Automated Synthesis of Multitolerance [1]

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

Abstract

We concentrate on automated synthesis of multitolerant programs, i.e., programs that tolerate multiple classes of faults and provide a (possibly) different level of fault-tolerance to each class. We consider three levels of fault-tolerance: (1) failsafe*, where in the presence of faults, the synthesized program guarantees safety, (2)* nonmasking*, where in the presence of faults, the synthesized program recovers to states from where its safety and liveness are satisfied, and (3)* masking *where in the presence of faults the synthesized program satisfies safety and recovers to states from where safety and liveness are satisfied.*

We focus on the automated synthesis of multitolerant programs in high atomicity where the program can read and write all its variables in an atomic step. We show that if one needs to add failsafe (respectively, nonmasking) fault-tolerance to one class of faults and masking fault-tolerance to another class of faults then such addition can be done in polynomial time in the state space of the fault-intolerant program. However, if one needs to add failsafe fault-tolerance to one class of faults and nonmasking fault-tolerance to another class of faults then the resulting problem is NP-complete. We find this result to be counterintuitive since adding failsafe and nonmasking fault-tolerance to the same *class of faults (which is equivalent to adding masking fault-tolerance to that class of faults) can be done in polynomial time, whereas adding failsafe fault-tolerance to one class of faults and nonmasking fault-tolerance to a* different *class of faults is NP-complete.*

Keywords: Fault-tolerance, Automatic addition of fault-tolerance, Formal methods, Program synthesis, Distributed programs

1 Introduction

Today's systems are often subject to multiple classes of faults and, hence, these systems need to provide appropriate level of fault-tolerance to each fault-class. Often it is undesirable or impractical to provide the same level of fault-tolerance to each class of faults. Hence, these systems need to tolerate multiple classes of faults, and (possibly) provide a different level of fault-tolerance to each class. To characterize such systems, the notion of multitolerance was introduced in [1]. The importance of such multitolerant systems can be easily observed from the fact that several methods for designing multitolerant programs as well as several instances of multitolerant programs can be readily found (e.g., [1–4]) in the literature.

In this paper, we focus on automated synthesis of multitolerant programs. Such automated synthesis has the advantage of generating fault-tolerant programs that (i) are correct by construction, and (ii) tolerate multiple classes of faults. Since the synthesized programs are correct by construction, there is no need for their proof of correctness.

One of the problems in automated synthesis of multitolerant programs is the complexity of such synthesis. Specifically, there exist situations where satisfying a specific fault-tolerance requirement for one class of faults conflicts with providing a different level of fault-tolerance to another fault-class. Hence, it is necessary to identify situations where synthesis of multitolerant programs can be performed efficiently and where heuristics need to be developed for adding multitolerance.

In our algorithms, we begin with a fault-intolerant program, i.e., a program that ensures that its specification is satisfied in the absence of faults although no guarantees are provided in the presence of faults. Subsequently, we add fault-tolerance to the given classes of faults while providing the required level of fault-tolerance to each of those classes. We consider three levels of fault-tolerance requirements, failsafe, nonmasking, and masking. Intuitively, in the presence of faults, a failsafe fault-tolerant program ensures that the safety is satisfied. In the presence of faults, a nonmasking fault-tolerant program recovers to states from where its safety and liveness specification is satisfied. And, a masking program satisfies both these properties (cf. Section 2 for precise definitions.)

Our algorithms are based on the algorithms in [5] where Kulkarni and Arora have presented algorithms for adding a *single* level of fault-tolerance to one class of faults. Specifically, in [5], the authors present sound and complete algorithms for adding failsafe, nonmasking, or masking fault-tolerance in the high atomicity model where a process can read and write all program variables in an atomic step. The complexity of these algorithms is polynomial in the state

[1]Email: sandeep@cse.msu.edu, ebnenasi@cse.msu.edu
Web: `http://www.cse.msu.edu/~{sandeep,ebnenasi}`
Tel: +1-517-355-2387, Fax: +1-517-432-1061
This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF grant EIA-0130724, and a grant from Michigan State University.

space of the fault-intolerant program.

Contributions of the paper. We focus on automated synthesis of high atomicity multitolerant programs in a stepwise fashion. The main results of the paper are as follows:

1. We present a sound and complete stepwise algorithm for the case where we add nonmasking fault-tolerance to one class of faults and masking fault-tolerance to another class of faults. The complexity of this algorithm is polynomial in the state space of the fault-intolerant program.
2. We present a sound and complete stepwise algorithm for the case where we add failsafe fault-tolerance to one class of faults and masking fault-tolerance to another class of faults. The complexity of this algorithm is also polynomial in the state space of the fault-intolerant program.
3. For the case where failsafe fault-tolerance is added to one fault-class and nonmasking fault-tolerance is added to another fault-class, we find a somewhat surprising result. We find that this problem is NP-complete. This result is surprising in that automating the addition of failsafe and nonmasking fault-tolerance to the *same* class of faults can be performed in polynomial time. However, addition of failsafe fault-tolerance to one class of faults and nonmasking fault-tolerance to a *different* class of faults is NP-complete.

Organization of the paper. The rest of the paper is organized as follows: In Section 2, we present preliminary concepts where we recall the definitions of programs, specifications, faults and fault-tolerance. Then, in Section 3, we present the formal definition of multitolerant programs and the problem of synthesizing a multitolerant program from a fault-intolerant program. Subsequently, in Section 4, we recall the relevant properties of algorithms in [5] that we use in automated addition of multitolerance. In Section 5, we present a sound and complete algorithm for the synthesis of multitolerant programs that provide nonmasking-masking multitolerance. Then, in Section 6, we present a sound and complete algorithm for the synthesis of multitolerant programs that provide failsafe-masking multitolerance. In Section 7, we present the NP-completeness proof for the case where failsafe-nonmasking multitolerance is added to fault-intolerant programs. Finally, in Section 8, we make concluding remarks and discuss future work.

2 Preliminaries

In this section, we give formal definitions of programs, problem specifications, faults, and fault-tolerance. The programs are specified in terms of their state space and their transitions. The definition of specifications is adapted from Alpern and Schneider [6]. The definition of faults and fault-tolerance is adapted from Arora and Gouda [7] and Kulkarni [8].

