variables. For each of these models, we presented a sound and complete algorithm for the three levels of fault-tolerance. We illustrated our algorithms by considering simple problems –the parking lot problem and the triple modular redundancy– where the algorithms are applied. Also, we noted that our algorithm has been used to derive other non-trivial fault-tolerant programs, e.g., Byzantine agreement and token ring circulation.

For the high atomicity model, where a process could read and write all variables in one atomic step, the complexity of our algorithms was polynomial in the state space of the fault-intolerant program. For the low atomicity model, where restrictions are imposed on what processes can read and write, the complexity of our algorithm was exponential in the state space of the fault-intolerant program. We further showed that the complexity of adding masking fault-tolerance in the low atomicity model is NP-hard.

For the algorithms in the high atomicity model, we also showed that the invariant output by our algorithm is maximal and that the fault-tolerant program output by our algorithm provides maximal non-determinism inside the invariant. As argued in Section 3, this property is desirable when designing multitolerable programs.
where one adds fault-tolerance to multiple classes of faults in a stepwise fashion.

To deal with the exponential complexity of the transformation problem in the low atomicity model, elsewhere [10,11], we have considered different approaches to reduce the complexity of the transformation problem. One way is to develop heuristics to obtain a deterministic polynomial implementation of Add_ft. In [10], Kulkarni, Arora and Chippada have identified a set of such heuristics. Using these heuristics, they have developed a tool that is used in Networked Embedded Software Technology (NEST) [8]. This tool provides a deterministic implementation of Add_ft, and it suffices for developing fault-tolerant solutions for several problems including Byzantine agreement and token rings. (The source code for the tool is available at http://www.cse.msu.edu/~sandep/software.) To our knowledge, the byzantine agreement solution in [10] is the first solution where byzantine faults are considered in automatic design of fault-tolerant programs.

Another way to reduce the complexity is to consider a weaker form of fault-tolerance. In [11], Kulkarni and Ebnenasir have considered the failsafe fault-tolerance. They have shown that the addition of failsafe fault-tolerance is also NP-hard. Furthermore, they have identified a class of programs and a class of specifications for which failsafe fault-tolerance can be added in polynomial time. They have also shown that programs and specifications for commonly encountered problems such as consensus, commit and agreement fall in this class. It follows that failsafe fault-tolerant programs for these problems can be designed in polynomial time. We are currently using these results to identify heuristics that strengthen the given specification (respectively, add determinism to the given fault-intolerant program) in such a way that the fault-tolerance can be added in polynomial time by using that modified specification (respectively, program).

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. Also, by reusing the fault-intolerant program, our algorithms have the potential for preserving efficiency of the fault-intolerant program.

Our low atomicity model is more general than the Read/Write model considered elsewhere (e.g., [15,23]) 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 shared memory interleaving model. More specifically, if a fault-tolerant program in the shared memory interleaving model exists although a fault-tolerant program in Read/Write model does not, our algorithm can still be used. Also, by considering general models where a process could read the state of multiple processes simultaneously, we expect that the state space of the problem at hand will be smaller as we do not need to consider intermediate states that occur in models such as Read/Write. The reduction in state space will be especially useful in the automation of fault-tolerance where the the complexity of algorithms is high. Moreover, tolerance refinement techniques (e.g., [4-8]) can be used to refine the generated program so that it uses even lower atomicity (e.g., Read/Write or message passing).

One of extensions of our work is to precompute a set of fault-tolerance components that often occur in fault-tolerant programs and use those components directly in the transformation algorithm. We have precomputed such components that deal with message loss and link/node failure. These components have been used in ad-hoc networks [27] and sensor networks [8]. We expect that if such components are precomputed then we can provide guarantees about their efficiency. Moreover, by using such efficient precomputed 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


Appendix A1: Concise Representation for Specifications

Following Alpern and Schneider [9], the safety specification identifies a set of bad prefixes that should not occur in program computations. We now show that for fusion closed and suffix closed specifications, it suffices to focus on prefixes of length 2. In other words, if we have a prefix \( \langle \alpha, s_0 \rangle \) that maintains \( \text{spec} \) then we can determine whether an extended prefix \( \langle \alpha, s_0, s_1 \rangle \) maintains \( \text{spec} \) by focusing on the transition \( (s_0, s_1) \), and ignoring \( \alpha \). Formally we state this in Lemma A.2 and provide its proof below. To prove Lemma A.2, we first need to prove Lemma A.1 (Both Lemmas A.1 and A.2 have been previously proved in [2,3]. For the reader’s convenience, we reiterate these proofs.)

