Abstract—In this paper, we classify multitolerant systems, i.e., systems that tolerate multiple classes of faults and provide potentially different levels of tolerance to them in terms of strong and weak multitolerance. Intuitively, this classification is based upon the guarantees provided by the program when one class of faults occurs while it is recovering from another class of faults. We focus on automated synthesis of weak multitolerant programs. Such weak multitolerance becomes necessary when it is impossible to provide strong multitolerance and/or when the probability of one class of faults occurring while the program is ‘recovering’ from a fault from another class is negligible.

By considering the levels of fault-tolerance provided to each class of faults, we evaluate five possible combinations for weak multitolerance. We find a counterintuitive result that if masking fault-tolerance is desired for one class of faults and masking (or failsafe) fault-tolerance is desired for another class of faults then the problem is NP-hard. This result is surprising since the corresponding problem for strong multitolerance can be solved in polynomial time. Also, we show that the problem of synthesizing weak multitolerance for other combinations is in P.

More broadly, this result demonstrates the role of assumptions, e.g., independence of occurrences of faults from different classes, in the complexity of automated synthesis.

Keywords—Program synthesis, Formal methods, Multitolerance, Fault tolerance

I. INTRODUCTION

There are two main requirements in providing fault-tolerance. The first requirement is that the safety properties of the program are preserved when faults occur. The second requirement is that the program recovers from faults so that its subsequent computation is correct. Intuitively, when both of these requirements are met, we denote the corresponding program as masking fault-tolerant. While masking fault-tolerance is ideal, due to feasibility and/or cost, one may choose to provide a weaker level of tolerance.

One weaker level of fault-tolerance is nonmasking (or stabilizing). In this level, the program provides recovery but may violate safety during recovery. Nonmasking fault-tolerance is desirable when the design of masking fault-tolerance is either expensive or impossible. For example, in [9], authors provide nonmasking fault-tolerance to memory safety bugs in Neutron, a version of the TinyOS operating system. In such a case, while one could technically design a masking fault-tolerant system, it is very expensive in terms of human effort. Other examples include algorithms [28] [31] for clock synchronization where faults such as failure and repair of nodes, random re-starts and initial lack of synchronization can corrupt clock values. In these examples, guaranteeing the safety property (e.g., clock drift is always limited) is expensive or impossible. Hence, nonmasking fault-tolerance is preferred so that the program will eventually recover to states where clocks remain synchronized. Other examples of nonmasking and stabilizing fault-tolerance include [10] [14] [34] [33].

Another weaker level of fault-tolerance is failsafe. In this level, the program always satisfies its safety properties in the presence of faults, but it may not need to resume satisfying its liveness properties when faults stop occurring. Thus, failsafe fault-tolerance is applied in the situations where safety property is obviously much more important than liveness. Note that if liveness is preserved the fault-tolerant model will be masking fault-tolerance. Since liveness is not ensured, implementation cost of failsafe is typically cheaper than masking fault-tolerance. Failsafe fault-tolerance is also utilized in systems at component level, e.g., one may choose to ensure that in case of faults, a component guarantees its own safety constraints although it may not satisfy its liveness constraints. Upon noticing this, other components could ensure that safety and liveness are satisfied for the overall system. Examples of such approach include [35]. It describes a mechanism to prevent a single faulty node from monopolizing the communication bus in a distributed hard real-time system. In such a case, failsafe fault-tolerance is imposed to enforce fail-silent behavior of the node. Other examples of failsafe system include [30] [32] [18] [20].

Since a system is often subject to multiple faults, system designers need to consider how the system acts when multiple faults occur simultaneously (by simultaneous, we mean that a fault from one class occurs before the system has recovered from a fault from another class). For example, an application may want to mask message loss, i.e., ensure that the program continues to satisfy its specification even if messages are lost. However, for more serious faults, e.g., node failure, it may only provide nonmasking fault-tolerance where the program eventually reorganizes itself to legitimate
states while some other properties (e.g., safety properties, satisfaction of requests generated during recovery) may not be met during recovery. Clearly, if the program provides nonmasking fault-tolerance for node failure, it cannot provide masking fault-tolerance if node failure and message loss occur simultaneously. It follows that the best one can do is to ensure that the program eventually recovers to states from where it satisfies its specification. In other words, the tolerance provided for the case where faults from both classes occur simultaneously is equal to the ‘minimum’ level of fault-tolerance provided to each class of faults. We denote such multitolerance as \textit{strong} multitolerance. In the above example, a \textit{strong} multitolerant program would ensure that nonmasking fault-tolerance would be provided if node failure and message loss occur simultaneously.

Another possible solution is to ensure that if only faults from one class occur then the program provides an appropriate level of fault-tolerance. However, it may not provide any tolerance if faults from two classes occur simultaneously. Such a situation can occur if a program that does the reorganization requires that message delivery is reliable and no messages are lost. We denote such multitolerance as \textit{weak} multitolerance.