2.1 Program

A program p is specified by a finite state space, S_p, and a set of transitions, δ_p, where δ_p is a subset of $\{(s_0, s_1) \mid s_0, s_1 \in S_p\}$. A state predicate of p is any subset of S_p. A state predicate S is closed in the program p (respectively, δ_p) iff $(\forall s_0, s_1 : (s_0, s_1) \in \delta_p : (s_0 \in S \Rightarrow s_1 \in S))$. A sequence of states, $\langle s_0, s_1, ...\rangle$, is a computation of p iff the following two conditions are satisfied: (1) $\forall j : j > 0 : (s_{j-1}, s_j) \in \delta_p$, and (2) if $\langle s_0, s_1, ...\rangle$ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in \delta_p$. A sequence of states, $\langle s_0, s_1, ...\rangle$, is a computation prefix of p iff $\forall j : j > 0 : (s_{j-1}, s_j) \in \delta_p$, i.e., a computation prefix need not be maximal.

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

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

2.2 Specification

A specification is a set of infinite sequences of states that is suffix closed and fusion closed. Suffix closure of the set means that if a state sequence σ is in that set then so are all the suffixes of σ. Fusion closure of the set means that if state sequences $\langle \alpha, s, \gamma\rangle$ and $\langle \beta, s, \delta\rangle$ are in that set then so are the state sequences $\langle \alpha, s, \delta\rangle$ and $\langle \beta, s, \gamma\rangle$, where α and β are finite prefixes of state sequences, γ and δ are suffixes of state sequences, and s is a program state.

Following Alpern and Schneider [6], we let the specification consist of a safety specification and a liveness specification. For a suffix closed and fusion closed specification, the safety specification can be specified as a set of bad transitions [8], that is, for program p, its safety specification is a subset of $\{(s_0, s_1) : s_0, s_1 \in S_p\}$. Hence, we say a transition (s_0, s_1) violates the safety of specification iff (s_0, s_1) belongs to the set of bad transitions. The liveness specification is not required in our algorithm; the liveness specification satisfied by the fault-intolerant program is preserved in the synthesized multitolerant program.

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

For a finite sequence (of states) α, we say that $\alpha = \langle s_0, s_1, \cdots, s_j\rangle$ maintains *spec* iff $\forall (s_i, s_{i+1}) : 0 \leq i \leq j-1 : (s_i, s_{i+1})$ does not violate *spec*. We say that p maintains (does not violate) *spec* from S iff (1) S is closed in p, and (2) every computation prefix of p that starts in a state in S maintains *spec*.

Notation. Whenever the specification is clear from the context, we will omit it; thus, S is an invariant of p abbreviates S is an invariant of p for *spec*.

2.3 Faults

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

We use $p[]f$ to denote the transitions obtained by taking the union of the transitions in p and the transitions in f. We say that a state predicate T is an f-span (read as fault-span) of p from S iff the following two conditions are satisfied: (1) $S \Rightarrow T$, and (2) T is closed in $p[]f$. Observe that for all computations of p that start at states where S is true, T is a boundary in the state space of p up to which (but not beyond which) the state of p may be perturbed by the occurrence of the transitions in f.

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

2.4 Fault-Tolerance

We now define what it means for a program to be failsafe/nonmasking/masking fault-tolerant. We say that p is failsafe f-tolerant (read as fault-tolerant) from S for $spec$ iff the following conditions hold: (1) p satisfies $spec$ from S, and (2) there exists T such that T is an f-span of p from S, and $p[]f$ maintains $spec$ from T.

Since a nonmasking fault-tolerant program need not satisfy safety in the presence of faults, p is nonmasking f-tolerant from S for $spec$ iff the following conditions hold: (1) p satisfies $spec$ from S, and (2) there exists T such that T is an f-span of p from S, and every computation of $p[]f$ that starts from a state in T contains a state of S.

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

Notation. 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". Also, whenever the specification $spec$ and the invariant S are clear from the context, we omit them; thus, "f-tolerant" abbreviates "f-tolerant from S for $spec$".

3 Problem Statement

In this section, we formally define the problem of synthesizing multitolerant programs from their fault-intolerant versions. Before defining the synthesis problem, we present our definition of multitolerance; i.e., we identify what it means for a program to be multitolerant in the presence of multiple classes of faults.

As mentioned in Section 2.4, a failsafe/nonmasking/masking fault-tolerant program guarantees to provide a desired level of fault-tolerance (i.e., failsafe/nonmasking/masking) in the presence of a specific class of faults. Now, we consider the case where faults from multiple fault-classes, say $f1$ and $f2$, occur in a given program computation.

There exist several possible choices in deciding the level of fault-tolerance that should be provided in the presence of multiple fault-classes. One possibility is to provide no guarantees when $f1$ and $f2$ occur in the same computation. With such a definition of multitolerance, the program would provide fault-tolerance if faults from $f1$ occur or if faults form $f2$ occur. However, no guarantees will be provided if both faults occur simultaneously.

Another possibility is to require that the fault-tolerance provided for the case where $f1$ and $f2$ occur simultaneously should be equal to the minimum level of fault-tolerance provided when either $f1$ occurs or $f2$ occurs. For example, if masking fault-tolerance is provided to $f1$ and failsafe fault-tolerance is provided to $f2$ then failsafe fault-tolerance should be provided for the case where $f1$ and $f2$ occur simultaneously. In our definition, we follow the latter approach. The following table illustrates the minimum level of fault-tolerance provided for different combinations of levels of fault-tolerance provided to individual classes of faults.

Fault-Tolerance	Failsafe	Nonmasking	Masking
Failsafe	Failsafe	No-Tolerance	Failsafe
Nonmasking	No-Tolerance	Nonmasking	Nonmasking
Masking	Failsafe	Nonmasking	Masking

In a special case, consider the situation where failsafe fault-tolerance is provided to both $f1$ and $f2$. From the above description, failsafe fault-tolerance should be provided for the fault class $f1 \cup f2$. By taking the union of all the fault-classes for which failsafe fault-tolerance is provided, we get one fault-class, say $f_{failsafe}$, for which failsafe fault-tolerance needs to be added. Likewise, we obtain the fault-class $f_{nonmasking}$ (respectively, $f_{masking}$) for which nonmasking (respectively, masking) fault-tolerance is provided.

Now, given (the transitions of) a fault-intolerant program, p, its invariant, S, its specification, $spec$, and a set of distinct classes of faults $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$, we define what it means for a synthesized program p', with invariant S', to be multitolerant by considering how p' behaves when (i) no faults occur; (ii) only one class of faults happens, and (iii) multiple classes of faults happen.

Definition. Program p' is *multitolerant* to $f_{failsafe}, f_{nonmasking}$, and $f_{masking}$ from S' for $spec$ iff (if and only if) the following conditions hold:

1. p' satisfies $spec$ from S' in the absence of faults.
2. p' is masking $f_{masking}$-tolerant from S' for $spec$.
3. p' is failsafe $(f_{failsafe} \cup f_{masking})$-tolerant from S' for $spec$.
4. p' is nonmasking $(f_{nonmasking} \cup f_{masking})$-tolerant from S' for $spec$. □

Remark. Since every program is failsafe/nonmasking/masking fault-tolerant to a class of faults whose set of transitions is empty, the above definition generalizes the cases where one of the classes of faults is not specified (e.g., $f_{masking} = \{\}$).