**Lemma A.1**

If \( \langle \alpha, s_0 \rangle \) maintains \( \text{spec} \), and \( \langle s_0, \beta \rangle \) maintains \( \text{spec} \)

Then \( \langle \alpha, s_0, \beta \rangle \) maintains \( \text{spec} \).

**Proof.**

\[
\langle \alpha, s_0 \rangle \text{ maintains } \text{spec} \land \langle s_0, \beta \rangle \text{ maintains } \text{spec}\\
= \{ \text{by definition of maintains } \}\\
(\exists \delta : \langle \alpha, s_0, \delta \rangle \in \text{spec}) \land (\exists \delta' : \langle s_0, \beta, \delta' \rangle \in \text{spec})\\
\Rightarrow \{ \text{by fusion closure of spec } \}\\
(\exists \delta' : \langle \alpha, s_0, \beta, \delta' \rangle \in \text{spec})\\
= \{ \text{by definition of maintains } \}\\
\langle \alpha, s_0, \beta \rangle \text{ maintains } \text{spec}.
\]

**Lemma A.2.** Let \( \alpha \) be finite sequence of states, and let \( \text{spec} \) be a specification.

If \( \langle \alpha, s_0 \rangle \text{ maintains } \text{spec} \)

Then \( \langle \alpha, s_0, s_1 \rangle \text{ maintains } \text{spec} \) iff \( \langle s_0, s_1 \rangle \text{ maintains } \text{spec} \).

**Proof.** If part:

\[
\langle \alpha, s_0, s_1 \rangle \text{ maintains } \text{spec}\\
= \{ \text{by definition of maintains } \}\\
(\exists \beta : \langle \alpha, s_0, s_1, \beta \rangle \in \text{spec})\\
\Rightarrow \{ \text{by suffix closure of spec } \}\\
(\exists \beta : \langle s_0, s_1, \beta \rangle \in \text{spec})\\
= \{ \text{by definition of maintains } \}\\
\langle s_0, s_1 \rangle \text{ maintains } \text{spec}.
\]
Only if part:
\[
\{ s_0, s_1 \} \text{ maintains } \text{spec} \land \{ \alpha, s_0 \} \text{ maintains } \text{spec} \\
\Rightarrow \{ \text{ by Lemma A.1 } \} \\
\{ \alpha, s_0, s_1 \} \text{ maintains } \text{spec}
\]

From Lemma A.2, it follows that the safety specification can be concisely represented by the set of ‘bad transitions’. Hence, we have assumed that for a given spec, the corresponding bad transitions is 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 \(s_0, s_1 \in S_p\).

Appendix A2: Adding Variables During Synthesis

Our algorithm assumes that the state space of the fault-tolerant program is the same as that of the fault-intolerant program. This restriction can be weakened by introducing additional variables in the fault-intolerant program itself. Let \(s_0\) be any state in the original fault-intolerant program. Thus, in the revised fault-intolerant program, there exists a set of states corresponding to state \(s_0\): for each state in this set the values of the original variables are the same as that in \(s_0\).

With the addition of new variables, the invariant and transitions of the fault-intolerant program are modified as follows: a state \(s'_0\) in the new state space is included in the invariant iff state \(s_0\) in the original state space is included in the invariant and \(s_0\) and \(s'_0\) agree on all the original variables. Likewise, for states \(s'_0, s'_1\) in the new state space, \((s'_0, s'_1)\) is included in the transitions of the fault-intolerant program iff there exist states \(s_0, s_1\) in the original state space such that \((s_0, s_1)\) is a transition of the fault-intolerant program, \(s_0\) and \(s'_0\) agree on all original variables, and \(s_1\) and \(s'_1\) agree on all original variables. Regarding faults, the modification depends upon assumptions about how they affect the new variables, e.g., a fault such as crash will not change the new variables whereas a fault such as transient will change the new variables arbitrarily. With the revised fault-intolerant program, its (new) invariant, (original) specification and (new) faults, we may now solve the transformation problem.

In fact, we can systematically add these new variables. For example, if we do not find a fault-tolerant program within the given state space, we can introduce one new variable (say, boolean) for each process in the program, and attempt to find a solution in the new state space. If no solution is found, we can continue the iteration. Note that this algorithm may not terminate. However, if a fault-tolerant program exists, it will terminate. Moreover, in the case of the high atomicity model (respectively, low atomicity model), the complexity of deriving that fault-tolerant program will be polynomial (respectively, exponential) in the state space of that fault-tolerant program.