Feasibility of Stepwise Addition of Multitolerance to High Atomicity Programs

Ali Ebnenasir and Sandeep S. Kulkarni

Michigan Technological University
Computer Science Technical Report
CS-TR-08-03
October 2008

Feasibility of Stepwise Addition of Multitolerance to High Atomicity Programs

Ali Ebnenasir

Department of Computer Science
Michigan Technological University
Houghton MI 49931 USA
Email: aebnenas@mtu.edu

Sandeep S. Kulkarni
Department of
Computer Science and Engineering
Michigan State University
East Lansing MI 48824 USA
Email: sandeep@cse.msu.edu

October 2008

Abstract

We present sound and (deterministically) complete algorithms for stepwise design of two families of multitolerant programs in a high atomicity program model, where a program can read and write all its variables in an atomic step. We illustrate 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 an addition can be done in separate polynomial-time (in the state space of the fault-intolerant program) steps *regardless of the order of addition*. This result has a significant methodological implication in that designers need not be concerned about unknown fault tolerance requirements that may arise due to unanticipated types of faults. Further, we illustrate that if one needs to add failsafe fault-tolerance to one class of faults and nonmasking fault-tolerance to a *different* class of faults, then the resulting problem is NP-complete in program state space. This is a counterintuitive result in that adding failsafe and nonmasking fault-tolerance to the *same* class of faults can be done in polynomial time. We also present sufficient conditions for polynomial-time addition of *failsafe-nonmasking* multitolerance to programs. Finally, we demonstrate the stepwise addition of multitolerance in the context of a token ring protocol and a repetitive agreement protocol that is subject to Byzantine and transient faults.

Keywords: Fault tolerance, Multitolerance, Automatic addition of fault tolerance, Formal methods, Program synthesis

Contents

1 Introduction

Today's systems are often subject to multiple classes of faults and, hence, it is desirable that these systems tolerate multiple classes of faults. Since it is often undesirable or impractical to provide the same level of fault-tolerance to each class of faults, Arora and Kulkarni [1] introduced the notion of *multitolerance*, where a multitolerant system (possibly) provides a different level of fault-tolerance to each fault-class. 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., [2, 3, 4, 1]) in the literature.

The design of finite-state multitolerant programs is complicated as different levels of fault-tolerance considered for different classes of faults may be inconsistent. To alleviate this problem, Arora and Kulkarni [1] have provided a method for stepwise design of multitolerance. In this method, multitolerance is added in a stepwise fashion where in each step, only one class of faults is considered. Furthermore, in each step, fault-tolerance to previously added faults is preserved. The choices made during stepwise addition of multitolerance affect the completeness of the design in that the stepwise addition may fail to yield a multitolerant program while one exists. To illustrate this, consider the case where we begin with a *fault-intolerant* program p – that meets its specification in the absence of faults although provides no guarantees in the presence of faults – where fault-tolerance is to be provided to two classes of faults, say f_1 and f_2. Arora and Kulkarni [1] have shown that their method is complete, i.e., if such a multitolerant program exists then one can use their method to find programs p_1 and p_2 where p_1 provides the desired level of fault-tolerance to f_1 and p_2 provides the desired level of fault-tolerance to both f_1 and f_2. In other words, their stepwise method can first design p_1 and then p_2. However, the method in [1] can generate several possible fault-tolerant programs after the first step (as well as the second step). In particular, in the first step, the synthesized program may be p_1 (considered above) or p_1' (another program). Moreover, it is possible that if the program synthesized in the first step is p_1', then the method may fail to find a multitolerant program in the second step. In this sense, the method in [1] is *non-deterministically* complete. In other words, the choices made during the addition of fault-tolerance to f_1 may prevent adding fault-tolerance to f_2 even though a fault-tolerant program that provides tolerance to both faults exists.

From the above discussion, it follows that a deterministically (sound and) complete stepwise method for multitolerance would be especially valuable, as it would allow us to consider only one class of faults at a time while ensuring that the choices made in the addition of fault-tolerance to one class of faults would not prevent addition of fault-tolerance to another class of faults. To the best of our knowledge, the design of such a deterministically complete method is still an open problem for arbitrary inputs (i.e., faults, programs, and specifications). Moreover, we are not even aware of the existence of such a deterministically sound and complete method for cases where restrictive inputs are considered.

In this paper, we show that for a class of high atomicity programs (where a process can read and write all program variables in an atomic step) and a specification in the *bad transitions* (BT) model [5] (where the safety specification can be characterized in terms of bad states and bad transitions that should not occur in program computations), a deterministically sound and complete solution exists in the following two scenarios:

- Failsafe fault-tolerance – where safety is guaranteed in the presence of faults – is provided to one class of faults, and masking fault-tolerance – where a masking program eventually recovers to its legitimate states and ensures that safety is always preserved – is provided to another class of faults, called failsafe-masking multitolerance.

- Nonmasking fault-tolerance – where a nonmasking program eventually recovers to its legitimate states although safety may be violated during recovery – is provided to one class of faults, and masking fault-tolerance is provided to another class of faults, called nonmasking-masking multitolerance. (see Section 2 for precise definitions of failsafe, nonmasking and masking fault-tolerance.)

Additionally, we also find a counterintuitive result that if failsafe fault-tolerance is required to one class of faults and nonmasking fault-tolerance is desired to another class of faults, then such a sound and deterministically complete algorithm is *unlikely* to exist if the complexity of adding one class of faults is to be polynomial. In particular, we show that the problem of adding failsafe fault-tolerance to one class of faults and nonmasking fault-tolerance to a different class of faults is NP-complete. This result is surprising in that adding failsafe and nonmasking fault-tolerance to the *same* class of faults is polynomial [6]. In general, this NP-completeness result implies the non-existence of a stepwise method (where the complexity of each step is polynomial) unless $P = NP$. For efficient stepwise addition of multitolerance, we present sufficient conditions where polynomial-time stepwise addition can be achieved. We also demonstrate an example program where these sufficient conditions are satisfied.

Moreover, our stepwise method is amenable to automation. Such automated synthesis has the advantage of generating fault-tolerant programs that are correct by construction, and tolerate multiple classes of faults. In our method, we begin with a fault-intolerant program and add fault-tolerance to the given classes of faults while providing the required level of fault-tolerance to each of those fault-classes. This addition is performed by reusing the algorithms in [6] where one adds fault-tolerance to a single class of faults. Furthermore, to illustrate the soundness and completeness of our method, we only rely on certain properties of the algorithms in [6] and not on their implementation; i.e., we *reuse* the algorithms in [6] as black boxes.

The rest of the paper is organized as follows: In Section 2, we present preliminary concepts. 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 demonstrate that, in general, stepwise addition of multitolerance is NP-complete (in program state space), which constitutes an impossibility result for efficient stepwise design of multitolerance unless $P = NP$. In Section 5, we investigate the feasibility of sound and complete stepwise addition of multitolerance for special cases. In Section 6, we illustrate two examples of stepwise addition of failsafe-nonmasking-masking and nonmasking-masking multitolerance. Finally, in Section 7, 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 defined in terms of their state space and their transitions. The definition of specifications is adapted from Alpern and Schneider [7]. The definitions of faults and fault-tolerance are adapted from Arora and Gouda [8] and Kulkarni [9].

2.1 Programs

A program $p = \langle S_p, \delta_p \rangle$ is defined by a finite state space, S_p, and a set of transitions, δ_p, where δ_p is a subset of $S_p \times S_p$. 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 (if and only if) $\forall s_0, s_1 : (s_0, s_1) \in \delta_p : (s_0 \in S \Rightarrow s_1 \in S)$. A sequence of states, $\sigma = \langle s_0, s_1, ... \rangle$ with $len(\sigma)$ states, is a computation of p iff the following two conditions are satisfied: (1) $\forall j : 0 < j < len(\sigma) : (s_{j-1}, s_j) \in \delta_p$, and (2) if σ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in \delta_p$. A *finite* sequence of states, $\langle s_0, s_1, ..., s_n \rangle$, is a computation prefix of p iff $\forall j : 0 < j \leq n : (s_{j-1}, s_j) \in \delta_p$.

The projection of a program p on a non-empty 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 the context, we use p and δ_p interchangeably. We also say that a state predicate S is true in a state s iff $s \in S$.

2.2 Specifications

Following Alpern and Schneider [7], we let the specification be a set of infinite sequences of states. We assume that this set is suffix-closed and fusion-closed. Suffix closure of the set means that if a state sequence σ is in that set then so are all the suffixes of σ. Fusion closure of the set means that if state sequences $\langle \alpha, s, \gamma \rangle$ and $\langle \beta, s, \delta \rangle$ are in that set then so are the state sequences $\langle \alpha, s, \delta \rangle$ and $\langle \beta, s, \gamma \rangle$, where α and β are finite prefixes of state sequences, γ and δ are suffixes of state sequences, and s is a program state.

We say a computation $\sigma = \langle s_0, s_1, \cdots \rangle$ satisfies (does not violate) *spec* iff $\sigma \in spec$. 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 satisfies *spec*. If p satisfies *spec* from S and $S \neq \{\}$, we say that S is an invariant of p for *spec*.

Note that since specifications contain only infinite sequences, a program can satisfy a specification from S only if all its computations from S are infinite, i.e., there are no deadlock states, where a deadlock state has no outgoing transitions. If a program is permitted to have *terminating* or *fixpoint* states where the program can stay forever, then this can be specified explicitly by providing a self-loop for those states. With this requirement, we can distinguish between permitted fixpoint states that may be present in the fault-intolerant program and *deadlock* states that could be created during synthesis when transitions are removed.

While a finite state sequence cannot satisfy a specification, *spec*, we can determine whether it has a potential to satisfy it. With this intuition, we say that α maintains *spec* iff there exists β such that $\alpha\beta$ (concatenation of α and β) is in *spec*. We say that a finite sequence α violates *spec* iff α does not maintain *spec*. Note that we overload the word violate to say that a finite sequence does not maintain *spec*.

Based on [7], given any *spec*, it can be expressed as an intersection of a safety specification and a liveness specification, each of which is also a set of infinite sequences of states. Such infinite sequences of states cannot be used as an input to a synthesis routine, especially if we are interested in identifying the complexity of the synthesis algorithm. Hence, we need to identify an equivalent (but concise) finite representation. We discuss this next.

Representation of safety during synthesis. For a suffix-closed and fusion-closed specification, the safety specification can be characterized by a set of bad transitions (see Page 26, Lemma 3.6 of [9]), that is, for program p, its safety specification can be characterized by a subset of $\{(s_0, s_1) : (s_0, s_1) \in S_p \times S_p\}$. These two representations are equivalent in that (1) given a set bad_{tr} of bad transitions, it corresponds to the infinite state sequences where no sequence contains any transition from bad_{tr}, and (2) given a specification *spec* in terms of a set of infinite sequences, the set of bad transitions bad_{tr} includes those transitions that do not appear in any sequence in *spec*. Since the set of transitions provides a concise representation for safety specification, we use that as an input to the synthesis algorithm.

Remark. If fusion or suffix closure is not provided then safety specification can be characterized in terms of finite-length prefixes [7]. We have shown in [5] that if one adopts such general model of safety specification instead of our restricted model (i.e., the *bad transitions* model) then the complexity of synthesis significantly increases from polynomial (in program state space) to NP-hard. Hence, for efficient synthesis, based on which tool support [10] can be provided, we represent safety with a set of bad transitions that must not occur in program computations.

Representation of liveness during synthesis. From [7], a specification, *spec*, is a liveness specification iff for any finite prefix α, α maintains *spec*. Our synthesis algorithms do not need liveness specification during synthesis. This is due to the fact that if the fault-intolerant program satisfies its liveness specification then, in the absence of faults, the fault-tolerant program also satisfies it.

Notation. Whenever the specification is clear from the context, we shall 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 $\{(s_0, s_1) : (s_0, s_1) \in S_p \times S_p\}$. We use $p[]f$ to denote the transitions obtained by taking the union of the transitions in p and the transitions in f. We say that a state predicate T is an f-span (read as fault-span) of p from S iff the following two conditions are satisfied: (1) $S \subseteq T$, and (2) T is closed in $p[]f$. Observe that for all computations of p that start in S, T is a boundary in the state space of p to which (but not beyond which) the state of p may be perturbed by the occurrence of f transitions.

We say that a sequence of states, $\sigma = \langle s_0, s_1, ... \rangle$ with $len(\sigma)$ states, is a computation of p in the presence of f iff the following three conditions are satisfied: (1) $\forall j : 0 < j < len(\sigma) : (s_{j-1}, s_j) \in (\delta_p \cup f)$, (2) if σ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in \delta_p$, and (3) $\exists n : n \geq 0 : (\forall j : j > n : (s_{j-1}, s_j) \in \delta_p)$. The first requirement captures that in each step, either a program transition or a fault transition is executed. The second requirement captures that faults do not have to execute. Finally, the third requirement captures that the number of fault occurrences in a computation is finite. This requirement is the same as that made in previous work (e.g., [11, 12, 8, 13]) to ensure that eventually recovery can occur.

2.4 Fault Tolerance

We now define what it means for a program to be failsafe/nonmasking/masking fault-tolerant. The intuition for these definitions is in terms of whether the program satisfies safety and whether the program recovers to states from where subsequent computations satisfy safety and liveness specifications. Intuitively, if only safety is satisfied in the presence of faults, the program is failsafe. If the program recovers to states from where subsequent computations satisfy the specification, then it is nonmasking fault-tolerant. If the program always satisfies safety as well as recovers to states from where subsequent computations satisfy the specification then it is masking fault-tolerant. Based on this intuition, we define the levels of fault-tolerance in terms of the following requirements:

1. In the absence of f, p satisfies *spec* from S.
2. There exists an f-span of p from S, denoted T.
3. $p[]f$ maintains *spec* from T.
4. Every computation of $p[]f$ that starts from a state in T contains a state of S.

We say a program p is failsafe f-tolerant from S for *spec* iff p meets the requirements 1, 2 and 3. The program p is nonmasking f-tolerant from S for *spec* iff p meets the requirements 1, 2 and 4. The program p is masking f-tolerant from S for *spec* iff p meets the requirements 1, 2, 3 and 4.
Notation. Whenever the specification *spec* and the invariant S are clear from the context, we shall 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. 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 f_1 and f_2 occur in the same computation. With such a definition of multitolerance, the program would provide fault-tolerance if faults from f_1 occur or if faults from f_2 occur. However, no guarantees will be provided if both faults occur simultaneously. Another possibility is to provide the minimum level of fault-tolerance when f_1 and f_2 occur. In this approach, we impose an ordering on levels of fault tolerance based on a *less than* relation, denoted $\prec$, which orders two levels of fault tolerance based on the level of guarantees they provide in the presence of faults. Thus, based on the definition of fault tolerance, we have failsafe $\prec$ masking, nonmasking $\prec$ masking, intolerant $\prec$ masking, intolerant $\prec$ failsafe and intolerant $\prec$ nonmasking. As a result, the fault-tolerance provided for cases where f_1 and f_2 occur simultaneously should be equal to the minimum level of fault-tolerance provided when either f_1 occurs or f_2 occurs. To illustrate this, let masking fault tolerance be required to f_1 and nonmasking fault tolerance be desired to f_2. Thus, the occurrence of f_2 allows violation of safety. If we were to require masking fault tolerance for cases where f_1 and f_2 occur simultaneously, then this would mean that *safety may be violated in the presence of f_2 alone, however, if f_1 occurs after the occurrence of f_2, then safety must be preserved.* This contradicts the notion that faults are undesirable and make it harder for a program to meet its specification. Thus, the level of fault-tolerance for any combination of fault-classes will be less than or equal to the level of fault-tolerance provided to each class. We follow this approach to define the notion of multitolerance in this section.

We use the $\prec$ relation to determine the level of fault tolerance that should be provided when multiple classes of faults occur. For instance, if masking fault tolerance is required to $f1$ and failsafe (respectively, nonmasking) fault-tolerance is desired to $f2$, then failsafe (respectively, nonmasking) fault-tolerance should be provided for the case where $f1$ and $f2$ occur simultaneously. However, if nonmasking fault-tolerance is provided to $f1$ and failsafe fault-tolerance is provided to $f2$, then no level of fault-tolerance will be guaranteed for the case where $f1$ and $f2$ occur simultaneously. Figure 1 illustrates the minimum level of fault-tolerance provided for different combinations of levels of fault-tolerance.

	Failsafe	**Nonmasking**	**Masking**
Failsafe	*Failsafe*	*Intolerant*	*Failsafe*
Nonmasking	*Intolerant*	*Nonmasking*	*Nonmasking*
Masking	*Failsafe*	*Nonmasking*	*Masking*

Figure 1: Minimum level of fault-tolerance (in *italic*) provided for combinations of two levels of fault-tolerance.