Now, using the definition of multitolerant programs, we identify the requirements of the problem of synthesizing a multitolerant program, p', from its fault-intolerant version, p. The problem statement is motivated by the goal of simply adding multitolerance and introducing no new behaviors in the absence of faults. This problem statement is the natural extension to the problem statement in [5] where fault-tolerance is added to a single class of faults.

Since we require p' to behave similar to p in the absence of faults, we stipulate the following conditions: First, we require S' to be a subset of S (i.e., $S' \subseteq S$). Otherwise, if there exists a state $s \in S'$ where $s \notin S$ then, *in the absence of faults*, p' can reach s and perform new computations that do not belong to p. Thus, p' will include new ways of satisfying $spec$ from s in the absence of faults. Second, we require $(p'|S') \subseteq (p|S')$. If $p'|S'$ includes a transition that does not belong to $p|S'$ then p' can include new ways for satisfying *spec in the absence of faults*. Thus, the problem of multitolerance synthesis is as follows:

The Synthesis Problem
Given p, S, $spec$, $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$
Identify p' and S' such that
$S' \subseteq S$,
$p'|S' \subseteq p|S'$, and
p' is multitolerant to $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$ from S' for $spec$. □

We state the corresponding decision problem as follows:

The Decision Problem
Given p, S, $spec$, $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$:
Does there exist a program p'*, with its invariant* S' *that satisfies the requirements of the synthesis problem*? □

4 Addition of Fault-Tolerance To One Fault-Class

In the synthesis of multitolerant programs, we reuse algorithms Add_Failsafe, Add_Nonmasking, and Add_Masking, presented by Kulkarni and Arora [5]. These algorithms respectively add failsafe/nonmasking/masking fault-tolerance to a *single* class of faults. Hence, we recall the relevant properties of these algorithms in this section. While we reiterate these algorithms in the Appendix A, we note that the description of the algorithms presented in this paper and their proofs depend *only* on the properties mentioned in this section and not on the actual implementation of the algorithms in [5].

The above-mentioned algorithms take a program p, its invariant S, its specification $spec$, a class of faults f, and synthesize an f-tolerant program p' (if any) with the invariant S'. The synthesized program p' and its invariant S' satisfy the following requirements: (i) $S' \subseteq S$; (ii) $p'|S' \subseteq p|S'$, and (iii) p' is failsafe (respectively, nonmasking or masking) f-tolerant from S' for $spec$.

The invariant S', calculated by Add_Failsafe (respectively, Add_Masking), has the property of being the largest such possible invariant for any failsafe (respectively, masking) program obtained by adding fault-tolerance to the given fault-intolerant program. In other words, if there exists a failsafe fault-tolerant program p'', with invariant S'' that satisfies the above requirements for adding fault-tolerance then $S'' \subseteq S'$. Also, if no sequence of fault transitions can violate the safety of specification from any state inside S then Add_Failsafe will not change the invariant of the fault-intolerant program. Hence, we make the following observations:

Observation 4.1. Let the input for Add_Failsafe be p, S, $spec$ and f. Let the output of Add_Failsafe be fault-tolerant program p' and invariant S'. If any program p'' with invariant S'' satisfies (i) $S'' \subseteq S$; (ii) $p''|S'' \subseteq p|S''$, and (iii) p'' is failsafe f-tolerant from S' for $spec$ then $S'' \subseteq S'$. □

Observation 4.2. Let the input for Add_Failsafe be p, S, $spec$ and f. Let the output of Add_Failsafe be fault-tolerant program p' and invariant S'. Unless there exists states in S from where a sequence of f transitions alone violates safety, $S' = S$. □

Likewise, the f-span of the masking f-tolerant program, say T', synthesized by the algorithm Add_Masking is the largest possible f-span. Thus, we make the following observation:

Observation 4.3. Let the input for Add_Masking be p, S, $spec$ and f. Let the output of Add_Masking be fault-tolerant program p', invariant S', and fault-span T'. If any program p'' with invariant S'' satisfies (i) $S'' \subseteq S$; (ii) $p''|S'' \subseteq p|S''$, (iii) p'' is masking f-tolerant from S' for $spec$, and (iv) T'' is the fault-span used for verifying the masking fault-tolerance of p'' then $S'' \subseteq S'$ and $T'' \subseteq T'$. □

The algorithm Add_Nonmasking only adds recovery transitions from states outside the invariant S to S. Thus, we make the following observations:

Observation 4.4. Add_Nonmasking does not add or remove any state of S. □

Observation 4.5. Add_Nonmasking does not add or remove any transition of $p|S$. □

Based on the Observations 4.1- 4.5, Kulkarni and Arora [5] show that the algorithms Add_Failsafe, Add_Nonmasking, and Add_Masking are sound and complete, i.e., the output of these algorithms satisfy the requirements for adding fault-tolerance to a *single* class of faults and these algorithms can find a fault-tolerant program if one exists.

Theorem 4.5. The algorithms Add_Failsafe, Add_Nonmasking, and Add_Masking are sound and complete [5]. □

5 Nonmasking-Masking Multitolerance

In this section, we present an algorithm for stepwise synthesis of multitolerant programs that are subject to two classes of faults $f_{nonmasking}$ and $f_{masking}$ for which respectively nonmasking and masking fault-tolerance is required. We also show that our synthesis algorithm is sound and complete.

Given a program p, with its invariant S, its specification $spec$, our goal is to synthesize a program p', with invariant S' that is multitolerant to $f_{nonmasking}$ and $f_{masking}$. By definition, p' must be masking $f_{masking}$-tolerant. In the presence of both $f_{nonmasking}$ and $f_{masking}$ (i.e., $f_{nonmasking} \cup f_{masking}$), p' must provide nonmasking fault-tolerance.

We proceed as follows: Using the algorithm Add_Masking, we synthesize a masking $f_{masking}$-tolerant program p_1, with invariant S', and fault-span $T_{masking}$. Now, since program p_1 is masking $f_{masking}$-tolerant, it provides safe recovery to its invariant, S', from every state in $(T_{masking} - S')$. Thus, in the presence of $f_{nonmasking} \cup f_{masking}$, if p_1 is perturbed to $(T_{masking} - S')$ then p_1 will satisfy the requirements of nonmasking fault-tolerance (i.e., recovery to S'). However, if $f_{nonmasking} \cup f_{masking}$ transitions perturb p_1 to states s, where $s \notin T_{masking}$, then recovery must be added from those states. Based on the Observations 4.4 and 4.5, it suffices to add recovery to $T_{masking}$ as provided recovery by p_1 from $T_{masking}$ to S' can be reused *even after* adding nonmasking fault-tolerance. Thus, the synthesis algorithm Add_Nonmasking_Masking is as shown in Figure 1.