Observe that \textit{strong} and \textit{weak} multitolerance are two extreme options in defining multitolerance. When faults from two classes occur simultaneously, one intermediate approach is to provide a tolerance that is in-between the tolerance prescribed by \textit{strong} and \textit{weak} multitolerance. As shown in Section III, the definition of \textit{weak} multitolerance in this paper allows us to model such \textit{intermediate} multitolerant programs.

Just like failsafe/nonmasking tolerance is an alternative to masking when it is impossible or expensive to afford masking tolerance, \textit{weak} multitolerance is an ideal alternative for \textit{strong} multitolerance in the following two situations: (1) it is impossible to guarantee any level of tolerance in a computation where faults from two classes occur simultaneously; (2) it may be possible to provide strong multitolerance by adding more redundancy, but this in turn is expensive. Hence, one has to do a tradeoff between this extra cost and the tolerance level. Examples include FTSS systems [6], where authors have shown impossibility of designing algorithms that tolerate crash faults and transient faults simultaneously although each fault could be tolerated separately. \textit{Weak} multitolerance is also desirable when the occurrences of faults from different classes are independent, therefore the probability of multiple classes of faults occurring in the same computation is negligible and, hence, can be ignored. Other examples include a triple modulo redundant (TMR) system. It is straightforward to observe that such a system tolerates intermittent Byzantine faults as well as intermittent benign faults. However, the program does not tolerate the situation where both types of faults occur in close proximity.

In this paper, we focus on automated addition of \textit{weak} multitolerance to an existing program. Typical scenarios where automation is desirable include: (1) a model checker found a counterexample in an existing program and, hence, the program needs to be modified to meet fault-tolerance requirements; (2) for the sake of separation of concerns, the designer can only focus on the functionality of the program and leave the issue of fault-tolerance to an automated technique. Moreover, while adding such fault-tolerance, it is desirable to preserve the functionality of the program in the absence of faults.

\textbf{Contributions of the paper.} We consider the case where \textit{weak} multitolerance is added to two classes of faults, \(f_1\) and \(f_2\). (These results can also be easily extended for the cases where three or more classes of faults are considered.) For each class of fault, we consider three levels of fault-tolerance: (1) failsafe, where in the presence of faults the program satisfies its safety specification, (2) nonmasking, where the program eventually recovers to legitimate states from where it satisfies its specification, and (3) masking, where the program satisfies both of these constraints. Thus, we consider five possible combinations \(MM, FM, FF, MN\) and \(NN\) where in each combination, the first letter denotes the level of fault-tolerance for \(f_1\) and the second letter denotes the level of fault-tolerance for \(f_2\). (We do not consider the combination \(FN\). We note that this combination is NP-complete and can be demonstrated using a variation of the proof in [26] where \(FN\) \textit{strong} multitolerance is considered.) The complexity of different combinations of \textit{weak} multitolerance is shown in Table I.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{F1} & \textbf{F2} & \textbf{Failsafe} & \textbf{Nonmasking} & \textbf{Masking} \\
\hline
\textbf{Failsafe} & P & NP-complete & NP-complete* \\
\hline
\textbf{Nonmasking} & NP-complete & P & P \\
\hline
\textbf{Masking} & NP-complete* & P & NP-complete* \\
\hline
\end{tabular}
\caption{The complexity of different types of \textit{weak} multitolerance}
\end{table}

We find that the results marked with an asterisk are especially surprising given that the corresponding problems can be solved in polynomial time for \textit{strong} multitolerance [26].

In broader terms, these results demonstrate the effect of ‘assumptions’ in designing fault-tolerant programs. In particular, while \textit{strong} multitolerance is ideal, as described above, there are cases where it is impossible. When faced with such impossibility results, a designer typically makes assumptions about the underlying system. (Very well studied problems in the context of distributed systems are consensus, leader election etc., where assumptions are made about the underlying system to overcome the impossibility result in [16]). One way to circumvent the impossibility in designing \textit{strong} multitolerance is to utilize \textit{weak} multitolerance where faults from each class are individually tolerated. However when faults from two classes occur simultaneously, tolerance
is not guaranteed. The results in this paper show that for certain combinations of tolerance level, the problem can be solved efficiently. However, for certain combinations, the complexity increases substantially.

Organization of the paper. In Section II, we present the formal definition of programs, specifications, faults and fault-tolerance. In Section III, we present the formal statement of the synthesis problem of weak multitolerance. In Section IV and V, we present the NP-completeness proof for the cases where MM and FM weak multitolerance are added to fault-intolerant programs. In Section VI, VII and VIII, we present the sound and complete algorithms for the synthesis of weak multitolerance programs that provide FF, MN and NN weak multitolerance. We present the related work in Section IX. We give a discussion in Section X. Finally, the conclusion and future work are described in Section XI.