When a program is subject to several classes of faults for which the same level of fault-tolerance is required, the addition of multitolerance amounts to making the union of all those faults as a single fault-class and providing the desired level of fault-tolerance for the union. For example, consider the situation where failsafe fault-tolerance is required for two classes of faults f_1 and f_2. From the above description, failsafe fault-tolerance should be provided for the fault class $f_f = f_1 \cup f_2$. Likewise, we obtain the fault-class f_n (respectively, f_m) for which nonmasking (respectively, masking) fault-tolerance is provided. Therefore, hereafter, f_f (respectively, f_n or f_m) denotes the union of all classes of faults for which failsafe (respectively, nonmasking or masking) fault-tolerance is required. We would like to note that while, in this case, we add fault tolerance to the union of fault-classes, it is feasible to apply the stepwise approach proposed in this paper to incrementally add fault tolerance to f_1 and then to f_2 (see Section 5 for details).

Now, given the transitions of a fault-intolerant program, p, its invariant, S, its specification, *spec*, and a set of classes of faults f_f, f_n, and f_m, we define what it means for a program p' (synthesized from p), with invariant S', to be multitolerant by considering how p' behaves when (i) no faults occur, and (ii) either one of faults f_f, f_n, and f_m occurs. Observe that if faults in f_f and f_m simultaneously occur in a computation then safety must be preserved in that computation. In other words, failsafe fault-tolerance must be provided in cases where faults from f_f and/or f_m occur. If faults from f_m alone occur then masking fault-tolerance must also be provided. Thus, the set of faults to which masking fault-tolerance is provided is a subset of the set of faults to which failsafe fault-tolerance is provided. With this intuition, we require that $f_m \subseteq f_f$. Likewise, we require that $f_m \subseteq f_n$. Therefore, we define multitolerant programs as follows:

Definition 3.1 Let f_m be a subset of $f_n \cap f_f$. Program p' is *multitolerant* to f_f, f_n, and f_m from S' for *spec* iff the following conditions hold:

1. p' satisfies *spec* from S' in the absence of faults.
2. p' is masking f_m-tolerant from S' for *spec*.
3. p' is failsafe f_f-tolerant from S' for *spec*.
4. p' is nonmasking f_n-tolerant from S' for *spec*. $\square$

For cases where only two types of faults are considered, we assign an appropriate value to the third fault-class. For example, if only f_m and f_n (where $f_m \subseteq f_n$) are considered then f_f is assigned to be equal to f_m. If only f_f and f_n are considered then f_m is assigned to be equal to $f_n \cap f_f$.

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 a natural extension of the problem statement in [6] 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 non-empty subset of S. Otherwise, there exists a state $s \in S'$ where $s \notin S$, and *in the absence of faults*, p' might reach s and perform new computations (i.e., new behaviors) that do not belong to p. Thus, p' might 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. Therefore, we define the multitolerance synthesis problem as follows:

The Multitolerance Synthesis Problem
Given p, S, *spec*, f_f, f_n, and f_m, identify p' and S' such that
$S' \subseteq S$,
$p'|S' \subseteq p|S'$, and
p' is multitolerant to f_f, f_n, and f_m from S' for *spec*. $\square$

We state the corresponding decision problem as follows:

The Multitolerance Decision Problem
Given p, S, *spec*, f_f, f_n, and f_m:
Does there exist a program p'*, with its invariant* S' *that satisfy the requirements of the synthesis problem*? $\square$

Notations. Given a fault-intolerant program p, specification *spec*, invariant S and classes of faults f_f, f_n, and f_m, we say that a program p' and a predicate S' *solve the (multitolerance) synthesis problem* iff p' and S' satisfy the three requirements of the synthesis problem. We say p' (respectively, S') solves the synthesis problem iff there exists S' (respectively, p') such that p', S' solve the synthesis problem.
Soundness and Completeness. An algorithm $\mathcal{A}$ is *sound* iff for all input instances consisting of a program p, its invariant S, its specification *spec*, and classes of faults f_f, f_n, and f_m, if $\mathcal{A}$ generates an output, then its output meets the requirements of the synthesis problem (i.e., solves the multitolerance synthesis problem). The algorithm $\mathcal{A}$ is *complete* iff when the answer to the multitolerance decision problem (as defined above) is affirmative, $\mathcal{A}$ always finds a multitolerant program p' with an invariant S' that solve the synthesis problem.

4 Impossibility of Stepwise Addition

In this section, we illustrate that, in general, synthesizing multitolerant programs from their fault-intolerant version is NP-complete. In Section 4.1, we present a polynomial-time mapping between a given instance of the 3-SAT problem and an instance of the (decision) problem of synthesizing multitolerance for the general case where failsafe-nonmasking-masking multitolerance is added to a program. Then, in Section 4.2, we show that the given 3-SAT instance is satisfiable iff the answer to the multitolerance decision problem (see Section 3) is affirmative; i.e., there exists a multitolerant program synthesized from the instance of the decision problem of multitolerance synthesis. We then illustrate the NP-completeness of the stepwise design of failsafe-nonmasking (FN) multitolerance.

4.1 Mapping 3-SAT to Adding Multitolerance

In this section, we present a polynomial-time reduction from any given instance of the 3-SAT problem to an instance of the decision problem defined in Section 3. A constructed 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_f, f_n, and f_m that perturb p. The problem statement for the 3-SAT problem is as follows:
3-SAT problem.

Given is a set of propositional variables, $a_1, a_2, ..., a_n$, 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. (A literal is a propositional variable or its negation.)

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

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. Corresponding to the propositional variables and disjunctions of the given 3-SAT instance, we include additional states outside the invariant. For each propositional variable a_i, we introduce the states x_i, x'_i, y_i, v_i (see Figure 2). For each disjunction $c_j = (a_i \vee \neg a_k \vee a_r)$ ($1 \leq i \leq n$, $1 \leq k \leq n$, and $1 \leq r \leq n$), we introduce a state z_j outside the invariant ($1 \leq j \leq M$).

The transitions of the fault-intolerant program. The only transition in $p|S$ is (s, s).
The transitions of f_m. The set of fault transitions for which masking fault-tolerance is required can take the program from s to y_i (corresponding to each a_i). For each disjunction c_j, we also introduce a fault

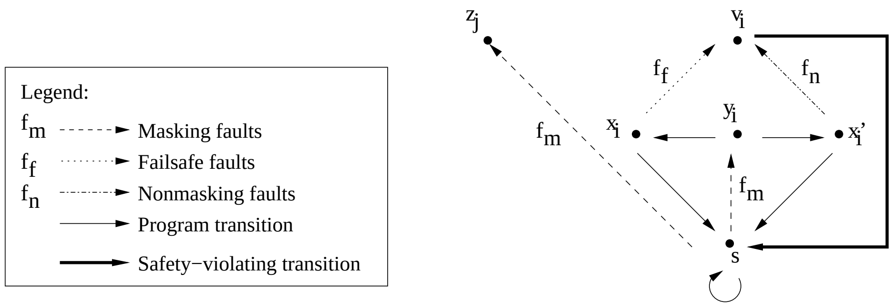

Figure 2: The states and the transitions corresponding to the propositional variables in the 3-SAT formula. The (v_i, s) transition violates safety.

transition that perturbs the program from state s to state z_j $(1 \leq j \leq M)$. Thus, f_m is equal to the set of transitions $\{(s, y_i) : 1 \leq i \leq n\} \cup \{(s, z_j) : 1 \leq j \leq M\}$.

The transitions of f_f. The transitions of f_f can perturb the program from x_i to v_i, for $1 \leq i \leq n$. Given that failsafe fault-tolerance should also be provided to f_m, the class of faults f_f would be equal to the set of transitions $f_m \cup \{(x_i, v_i) : 1 \leq i \leq n\}$.

The transitions of f_n. The transitions of f_n can perturb the program from x'_i to v_i. Moreover, recovery should be provided in the presence of f_m, thus $f_n = f_m \cup \{(x'_i, v_i) : 1 \leq i \leq n\}$.

The safety specification of the fault-intolerant program, p. None of the fault transitions, namely f_f, f_n, and f_m identified above violates safety. In addition, for each propositional variable a_i $(1 \leq i \leq n)$, the following transitions do not violate safety (see Figure 2):

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

For each disjunction $c_j = a_i \vee \neg a_k \vee a_r$, the following transitions do not violate safety:

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

The safety specification of the instance of the multitolerance problem forbids the execution of any transition except those identified above. For example, observe that, in Figure 2, the set of transitions (v_i, s), for $1 \leq i \leq n$, violates safety.

4.2 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 4.1.

Lemma 4.1 If the given 3-SAT formula is satisfiable then there exists a multitolerant program that solves the instance of the multitolerance synthesis problem identified in Section 4.1.
Proof. Since the 3-SAT formula is satisfiable, there exists an assignment of truth values to the propositional variables a_i, $1 \leq i \leq n$, such that each c_j, $1 \leq j \leq M$, is *true*. Now, we identify a multitolerant program, p', that is obtained by adding multitolerance to the fault-intolerant program p identified in Section 4.1. 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 follows. (We illustrate a partial structure of p' where $a_i = true$, $a_k = false$, and $a_r = true$ $(1 \leq i, k, r \leq n)$ in Figure 3.)

- For each propositional variable a_i, $1 \leq i \leq n$, if a_i is *true* then we include the transitions (y_i, x_i) and (x_i, s). Moreover, for each disjunction c_j that includes a_i, we include the transition (z_j, x_i). Thus, in the presence of f_m alone, p' guarantees recovery to s through x_i while preserving safety; i.e., *safe recovery* to invariant.

- For each propositional variable a_i, $1 \leq i \leq n$, if a_i is *false* then we include (y_i, x'_i) and (x'_i, s) to provide safe recovery to the invariant. Moreover, corresponding to each disjunction c_j that includes $\neg a_i$, we include transition (z_j, x'_i). In this case, since state v_i can be reached from x'_i by faults f_n, we include transition (v_i, s) so that in the presence of f_n program p' recovers to s.

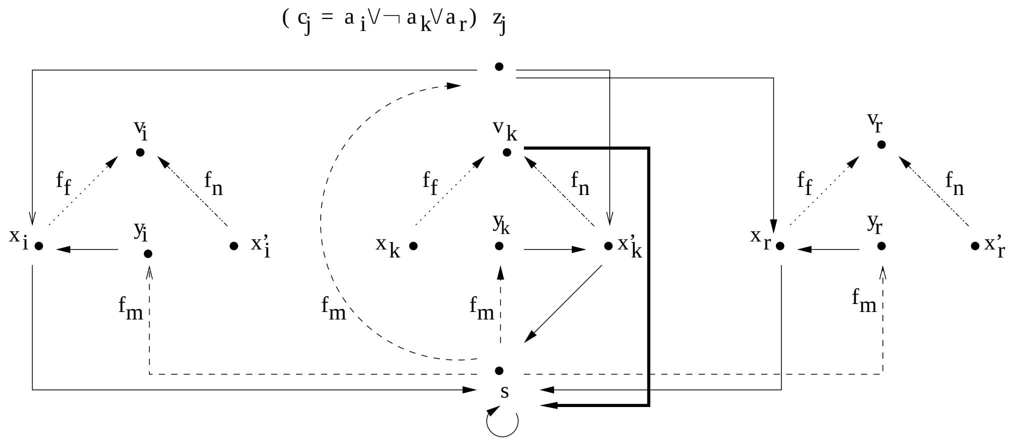

Figure 3: The partial structure of the multitolerant program where $a_i = true, a_k = false$ and $a_r = true$. For failsafe f_f-tolerance, the deadlock states v_i and v_r are permitted as they are reachable only in computations of $p[]f_f$.

Now, we show that p' is multitolerant in the presence of faults f_f, f_n, and f_m.

- **p' in the absence of faults.** $p'|S = p|S$. Thus, p' satisfies *spec* in the absence of faults.
- **Masking f_m-tolerance.** If the faults from f_m 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 $p'[]f_m$. Thus, any computation of $p'[]f_m$ eventually reaches a state in the invariant while preserving safety. If a_i is *false* then there exists exactly one sequence of transitions, $\langle (y_i, x'_i), (x'_i, s) \rangle$, in $p'[]f_m$. By the same argument, any computation of $p'[]f_m$ reaches a state in the invariant without violating safety.

  In the second case, since c_j evaluates to *true*, one of the literals in c_j 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, $\neg a_k$) is a literal in c_j and a_k (respectively, $\neg a_k$) 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 $p'[]f_m$ except (x_k, s) (respectively, (x'_k, s)), every computation of p' reaches the invariant without violating safety. Thus, p' is masking f_m-tolerant.
- **Failsafe f_f-tolerance.** Based on the case considered above, if only faults from f_m occur then the program is also failsafe fault-tolerant. Hence, we consider only the case where at least one fault from $f_f - f_m$ has occurred. Faults in $f_f - f_m$ occur only in state x_i, $1 \leq i \leq n$. 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 $f_f - f_m$ occurs p' simply stops. Note that a failsafe program is allowed to halt outside its invariant without violating safety. Therefore, p' is failsafe f_f-tolerant.
- **Nonmasking f_n-tolerance.** Consider the case where at least one fault transition of $f_n - f_m$ has occurred. Faults in $f_n - f_m$ occur only in state x'_i, $1 \leq i \leq n$. 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_n, p' recovers to $\{s\}$. $\square$

Lemma 4.2 If there exists a multitolerant program that solves the instance of the synthesis problem identified in Section 4.1 then the given 3-SAT formula is satisfiable.

Proof. Suppose that there exists a multitolerant program p' derived from the fault-intolerant program, p, identified in Section 4.1. Since the invariant of p', S', is non-empty, $S = \{s\}$ and $S' \subseteq S$, S' must include state s. Thus, $S' = S$. Since each y_i, $1 \leq i \leq n$, is directly reachable from s by a fault from f_m, 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*. If p' includes (y_i, x'_i) then we assign a_i to be *false*. This way, each propositional variable in the 3-SAT formula will get at least one truth assignment. Now, we show that the truth assignment to each propositional variable is consistent and that each disjunction in the 3-SAT formula evaluates to *true*.

- *Each propositional variable gets a unique truth assignment.* Suppose that there exists a propositional variable 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, faults from f_m and f_n have occurred. Hence, p' must at least provide recovery from v_i to invariant. Moreover, v_i can be reached by the following transitions (s, y_i), (y_i, x_i), and (x_i, v_i). In this case, faults from f_m and f_f have occurred. Hence, p' must ensure safety. Since it is impossible to provide safe recovery from v_i to s, the propositional variable a_i must be assigned only one truth value.

- *Each disjunction is true.* Let $c_j = a_i \vee \neg a_k \vee a_r$ be a disjunction in the given 3-SAT formula. Note that state z_j can be reached by the occurrence of f_m from s. Thus, 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 propositional variables. Consider the case where p' contains transition (z_j, x_i). Thus, p' can reach x_i in the presence of f_m alone. Moreover, let a_i be *false*. Then p' contains the transition (y_i, x'_i). Thus, x'_i can also be reached by the occurrence of f_m alone. Based on the above proof for unique assignment of truth values to propositional variables, p' cannot reach x_i and x'_i in the presence of f_m alone. Hence, if (z_j, x_i) is included in p' then a_i must have been assigned the truth value *true*; i.e., c_j becomes *true*. Likewise, if (z_j, x'_k) is included in p' then a_k must be assigned *false*. Thus, each disjunction evaluates to *true*. □

Theorem 4.3 The problem of synthesizing multitolerant programs from their fault-intolerant versions is NP-complete.

Proof. Based on Lemmas 4.1 and 4.2, the NP-hardness of the multitolerance synthesis problem follows. We have omitted the proof of NP membership since it is straightforward. (see Appendix B for the proof of NP membership). □

NP-completeness of failsafe-nonmasking (FN) multitolerance. In order to illustrate the NP-completeness of FN multitolerance, we extend the NP-completeness proof of synthesizing multitolerance in that we replace the f_m fault transition (s, y_i) with a sequence of transitions of f_f and f_n as shown in Figure 4. Likewise, we replace fault transition (s, z_j) with a structure similar to Figure 4. Thus, y_i (respectively, z_i) is reachable by f_f faults alone and by f_n faults alone. As a result, v_i is reachable in the computations of $p'[]f_f$ and in the computations of $p'[]f_n$. Thus, to add multitolerance, safe recovery must be added from v_i to s (see Figure 2). Now, we note that with this mapping, the proofs of Lemmas 4.1 and 4.2, and Theorem 4.3 can be easily extended to show that synthesizing FN multitolerance is NP-complete.

Theorem 4.4. The problem of synthesizing failsafe-nonmasking multitolerant programs from their fault-intolerant version is NP-complete. □

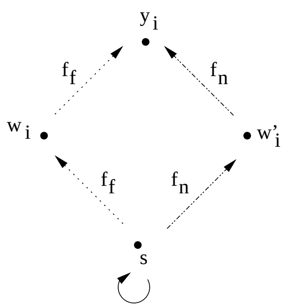

Figure 4: A proof sketch for NP-completeness of synthesizing failsafe-nonmasking multitolerance.