Add_Nonmasking_Masking(p: transitions, $f_{nonmasking}, f_{masking}$: fault,
S: state predicate, $spec$: safety specification)
{
$p_1, S', T_{masking}$:= *Add_Masking*$(p, f_{masking}, S, spec)$;
if $(S' = \{\})$ `declare no multitolerant program` p' `exists;`
return $\emptyset, \emptyset$;
p', T' := *Add_Nonmasking*$(p_1, f_{nonmasking} \cup f_{masking}, T_{masking}, spec)$;
return p', S';
}

Figure 1. Synthesizing nonmasking-masking multitolerance.

Now, in Theorem 5.1, we show the soundness of Add_Nonmasking_Masking, i.e., we show that the output of Add_Nonmasking_Masking satisfies the requirements of the problem statement in Section 3. Subsequently, in Theorem 5.2, we show the completeness of Add_Nonmasking_Masking, i.e., we show that if a multitolerant program can be designed for the given fault-intolerant program then Add_Nonmasking_Masking will not declare failure.

Theorem 5.1. The algorithm Add_Nonmasking_Masking is sound.

Proof. Based on the soundness of Add_Masking (cf. Theorem 4.5), $S' \subseteq S$.

Also, using the soundness of Add_Masking, we have $p_1|S' \subseteq p|S'$. In addition, based on the Observation 4.5, we have $p_1|S' = p'|S'$. As a result, we have $p'|S' \subseteq p|S'$.

Now, we show that p' is multitolerant to $f_{nonmasking}$ and $f_{masking}$ from S' for $spec$:

1. **Absence of faults.** From the soundness of Add_Masking, it follows that p_1 satisfies $spec$ from S' in the absence of faults. Since Add_Nonmasking does not add to (respectively, remove from) any transitions of $p_1|S'$ (cf. Observation 4.5), it follows that p' satisfies $spec$ from S'.
2. **Masking $f_{masking}$-tolerance.** From the soundness of Add_Masking, p_1 is masking $f_{masking}$-tolerant from S' for $spec$. Also, based on the Observation 4.4 and 4.5, Add_Nonmasking preserves masking $f_{masking}$-tolerance property of p_1 since $p_1|T_{masking} = p'|T_{masking}$. Therefore, p' is masking $f_{masking}$-tolerant from S' for $spec$.
3. **Nonmasking $(f_{nonmasking} \cup f_{masking})$-tolerance.** From the soundness of Add_Nonmasking, we know that p' is nonmasking $(f_{nonmasking} \cup f_{masking})$-tolerant from $T_{masking}$ for $spec$. Also, based on the Observation 4.4 and 4.5, Add_Nonmasking preserves masking $f_{masking}$-tolerance property of p_1 since $p_1|T_{masking} = p'|T_{masking}$. Thus, recovery from $T_{masking}$ to S' is guaranteed in the presence of $f_{nonmasking} \cup f_{masking}$. Therefore, p' is nonmasking $(f_{nonmasking} \cup f_{masking})$-tolerant from S' for $spec$.

Based on the above discussion, it follows that p' is multitolerant to $f_{nonmasking}$ and $f_{masking}$ from S' for $spec$. Therefore, Add_Nonmasking_Masking is sound. □

Theorem 5.2. The algorithm Add_Nonmasking_Masking is complete.

Proof. Add_Nonmasking_Masking declares that a multitolerant program does not exist only when Add_Masking does not find a masking $f_{masking}$-tolerant program. Since the synthesized program must be masking $f_{masking}$-tolerant, from the completeness of Add_Masking, completeness of Add_Nonmasking_Masking follows. □

6 Failsafe-Masking Multitolerance

In this section, we investigate the stepwise synthesis of programs that are multitolerant to two classes of faults $f_{failsafe}$ and $f_{masking}$ for which we respectively require failsafe and masking fault-tolerance. We present a sound and complete algorithm for synthesizing failsafe-masking multitolerant programs.

Let p be the input fault-intolerant program with its invariant S, its specification $spec$, and p' be the synthesized multitolerant program with its invariant S'. Since the multitolerant program p' must maintain safety of $spec$ from every reachable state in the computations of $p'[](f_{failsafe} \cup f_{masking})$ and $p'[]f_{masking}$, p' must not reach a state from where safety is violated by a sequence of $f_{failsafe} \cup f_{masking}$ transitions. Hence, we calculate a set of states, say ms (cf. Figure 2), from where safety of $spec$ is violated by a sequence of transitions of $f_{failsafe} \cup f_{masking}$. Also, p' must not execute transitions that take p' to a state in ms. Hence, we define mt to include these transitions as well as the transitions that violate safety of $spec$.

Now, since p' should be masking $f_{masking}$-tolerant, we use the algorithm Add_Masking to synthesize a program p_1 given the input parameters $p - mt$, $f_{masking}$, $S - ms$, and mt. We only consider faults $f_{masking}$ because p_1 need not be masking fault-tolerant to $f_{failsafe}$. Since a multitolerant program must not reach a state of ms, we use the state predicate $S - ms$ as the input invariant to Add_Masking. Finally, we use mt transitions in place of the $spec$ parameter (i.e., the fourth parameter of Add_Masking). Since Add_Masking treats mt as a set of safety-violating transitions, it does not include them in the synthesized program p_1. Thus, starting from a state in S', a computation of $p_1[]f_{masking}$ does not reach a state in ms. As a result, if $T_{masking}$ contains a state s in ms, s can be removed while preserving the masking $f_{masking}$-tolerance property of p_1. Hence, we make the following observation:

Observation 6.1. In the output of the algorithm Add_Masking (cf. Figure 2), removing ms states from $T_{masking}$ preserves masking $f_{masking}$-tolerance property of p_1. $\Box$

Now, if faults $f_{failsafe} \cup f_{masking}$ perturb p_1 to a state s, where $s \notin T_{masking}$ then our synthesis algorithm will have to ensure that safety is maintained. To achieve this goal, we add failsafe $(f_{failsafe} \cup f_{masking})$-tolerance to p_1 from $(T_{masking} - ms)$ using the algorithm Add_Failsafe.