5 Feasibility of Stepwise Addition

While, in Section 4, we illustrate that the general case problem of synthesizing multitolerant programs is NP-complete, in our previous work [14], we have presented sound and complete polynomial algorithms for two special cases, namely, *nonmasking-masking* (NM) multitolerance and *failsafe-masking* (FM) multitolerance, where we add nonmasking (respectively, failsafe) fault-tolerance to one fault-class and masking fault-tolerance to another class of faults. Our algorithms in [14] for the addition of FM and NM multitolerance use the existing algorithms (presented by Kulkarni and Arora [6]) for the addition of fault tolerance to a single class of faults. While our algorithms in [14] are deterministically sound and complete, there are two problems with the use of these algorithms in practice. First, these are white-box algorithms in that designers should have knowledge about the internal working of the algorithms in [6]. Second, the algorithms in [14] cannot be used in a stepwise fashion. For instance, in the design of FM multitolerance, consider the case where only the class of faults f_f is known in early stages of development. As such, we can synthesize a failsafe f_f-tolerant program p_1. Now, if we perform some correctness-preserving maintenance or quality-of-service (e.g., performance) improvements on p_1, then, upon detecting the class of faults f_m (for which masking fault tolerance is required), it is desirable to reuse p_1 in the design of the FM multitolerant program instead of the original intolerant program. To address this problem, in this section, we present a stepwise approach in which we reuse the algorithms in [6] as black boxes. First, in Section 5.1, we represent the properties of the algorithms presented in [6]. Then, in Sections 5.2 and 5.3, we respectively investigate the stepwise addition of NM and FM multitolerance. Finally, in Section 5.4, we present some sufficient conditions for polynomial-time addition of FN multitolerance.

5.1 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 [6]. These algorithms take a program p, its invariant S, its specification *spec*, a class of faults f, and synthesize a failsafe/nonmasking/masking f-tolerant program p' (if one exists) with a new invariant S' and an f-span T'. The synthesized program p' and its invariant S' satisfy the following requirements: (1) $S' \subseteq S$; (2) $p'|S' \subseteq p|S'$, and (3) p' is failsafe/nonmasking/masking f-tolerant from S' for *spec*. The Add_Failsafe algorithm removes *offending states* in S and *offending transitions* in $p|S$, where from an offending state a sequence of fault transitions alone may violate safety of *spec*. An offending transition either violates safety of *spec* or reaches an offending state. Add_Failsafe ensures that (1) no deadlock states are created in S', and (2) $p'|T'$ does not include any offending transitions. In addition to removing offending states/transitions, the Add_Masking algorithm adds new recovery transitions from $(T'-S')$ to S' while preserving the safety of *spec*. The Add_Nonmasking algorithm only adds recovery transitions to the invariant. In this section, we recall the relevant properties of these algorithms. While we reiterate these algorithms in Appendix A, we note that the description of the multitolerance algorithms and their proofs depend *only* on the properties mentioned in this section and not on the actual implementation of the algorithms in [6].

For Add_Failsafe and Add_Masking, the invariant S' 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. More precisely, if there exists a failsafe (respectively, masking) fault-tolerant program p'', with invariant S'' that satisfies $S'' \subseteq S$, $p''|S'' \subseteq p|S''$, and p'' is failsafe (respectively, masking) f-tolerant from S'' for *spec*, then $S'' \subseteq S'$. Moreover, if the invariant S does not include any offending states, then Add_Failsafe will not change the invariant of the fault-intolerant program. Now, let the input for Add_Failsafe be p, S, *spec* and f. Let the output of Add_Failsafe be the fault-tolerant program p' and invariant S'. We state the following properties:

Property 5.1.1 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 $S'' \subseteq S'$. $\square$

Property 5.1.2 If there exist no offending states in S, then $S' = S$ and $p'|S' = p|S'$. $\square$

Likewise, the f-span of the masking f-tolerant program, say T', synthesized by the algorithm Add_Masking is the largest possible f-span for a masking program synthesized from p. Thus, we state the following properties:

Property 5.1.3 Let the input for Add_Masking be p, S, *spec* and f. Let the output of Add_Masking be the fault-tolerant program p', invariant $S' \subseteq S$, 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 $T'' \subseteq T'$. □

Property 5.1.4 Let the input for Add_Masking be p, S, *spec* and f and T be the set of states reachable by computations of $p[]f$. Let the output of Add_Masking be fault-tolerant program p', invariant $S' \subseteq S$, and fault-span T'. If there exist some offending states in T, then T' does not include such states, and as a result, T' is a proper subset of T (i.e., $T' \subset T$). □

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

Property 5.1.5 Add_Nonmasking does not add/remove any state to/from S. □

Property 5.1.6 Add_Nonmasking does not add/remove any transition to/from $p|S$. □

Based on the Properties 5.1.1– 5.1.6, Kulkarni and Arora [6] show that the algorithms Add_Failsafe, Add_Nonmasking, and Add_Masking are sound and complete, i.e., the output of these algorithms satisfies the three requirements (mentioned in the beginning of this section) for adding fault-tolerance to a *single* class of faults and these algorithms can find a fault-tolerant version of the fault-intolerant program if one exists.

Theorem 5.1.7 The algorithms Add_Failsafe, Add_Nonmasking, and Add_Masking are sound and complete. (see [6] for proof.) □

5.2 Nonmasking-Masking (NM) Multitolerance

In this section, we present a sound and complete algorithm (see Figure 5) for stepwise design of NM multitolerant programs from their fault-intolerant versions that are subject to two classes of faults f_n and f_m, where $f_m \subseteq f_n$. Formally, given a program p, with its invariant S, its specification *spec*, our goal is to synthesize a program p', with invariant S' that is NM multitolerant to f_n and f_m from S' for *spec*. By definition, p' must be masking f_m-tolerant and nonmasking f_n-tolerant. Towards this end, we proceed as follows: Using the algorithm Add_Masking, we synthesize a masking f_m-tolerant program p_1, with invariant S_1, and fault-span T_m (Line 1 in Figure 5). Now, since program p_1 is masking f_m-tolerant, it provides safe recovery to its invariant, S_1, from every state in $(T_m - S_1)$. Thus, in the presence of f_n, if p_1 is perturbed to $(T_m - S_1)$ then p_1 will satisfy the requirements of nonmasking fault-tolerance (i.e., recovery to S_1). However, if f_n perturbs p_1 to a state s, where $s \notin T_m$, then recovery must be added from s. Based on Properties 5.1.5 and 5.1.6, it suffices to add recovery to T_m as provided recovery by p_1 from T_m to S_1 can be reused *even after* adding nonmasking fault-tolerance. We invoke Add_Nonmasking (Line 3 in Figure 5) with T_m as an invariant of p_1.

```
Add_Masking_Nonmasking(p: transitions, $f_n, f_m$: fault,
                                    S: state predicate, spec: safety specification)
{
  $p_1, S_1, T_m := Add\_Masking(p, f_m, S, spec)$;                     (1)
  if ($S_1 = \{\}$)   declare no multitolerant program $p'$ exists;       (2)
                 return $\emptyset, \emptyset$;
  $p', S', T' := Add\_Nonmasking(p_1, f_n, T_m, spec)$;                 (3)
  return $p', S'$;                                                      (4)
}
```

Figure 5: Stepwise addition of NM multitolerance.

Theorem 5.2.1 The Add_Masking_Nonmasking algorithm is sound.

Proof. Based on the soundness of Add_Masking (see Theorem 5.1.7), we have $S_1 \subseteq S$. The equality $S_1 = S'$ follows from the Property 5.1.5. Also, using the soundness of Add_Masking, we have $p_1|S_1 \subseteq p|S_1$ (i.e., $p_1|S' \subseteq p|S'$). In addition, based on the Property 5.1.6, 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_n and f_m from S' for *spec*:

1. **Absence of faults.** From the soundness of Add_Masking, it follows that p_1 satisfies *spec* from S_1 $(= S')$ in the absence of faults. Add_Nonmasking does not add/remove any transitions to/from $p_1|S'$ (Property 5.1.6). Thus, it follows that p' satisfies *spec* from S'.

2. **Masking f_m-tolerance.** From the soundness of Add_Masking, p_1 is masking f_m-tolerant from $S_1(= S')$ for *spec*. Also, based on the Property 5.1.5 and 5.1.6, since $p_1|T_m = p'|T_m$, Add_Nonmasking preserves masking f_m-tolerance property of p_1. Therefore, p' is masking f_m-tolerant from S' for *spec*.

3. **Nonmasking f_n-tolerance.** From the soundness of Add_Nonmasking, we know that p' is nonmasking f_n-tolerant from T_m for *spec*. Also, since Add_Nonmasking preserves masking f_m-tolerance property of p_1, recovery from T_m to S' is guaranteed in the presence of f_n. Therefore, p' is nonmasking f_n-tolerant from S' for *spec*. □

Theorem 5.2.2. The algorithm Add_Masking_Nonmasking is complete.
Proof. Add_Masking_Nonmasking declares that a multitolerant program does not exist only when Add_Masking does not find a masking f_m-tolerant program. Therefore, the completeness of Add_Masking_Nonmasking follows from the completeness of Add_Masking. □

The reader may question what would have happened if nonmasking f_n-tolerance and masking f_m-tolerance had been added in a reverse order. One scenario for such a requirement is the case where f_m is unknown at the early stages of design, and we first reuse the algorithm Add_Nonmasking (Line 1 in Figure 6) to synthesize a program p_1 that is nonmasking f_n-tolerant from S_1 for *spec*. Before detecting f_m, we may perform some correctness-preserving revisions (e.g., for performance enhancement) on p_1. Now, upon detecting f_m, it is desirable to add masking f_m-tolerance to p_1 instead of the original fault-intolerant program p. Note that the recovery provided by p_1 may not guarantee safety. Hence, we apply the Add_Masking_Nonmasking algorithm to p_1 (Line 2 in Figure 6) to synthesize a NM multitolerant program. Notice that invoking Add_Masking instead of Add_Masking_Nonmasking may destroy the nonmasking property of p_1. Moreover, while the algorithm in Figure 6 may seem simple, it illustrates the potential of our stepwise addition method in building new aggregate algorithms from simpler addition algorithms.

```
Add_Nonmasking_Masking($p$: transitions, $f_n, f_m$: fault,
                                    $S$: state predicate, $spec$: safety specification)
{
  $p_1, S_1, T_n := Add\_Nonmasking(p, f_n, S, spec)$;                    (1)
  return $Add\_Masking\_Nonmasking(p_1, f_n, f_m, S_1, spec)$;            (2)
}
```

Figure 6: Stepwise addition of NM multitolerance in the reverse order.

Theorem 5.2.3 The Add_Nonmasking_Masking algorithm is sound and complete. □

5.3 Failsafe-Masking (FM) Multitolerance

In this section, we investigate the stepwise addition of Failsafe-Masking (FM) multitolerance to high atomicity programs that tolerate two classes of faults f_f and f_m for which failsafe and masking fault-tolerance are respectively required, where $f_m \subseteq f_f$. We start by reusing the Add_Failsafe algorithm (Line 1 in Figure 7), where we add failsafe f_f-tolerance to p. The resulting program p_1 provides failsafe f_f-tolerance from its invariant S_1. To add masking f_m-tolerance to p_1, we use the Add_Masking algorithm. Based on Property 5.1.4, Add_Masking ensures that T_m does not include any state from where a sequence of f_m transitions alone violates safety. However, since $f_m \subseteq f_f$, in the addition of masking f_m-tolerance to p_1, the fault-span of the resulting program may include states from where transitions of $f_f - f_m$ alone violate safety. Even though such states do not belong to T_1, we need to ensure that during the addition of masking f_m-tolerance (Line 4 in Figure 7) they are not included in the fault-span T_m. Towards this end, we strengthen the safety specification (Line 3 in Figure 7), denoted *spec'*, by including the transitions that reach outside T_1. Finally, we invoke the Add_Masking algorithm to add masking f_m-tolerance to p_1 from S_1 for *spec'*.

```
Add_Failsafe_Masking($p$: transitions, $f_f, f_m$: fault, $S$: state predicate, $spec$: safety specification)
{
  $p_1, S_1, T_1 := Add\_Failsafe(p, f_f, S, spec)$;                              (1)
  if ($S_1 = \{\}$)   declare no multitolerant program $p'$ exists;              (2)
                      return $\emptyset, \emptyset$;
  $spec' := spec \cup \{(s_0, s_1) : s_0 \in T_1 \ \wedge \ s_1 \notin T_1\}$;      (3)
  $p', S', T_m := Add\_Masking(p_1, f_m, S_1, spec')$;                             (4)
  if ($S' = \{\}$)   declare no multitolerant program $p'$ exists;               (5)
                     return $\emptyset, \emptyset$;
  return $p', S'$;                                                                (6)
}
```

Figure 7: Stepwise addition of FM multitolerance.

Theorem 5.3.1. The Add_Failsafe_Masking algorithm is sound.
Proof. Using the soundness of Add_Failsafe, we have $S_1 \subseteq S$. Based on the Property 5.1.3, we have $S' \subseteq S_1$. Also, from the soundness of Add_Failsafe, it follows that $p_1|S_1 \subseteq p|S_1$. Since $S' \subseteq S_1$ and $p_1|S_1 \subseteq p|S_1$, we have $p_1|S' \subseteq p|S'$. From the soundness of Add_Masking, it follows that $p'|S' \subseteq p_1|S'$. Therefore, we have $p'|S' \subseteq p|S'$.

Now, we show that p' (see Figure 7) is indeed multitolerant to f_f and f_m from S' for *spec*.

1. **Absence of faults.** From the soundness of Add_Failsafe, it follows that p_1 satisfies *spec* from S_1 in the absence of faults. Also, based on the soundness of Add_Masking, p' satisfies *spec'* from S' in the absence of faults. Since *spec'* is a strengthened version of *spec* (i.e., the set of bad transitions ruled out by *spec* is a subset of the set of bad transitions ruled out by *spec'*), it follows that p' satisfies *spec* from S' in the absence of faults.

2. **Failsafe f_f-tolerance.** From the soundness of Add_Failsafe, p_1 is failsafe f_f-tolerant from S_1 for *spec*. Thus, no computation of p_1 violates safety of specification from T_1 (respectively, from S_1). Also, based on the soundness of Add_Masking, p' does not execute any transitions that violate *spec'*; i.e., no state outside T_1 becomes reachable due to the addition of masking f_m-tolerance. Since $S' \subseteq S_1$, no computation of p' will ever violate *spec* in the presence of f_f from S'. Therefore, p' is failsafe f_f-tolerant from S' for *spec*.

3. **Masking f_m-tolerance.** The soundness of Add_Masking guarantees that p' is masking f_m-tolerant from S' for *spec'*. Since *spec'* is a strengthened version of *spec*, it follows that p' is masking f_m-tolerant from S' for *spec*.

□

Theorem 5.3.2. The Add_Failsafe_Masking algorithm is complete.
Proof. If there exists a program p'', with invariant S'', and fault-span T'' that is multitolerant to f_f and f_m then p'' must be failsafe f_f-tolerant from S'' for *spec*. Our algorithm declares failure only if there is no such failsafe program synthesized from p (due to the completeness of Add_Failsafe). Also, since p'' is multitolerant, it must provide masking f_m-tolerance from S'' in the presence of f_m faults. Since our algorithm declares failure only if no program can be synthesized from p that meets both the requirements of failsafe f_f-tolerance and masking f_m-tolerance, it follows that Add_Failsafe_Masking is complete. □

Now, we pose the following question: *Is the stepwise design of FM multitolerance possible if masking f_m-tolerance is added to p before the addition of failsafe f_f-tolerance*? To address this question, we present the Add_Masking_Failsafe algorithm (see Figure 8)

The Add_Masking_Failsafe algorithm first adds masking f_m-tolerance to p (Line 1 in Figure 8) using the Add_Masking algorithm. If such an addition of masking f_m-tolerance succeeds, then we reuse our

```
Add_Masking_Failsafe(p: transitions, $f_f, f_m$: fault, S: state predicate, spec: safety specification)
{
  $p_1, S_1, T_1 := Add\_Masking(p, f_m, S, spec)$;                                  (1)
  if ($S_1 = \{\}$)   declare no multitolerant program $p'$ exists;                (2)
                      return $\emptyset, \emptyset$;
  return $Add\_Failsafe\_Masking(p_1, f_f, f_m, S_1, spec)$;                          (3)
}
```

Figure 8: Stepwise addition of FM multitolerance in the reverse order.

Add_Failsafe_Masking algorithm to generate a FM multitolerant program.

Theorem 5.3.3. The Add_Masking_Failsafe algorithm is sound and complete. (Proof is straightforward, hence omitted.) □

We emphasize that while the algorithms for adding NM and FM multitolerance combine existing algorithms, the use of our algorithms ensures that the decisions taken in adding fault-tolerance to the known classes of faults will not adversely affect the ability to tolerate other classes of faults. Our stepwise approach is also applicable for the case where the same level of fault tolerance is required against two different classes of faults f_1 and f_2. To the best of our knowledge, the algorithms in Sections 5.2 and 5.3 are the first deterministically sound and complete algorithms for stepwise addition of multitolerance.

5.4 Sufficient Conditions for Polynomial-Time Addition of Failsafe-Nonmasking

In order to deal with the exponential complexity of adding failsafe-nonmasking (FN) multitolerance, we pose the following questions: *Under what conditions the addition of FN multitolerance can be done in polynomial time?* In other words, *what conditions should be imposed on faults, specifications, and programs so that adding FN multitolerance could be done in polynomial time?* The NP-completeness of adding FN multitolerance (see Theorem 4.4) is due to the following issues: (i) the existence of states outside the invariant that are reachable in the presence of f_f alone and in the presence of f_n alone, and (ii) the impossibility of adding safe recovery from such states.[1]

Let p be a program with its invariant S, its specification *spec*, and classes of faults f_f and f_n for which we respectively require failsafe f_f-tolerance and nonmasking f_n-tolerance. Moreover, let (i) T_f be the set of states reachable by the computations of $p[]f_f$ from S, and (ii) T_n be the set of states reachable by the computations of $p[]f_n$ from S. Further, let the specification *spec* be fault-safe for faults f_f, where fault-safe specifications identify a class of specifications that are not directly violated by fault transitions.

Definition 5.4.1 A specification *spec* is fault-safe for faults f (denoted f-safe) iff the following condition is satisfied.

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

We have adopted the definition of fault-safe specifications from [15]. The examples of fault-safe specifications include important problems such as Byzantine agreement, consensus and commit (see [15]). If the specification *spec* is f_f-safe then no sequence of f_f transitions alone will violate the safety of *spec* from S. As a result, the invariant of the multitolerant program, S', will be equal to S (see Property 5.1.2). If the set of states that are reachable outside the invariant in the computations of $p[]f_f$ is disjoint from the set of states that are reachable in the computations of $p[]f_n$ then the program p can distinguish the occurrence of f_f from the occurrence of f_n by respectively detecting the state predicates $(T_f - S)$ and $(T_n - S)$. Thus, in order to guarantee FN multitolerance, program p should guarantee (i) recovery to S if p detects that fault f_n has occurred, and (ii) safety if p detects that fault f_f has occurred.

Definition 5.4.2 We say f_f and f_n are *mutually exclusive* with respect to program p and its invariant S if and only if $((T_f - S) \cap (T_n - S)) = \emptyset$. □

[1]Without loss of generality, in this section, we concentrate on the cases where $f_f \cap f_n = \emptyset$; i.e., if $f_f \cap f_n \neq \emptyset$ and adding safe recovery against $f_f \cap f_n$ is possible, then two disjoint classes of faults can easily be drawn using the symmetric difference of f_f and f_n.

Next, we present the Add_Failsafe_Nonmasking algorithm (see Figure 9) that adds FN multitolerance to p in polynomial time if (i) faults f_f and f_n are mutually exclusive with respect to p and its invariant S, and (ii) the specification *spec* is f_f-safe.

```
Add_Failsafe_Nonmasking($p$: transitions, $f_f, f_n$: fault,
                        $S$: state predicate, $spec$: safety specification)
{
  $p_1, S_1, T_f$ := Add_Failsafe($p, f_f, S, spec$);
  $p', S', T_n$ := Add_Nonmasking($p_1, f_n, S_1, spec$);
  return $p', S'$;
}
```

Figure 9: Synthesizing failsafe-nonmasking multitolerance for mutually exclusive faults.

The algorithm Add_Failsafe_Nonmasking reuses the Add_Failsafe algorithm from [6] to add failsafe f_f-tolerance to p. Thus, program p_1 is failsafe f_f-tolerant for *spec* from S_1. Since *spec* is f_f-safe, Add_Failsafe does not remove any states from S, and as a result, $S_1 = S$. For this reason, this step of the algorithm is always successful; i.e., Add_Failsafe always finds a failsafe f_f-tolerant program. In the next step, we reuse the Add_Nonmasking algorithm from [6] to add nonmasking f_n-tolerance to p_1.

Theorem 5.4.4. If f_f and f_n are mutually exclusive and *spec* is f_f-safe then the algorithm Add_Failsafe_Nonmasking is sound.

Proof. Since *spec* is f_f-safe, based on the Properties 5.1.2 and 5.1.6, Add_Failsafe and Add_Nonmasking do not add/remove any states (respectively, transitions) to/from S (respectively, $p|S$). Hence, we have $S_1 = S = S'$ and $p_1|S_1 = p'|S' = p|S'$. Now, we show that p' is multitolerant to f_f and f_n from S' for *spec*:

1. **Absence of faults.** Since the equalities $S = S'$ and $p'|S' = p|S$ hold, it follows that every computation of p' starting in S' is a computation of p. Thus, p' satisfies *spec* from S'.

2. **Failsafe f_f-tolerance.** From the soundness of Add_Failsafe, p_1 is failsafe f_f-tolerant from S_1 for *spec*. Since $S' = S_1$, p_1 is failsafe f_f-tolerant from S' for *spec*. Based on the Properties 5.1.5 and 5.1.6, Add_Nonmasking does not add (respectively, remove) any transition in $p_1|S$. Also, since $(T_f - S)$ and $(T_n - S)$ are disjoint (by mutual exclusivity of f_f and f_n), Add_Nonmasking does not add any transitions to $p_1|(T_f - S)$. Hence, we have $p_1|T_f = p'|T_f$. Therefore, in the presence of f_f, p' will never execute a safety violating transition, and as a result, p' is failsafe f_f-tolerant from S' for *spec*.

3. **Nonmasking f_n-tolerance.** Since $S' = S$ and recovery is provided from $(T_n - S)$ to S, p' is nonmasking f_n-tolerant from S' for *spec*. $\square$

Theorem 5.4.5. The algorithm Add_Failsafe_Nonmasking has polynomial-time complexity. $\square$

6 Examples

In this section, we present two examples for stepwise addition of multitolerance. First, in Section 6.1, we present a failsafe-nonmasking-masking multitolerant token ring program that is subject to three classes of faults. Second, in Section 6.2, we present a nonmasking-masking repetitive Byzantine agreement protocol. (The Promela [16] model of the intermediate and final multitolerant programs of these examples are available in Appendixes C and D.)

6.1 Multitolerant Token Passing

In this section, we demonstrate how our stepwise algorithms facilitate the addition of multitolerance to a token ring program that is subject to three classes of faults.

The Two-Ring Token Passing (TRTP) program. The TRTP program includes 8 processes located in two rings A and B (see Figure 10). In Figure 10, the arrows show the direction of token passing. Process PA_i (respectively, PB_i), $0 \leq i \leq 2$, is the predecessor of PA_{i+1} (respectively, PB_{i+1}). Process PA_3 (respectively, PB_3) is the predecessor of PA_0 (respectively, PB_0). Each process PA_i (respectively, PB_i), $0 \leq i \leq 3$, has an integer variable a_i (respectively, b_i) with the domain {-1, 0, 1, 2, 3}, where -1 represents a detectably corrupted value. Moreover, process PA_i (respectively, PB_i), $0 \leq i \leq 3$, has a Boolean variable, denoted upa_i (respectively, upb_i), that represents whether or not that process has crashed.

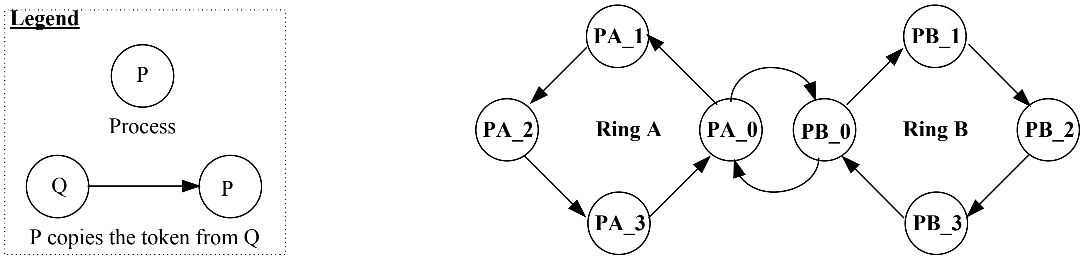

Figure 10: The two-ring token passing program.

Process PA_i, for $1 \leq i \leq 3$, *has the token* iff $(a_{i-1} = a_i \oplus 1) \wedge upa_i \wedge (a_{i-1} \neq -1) \wedge (a_i \neq -1)$, where $\oplus$ denotes addition modulo 4. Intuitively, PA_i has the token iff a_i is one unit less a_{i-1}, PA_i has not crashed, and a_i and a_{i-1} are not detectably corrupted. Process PA_0 has the token iff $(a_0 = a_3) \wedge (b_0 = b_3) \wedge (a_0 = b_0) \wedge upa_0 \wedge (a_0 \neq -1) \wedge (a_3 \neq -1)$; i.e., PA_0 has the same value as its predecessor and that value is equal to the values held by PB_0 and PB_3, PA_0 has not crashed and a_0 and a_3 are not corrupted. Process PB_0 has the token iff $(b_0 = b_3) \wedge (a_0 = a_3) \wedge ((b_0 \oplus 1) = a_0) \wedge upb_0 \wedge (b_0 \neq -1) \wedge (b_3 \neq -1)$. That is, PB_0 has the same value as its predecessor and that value is one unit less than the values held by PA_0 and PA_3, PB_0 has not crashed and b_0 and b_3 are not corrupted. Process PB_i $(1 \leq i \leq 3)$ *has the token* iff $(b_{i-1} = b_i \oplus 1) \wedge upb_i \wedge (b_{i-1} \neq -1) \wedge (b_i \neq -1)$. The TRTP program also has a Boolean variable $turn$; ring A executes only if $turn = true$, and if ring B executes then $turn = false$. We use Dijkstra's guarded commands language [17] as a shorthand for representing the set of program transitions. A guarded command (action) is of the form $grd \rightarrow stmt$, where grd is a state predicate and $stmt$ is a statement that updates program variables. The guarded command $grd \rightarrow stmt$ includes all program transitions $\{(s_0, s_1) : grd$ holds at s_0 and the *atomic* execution of $stmt$ at s_0 takes the program to state $s_1\}$. Using the following actions, the processes circulate the token in rings A and B $(i = 1, 2, 3)$:

$$\begin{array}{llcll}
AC_0: & (a_0 = a_3) \wedge turn & \longrightarrow & \text{if } (a_0 = b_0) & a_0 := a_3 \oplus 1; \\
 & & & \text{else} & turn := false; \\
AC_i: & (a_{i-1} = a_i \oplus 1) & \longrightarrow & a_i := a_{i-1}; &
\end{array}$$

Notice that the action AC_i is a parameterized action for processes PA_1, PA_2 and PA_3. The actions of the processes in ring B are as follows $(i = 1, 2, 3)$:

$$\begin{array}{llcll}
BC_0: & (b_0 = b_3) \wedge \neg turn & \longrightarrow & \text{if } (a_0 \neq b_0) & b_0 := b_3 \oplus 1; \\
 & & & \text{else} & turn := true; \\
BC_i: & (b_{i-1} = b_i \oplus 1) & \longrightarrow & b_i := b_{i-1}; &
\end{array}$$

Invariant. Consider a state s where $(\forall i : 0 \leq i \leq 3 : (a_i = 0) \wedge (b_i = 0))$ and $turn$ holds in s. The invariant of the TRTP program contains all the states that are reached from s by the execution of actions AC_i and BC_i, for $0 \leq i \leq 3$. Let $v(s)$ denotes the value of a variable v in s. Starting from a state s_0 where $(turn(s_0) = true) \wedge (\forall i : 0 \leq i \leq 3 : (a_i(s_0) = 0) \wedge (b_i(s_0) = 0))$, process PA_0 has the token and starts circulating the token until the program reaches the state s_1, where $(turn(s_1) = false) \wedge (\forall i : 0 \leq i \leq 3 : (a_i(s_1) = 1) \wedge (b_i(s_1) = 0))$; i.e., PB_0 has the token. Process PB_0 circulates the token until the program reaches a state s_2, where $(turn(s_2) = true) \wedge (\forall i : 0 \leq i \leq 3 : (a_i(s_2) = 1) \wedge (b_i(s_2) = 1))$, process PA_0 again has the token. This way the token circulation continues in both rings. The invariant $I_{TRTP} = I_{up} \wedge I_A \wedge I_B$ includes all states satisfying the following conditions:

$$I_{up} = \quad \forall i : 0 \leq i \leq 3 : (upa_i \wedge upb_i \;\wedge\; (a_i \neq -1) \;\wedge\; (b_i \neq -1))$$

$$\begin{aligned} I_A = \quad & (\forall i : 0 \leq i \leq 3 : a_i = a_{i\oplus 1}) \;\vee\; ((turn = true) \;\wedge\; (\exists j : 1 \leq j \leq 3 : (a_{j-1} = a_j \oplus 1) \;\wedge \\ & \qquad (\forall k : 0 \leq k < j-1 : a_k = a_{k+1}) \;\wedge\; (\forall k : j \leq k < 3 : a_k = a_{k+1})) \end{aligned}$$

$$\begin{aligned} I_B = \quad & (\forall i : 0 \leq i \leq 3 : b_i = b_{i\oplus 1}) \;\vee\; ((turn = false) \;\wedge\; (\exists j : 1 \leq j \leq 3 : (b_{j-1} = b_j \oplus 1) \;\wedge \\ & \qquad (\forall k : 0 \leq k < j-1 : b_k = b_{k+1}) \;\wedge\; (\forall k : j \leq k < 3 : b_k = b_{k+1})) \end{aligned}$$

The predicate I_{up} represents the set of states where no process has crashed and no variable is corrupted. The states where either all a (respectively, b) values are equal or it is the turn of ring A (respectively, B) and there is only one token in ring A (respectively, B) belong to the predicate I_A (respectively, I_B).
Safety specification. The safety specification of TRTP stipulates that in each state *at most* one token exists. This requirement could be due to some practical constraints where, for example, TRTP is used as an underlying protocol for assuring mutual exclusion and the process that has the token is allowed to access a shared resource. Additionally, no non-faulty process is allowed to copy the value of its detectably faulty predecessor.
Read/write constraints. While in the model represented in Section 2, each process can read/write all program variables in an atomic step, we consider the TRTP example under certain read/write constraints (imposed on processes with respect to the variables of other processes) in order to illustrate the applicability of our approach in the design of multitolerant programs in more concrete models. Specifically, process PA_i (respectively, PB_i), $1 \leq i \leq 3$, is allowed to read the state of its predecessor and write only a_i (respectively, b_i). Process PA_0 (respectively, PB_0) can read the state of its predecessor, PB_0 and PB_3 (respectively, PA_0 and PA_3) and *turn*. Process PA_0 (respectively, PB_0) is permitted to write only a_0 (respectively, b_0) and *turn*.
Faults f_m The class of faults f_m may detectably corrupt the state of only one process (i.e., set its value to -1) in one of the rings if no process is corrupted. Such faults may represent cases where a process fails and restarts.

$$\begin{aligned} \text{FM:} \quad & (\forall j : 0 \leq j \leq 3 : (a_j \neq -1) \;\wedge\; (b_j \neq -1)) \\ & \qquad \longrightarrow \quad a_0 := -1 \mid a_1 := -1 \mid a_2 := -1 \mid a_3 := -1 \mid \\ & \qquad\qquad\quad b_0 := -1 \mid b_1 := -1 \mid b_2 := -1 \mid b_3 := -1; \end{aligned}$$

The notation | represents the non-deterministic execution of only one of the assignments separated by |.
Faults f_f In addition to corrupting the state of only one process in a specific ring (by the action FM), this class of faults may also cause a process to crash in a detectable manner; i.e., set its *up* value to false. In addition to the action FM, the fault f_f includes the following action:

$$\begin{aligned} \text{FF:} \quad & (\forall j : 0 \leq j \leq 3 : (upa_j \;\wedge\; upb_j)) \\ & \quad \longrightarrow \quad upa_0 := false \mid upa_1 := false \mid upa_2 := false \mid upa_3 := false \mid \\ & \qquad\qquad upb_0 := false \mid upb_1 := false \mid upb_2 := false \mid upb_3 := false; \end{aligned}$$

Faults f_n The TRTP program is also subject to undetectable transient faults, denoted f_n, that may perturb the value of a and b variables non-deterministically. Note that since f_n transitions may generate multiple tokens, it is impossible to ensure that there is only one token at all times (i.e., safety is directly violated by faults f_n).