Add_Failsafe_Masking(p: transitions, $f_{failsafe}, f_{masking}$: fault,
S: state predicate, $spec$: safety specification)
{
$ms := \{s_0 : \exists s_1, s_2, ... s_n :$
$(\forall j : 0 \leq j < n : (s_j, s_{(j+1)}) \in (f_{failsafe} \cup f_{masking})) \wedge$
$(s_{(n-1)}, s_n)$ violates $spec$ };
$mt := \{(s_0, s_1) : ((s_1 \in ms) \vee (s_0, s_1)$ violates $spec$) };
$p_1, S', T_{masking} := Add\_Masking(p - mt, f_{masking}, S - ms, mt)$;
if $(S' = \{\})$ `declare no multitolerant program` p' `exists;`
return $\emptyset, \emptyset$;
$p', T' := Add\_Failsafe(p_1, f_{failsafe} \cup f_{masking}, T_{masking} - ms, mt)$;
return p', S';

Figure 2. Synthesizing failsafe-masking multitolerance.

The algorithm Add_Failsafe takes the program p_1, faults $f_{failsafe} \cup f_{masking}$, the state predicate $(T_{masking} - ms)$, and the set of mt transitions as the set of transitions that the multitolerant program is not allowed to execute. Since the input invariant to Add_Failsafe (i.e., $(T_{masking} - ms)$) has no ms state, based on the Observation 4.2, the algorithm Add_Failsafe does not remove any state of $(T_{masking} - ms)$. Also, Add_Failsafe does not remove any transition of $p_1|(T_{masking} - ms)$. Thus, we have $(p'|(T_{masking} - ms)) = (p_1|(T_{masking} - ms))$ and $p'|S' = p_1|S'$.

Theorem 6.1. The algorithm Add_Failsafe_Masking is sound and complete. $\Box$

Since the proof of Theorem 6.1 is similar to the proofs of Theorems 5.1 and 5.2, we relegate the proof to the Appendix B.

7 Failsafe-Nonmasking-Masking Multitolerance

In this section, we show that, in general, the problem of synthesizing multitolerant programs from their fault-intolerant version is NP-complete. Towards this end, in Section 7.1, we show that the problem of synthesizing multitolerant programs from their fault-intolerant version is in NP by designing a non-deterministic polynomial algorithm. Afterwards, in Section 7.2, we present a mapping between a given instance of the 3-SAT problem and an instance of the (decision) problem of synthesizing multitolerance. Then, in Section 7.3, we show that the given 3-SAT instance is satisfiable iff the answer to the decision problem is affirmative; i.e., there exists a multitolerant program synthesized from the instance of the decision problem of multitolerance synthesis.

7.1 Non-Deterministic Synthesis Algorithm

In this section, we first identify the difficulties of adding multitolerance to three distinct classes of faults $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$. Then, we present a non-deterministic solution for adding multitolerance to fault-intolerant programs.

For a program p that is subject to three classes of faults $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$, consider the cases where there exists a state s such that (i) s is reachable in the computations of $p[](f_{failsafe} \cup f_{masking})$ from invariant, (ii) s is reachable in the computations of $p[](f_{nonmasking} \cup f_{masking})$ from invariant, and (iii) no safe recovery is possible from s to the invariant.

In such cases, we have the following options: (i) ensure that s is unreachable in the computations of $p[](f_{failsafe} \cup f_{masking})$ and add a recovery transition (that violates safety) from s to the invariant, or (ii) ensure that s is unreachable in the computations of $p[](f_{nonmasking} \cup f_{masking})$ and leave s as a deadlock state. Moreover, the choice made for this state affects other similar states. Hence, one needs to explore all possible choices for each such state s, and as a result, brute-force exploration of these options requires exponential time in the state space.

Now, given a program p, with its invariant S, its specification $spec$, and three classes of faults $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$, we present the non-deterministic algorithm Add_Multitolerance. In our non-deterministic algorithm, first, we guess a program p', its invariant S', and three fault-spans $T_{failsafe}$, $T_{nonmasking}$, and $T_{masking}$. Then, we verify a set of conditions that ensure the multitolerance property of p'. We have shown our algorithm in Figure 3.

Theorem 7.1 The algorithm Add_Multitolerance is sound and complete. $\Box$

Theorem 7.2 The problem of synthesizing multitolerant programs from their fault-intolerant versions is in NP. $\Box$

Since the Add_Multitolerance algorithm simply verifies the conditions needed for multitolerance, the proof is straightforward, and hence, omitted.

7.2 Mapping 3-SAT To Multitolerance

In this section, we give an algorithm for polynomial-time mapping of any given instance of the 3-SAT problem into an instance of the decision problem defined in Section 3. The instance of the decision problem of synthesizing multitolerance consists of the fault-intolerant program, p, its invariant, S, its specification, and three classes of faults $f_{failsafe}, f_{nonmasking}$, and $f_{masking}$ that perturb p. The problem statement for the 3-SAT problem is as follows:

3-SAT problem.

Given is a set of literals, $a_1, a_2, ..., a_n$ and $a'_1, a'_2, ..., a'_n$, where a_i and a'_i are complements of each other, and a Boolean formula $c = c_1 \wedge c_2 \wedge ... \wedge c_M$, where each c_j is a disjunction of exactly three literals.

Does there exist an assignment of truth values to $a_1, a_2, ..., a_n$ such that c is satisfiable?

Add_Multitolerance (p: transitions, $f_{failsafe}, f_{nonmasking}, f_{masking}$: fault, S: state predicate, $spec$: safety specification)

$$
\begin{array}{ll}
\{ & \\
\quad ms := \{s_0 : \exists s_1, s_2, ...s_n : (\forall j : 0 \le j < n : (s_j, s_{(j+1)}) \in (f_{failsafe} \cup f_{masking})) \wedge & \\
\qquad\qquad (s_{(n-1)}, s_n) \text{ violates } spec \}; & (1) \\
\quad mt := \{(s_0, s_1) : ((s_1 \in ms) \vee (s_0, s_1) \text{ violates } spec) \}; & (2) \\
& \\
\quad \text{Guess } p', S', T_{failsafe}, T_{nonmasking}, T_{masking}; & (3) \\
\quad \text{Verify the following conditions:} & \\
\qquad S' \subseteq S;\ S' \neq \{\};\ S' \subseteq T_{failsafe};\ S' \subseteq T_{nonmasking};\ S' \subseteq T_{masking}; & (4) \\
\qquad (\forall s_0 : s_0 \in S' : (\exists s_1 :: (s_0, s_1) \in p')); & (5) \\
\qquad p'|S' \subseteq p|S';\ S' \text{ is closed in } p'; & (6) \\
& \\
\qquad T_{masking} \text{ is closed in } p'[]f_{masking}; & (7) \\
\qquad T_{masking} \cap ms = \emptyset;\ (p'|T_{masking}) \cap mt = \emptyset; & (8) \\
\qquad (\forall s_0 : s_0 \in T_{masking} : (\exists s_1 :: (s_0, s_1) \in p'));\ (p'|(T_{masking} - S')) \text{ is acyclic}; & (9) \\
& \\
\qquad T_{failsafe} \text{ is closed in } p'[](f_{failsafe} \cup f_{masking}); & (10) \\
\qquad T_{failsafe} \cap ms = \emptyset;\ (p'|T_{failsafe}) \cap mt = \emptyset; & (11) \\
& \\
\qquad T_{nonmasking} \text{ is closed in } p'[](f_{nonmasking} \cup f_{masking}); & (12) \\
\qquad (\forall s_0 : s_0 \in T_{nonmasking} : (\exists s_1 :: (s_0, s_1) \in p'));\ (p'|(T_{nonmasking} - S')) \text{ is acyclic}; & (13) \\
\} &
\end{array}
$$

Figure 3. A non-deterministic polynomial algorithm for synthesizing multitolerance.