$$\begin{aligned} \text{FNA:} \quad & true \longrightarrow \quad a_0 := 0|1|2|3,\ a_1 := 0|1|2|3,\ a_2 := 0|1|2|3,\ a_3 := 0|1|2|3; \\ \text{FNB:} \quad & true \longrightarrow \quad b_0 := 0|1|2|3,\ b_1 := 0|1|2|3,\ b_2 := 0|1|2|3,\ b_3 := 0|1|2|3; \end{aligned}$$

In addition to corrupting the state of processes by the action FM, the class of faults f_n may non-deterministically assign a value between 0 and 3 to any variable.
Adding Failsafe_Masking. We use our stepwise algorithm in Figure 7 to add failsafe f_f-tolerance and masking f_m-fault tolerance. When we apply the Add_Failsafe algorithm, we generate the following failsafe

program TRTP' ($i = 1, 2, 3$). The **bold** font represents the new constraints/actions added to the intolerant program.

$$
\begin{array}{lll}
AC_0' : & (a_0 = a_3) \;\wedge\; \mathbf{upa_0} \;\wedge\; (\mathbf{a_3 \neq -1}) \;\wedge\; turn \;\longrightarrow & \\
& \quad \text{if } (((a_0 = b_0) \wedge (\mathbf{b_0 \neq -1})) \;\vee\; \neg\mathbf{upb_0}) & a_0 := a_3 \oplus 1; \\
& \quad \text{else} & turn := false; \\
AC_{01} : & \neg\mathbf{upa_3} \;\wedge\; \mathbf{upa_0} & \longrightarrow \; \mathbf{upa_0} := \mathbf{false};^2 \\
AC_{02} : & \neg\mathbf{upb_0} \;\wedge\; \neg\mathbf{upb_1} \;\wedge\; \neg\mathbf{upb_2} \;\wedge\; \neg\mathbf{upb_3} \;\wedge\; \mathbf{upa_0} \;\wedge\; \neg\mathbf{turn} & \\
& & \longrightarrow \; \mathbf{turn} := \mathbf{true}; \\
AC_i' : & (a_{i-1} = a_i \oplus 1) \wedge \mathbf{upa_i} \;\wedge\; (\mathbf{a_{i-1} \neq -1}) & \longrightarrow \; a_i := a_{i-1}; \\
AC_{i1} : & \neg\mathbf{upa_{i-1}} \;\wedge\; \mathbf{upa_i} & \longrightarrow \; \mathbf{upa_i} := \mathbf{false};
\end{array}
$$

Observe that the failsafe program contains new actions and new constraints in the guards of original actions. Specifically, the guards of actions AC_0 and AC_i have been strengthened in AC_0' and AC_i' to prevent a crashed process from executing and from copying a corrupted value. Moreover, if the process PB_0 has crashed then the token continues to be circulated in ring A (see action AC_0'). If a process has not crashed but its predecessor has crashed, then that process changes its state to crashed (see actions AC_{01} and AC_{i1})[2]. If all processes in ring B have crashed and the value of *turn* is not *true*, then *turn* will be set to true to permit token circulation in ring A. Due to space constraints, we omit the revised actions $BC_0', BC_{01}, BC_{02}, BC_i', BC_{i1}$ ($1 \leq i \leq 3$) of processes in ring B as symmetric revisions are performed on them in the failsafe f_f-tolerant program TRTP'.

After strengthening the safety specification as prescribed in Step 3 of Figure 7, we use Add_Masking to add masking f_m-tolerance to TRTP' in order to generate the program TRTP". The guard of the action AC_0' (respectively, BC_0') is weakened (see actions AC_0'' and BC_0'' below) to include recovery transitions that correct the state of the process PA_0 (respectively, PB_0) once corrupted by the fault action FM.

$$
\begin{array}{lll}
AC_0'' : & ((a_0 = a_3) \vee (\mathbf{a_0 = -1})) \;\wedge\; upa_0 \;\wedge\; (a_3 \neq -1) \;\wedge\; turn \;\longrightarrow & \\
& \quad \text{if } (((a_0 = b_0) \wedge (b_0 \neq -1)) \vee (\mathbf{a_0 = -1}) \;\vee\; \neg upb_0) & a_0 := a_3 \oplus 1; \\
& \quad \text{else} & turn := false; \\
BC_0'' : & ((b_0 = b_3) \vee (\mathbf{b_0 = -1})) \;\wedge\; upb_0 \;\wedge\; (b_3 \neq -1) \;\wedge\; \neg turn \longrightarrow & \\
& \quad \text{if } (((a_0 \neq b_0) \wedge (a_0 \neq -1)) \vee (\mathbf{b_0 = -1}) \;\vee\; \neg upa_0) & b_0 := b_3 \oplus 1; \\
& \quad \text{else} & turn := true;
\end{array}
$$

In the final step, we add nonmasking f_n-tolerance to TRTP" to generate a failsafe-masking-nonmasking program. *Note that the TRTP example meets the sufficient conditions identified in Section 5.4 for polynomial-time addition of failsafe-nonmasking multitolerance.* The final multitolerant program includes new recovery actions AC_{i2} and BC_{i2} respectively added to the set of actions of processes PA_i and PB_i, for $1 \leq i \leq 3$.

$$
\begin{array}{lll}
AC_{i2} : & (\mathbf{a_{i-1} \neq a_i \oplus 1}) \;\wedge\; (\mathbf{a_{i-1} \neq a_i}) \;\wedge\; \mathbf{upa_i} \;\wedge\; (\mathbf{a_{i-1} \neq -1}) & \longrightarrow \; \mathbf{a_i := a_{i-1}}; \\
BC_{i2} : & (\mathbf{b_{i-1} \neq b_i \oplus 1}) \;\wedge\; (\mathbf{b_{i-1} \neq b_i}) \;\wedge\; \mathbf{upb_i} \;\wedge\; (\mathbf{b_{i-1} \neq -1}) & \longrightarrow \; \mathbf{b_i := b_{i-1}};
\end{array}
$$

Observe that the final program has the following properties (see Figure 11): (1) If faults f_m occur, then there is at most one token at all times (i.e., safety) and every process is guaranteed to receive the token; (2) If faults f_f occur, then there is at most one token at all times and it is circulated amongst a *subset of processes*; (3) If faults f_n occur, then there may be multiple tokens, nonetheless, the program will eventually recover to states from where at most one token exists and every process will receive the token, and (4) If faults from $f_f \cup f_n$ occur, then the program will eventually recover to states from where at most one token exists and a *subset of processes* will receive the token. In this case, the program is the same as the self-stabilizing token ring program designed by Dijkstra [11] if we abstract out the *up* variables.

[2]We have overloaded the meaning of the *up* variables in that if the predecessor of a process has crashed, then that process is considered to be crashed as well. We could use another variable for this purpose, but for simplicity we have overloaded *up*.

Property / Fault-class	Always at most one token?	Token circulation among	Recovery to at most one token?
f_m	Yes	All processes	Yes
f_f	Yes	Subset of processes	Yes
f_n	No	All processes	Yes
$f_f \cup f_n$	No	Subset of processes	Yes

Figure 11: The properties of the multitolerant program TRTP".

Remark. While we have presented the TRTP program in the context of 4 processes in each ring, the example can be generalized for any fixed number of processes. Moreover, observe that the number of rings can also be increased, where one process from each ring participates in a higher level ring of processes in which token circulation determines which ring is active. This example can also be extended so that every process participates in several rings, thereby ensuring that non-faulty processes receive the token even if one or more processes fail.

6.2 Nonmasking-Masking Repetitive Byzantine Agreement

In this section, we synthesize a repetitive agreement protocol that provides masking fault tolerance to Byzantine faults and nonmasking fault tolerance to transient faults; i.e., nonmasking-masking multitolerant.
The fault-intolerant Repetitive Byzantine (RB) program. The RB program includes a general process, denoted P_g, and three non-general processes, denoted P_1, P_2, P_3. The program computations consist of consecutive rounds of decision making, where in each round P_g casts a binary decision and the non-generals copy the decision of the general and finalize their decision in the current round with an agreement on the same value. The process P_g has a decision variable, denoted d_g, with the domain {0, 1}. Each process P_i, for $1 \leq i \leq 3$, also has a decision variable, denoted d_i, with the domain {-1, 0, 1}, where -1 represents an undecided state for that process. To distinguish consecutive rounds of decision making from each other, each non-general process P_i uses Boolean variables sn_i and sn_old_i respectively representing the sequence number of the current round and that of the previous round. Process P_g has a Boolean variable sn_g representing the sequence number of the general. Process P_i copies its decision in an *output* decision variable d_old_i that should be read at the end of the current round. To determine whether or not a process is Byzantine, each process has a Boolean variable b. Now, if all sequence numbers are equal, the general starts the next round by toggling sn_g (see action G below) and resetting all b values. Since we allow the Byzantine process to change in every round, when the general begins a new round by executing action G, all processes are assumed to be non-Byzantine. Subsequently, a fault BF_1 (described later) can cause one of the processes to become Byzantine. Thus, our definition of Byzantine faults is a generalization of that in [18], where the same process is assumed to be Byzantine at all times.

$$\begin{array}{lll} G: & (sn_g = sn_1 = sn_2 = sn_3) & \longrightarrow & sn_g := \neg sn_g;\ d_g := 0|1; \\ & & & b_g := false;\ b_1 := false;\ b_2 := false;\ b_3 := false; \end{array}$$

Each non-general process P_i copies the decision of the general when it starts a new round (i.e., $sn_i \neq sn_g$) and it has not yet decided (see action A_{i0}). If P_i has not yet output its decision in the current round, then it will do so (see action A_{i1}). After outputting its decision, P_i finalizes the current round by toggling its sequence number and resetting d_i (see action A_{i2}).

$$\begin{array}{llll} A_{i0}: & (d_i = -1) \wedge (sn_i \neq sn_g) & \longrightarrow & d_i := d_g; \\ A_{i1}: & (d_i \neq -1) \wedge (sn_i = sn\_old_i) & \longrightarrow & d\_old_i := d_i;\ sn\_old_i := \neg sn\_old_i; \\ A_{i2}: & (d_i \neq -1) \wedge (sn_i \neq sn\_old_i) & \longrightarrow & d_i := -1;\ sn_i := \neg sn_i; \end{array}$$

Byzantine faults (f_m). At the start of each round, the Byzantine faults f_m may cause at most one process to become Byzantine if no process is Byzantine. At any time, a Byzantine process may arbitrarily change its decision in the round it has become Byzantine.

$$
\begin{array}{lll}
BF_1: & (sn_1 = sn_2 = sn_3) \ \wedge \ (sn_1 \neq sn_g) \ \wedge & \\
& \qquad\qquad \neg b_g \ \wedge \ \neg b_1 \ \wedge \ \neg b_2 \ \wedge \ \neg b_3 \longrightarrow & b_g := true \mid \\
& & b_i := true; \\
BF_2: & b_g \longrightarrow & d_g := 0|1; \\
BF_{3i}: & b_i \longrightarrow & d_i := 0|1;\ d\_old_i := 0|1;
\end{array}
$$

Safety specification. The safety specification requires validity and agreement in every round. In other words, at the end of every round (i.e., when the guard of action G is enabled), if the general is non-Byzantine then its decision (d_g) must match the decision of all non-Byzantine non-generals (d_old) from that round (i.e., *validity*). If the general is Byzantine then the decision of all non-Byzantine non-generals (d_old) must match each other (i.e., *agreement*).

Adding masking f_m-tolerance. We use the Add_Masking algorithm to generate masking program. The action A_{i0} remains unchanged. However, the algorithm revises the actions A_{i1} and A_{i2} to A'_{i1} and A'_{i2}, and adds a new action A_{i3} as follows $(1 \leq i \leq 3)$:

$$
\begin{array}{lll}
A'_{i1}: & (d_i \neq -1) \ \wedge \ (sn\_old_i = sn_i) \wedge (\mathbf{sn_1 = sn_2 = sn_3}) \ \wedge & \\
& \qquad (\forall \mathbf{i : 1 \leq i \leq 3 : d_i \neq -1}) \ \wedge \ (\mathbf{sn_i \neq sn_g}) \ \longrightarrow & d\_old_i := d_i; \\
& & sn\_old_i := \neg sn\_old_i; \\
A'_{i2}: & (d_i \neq -1) \wedge (\forall \mathbf{i : 1 \leq i \leq 3 : sn\_old_i = sn_g}) \ \wedge \ (\mathbf{d_i} = maj) \ \wedge & \\
& \qquad (\mathbf{sn_i \neq sn_g}) \longrightarrow & d_i := -1; \\
& & sn_i := \neg sn_i; \\
A_{i3}: & (\mathbf{d_i \neq -1}) \ \wedge \ (\forall \mathbf{i : 1 \leq i \leq 3 : sn\_old_i = sn_g}) \ \wedge \ (\mathbf{d_i \neq \ maj}) \ \wedge & \\
& \qquad (\mathbf{sn_i \neq sn_g}) \longrightarrow & \mathbf{d_i := maj}; \\
& & \mathbf{d\_old_i := maj};
\end{array}
$$

where $maj = \ majority(d\_old\_1, d\_old\_2, d\_old\_3)$.

Observe that the program RB$'$ consisting of actions $G, A_{i0}, A'_{i1}, A'_{i2}$ and A_{i3}, for $1 \leq i \leq 3$, is masking f_m-tolerant.

Transient faults. In addition to the class of faults f_m, the transient faults f_n perturb the state of program RB$'$ and change the decision values and sequence numbers by the following action (Notice that the faults may perturb several processes; i.e., $1 \leq i \leq 3$.):

$$
\begin{array}{llll}
\text{TF:} \quad true \longrightarrow & d_i := 0|1; & d\_old_i := 0|1; \\
& sn_i := 0|1; & sn\_old_i := 0|1; \\
& d_g := 0|1; & sn_g := 0|1;
\end{array}
$$

Due to the occurrence of transient faults, the masking program may find itself in a state where some non-general process P_i wrongly believes that it has finalized its current round; i.e., $(d_i = -1) \wedge (sn_i \neq sn\_old_i)$ or $(d_i \neq -1) \wedge (sn_i = sn_g)$ holds. Further, P_i may incorrectly believe that it has not yet finalized its current round while the other non-generals have; i.e., $((d_i \neq -1) \wedge (sn_i = sn\_old_i)) \ \wedge \ (((d_j = -1) \wedge (sn_j = sn_g)) \ \vee \ ((d_k = -1) \wedge (sn_k = sn_g)))$. In such states, RB$'$ simply deadlocks. To ensure that the masking program will eventually continue its repetitive rounds of decision making, we add nonmasking fault tolerance to Byzantine and transient faults such that from any arbitrary state, the program recovers to its invariant $roundInv \equiv (Inv_1 \ \wedge \ Inv_2)$, where

$$
\begin{array}{l}
Inv_1 = (\forall i : 1 \leq i \leq 3 : (d_i = -1) \Rightarrow (sn_i = sn\_old_i)) \ \wedge \\
\qquad\quad (\forall i : 1 \leq i \leq 3 : (sn_i = sn_g) \Rightarrow (d_i = -1)) \\
Inv_2 = \forall i : 1 \leq i \leq 3 : (\forall j : (1 \leq j \leq 3) \wedge (i \neq j) : \\
\qquad\qquad ((d_i \neq -1) \wedge (sn_i = sn\_old_i) \wedge (d_j = -1)) \Rightarrow (sn_j \neq sn_g)))
\end{array}
$$

The following new recovery actions are added to the set of program actions, where $(1 \leq i, j, k \leq 3)$ and in action R_{ijk} the condition $(i \neq j) \ \wedge \ (i \neq k) \ \wedge \ (k \neq j)$ holds.

```
$R_{1i}$:   $(d_i \neq -1) \wedge (sn_i = sn_g)$                                    $\longrightarrow$ $d_i := -1$;
$R_{2i}$:   $(d_i = -1) \wedge (sn_i \neq sn\_old_i)$                               $\longrightarrow$ $sn\_old_i := sn_i$;
$R_{ijk}$:  $(d_i \neq -1) \wedge (sn_i = sn\_old_i) \wedge$
                 $(((d_j = -1) \wedge (sn_j = sn_g)) \vee ((d_k = -1) \wedge (sn_k = sn_g)))$
                                                                                    $\longrightarrow$ $sn_i = sn_g$;
                                                                                         $sn\_old_i := sn_g$;
                                                                                         $d_i := -1$
```

The above actions guarantee that from any arbitrary state, the nonmasking-masking multitolerant program RB$''$ recovers to *roundInv*; i.e., RB$''$ is self-stabilizing [11] to Byzantine and transient faults.

7 Conclusions and Future Work

In this paper, we investigated the problem of stepwise design of multitolerant programs from their fault-intolerant versions. A program that is subject to multiple classes of faults, and provides a different level of fault-tolerance to each fault-class is a *multitolerant* program. We considered three levels of fault-tolerance, *failsafe*, *nonmasking* and *masking*. Our contribution in this paper is two-fold: First, for cases where one needs to add failsafe fault-tolerance to one class of faults and nonmasking fault-tolerance to a *different* class of faults, we found a surprising result that this problem is NP-complete (in program state space). Since adding fault-tolerance to a single class of faults is in P [6], this implies that a stepwise method, where the number of steps is constant, for failsafe-nonmasking multitolerance does not exist (unless P=NP). To deal with this NP-completeness result, we identified classes of programs, specifications and faults for which failsafe-nonmasking multitolerance can be designed in a stepwise manner (in polynomial time).

Second, we investigated the feasibility of a stepwise method that is sound and deterministically complete. Such a method is highly desirable, as it is difficult to anticipate all classes of faults while designing fault-tolerant programs, and upon detecting new classes of faults, new levels of fault-tolerance should be added to the design at hand while preserving existing levels of fault-tolerance. We presented such a sound and deterministically complete design method for *special cases* where one adds failsafe (respectively, nonmasking) fault-tolerance to one class of faults and masking fault-tolerance to another class of faults. More importantly, we showed that such an addition is feasible regardless of the order in which different faults are considered. This result has a significant impact for designers in that they can reuse an existing design in future additions of fault-tolerance no matter what classes of faults are currently known!

Our algorithms for adding multitolerance reuse existing algorithms in [6] based on certain properties of them; they do not rely on the implementation of the existing algorithms. Thus, improvements to the implementation of algorithms for adding fault-tolerance to a single class of faults (presented in [6]) can be automatically applied in synthesis of multitolerant protocols. In this regard, we have illustrated that approaches in symbolic synthesis [19] and distributed synthesis algorithms [20] can be effectively used to synthesize programs with reachable states exceeding 2^{100}. We are currently utilizing such techniques for the addition of multitolerance.

References

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

[2] V. Hadzilacos E. Anagnostou. Tolerating transient and permanent failures. *Proceedings of the 7th International Workshop on Distributed Algorithms. Les Diablerets, Switzerland*, pages 174–188, September 1993.

[3] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Proceedings of the Second Workshop on Self-Stabilizing Systems*, page 255, 1995.

[4] S. Tsang and E. Magill. Detecting feature interactions in the intelligent network. *Feature Interactions in Telecommunications Systems II, IOS Press*, 1994. available at http://www.comms.eee.strath.ac.uk/ fi/papers.html.

[5] S. S. Kulkarni and A. Ebnenasir. The effect of the safety specification model on the complexity of adding masking fault-tolerance. *IEEE Transaction on Dependable and Secure Computing*, pages 348–355, October-December 2005.

[6] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 82–93, 2000.

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

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

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

[10] Ali Ebnenasir, Sandeep S. Kulkarni, and Anish Arora. FTSyn: a framework for automatic synthesis of fault-tolerance. *International Journal on Software Tools for Technology Transfer*, 10(5):455–471, 2008.

[11] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[12] Anish Arora and Sandeep S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Transactions on Software Engineering*, 24(6):435–450, 1998.

[13] G. Varghese. *Self-stabilization by local checking and correction.* PhD thesis, MIT/LCS/TR-583, 1993.

[14] Sandeep S. Kulkarni and Ali Ebnenasir. Automated synthesis of multitolerance. *In Proceedings of International Conference on Dependable Systems and Networks (DSN), Palazzo dei Congressi, Florence, Italy*, pages 209–218, July 2004.

[15] S. S. Kulkarni and A. Ebnenasir. Complexity issues in automated synthesis of failsafe fault-tolerance. *IEEE Transactions on Dependable and Secure Computing*, 2(3):201–215, July-September 2005.

[16] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

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

[18] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[19] B. Bonakdarpour and S. S. Kulkarni. Exploiting symbolic techniques in automated synthesis of distributed programs. In *IEEE International Conference on Distributed Computing Systems - ICDCS*, pages 3–11, 2007.

[20] Ali Ebnenasir. Diconic addition of failsafe fault-tolerance. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE)*, pages 44–53, 2007.

[21] R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 56 – 69, 1998.

8 Appendix A: Synthesis Algorithms For Adding Fault-Tolerance to One Class of Faults

As mentioned in Section 5.1, the knowledge of the implementation of Add_Failsafe, Add_Nonmasking, and Add_Masking is not required for the understanding of the algorithms presented in this paper. Our results only use the properties in Section 5.1. However, for the convenience of the interested reviewer, we recall the implementation of these algorithms, next.

The algorithm Add_Failsafe takes a fault-intolerant program p, its invariant S, its safety specification *spec*, and a class of faults f. The output of the algorithm is a failsafe f-tolerant program p' with its invariant S'.

```
Add_Failsafe($p, f$ : transitions, $S$ : state predicate, $spec$ : specification)
{
    $ms := \{s_0 : \exists s_1, s_2, ...s_n :$
                $(\forall j : 0 \leq j < n : (s_j, s_{(j+1)}) \in f) \quad \wedge \quad (s_{(n-1)}, s_n)$ violates $spec$ };
    $T'$:= true$-ms$;
    $mt := \{(s_0, s_1) : ((s_1 \in ms) \vee (s_0, s_1)$ violates $spec)$ };
    $S'$ := ConstructInvariant($S - ms, p - mt$);
    if ($S' = \{\}$)  declare no failsafe $f$-tolerant program $p'$;
                return $\emptyset, \emptyset$;
    else $p'$ :=ConstructTransitions($p - mt, S'$); // One can add any subset of transitions in
            // $((T' - S') \times T') - mt$ to $p'$ without violating the properties V-A.1 and V-A.2
    return $p', S', T'$;
}

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

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

Figure 12: The synthesis algorithm for adding failsafe fault-tolerance.

The routine ConstructInvariant calculates the largest subset of the state predicate S where the computations of program p are infinite. Also, ConstructTransition routine removes transitions of its first parameter, p, that violate the closure of its second parameter, state predicate S.

```
Add_Nonmasking($p, f$ : transitions, $S$ : state predicate, $spec$ : specification)
{
    $S' := S$;
    $T' := \{s : s$ is reachable from $S$ by computations of $p[]f\}$;
    $p' := (p|S) \cup \{(s_0, s_1) : s_0 \notin S \wedge s_1 \in S\}$
    return $p', S', T'$;
}
```

Figure 13: The synthesis algorithm for adding nonmasking fault-tolerance.

The algorithm Add_Nonmasking adds nonmasking fault-tolerance to a fault-intolerant program p.

The algorithm Add_Masking takes a fault-intolerant program p, its invariant S, its safety specification *spec*, and a class of faults f, and synthesizes a masking f-tolerant program p' with its invariant S'. The routine ConstructFaultSpan calculates the largest subset of its first parameter, state predicate T, that is closed in faults f.

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

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

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

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

Figure 14: The synthesis algorithm for adding masking fault-tolerance.

9 Appendix B: Non-Deterministic Addition of Multitolerance

In this section, we first identify the difficulties of adding multitolerance to three classes of faults f_f, f_n, and f_m, where $f_m = f_f \cap f_n$, failsafe fault-tolerance is required to f_f, nonmasking fault-tolerance is required to f_n and masking fault-tolerance is required to f_m. Then, we present a non-deterministic solution (see Figure 15) for adding multitolerance to fault-intolerant programs.

For a program p that is subject to three classes of faults f_f, f_n, and f_m, consider the cases where there exists a state s such that (i) s is reachable in the computations of $p[]f_f$ from invariant, (ii) s is reachable in the computations of $p[]f_n$ from invariant, and (iii) no safe recovery (i.e., recovery during which safety is preserved) is possible from s to the invariant.

In such cases, we have the following options: (1) ensure that s is unreachable in the computations of $p[]f_f$ and add a recovery transition (that violates safety) from s to the invariant, or (2) ensure that s is unreachable in the computations of $p[]f_n$ 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_f, f_n, and f_m, 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_f, T_n, and T_m. Then, we verify a set of conditions that ensure the multitolerance property of p'. We have shown our algorithm in Figure 15.

Theorem B.1 The algorithm Add_Multitolerance is sound and complete.

Proof. Since this algorithm simply verifies the conditions needed for multitolerance, the proof is straightforward. □

Theorem B.2 The problem of synthesizing multitolerant programs from their fault-intolerant versions is in NP. □

```
Add_Multitolerance ($p$: transitions, $f_f, f_n, f_m$: fault, $S$: state predicate,
                                                        $spec$: safety specification)
{
  $ms := \{s_0 : \exists s_1, s_2, ...s_n : (\forall j : 0 \leq j < n : (s_j, s_{(j+1)}) \in (f_f \cup f_m)) \wedge$
                                   $(s_{(n-1)}, s_n)$ violates $spec$ };      (1)
  $mt := \{(s_0, s_1) : ((s_1 \in ms) \vee (s_0, s_1)$ violates $spec)$ };      (2)

  Guess $p', S', T_f, T_n, T_m$;      (3)
  Verify the following conditions:
      $S' \subseteq S$; $S' \neq \{\}$; $S' \subseteq T_f$; $S' \subseteq T_n$; $S' \subseteq T_m$;      (4)
      $(\forall s_0 : s_0 \in S' : (\exists s_1 :: (s_0, s_1) \in p'))$;      (5)
      $p'|S' \subseteq p|S'$; $S'$ is closed in $p'$;      (6)

      $T_m$ is closed in $p'[]f_m$;      (7)
      $T_m \cap ms = \emptyset$; $(p'|T_m) \cap mt = \emptyset$;      (8)
      $(\forall s_0 : s_0 \in T_m : (\exists s_1 :: (s_0, s_1) \in p'))$; $(p'|(T_m - S'))$ is acyclic;      (9)

      $T_f$ is closed in $p'[]f_f$;      (10)
      $T_f \cap ms = \emptyset$; $(p'|T_f) \cap mt = \emptyset$;      (11)

      $T_n$ is closed in $p'[]f_n$;      (12)
      $(\forall s_0 : s_0 \in T_n : (\exists s_1 :: (s_0, s_1) \in p'))$; $(p'|(T_n - S'))$ is acyclic;      (13)
}
```

Figure 15: A non-deterministic polynomial algorithm for synthesizing multitolerance.

10 Appendix C: The Promela Models of the Multitolerant Two-Ring Token Passing (TRTP)

In this section, we present the Promela models of the failsafe, failsafe-masking and failsafe-nonmasking-masking multitolerant TRTP programs presented in Section 6.1. The following code represents the failsafe program.

```
short a0 ;
short a1 ;
short a2 ;
short a3 ;

bool up_a0 = true;
bool up_a1 = true;
bool up_a2 = true;
bool up_a3 = true;

short b0 ;
short b1 ;
short b2 ;
short b3 ;

bool turn;

short count = 30;

bool up_b0 = true;
bool up_b1 = true;
bool up_b2 = true;
bool up_b3 = true;

bool ringAperturbed = false;
bool ringBperturbed = false;

#define N 4  /* number of states in each process */

#define allEqualA ((a0 == a1) && (a1 == a2) && (a2 == a3) && (a0 != -1) &&
                    up_a0 && up_a1 && up_a2 && up_a3)

#define allEqualB ((b0 == b1) && (b1 == b2) && (b2 == b3) && (b0 != -1) &&
                    up_b0 && up_b1 && up_b2 && up_b3)

#define workA  ((((a1 == a2) && (a2 == a3) && (a0 == ((a1+1) % N))  ) ||
                   ((a0 == a1) && (a2 == a3) && (a1 == ((a2+1)%N))  ) ||
                                   ((a0 == a1) && (a1 == a2) && ( ((a3+1) % N) == a2)  ) ) &&
                                   (up_a0 && up_a1 && up_a2 && up_a3) &&
                                   (a0 != -1) && (a1 != -1)&& (a2 != -1) && (a3 != -1))

#define workB  ((((b1 == b2) && (b2 == b3) && (b0 == ((b1+1) % N))  ) ||
                    ((b0 == b1) && (b2 == b3) && (b1 == ((b2+1)%N)) ) ||
                                        ((b0 == b1) && (b1 == b2) && ( ((b3+1) % N) == b2)  ) ) &&
                                        (up_b0 && up_b1 && up_b2 && up_b3) &&
                                        (b0 != -1) && (b1 != -1)&& (b2 != -1) && (b3 != -1))


#define tokenA0        ((a0 == a3) && (b0 == b3) && (a0 == b0) &&  up_a0  && (a0 != -1) && (a3 != -1))
#define tokenA1        ((a0 == ((a1+1)%N)) && up_a1 &&  (a0 != -1) && (a1 != -1))
#define tokenA2        ((a1 == ((a2+1)%N)) && up_a2 &&  (a1 != -1) && (a2 != -1))
#define tokenA3        ((a2 == ((a3+1)%N)) && up_a3 &&  (a2 != -1) && (a3 != -1))

#define tokenB0        ((b0 == b3) && (a0 == a3) && (((b0 + 1) %N) == a0)  && up_b0  && (b0 != -1) && (b3 != -1))
#define tokenB1        ((b0 == ((b1+1)%N)) && up_b1   && (b0 != -1) && (b1 != -1))
#define tokenB2        ((b1 == ((b2+1)%N)) && up_b2   && (b1 != -1) && (b2 != -1))
#define tokenB3        ((b2 == ((b3+1)%N)) && up_b3  && (b2 != -1) && (b3 != -1))

#define atmostOneTokenInRingA  ((tokenA0 && !tokenA1 && !tokenA2 && !tokenA3) ||
                                (!tokenA0 && tokenA1 && !tokenA2 && !tokenA3) ||
                                (!tokenA0 && !tokenA1 && tokenA2 && !tokenA3) ||
                                (!tokenA0 && !tokenA1 && !tokenA2 && tokenA3) ||
                                (!tokenA0 && !tokenA1 && !tokenA2 && !tokenA3))

#define atmostOneTokenInRingB  ((tokenB0 && !tokenB1 && !tokenB2 && !tokenB3) ||
                                (!tokenB0 && tokenB1 && !tokenB2 && !tokenB3) ||
                                (!tokenB0 && !tokenB1 && tokenB2 && !tokenB3) ||
                                (!tokenB0 && !tokenB1 && !tokenB2 && tokenB3) ||
                                (!tokenB0 && !tokenB1 && !tokenB2 && !tokenB3))

#define atMostOneToken ((!tokenB0 && !tokenB1 && !tokenB2 && !tokenB3) &&
                          ((tokenA0 && !tokenA1 && !tokenA2 && !tokenA3) ||
                           (!tokenA0 && tokenA1 && !tokenA2 && !tokenA3) ||
                           (!tokenA0 && !tokenA1 && tokenA2 && !tokenA3) ||
                           (!tokenA0 && !tokenA1 && !tokenA2 && tokenA3) ||
                           (!tokenA0 && !tokenA1 && !tokenA2 && !tokenA3)) ) ||
                        ((!tokenA0 && !tokenA1 && !tokenA2 && !tokenA3) &&
                           ((tokenB0 && !tokenB1 && !tokenB2 && !tokenB3) ||
                            (!tokenB0 && tokenB1 && !tokenB2 && !tokenB3) ||
                            (!tokenB0 && !tokenB1 && tokenB2 && !tokenB3) ||
                            (!tokenB0 && !tokenB1 && !tokenB2 && tokenB3) ||
                            (!tokenB0 && !tokenB1 && !tokenB2 && !tokenB3)) )

#define inv  ((allEqualA || (workA && turn)) && (allEqualB || (workB && !turn)))

#define ringAup                (up_a0 && up_a1 && up_a2 && up_a3)
#define ringBup                (up_b0 && up_b1 && up_b2 && up_b3)

#define ringAcorrupted  ((a0 == -1) || (a1 == -1) || (a2 == -1) || (a3 == -1) )
#define ringBcorrupted  ((b0 == -1) || (b1 == -1) || (b2 == -1) || (b3 == -1) )
```

```
/* Properties to check in the absence of faults */
/* [] inv */
/* [] (atMostOneToken)  */
/* [] (!workA || !workB)  if one works, the other does not.  */
/* [] (allEqualA -> <> workA) */
/* [] (allEqualB -> <> workB) */
/* [] (workA -> <> allEqualA) */
/* [] (workB -> <> allEqualB) */
/* [] (workA -> <> workB) */
/* [] (workB -> <> workA) */

/* Properties to check in the presence of failsafe and failsafe-masking faults */
/*  []  atMostOneToken */
/* [] (!ringAup -> <> workB) */
/* [] (!ringBup -> <> workA) */
/* [] (!ringAup -> <> allEqualB) */
/* [] (!ringBup -> <> allEqualA) */

/* Properties to check in the presence of masking faults */
/* []  atMostOneToken && (<>  []  inv) */

/* Properties to check in the presence of non-masking faults */
/* <> [] inv */

proctype process_a0 ()
{
do
::atomic{
        (a0 == a3) && up_a0 && turn  && (a3 != -1)
        ->  if
                ::((a0 == b0) && (b0 != -1)) || !up_b0 -> a0 = (a3+1) % N;
                :: else if
                                            :: up_b0  -> turn = false;
                            :: else skip;
                        fi;
            fi;
}