Next, we identify each entity of the instance of the problem of multitolerance synthesis, based on the given instance of the 3-SAT formula.

The state space and the invariant of the fault-intolerant program, p**.** The invariant, S, of the fault-intolerant program, p, includes only one state, say s. Based on the literals and disjunctions of the given 3-SAT instance, we include additional states outside the invariant. Specifically, for each literal a_i and its complement, we introduce the following states (cf. Figure 4):

- x_i, x'_i, y_i, v_i

And, for each disjunction $c_j = (a_i \vee a'_k \vee a_r)$ ($1 \le i \le n$, $1 \le k \le n$, and $1 \le r \le n$), we introduce a state z_j outside the invariant ($1 \le j \le M$).

The transitions of the fault-intolerant program. The only transition in the fault-intolerant program is a self-loop (s, s).

The transitions of $f_{failsafe}$**.** The transitions of $f_{failsafe}$ can perturb the program from x_i to v_i. Thus, the class of faults $f_{failsafe}$ is equal to the set of transitions $\{(x_i, v_i) : 1 \le i \le n\}$.

The transitions of $f_{nonmasking}$**.** The transitions of $f_{nonmasking}$ can perturb the program from x'_i to v_i. Thus, we have $f_{nonmasking} = \{(x'_i, v_i) : 1 \le i \le n\}$.

The transitions of $f_{masking}$**.** The transitions of $f_{masking}$ can take the program from s to y_i. Also, for each disjunction c_j, we introduce a fault transition that perturbs the program from state s to state z_j ($1 \le j \le M$). Thus, the class of faults $f_{masking}$ is equal to the set of transitions $\{(s, y_i) : 1 \le i \le n\} \cup \{(s, z_j) : 1 \le j \le M\}$.

The safety specification of the fault-intolerant program, p**.** None of the fault transitions, namely $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$ identified above violate safety. In addition, for each literal a_i and its complement a'_i ($1 \le i \le n$), the following transitions do not violate safety (cf. Figure 4):

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

And, for each disjunction $c_j = a_i \vee a'_k \vee a_r$, the following transitions do not violate safety:

- $(z_j, x_i), (z_j, x'_k), (z_j, x_r)$

All transitions except those identified above violate safety of specification. Also, observe that the transition (v_i, s), shown in Figure 4, violates safety.

7.3 Reduction From 3-SAT

In this section, we show that the given instance of 3-SAT is satisfiable iff multitolerance can be added to the problem instance identified in Section 7.2. Specifically, in Lemma 7.3, we show that if the given instance of the 3-SAT formula is satisfiable then there exists a multitolerant program that solves the instance of the multitolerance synthesis problem identified in Section 7.2. Then, in Lemma 7.4, we show that if there exists a multitolerant program that solves the instance of the multitolerance synthesis problem, identified in Section 7.2, then the given 3-SAT formula is satisfiable.

Lemma 7.3 If the given 3-SAT formula is satisfiable then there exists a multitolerant program that solves the instance of the addition problem identified in Section 7.2.

Proof. Since the 3-SAT formula is satisfiable, there exists an assignment of truth values to the literals a_i, $1 \le i \le n$, such that each c_j, $1 \le j \le M$, is *true*. Now, we identify a multitolerant program, p', that is obtained by adding multitolerance to the fault-intolerant program p identified in Section 7.2.

The invariant of p' is the same as the invariant of p (i.e., $\{s\}$). We derive the transitions of the multitolerant program p' as

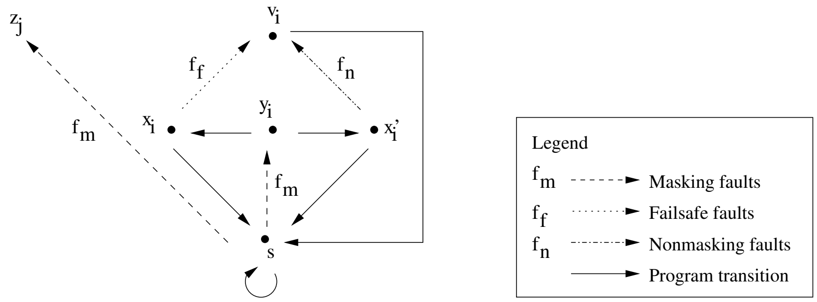

Figure 4. The states and the transitions corresponding to the literals in the 3-SAT formula.

follows. (As an illustration, we have shown the partial structure of p' where $a_i = true$, $a_k = false$, and $a_r = true$ $(1 \leq i, k, r \leq n)$ in Figure 5.)

- For each literal a_i, $1 \leq i \leq n$, if a_i is *true* then we will include the transitions (y_i, x_i) and (x_i, s). Thus, in the presence of $f_{masking}$ alone, p' provides safe recovery to s through x_i.
- For each literal a_i, $1 \leq i \leq n$, if a_i is *false* then we will include (y_i, x_i') and (x_i', s) to provide safe recovery to the invariant. In this case, since state v_i can be reached from x_i' by faults $f_{nonmasking}$, we include transition (v_i, s) so that in the presence of $f_{masking}$ and $f_{nonmasking}$ program p' provides nonmasking fault-tolerance.
- For each disjunction c_j that includes a_i, we include the transition (z_j, x_i) iff a_i is *true*. And, for each disjunction c_j that includes a_i', we include transition (z_j, x_i') iff a_i is *false*.

Now, we show that p' is multitolerant in the presence of faults $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$.

- **p' in the absence of faults.** $p'|S = p|S$. Thus, p' satisfies $spec$ in the absence of faults.
- **Masking tolerance to $f_{masking}$.** If the faults from $f_{masking}$ occur then the program can be perturbed to (1) y_i, $1 \leq i \leq n$, or (2) z_j, $1 \leq j \leq M$.