::atomic{ !up_a3 && up_a0 -> up_a0 = false; }

::atomic{ (!up_b0 && !up_b1 && !up_b2 && !up_b3) && up_a0 && !turn -> turn = true; }
od
}

proctype process_a1 ()
{
do
::atomic{
        (a0 == ((a1+1)%N))  && up_a1 && (a0 != -1) -> a1 = a0; }

::atomic{ !up_a0 && up_a1 -> up_a1 = false; }
od
}

proctype process_a2 ()
{
do
::atomic{
       (a1 == ((a2+1)%N)) && up_a2 && (a1 != -1)  -> a2 = a1;  }

::atomic{ !up_a1 && up_a2 -> up_a2 = false; }
od
}

proctype process_a3 ()
{
do
```

```
::atomic{

        (a2 == ((a3+1)%N)) &&  up_a3  && (a2 != -1) -> a3 = a2;  }

::atomic{ !up_a2 && up_a3 -> up_a3 = false; }
od
}


/*  Processes of ring B */
proctype process_b0 ()
{
do

::atomic{
        (b0 == b3) && up_b0 && !turn  && (b3 != -1)
                -> if
                    :: ((a0 != b0) && (a0 != -1)) || !up_a0  -> b0 = (b3+1) % N;
                    :: else if :: up_a0 -> turn = true;
                                                    :: else skip;
                            fi;
                   fi
                        }
::atomic{ !up_b3 && up_b0 -> up_b0 = false; }
::atomic{ (!up_a0 && !up_a1 && !up_a2 && !up_a3) && up_b0 && turn -> turn = false; }
od
}

proctype process_b1 ()
{
do
::atomic{
        (b0 == ((b1+1)%N))  && up_b1 && (b0 != -1) -> b1 = b0;   }

::atomic{ !up_b0 && up_b1 -> up_b1 = false;   }
od
}

proctype process_b2 ()
{
do
::atomic{
        (b1 == ((b2+1)%N)) && up_b2 && (b1 != -1) -> b2 = b1;   }

::atomic{ !up_b1 && up_b2 -> up_b2 = false;}
od
}

proctype process_b3 ()
{
do
::atomic{
        (b2 == ((b3+1)%N)) && up_b3  && (b2 != -1) -> b3 = b2;   }

::atomic{ !up_b2 && up_b3 -> up_b3 = false;}
od
}

proctype failsafe_fault() {
if
    :: atomic{ up_a0 && up_b0 && up_b1 && up_b2 && up_b3 && !ringBcorrupted -> up_a0 =
false;}
    :: atomic{ up_a1 && up_b0 && up_b1 && up_b2 && up_b3 && !ringBcorrupted -> up_a1 =
false;}
    :: atomic{ up_a2 && up_b0 && up_b1 && up_b2 && up_b3 && !ringBcorrupted -> up_a2 =
false;}
    :: atomic{ up_a3 && up_b0 && up_b1 && up_b2 && up_b3 && !ringBcorrupted -> up_a3 =
false;}
    :: atomic{ up_b0 && up_a0 && up_a1 && up_a2 && up_a3 && !ringAcorrupted  -> up_b0
= false;}
    :: atomic{ up_b1 && up_a0 && up_a1 && up_a2 && up_a3  && !ringAcorrupted-> up_b1 =
false;}
    :: atomic{ up_b2 && up_a0 && up_a1 && up_a2 && up_a3  && !ringAcorrupted-> up_b2 =
false;}
    :: atomic{ up_b3 && up_a0 && up_a1 && up_a2 && up_a3  && !ringAcorrupted-> up_b3 =
false;}
fi;
}

proctype masking_fault() {
if
    :: atomic{ (a0 != -1) && (b0 != -1) && (b1 != -1) && (b2 != -1) && (b3 != -1) &&
ringBup ->  a0 = -1;}
    :: atomic{ (a1 != -1) && (b0 != -1) && (b1 != -1) && (b2 != -1) && (b3 != -1)  &&
ringBup ->  a1 = -1;}
    :: atomic{ (a2 != -1) && (b0 != -1) && (b1 != -1) && (b2 != -1) && (b3 != -1)  &&
ringBup ->  a2 = -1;}
    :: atomic{ (a3 != -1) && (b0 != -1) && (b1 != -1) && (b2 != -1) && (b3 != -1)  &&
ringBup ->  a3 = -1;}
    :: atomic{ (b0 != -1) && (a0 != -1) && (a1 != -1) && (a2 != -1) && (a3 != -1)  &&
ringAup  ->  b0 = -1;}
    :: atomic{ (b1 != -1) && (a0 != -1) && (a1 != -1) && (a2 != -1) && (a3 != -1)  &&
ringAup  ->  b1 = -1;}
    ::atomic{  (b2 != -1) && (a0 != -1) && (a1 != -1) && (a2 != -1) && (a3 != -1) &&
ringAup ->  b2 = -1;}
    :: atomic{ (b3 != -1) && (a0 != -1) && (a1 != -1) && (a2 != -1) && (a3 != -1) &&
ringAup ->  b3 = -1;}
fi;
}

proctype nonmasking_fault() {
do
::atomic{
count != 0 -> count-- ;
              if
              ::true -> a0 = 0;
              ::true -> a0 = 1;
              ::true -> a0 = 2;
              ::true -> a0 = 3;

              ::true -> a1 = 0;
              ::true -> a1 = 1;
              ::true -> a1 = 2;
              ::true -> a1 = 3;

              ::true -> a2 = 0;
              ::true -> a2 = 1;
              ::true -> a2 = 2;
              ::true -> a2 = 3;

              ::true -> a3 = 0;
              ::true -> a3 = 1;
              ::true -> a3 = 2;
              ::true -> a3 = 3;
              fi
}

::atomic{
count != 0 -> count-- ;
              if
              ::true -> b0 = 0;
              ::true -> b0 = 1;
              ::true -> b0 = 2;
              ::true -> b0 = 3;
```

```

               ::true -> b1 = 0;
               ::true -> b1 = 1;
               ::true -> b1 = 2;
               ::true -> b1 = 3;

               ::true -> b2 = 0;
               ::true -> b2 = 1;
               ::true -> b2 = 2;
               ::true -> b2 = 3;

               ::true -> b3 = 0;
               ::true -> b3 = 1;
               ::true -> b3 = 2;
               ::true -> b3 = 3;
               fi
}
od
}

init {
run process_a0();
run process_a1();
run process_a2();
run process_a3();

run process_b0();
run process_b1();
run process_b2();
run process_b3();

run failsafe_fault();
run masking_fault();
run nonmasking_fault()
}
```

The processes of the failsafe-masking program are as follows:

```
proctype process_a0 ()
{
do
::atomic{
        ((a0 == a3) ||  (a0 == -1)) && up_a0 && turn  && (a3 != -1)
        ->  if
                ::((a0 == b0) && (b0 != -1)) ||  (a0 == -1) || !up_b0 -> a0 = (a3+1) % N;
                :: else if
                                             :: up_b0  -> turn = false;
                            :: else skip;
                        fi;
            fi;
}

::atomic{ !up_a3 && up_a0 -> up_a0 = false; }

::atomic{ (!up_b0 && !up_b1 && !up_b2 && !up_b3) && up_a0 && !turn -> turn = true; }
od
}

proctype process_a1 ()
{
do
::atomic{ (a0 == ((a1+1)%N))  && up_a1 && (a0 != -1) -> a1 = a0; }

::atomic{ !up_a0 && up_a1 -> up_a1 = false; }
od
}

proctype process_a2 ()
{
```

```
do
::atomic{ (a1 == ((a2+1)%N)) && up_a2 && (a1 != -1)  -> a2 = a1;  }

::atomic{ !up_a1 && up_a2 -> up_a2 = false; }
od
}

proctype process_a3 ()
{
do
::atomic{ (a2 == ((a3+1)%N)) &&  up_a3  && (a2 != -1) -> a3 = a2;  }

::atomic{ !up_a2 && up_a3 -> up_a3 = false; }
od
}


/*  Processes of ring B */
proctype process_b0 ()
{
do

::atomic{
        ((b0 == b3) ||  (a0 == -1)) && up_b0 && !turn  && (b3 != -1)
                -> if
                    :: ((a0 != b0) && (a0 != -1)) ||  (a0 == -1) || !up_a0  -> b0 = (b3+1) % N;
                    :: else if :: up_a0 -> turn = true;
                                            :: else skip;
                            fi;
                   fi
                        }
::atomic{ !up_b3 && up_b0 -> up_b0 = false; }
::atomic{ (!up_a0 && !up_a1 && !up_a2 && !up_a3) && up_b0 && turn -> turn = false; }
od
}

proctype process_b1 ()
{
do
::atomic{ (b0 == ((b1+1)%N))  && up_b1 && (b0 != -1) -> b1 = b0;   }

::atomic{ !up_b0 && up_b1 -> up_b1 = false;   }
od
}

proctype process_b2 ()
{
do
::atomic{ (b1 == ((b2+1)%N)) && up_b2 && (b1 != -1) -> b2 = b1;   }

::atomic{ !up_b1 && up_b2 -> up_b2 = false;}
od
}

proctype process_b3 ()
{
do
::atomic{ (b2 == ((b3+1)%N)) && up_b3  && (b2 != -1) -> b3 = b2;   }

::atomic{ !up_b2 && up_b3 -> up_b3 = false;}
od
}
```

The processes of the failsafe-nonmasking-masking program are as follows:

```
proctype process_a0 ()
{
do
```

```
::atomic{
        ((a0 == a3) || (a0 == -1)) && up_a0 && turn && (a3 != -1)
        ->  if
                ::((a0 == b0) && (b0 != -1)) || (a0 == -1) || !up_b0 -> a0 = (a3+1) % N;
                :: else if
                                            :: up_b0  -> turn = false;
                            :: else skip;
                        fi;
            fi;
}

::atomic{ !up_a3 && up_a0 -> up_a0 = false; }

::atomic{ (!up_b0 && !up_b1 && !up_b2 && !up_b3) && up_a0 && !turn -> turn = true; }
od
}

proctype process_a1 ()
{
do
::atomic{  ( (a0 == ((a1+1)%N))   || (a1 == -1) ) && up_a1  && (a0 != -1)
                                               -> a1 = a0;  }

::atomic{  (a0 != ((a1+1)%N)) && (a0 != a1) && up_a1 && (a0 != -1)
                                                -> a1 = a0;  }
::atomic{ !up_a0 && up_a1 -> up_a1 = false; }
od
}

proctype process_a2 ()
{
do
::atomic{
      ( (a1 == ((a2+1)%N))  || (a2 == -1)) && up_a2 && (a1 != -1) -> a2 = a1;  }

::atomic{
        (a1 != ((a2+1)%N)) && (a2 != a1) && up_a2 && (a1 != -1)
                                                -> a2 = a1;  }
::atomic{ !up_a1 && up_a2 -> up_a2 = false; }
od
}

proctype process_a3 ()
{
do
::atomic{ ((a2 == ((a3+1)%N))  || (a3 == -1)) &&  up_a3  && (a2 != -1)
                                                        -> a3 = a2;  }

::atomic{ (a2 != ((a3+1)%N)) && (a3 != a2) && up_a3 && (a2 != -1)
                                                -> a3 = a2;  }
::atomic{ !up_a2 && up_a3 -> up_a3 = false; }
od
}


/*  Processes of ring B */
proctype process_b0 ()
{
do

::atomic{
        ((b0 == b3)  || (b0 == -1))  && up_b0 && !turn && (b3 != -1)
                -> if
                    :: ((a0 != b0) && (a0 != -1)) || (b0 == -1) || !up_a0  -> b0 = (b3+1) % N;
                    :: else if :: up_a0 -> turn = true;
                                                   :: else skip;
                            fi;
                    fi
                         }
::atomic{ !up_b3 && up_b0 -> up_b0 = false; }
::atomic{ (!up_a0 && !up_a1 && !up_a2 && !up_a3) && up_b0 && turn -> turn = false; }
od
}

proctype process_b1 ()
{
do
::atomic{ ((b0 == ((b1+1)%N)) || (b1 == -1))  && up_b1 && (b0 != -1)
                                        -> b1 = b0;   }

::atomic{ (b0 != ((b1+1)%N)) && (b0 != b1) && up_b1 && (b0 != -1)
                                        -> b1 = b0;   }
::atomic{ !up_b0 && up_b1 -> up_b1 = false;   }
od
}

proctype process_b2 ()
{
do
::atomic{ ((b1 == ((b2+1)%N)) || (b2 == -1)) && up_b2  && (b1 != -1)
                                        -> b2 = b1;   }

::atomic{ (b1 != ((b2+1)%N)) && (b1 != b2) && up_b2 && (b1 != -1)
                                        -> b2 = b1;   }
::atomic{ !up_b1 && up_b2 -> up_b2 = false;}
od
}

proctype process_b3 ()
{
do
::atomic{  ((b2 == ((b3+1)%N)) || (b3 == -1))  && up_b3 && (b2 != -1)
                                        -> b3 = b2;   }

::atomic{ (b2 != ((b3+1)%N)) && (b2 != b3) && up_b3 && (b2 != -1)
                                        -> b3 = b2;  }
::atomic{ !up_b2 && up_b3 -> up_b3 = false;}
od
}
```

11 Appendix D: The Promela Model of the Multitolerant Repetitive Byzantine Agreement

In this section, we present the Promela model of the nonmasking-masking multitolerant version of the repetitive Byzantine agreement program presented in Section 6.2. The masking program includes all processes except the stabilize() process that includes necessary actions for recovery in the presence of transient faults.

```
short dg;
short d1 = -1 ;
short d2 = -1;
short d3 = -1;

short d1_new ;
short d2_new  ;
short d3_new ;

bool sng ;
bool sn1 ;
bool sn2 ;
bool sn3 ;

bool sn1_new ;
bool sn2_new ;
bool sn3_new ;

bool bg ;
bool b1 ;
bool b2 ;
bool b3 ;

int count = 10;

#define equalSnSg ((sn1 == sn2) && (sn2 == sn3) && (sn1 == sng))

#define equalSn ((sn1 == sn2) && (sn2 == sn3))

#define equalSn_new ((sn1_new == sn2_new) && (sn2_new == sn3_new))

#define allBot        ((d1 == d2) && (d2 == d3) && (d1 == -1))

#define allNotBot        ((d1 != -1) && (d2 != -1) && (d3 != -1))


#define initRound  (equalSN && allBot)

#define allAgree ((d1 == d2) && (d2 == d3) && (d1 != -1))

#define faultInv ((!bg && !b1 && !b2 && !b3) || (bg && !b1 && !b2 && !b3) ||
                  (!bg && b1 && !b2 && !b3) || (!bg && !b1 && b2 && !b3) ||
                                (!bg && !b1 && !b2 && b3))

#define validityIP ((!equalSn || (sn1 == sng) || bg) ||
                    ((b1 || (d1 == -1) || (d1 == dg)) &&
                                        (b2 || (d2 == -1)  || (d2 == dg)) &&
                                        (b3 || (d3 == -1)  || (d3 == dg)) ) )

#define agreementIP ((!equalSn || (sn1 == sng) || !bg ) ||
                     (d1 == -1) || (d2 == -1) || (d3 == -1) ||
                                         ((d1 == d2) && (d2 == d3)) )

#define validityTP ((!equalSn_new || (sn1_new != sng) || bg) ||
                    ((b1 || (d1 == -1)  || (d1_new == dg)) &&
                                         (b2 || (d2 == -1)  || (d2_new == dg)) &&
                                         (b3 || (d3 == -1)  || (d3_new == dg)) ) )

#define agreementTP ((!equalSnSg || (d1 != -1) || (d2 != -1) || (d3 != -1) || !bg) ||
                     ((d1_new == d2_new) && (d2_new == d3_new)) )

#define inv1 (((d1 != -1) || (sn1 == sn1_new)) &&
              ((sn1 != sng) || (d1 == -1)) && ((d2 != -1) || (sn2 == sn2_new)) &&
                          ((sn2 != sng) || (d2 == -1)) && ((d3 != -1) || (sn3 == sn3_new)) &&
                          ((sn3 != sng) || (d3 == -1)) )

#define inv2 (((d1 == -1) || (sn1 != sn1_new) || (d2 != -1) || (sn2 != sng)) &&
              ((d1 == -1) || (sn1 != sn1_new) || (d3 != -1) || (sn3 != sng)) &&
                          ((d2 == -1) || (sn2 != sn2_new) || (d1 != -1) || (sn1 != sng)) &&
                          ((d2 == -1) || (sn2 != sn2_new) || (d3 != -1) || (sn3 != sng)) &&
                          ((d3 == -1) || (sn3 != sn3_new) || (d1 != -1) || (sn1 != sng)) &&
                          ((d3 == -1) || (sn3 != sn3_new) || (d2 != -1) || (sn2 != sng)) )