  In the first case, if a_i is *true* then there exists exactly one sequence of transitions, $\langle (y_i, x_i), (x_i, s) \rangle$, in $p'[]f_{masking}$. Thus, any computation of $p'[]f_{masking}$ eventually reaches a state in the invariant. Moreover, starting from y_i the computations of $p'[]f_{masking}$ do not violate the safety specification. And, if a_i is *false* then there exists exactly one sequence of transitions, $\langle (y_i, x_i'), (x_i', s) \rangle$, in $p'[]f_{masking}$. By the same argument, even in this case, any computation of $p'[]f_{masking}$ reaches a state in the invariant and does not violate the safety specification during recovery.

  In the second case, since c_j evaluates to *true*, one of the term in c_j (a literal or its complement) evaluates to *true*. Thus, there exists at least one transition from z_j to some state x_k (respectively, x_k') where a_k (respectively, a_k') is a literal in c_j and a_k (respectively, a_k') evaluates to *true*. Moreover, the transition (z_j, x_k) is included in p' iff a_k evaluates to *true*. Thus, (z_j, x_k) is included in p' iff (x_k, s) is included in p'. Since from x_k (respectively, x_k'), there exists no other transition in $p'[]f_{masking}$ except (x_k, s), every computation of p' reaches the invariant without violating safety. Based, on the above discussion, p' is masking tolerant to $f_{masking}$.
- **Failsafe tolerance to $f_{masking} \cup f_{failsafe}$.** Clearly, based on the case considered above, if only faults from $f_{masking}$ occur then the program is also failsafe fault-tolerant. Hence, we consider only the case where at least one fault from $f_{failsafe}$ has occurred.

  Faults in $f_{failsafe}$ occur only in state x_i, $1 \leq i \leq n$. And, p' reaches x_i iff a_i is assigned $true$ in the satisfaction of the given 3-SAT formula. Moreover, if a_i is *true* then there is no transition from v_i. Thus, after a fault transition of class $f_{failsafe}$ occurs p' simply stops. Therefore, p' does not violate safety.
- **Nonmasking tolerance to $f_{masking} \cup f_{nonmasking}$.** This proof is similar to the proof of failsafe fault-tolerance shown above. Specifically, we only need to consider the case where at least one fault transition of class $f_{nonmasking}$ has occurred.

  Faults in $f_{nonmasking}$ occur only in state x_i', $1 \leq i \leq n$. And, p' reaches x_i' iff a_i is assigned *false* in the satisfaction of the given 3-SAT formula. Moreover, if a_i is *false* then the only transition from v_i is (v_i, s). Thus, in the presence of $f_{masking}$ and $f_{nonmasking}$, p' recovers to its invariant. (Note that the recovery in this case violates safety.) $\square$

Lemma 7.4 If there exists a multitolerant program that solves the instance of the synthesis problem identified earlier then the given 3-SAT formula is satisfiable.

Proof. Suppose that there exists a multitolerant program p' derived from the fault-intolerant program, p, identified in Section 3. Since the invariant of p', S', is non-empty and $S' \subseteq S$, S' must include state s. Thus, $S' = S$. Also, since each y_i, $1 \leq i \leq n$, is directly reachable from s by a fault from $f_{masking}$, p' must provide safe recovery from y_i to s. Thus, p' must include either (y_i, x_i) or (y_i, x_i'). We make the following truth assignment as follows: If p' includes (y_i, x_i) then we assign a_i to be *true*. And, if p' includes (y_i, x_i') then we assign a_i to be *false*. Clearly, each literal in the 3-SAT formula will get at least one truth assignment. Now, we show that the truth assignment to each literal is consistent and that each disjunct in the 3-SAT formula evaluates to *true*.

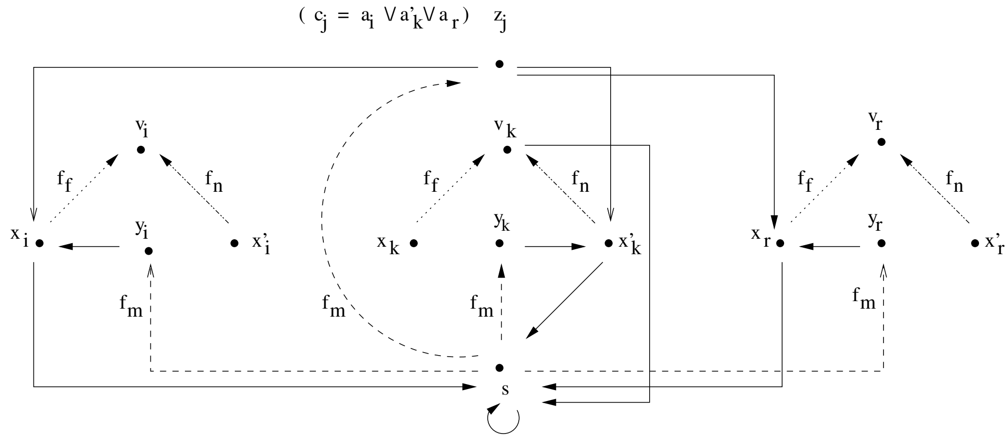

Figure 5. The partial structure of the multitolerant program

- *Each literal gets a unique truth assignment.* Suppose that there exists a literal a_i, which is assigned both *true* and *false*, i.e., both (y_i, x_i) and (y_i, x'_i) are included in p'. Now, v_i can be reached by the following transitions (s, y_i), (y_i, x'_i), and (x'_i, v_i). In this case, only faults from $f_{masking}$ and $f_{nonmasking}$ have occurred. Hence, p' must provide recovery from v_i to invariant. Also, v_i can be reached by the following transitions (s, y_i), (y_i, x_i), and (x_i, v_i). In this case, only faults from $f_{masking}$ and $f_{failsafe}$ have occurred. Hence, p' must ensure safety. Based on the above discussion, p' must provide a safe recovery to the invariant from v_i. Based on the definition of the safety specification identified in Section 7.2, this is not possible. Thus, literal a_i must be assigned only one truth value.
- *Each disjunction is true.* Let $c_j = a_i \vee a'_k \vee a_r$ be a disjunction in the given 3-SAT formula. The corresponding state added in the instance of the multitolerance problem is z_j. Note that state z_j can be reached by the occurrence of a fault from $f_{masking}$ from s. Hence, p' must provide safe recovery from z_j. Since the only safe transitions from z_j are those corresponding to states x_i, x'_k and x_r, p' must include at least one of the transitions (z_j, x_i), (z_j, x'_k), or (z_j, x_r).

  Now, we show that the transition included from z_j is consistent with the truth assignment of literals. Specifically, consider the case where p' contains transition (z_j, x_i) and a_i is assigned *false*, p' can reach x_i in the presence of faults from $f_{masking}$ alone. Moreover, if a_i is assigned *false* then p' contains the transition (y_i, x'_i). Thus, x'_i can also be reached by the occurrence of faults from $f_{masking}$ alone. Based on the above proof for unique assignment of truth values to literals, p' cannot reach x_i and x'_i in the presence of $f_{masking}$ alone. Hence, if (z_j, x_i) is included in p' then a_i must have been assigned truth value *true*. Likewise, if (z_j, x'_k) is included in p' then a_k must be assigned truth value *false*. Thus, with the truth assignment considered above, each disjunction must evaluate to *true*. □

Theorem 7.5 The problem of synthesizing multitolerant programs from their fault-intolerant versions is NP-complete. □

7.4 Failsafe-Nonmasking Multitolerance

In this section, we extend the NP-completeness proof of synthesizing multitolerance for the case where we add failsafe fault-tolerance to one class of faults, say $f_{failsafe}$, and we add nonmasking fault-tolerance to another class of faults, say $f_{nonmasking}$.

Our mapping for this case is similar to that in Section 7.2. We replace the $f_{masking}$ fault transition (s, y_i) with a sequence of transitions of $f_{failsafe}$ and $f_{nonmasking}$ as shown in Figure 6. Likewise, we replace fault transition (s, z_j) with a structure similar to Figure 6. Thus, y_i (respectively, z_i) is reachable by $f_{failsafe}$ faults alone and by $f_{nonmasking}$ faults alone. As a result, v_i is reachable in the computations of $p'[]f_{failsafe}$ and in the computations of $p'[]f_{nonmasking}$. Thus, to add multitolerance, safe recovery must be added from v_i to s (cf. Figure 4). Now, we note that with this mapping, the proofs of Lemmas 7.3 and 7.4 and Theorem 7.5 can be easily extended to show that synthesizing failsafe-nonmasking multitolerance is NP-complete. Thus, we have

Corollary 7.6. The problem of synthesizing failsafe-nonmasking multitolerant programs from their fault-intolerant version is NP-complete. □

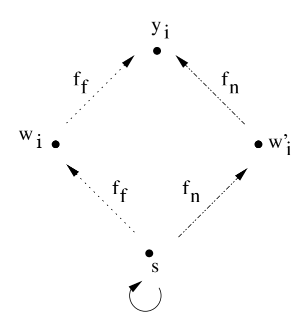

Figure 6. A proof sketch for NP-completeness of synthesizing failsafe-nonmasking multitolerance.

8 Conclusion and Future Work

In this paper, we investigated the problem of synthesizing multitolerant programs from their fault-intolerant versions. The input to the synthesis algorithm included the fault-intolerant program, different classes of faults to which fault-tolerance had to be added, and the level of tolerance provided for each class of faults. Our algorithms ensured that the synthesized program provided the specified level of fault-tolerance if a fault from any single class had occurred. Moreover, it ensured that if faults from multiple classes occurred then the program would provide the minimal level of fault-tolerance provided to each of those classes.

We considered three levels of fault-tolerance, *failsafe*, *nonmasking* and *masking*. We presented a sound and complete algorithm for the case where failsafe (respectively, nonmasking) fault-tolerance would be added to one class of faults and masking fault-tolerance would be provided to another class of faults. Thus, in these cases, if a multitolerant program could be synthesized for the given input program, our algorithms always would produce one such fault-tolerant algorithm. The complexity of these algorithms is polynomial in the state space of the fault-intolerant program.

For the case where one needs to add failsafe fault-tolerance to one class of faults and nonmasking fault-tolerance to another class of faults, we found a surprising result. Specifically, we showed that this problem is NP-complete. As mentioned earlier, this result was counterintuitive as adding failsafe and nonmasking fault-tolerance to the *same* class of faults can be done in polynomial time. However, adding failsafe fault-tolerance to one class of faults and nonmasking fault-tolerance to another class of faults is NP-complete.

Our synthesis approach is different from specification-based approaches [9–12] where one synthesizes a fault-tolerant program from its temporal logic specification. Hence, our approach is desirable when one needs extend an existing system by adding fault-tolerance. Also, the synthesis algorithms of [5, 13, 14] add fault-tolerance to only one class of faults whereas we address the synthesis of programs that simultaneously tolerate multiple classes of faults. To our knowledge, ours is the first algorithm for automated design of multitolerant programs.

Although the results focused in this paper deal with the high atomicity model, we note that the algorithms in high atomicity model are important in synthesizing distributed fault-tolerant programs as well. Specifically, our algorithms identify a limit upto which even *highly powerful* processes can add the necessary multitolerance. Thus, the output of these algorithms can be used in identifying the limits that distributed processes — along with their limitation on reading and writing variables of the program — can achieve in terms of adding the necessary multitolerance. As an illustration, we note that in [14], we have identified how algorithms in high atomicity can be systematically used in adding fault-tolerance to a single class of faults.

As an extension to our work we plan to explore the polynomial boundary of synthesizing multitolerant programs by identifying necessary and sufficient conditions for polynomial synthesis of multitolerant programs. Some of the sufficient conditions identified in this paper include the cases where (i) only failsafe and masking fault-tolerance is added, and (ii) only nonmasking and masking fault-tolerance is added. Also, we intend to identify heuristics by which we can synthesize multitolerant programs in polynomial time. Another extension to our work is to use these heuristics and algorithms in synthesizing multitolerant distributed programs.

References

[1] Sandeep S. Kulkarni A. Arora. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 1998.

[2] V. Hadzilacos E. Anagnostou. Tolerating transient and permanent failures. *Proceedings of the 7th International Workshop on Distributed Algorithms. Les Diablerets, Switzerland*, 1993.

[3] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Proceedings of the Second Workshop on Self-Stabilizing Systems*, 1995.

[4] S. Tsang and E. Magill. Detecting feature interactions in the intelligent network. *Feature Interactions in Telecommunications Systems II, IOS Press*, 1994.

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

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

[7] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

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

[9] E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

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

[11] P. Attie and A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM TOPLAS (a preliminary version of this paper appeared in PODC96)*, 23(2), March 2001.

[12] O. Kupferman and M.Y. Vardi. Synthesizing distributed systems. In *Proc. 16th IEEE Symp. on Logic in Computer Science*, July 2001.

[13] S. S. Kulkarni and A. Ebnenasir. The complexity of adding failsafe fault-tolerance. *International Conference on Distributed Computing Systems*, 2002.

[14] S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. *International Conference on Distributed Computing Systems*, 2003.