#define roundInv inv1 && inv2

/* Properties to check in the absence of faults */
/* []<> (equalSnSg && allBot)  */
/* [] validityIP */
/* [] agreementIP */

/* Properties to check in the presence of Byzantine faults */
/* []<> equalSnSg  */
/* [] validityTP */
/* [] agreementTP */

/* Properties to check in the presence of non-masking faults */
/* <> [] roundInv */
/* [] (inv1 -> <>equalSnSg) */
/* [] (roundInv -> <>(allBot&& equalSnSg)) */
/* [] (inv1 -> <> [] validityTP) */
/* [] (inv1 -> <> [] agreementTP) */

proctype process_g ()
{
do
::atomic{
        equalSnSg  -> sng = !sng; bg = false; b1 = false; b2 = false; b3 = false;
                                        if
                                                :: true -> dg = 0;
                                                :: true -> dg = 1;
                                        fi
}
od
}

proctype process_1 ()
{
do
::atomic{    (d1 == -1) && (sn1 != sng)  -> d1 = dg; }

:: atomic{ (d1 != -1) && (sn1_new == sn1) && equalSn && allNotBot && (sn1 != sng)
                                    -> d1_new = d1;
                                       sn1_new = !sn1_new;  }

::atomic{ (d1 != -1) && ((sn1_new == sng) && (sn2_new == sng)  && (sn3_new == sng) ) &&
                         ((d1 != d2_new) && (d2_new == d3_new)) && (sn1 != sng)
                                    -> d1 = d2_new;
                                       d1_new = d2_new;  }

::atomic{ (d1 != -1) && ((sn1_new == sng) && (sn2_new == sng)  && (sn3_new == sng) ) &&
                        ((d1 == d2_new) || (d1 == d3_new)) && (sn1 != sng)
                                    -> d1 = -1;
                                                            sn1 = !sn1;  }
od
}
```

```
proctype process_2 ()
{
do

::atomic{ (d2 == -1) && (sn2 != sng)  -> d2 = dg;   }

:: atomic{ (d2 != -1) && (sn2_new == sn2) && equalSn && allNotBot && (sn2 != sng)
                                    -> d2_new = d2;
                                       sn2_new = !sn2_new;  }

::atomic{ (d2 != -1) && ((sn1_new == sng) && (sn2_new == sng)  && (sn3_new == sng)) &&
                         ((d2 != d1_new) && (d1_new == d3_new)) && (sn2 != sng)
                                    -> d2 = d1_new;
                                                                           d2_new = d1_new;  }

::atomic{ (d2 != -1) && ((sn1_new == sng) && (sn2_new == sng)  && (sn3_new == sng)) &&
                         ((d2 == d1_new) || (d2 == d3_new)) && (sn2 != sng)
                                    -> d2 = -1;
                                                                           sn2 = !sn2;  }
od
}

proctype process_3 ()
{
do
::atomic{  (d3 == -1) && (sn3 != sng)  -> d3 = dg; }

:: atomic{ (d3 != -1) && (sn3_new == sn3) && equalSn && allNotBot && (sn3 != sng)
                                    -> d3_new = d3;
                                       sn3_new = !sn3_new;  }

::atomic{ (d3 != -1) && ((sn1_new == sng) && (sn2_new == sng)  && (sn3_new == sng)) &&
                         ((d3 != d2_new) && (d2_new == d1_new)) && (sn3 != sng)
                                    -> d3 = d2_new;
                                                                           d3_new = d2_new; }

::atomic{ (d3 != -1) && ((sn1_new == sng) && (sn2_new == sng)  && (sn3_new == sng)) &&
                        ((d3 == d2_new) || (d3 == d1_new)) && (sn3 != sng)
                                    -> d3 = -1;
                                                                           sn3 = !sn3;  }
od
}

proctype stabilize(){

do
:: atomic{(d1 == -1) && (sn1 != sn1_new) -> sn1_new = sn1 ; }
:: atomic{(d1 != -1) && (sn1 == sng) -> d1 = -1; }

:: atomic{(d2 == -1) && (sn2 != sn2_new) -> sn2_new = sn2; }
:: atomic{(d2 != -1) && (sn2 == sng) -> d2 = -1; }

:: atomic{(d3 == -1) && (sn3 != sn3_new) -> sn3_new =sn3; }
:: atomic{(d3 != -1) && (sn3 == sng) -> d3 = -1;  }

:: atomic{ (d1 != -1) && (sn1 == sn1_new) &&
            (((d2 == -1) && (sn2 == sng) ) || ((d3 == -1) && (sn3 == sng)) )
                        -> sn1 = sng ; sn1_new = sng ; d1 = -1; }

:: atomic{ (d2 != -1) && (sn2 == sn2_new) &&
            (((d1 == -1) && (sn1 == sng) ) || ((d3 == -1) && (sn3 == sng)) )
                        -> sn2 = sng ; sn2_new = sng; d2 = -1; }

:: atomic{ (d3 != -1) && (sn3 == sn3_new) &&
            (((d1 == -1) && (sn1 == sng) ) || ( (d2 == -1) && (sn2 == sng)) )
                    -> sn3 = sng ; sn3_new = sng; d3 = -1; }
od
```

```
}

proctype process_ByzFaults ()
{
do
::atomic{ (sn1 == sn2) && (sn3 == sn2) && (sng != sn1) && (!bg && !b1 && !b2 && !b3)  ->

if
                                        :: true -> bg = true;
                                                        if :: true -> dg = 0;
                                                           :: true -> dg = 1;
                                                        fi;
                                        :: true -> b1 = true;
                                                        if :: true -> d1 = 0;
                                                           :: true -> d1 = 1;
                                                        fi;
                                        :: true -> b2 = true;
                                                        if :: true -> d2 = 0;
                                                           :: true -> d2 = 1;
                                                        fi;
                                        :: true -> b3 = true;
                                                        if :: true -> d3 = 0;
                                                           :: true -> d3 = 1;
                                                        fi;

fi
}
od
}

proctype process_transFaults ()
{
do
::atomic{
count != 0 -> atomic{ count-- ;
        if
                :: true -> sn1 = 0;
                :: true -> sn1 = 1;

                :: true -> sn2 = 0;
                :: true -> sn2 = 1;

                :: true -> sn3 = 0;
                :: true -> sn3 = 1;

                :: true -> d1 = 0;
                :: true -> d1 = 1;

                :: true -> d2 = 0;
                :: true -> d2 = 1;

                :: true -> d3 = 0;
                :: true -> d3 = 1;

                :: true -> sn1_new = 0;
                :: true -> sn1_new = 1;

                :: true -> sn2_new = 0;
                :: true -> sn2_new = 1;

                :: true -> sn3_new = 0;
                :: true -> sn3_new = 1;

                :: true -> d1_new = 0;
                :: true -> d1_new = 1;

                :: true -> d2_new = 0;
                :: true -> d2_new = 1;

                :: true -> d3_new = 0;
                :: true -> d3_new = 1;

                :: true -> dg = 0;
                :: true -> dg = 1;

                :: true -> sng = 0;
                :: true -> sng = 1;

        fi;
}
}
od
}

init {
run process_g();
run process_1();
run process_2();
run process_3();

run stabilize();

run process_ByzFaults ();
run process_transFaults ()
}
```

12 Appendix E: Multitolerant Cruise Control Program

In this section, we present an example of adding multitolerance to existing programs. Specifically, we add failsafe-masking multitolerance to the controlling software of a cruise control system. The cruise control example in this section is a simplified version of the example in [21]. First, we introduce the Cruise Control (CC) program, its specification, its invariant, and the classes of faults that perturb the CC program. Then, we illustrate how we use Add_Failsafe_Masking algorithm (see Figure 7) to add multitolerance to the CC program.

The fault-intolerant CC program. The CC program has 4 input variables *Ignition, EngState, Brake*, and *Lever* that represent the values of the input signals. (To distinguish variables from their values, system variables start with capitalized letters.) The domain of the variable *Ignition* contains values of *on* and *off* that represent the state of the ignition switch. The variable *EngState* represents the working state of the engine with the domain { *running, off* }. The variable *Brake* illustrates the state of an input signal from the brakes that shows whether or not the driver has applied the brakes. The domain of *Brake* is equal to { *notApplied, applied, unknown* }. The *Lever* variable represents the position of the cruise control lever set by the driver. The cruise control lever can be in three positions *off, constant*, and *resume*. Also, the CC program has a variable *SysMode* that stores its operating mode. The CC program can be in the following modes: *off, inactive, cruise*, and *override*. If the CC program is in none of the above-mentioned modes then its mode is *unknown*. We represent the fault-intolerant CC program by the actions A_1-A_9.

If the program is in *off* mode and the ignition is on then the program transitions to the *inactive* mode (Action A_1). The program transitions to the *off* mode if it is in the *inactive* mode and the ignition turns off (Action A_2). In the *inactive* mode, if (i) the lever is set on *constant*, (ii) the ignition is on, (iii) the engine is running, and (iv) the driver has not applied the brakes then the program transitions to the *Cruise* mode (Action A_3). In the *cruise* mode, the program transitions to the *off* mode if the ignition turns off (Action A_4). Also, in the *cruise* mode, the program transitions to the *inactive* mode when the engine turns off (Action A_5). If the lever is off or the driver applies the brakes then the program moves to the *override* mode from *cruise* mode (Actions A_6). In the *override* mode, the program goes to the *off* mode if the ignition turns off (Action A_7). If the engine turns off then the program will transition to the *inactive* mode from the *override* mode (Action A_8). Finally, in the *override* mode, the program transitions to the *cruise* mode if (i) the ignition is on, (ii) the engine is running, (iii) the brakes have not been applied, and (iv) the lever is on *constant* or *resume* (Action A_9).

```
A_1 :  ((SysMode = off) ∧ (Ignition = on))                    ⟶  SysMode := inactive;
A_2 :  ((SysMode = inactive) ∧ (Ignition = off ))             ⟶  SysMode := off;
A_3 :  ((SysMode = inactive) ∧ (Lever = constant) ∧ (Ignition = on)  ∧
                              (EngState = running) ∧ (Brake = notApplied))
                                                              ⟶  SysMode := cruise;
A_4 :  ((SysMode = cruise) ∧ (Ignition = off))                ⟶  SysMode := off;
A_5 :  ((SysMode = cruise) ∧ (EngState = off))                ⟶  SysMode := inactive;
A_6 :  ((SysMode = cruise) ∧ ((Brake = applied) ∨ (Lever = off)) )
                                                              ⟶  SysMode := override;
A_7 :  ((SysMode = override) ∧ (Ignition = off))              ⟶  SysMode := off;
A_8 :  ((SysMode = override) ∧ (EngState = off))              ⟶  SysMode := inactive;
A_9 :  ((SysMode = override) ∧ (Ignition = on) ∧ (EngState = running)  ∧
        (Brake = notApplied) ∧ ((Lever = constant) ∨ (Lever = resume)))
                                                              ⟶  SysMode := cruise;
```

The invariant of the CC program. In general, the CC program should be in one of the modes *off*, *inactive*, *cruise*, or *override*, and the brakes subsystem should also be working properly. Also, if the program is in the *off* mode then the ignition should be off. If the program is in the *inactive* mode then the ignition should be on. If the program is in the *cruise* mode, then ignition should be on, the engine should be running, the lever should not be off, and the brakes must not be applied. In the *override* mode, the ignition is on and the engine should be running. Hence, we represent the invariant of the CC program by the state predicate INV_{CC}, where

$$INV_{CC} = \{ \text{s} : ((\text{SysMode(s)} \neq \text{unknown}) \wedge (\text{Brake(s)} \neq \text{unknown})) \} \cap I_1, \text{ where}$$

$$\begin{aligned}
I_1 = \{\text{s}: & ((\text{SysMode(s)} = \text{off}) \vee (\text{SysMode(s)} = \text{inactive}) \vee \\
& \qquad (\text{SysMode(s)} = \text{cruise}) \vee (\text{SysMode(s)} = \text{override})) \wedge \\
& ((\text{SysMode(s)} = \text{off}) \Rightarrow (\text{Ignition(s)} = \text{off})) \wedge \\
& ((\text{SysMode(s)} = \text{inactive}) \Rightarrow (\text{Ignition(s)} = \text{on})) \wedge \\
& ((\text{SysMode(s)} = \text{cruise}) \Rightarrow ((\text{ignition(s)} = \text{on}) \wedge (\text{EngState(s)} = \text{running}) \wedge \\
& \qquad (\text{Brake(s)} \neq \text{applied}) \wedge (\text{Lever(s)} \neq \text{off}))\ ) \wedge \\
& ((\text{SysMode(s)} = \text{override}) \Rightarrow ((\text{Ignition(s)} = \text{on}) \wedge (\text{EngState(s)} = \text{running}))) \}
\end{aligned}$$

Notation. $x(s)$ denotes the value of a program variable x in state s.

The class of f_f faults. The malfunction of the brake subsystem may corrupt the value of *Brake* to an unknown value. In addition, faults may perturb the cruise control system to an unknown operating mode. We represent the set of transitions of f_f by the following actions:

$$\begin{array}{lll}
FS: \ (\text{Brake} \neq \text{unknown}) & \longrightarrow & \text{Brake} := \text{unknown}; \\
FM: \ (\text{SysMode} \neq \text{unknown}) & \longrightarrow & \text{SysMode} := \text{unknown};
\end{array}$$

The class of f_m faults. Embedded computing systems are deployed in harsh environments, and as a result, such systems are subject to the faults that perturb the internal state of the system. The mode of the CC program may be perturbed to an unknown state when the program is in one of its regular modes. In such situations, the system should recover to one of its regular modes without violating safety. Thus, masking fault-tolerance should be provided to the fault action FM, which denotes the class of f_m faults (observe that $f_m \subseteq f_f$ holds).

The safety specification of the CC program. The specification of the CC program stipulates that as long as the driver has applied the brakes the CC program must never transition to the cruise mode. Also, in cases where the signal coming from the brake subsystem is corrupted then it is not safe for the program to transitions to the cruise mode. Finally, the program must not transition to the cruise mode in cases where it is in an unknown mode. Hence, we represent the safety specification of the CC program by the following set of bad transitions:

$$\begin{aligned}
\text{spec}_{CC} = \{ & (s_0, s_1) : \\
& ((\text{Brake}(s_0) = \text{applied}) \wedge (\text{Brake}(s_1) = \text{applied}) \wedge (\text{SysMode}(s_1) = \text{cruise})) \vee \\
& ((\text{Brake}(s_0) = \text{unknown}) \wedge (\text{SysMode}(s_1) = \text{cruise})) \vee \\
& ((\text{SysMode}(s_0) = \text{unknown}) \wedge (\text{SysMode}(s_1) = \text{cruise})) \}
\end{aligned}$$

We trace the application of the Add_Failsafe_Masking algorithm (see Figure 7) for the CC program. After the addition of multitolerance, we require the synthesized program to be (i) failsafe fault-tolerant in the presence of f_f (i.e., the set of transitions represented by actions FS and FM), and (ii) masking fault-tolerant in the presence of f_m (i.e., the set of transitions represented by the action FM).

Adding failsafe f_f-tolerance from S for *spec*. The fault transitions of f_f may perturb the state of the program from INV_{CC} to states where both *SysMode* and *Brake* are unknown. Thus, the fault-span T_1 would be equal to $((SysMode(s) = unknown) \vee (Brake(s) = unknown))$. The program CC transitions to the *Cruise* mode either by the action A_3 or by the action A_9. The guards of these actions are enabled if *Brake* = *notApplied*. As a result, by construction, program CC will not execute any safety-violating transition from states where $Brake = unknown$ or $SysMode = unknown$.

Calculating *spec*′. Since, in the context of this example, the transitions of f_f do not directly violate $spec_{CC}$ and T_1 is closed in program transitions (by definition), we have $spec = spec'$.

Adding masking f_m-tolerance from INV_{CC} for $spec_{CC}$. The invocation of Add_Masking generates a program p' that is masking f_m-tolerance from INV_{CC} for *spec*. The invariant S' of p' is equal to the invariant of the fault-intolerant program; i.e., INV_{CC}. The program p' contains a set of transitions that recover p' from an unknown mode. We represent these transitions by actions M_{10} and M_{11} as follows:

$$\begin{array}{lll}
M_1: & A_1 & \\
 & \cdots & \\
M_9: & A_9 & \\
M_{10}: & ((\text{SysMode} = \text{unknown}) \wedge (\text{Ignition} = \text{off})) & \longrightarrow \quad \text{SysMode} := \text{off}; \\
M_{11}: & ((\text{SysMode} = \text{unknown}) \wedge (\text{Ignition} = \text{on})) & \longrightarrow \quad \text{SysMode} := \text{inactive};
\end{array}$$

We leave it to the readers to investigate that the application of the Add_Masking_Failsafe algorithm in Figure 8 also creates the same failsafe-masking multitolerant program[3].

[3] Even though in this case reversing the order of adding failsafe and masking creates the same program, it may not be the case in general.