II. Preliminaries

In this section, we give formal definitions of programs, problem specifications, faults, and fault-tolerance. The programs are specified in terms of their state space and their transitions. The definition of specifications is adapted from Alpern and Schneider [1]. The definition of faults and fault-tolerance is adapted from Arora and Gouda [2] and Kulkarni [22].

A. Program

Definition 2.1: (program) A program \( P \) is a tuple \( \langle S_P, \psi_P \rangle \), where \( S_P \) is the state space (i.e., the set of all possible states), and \( \psi_P \) is a set of transitions, where \( \psi_P \) is a subset of \( S_P \times S_P \).

Definition 2.2: (state predicate) A state predicate \( S \) is any subset of \( S_P \).

Definition 2.3: (core) A state predicate \( S \) is closed in program \( P = \langle S_P, \psi_P \rangle \) (or briefly \( \psi_P \)) iff \( \forall (s_0, s_1) \in \psi_P : ((s_0 \in S) \Rightarrow (s_1 \in S)) \).

Definition 2.4: (computation) A computation of \( P = \langle S_P, \psi_P \rangle \) (or briefly \( \psi_P \)) is a finite or infinite state sequence: \( \pi = \langle s_0, s_1, \ldots \rangle \) s.t. the following conditions are satisfied:
1. \( \forall j : 0 < j < \text{lengthof}(\pi) : (s_{j-1}, s_j) \in \psi_P \),
2. \( \pi \) is infinite and terminates in \( s_f \) then there does not exist any state \( s \) such that \( (s_f, s) \in \psi_P \).

Definition 2.5: (projection) The Projection of a set \( \psi \) of transitions on a state predicate \( S \) (denoted as \( \psi|S \)) is the following set of transitions: \( \psi|S = \{ (s_0, s_1) : (s_0, s_1) \in \psi_P \land s_0, s_1 \in S \} \).

The projection of program \( P \) on state predicate \( S \) (denoted as \( P|S \)), is the program \( \langle S_P, \{ (s_0, s_1) : (s_0, s_1) \in \psi_P \land s_0, s_1 \in S \} \rangle \).

B. Specification

Definition 2.6: (safety specification) The safety specification is specified as a set of bad transitions [22], i.e., for program \( P \), its safety specification is a subset of \( S_P \times S_P \).

Hence, we say a transition \( (s_0, s_1) \) violates the safety specification \( sspec \) iff \( (s_0, s_1) \in sspec \). A sequence \( \pi = \langle s_0, s_1, \ldots \rangle \) satisfies \( sspec \) iff \( \forall j : 0 < j < \text{lengthof}(\pi) : (s_{j-1}, s_j) \notin sspec \).

Definition 2.7: (liveness specification) A liveness specification is specified in terms of a set of sequences. A sequence \( \pi = \langle s_0, s_1, \ldots \rangle \) satisfies \( sspec \) iff some suffix of \( \pi \) is in the set of sequences specified by \( lspec \).

A specification \( spec \) for program \( P \) consists of a safety specification, say \( sspec \), and a liveness specification, say \( lspec \) [1]. In other words, a sequence satisfies the specification \( spec \) iff it satisfies the corresponding safety and liveness specification.

Remark 2.1: In our synthesis problem in Section III, we begin with an initial program that satisfies its specification (including the liveness specification). We will show that our synthesis techniques preserve the liveness specification. Hence, the liveness specification need not be specified explicitly.

We now define what it means for a program \( P \) to satisfy a specification.

Definition 2.8: (satisfies) Let \( P = \langle S_P, \psi_P \rangle \) be a program, \( S \) be a state predicate, and \( spec \) be a specification for \( P \). We write \( P \models_S spec \) and say that \( P \) satisfies \( spec \) from \( S \) iff (1) \( S \) is closed in \( \psi_P \), and (2) every computation of \( P \) that starts from a state in \( S \) satisfies \( spec \).

Definition 2.9: (invariant) Let \( P = \langle S_P, \psi_P \rangle \) be a program, \( S \) be a state predicate, and \( spec \) be a specification for \( P \). If \( P \models_S spec \) and \( S \neq \{ \} \), we say that \( S \) is an invariant of \( P \) for \( spec \).

Whenever the specification is clear from the context, we will omit it; thus, “\( S \) is an invariant of \( P \)” abbreviates “\( S \) is an invariant of \( P \) for \( spec \)”. Note that Definition 2.8 introduces the notion of satisfaction with respect to computations. In case of computations prefixes that are not necessarily maximal, we characterize them by determining whether they can be extended to an infinite computation that satisfies the specification.

Definition 2.10: (maintains) Program \( P \) maintains \( spec \) from \( S \) iff (1) \( S \) is closed in \( \psi_P \), and (2) for all computation prefixes \( \alpha \) of \( P \), there exists a computation suffix \( \beta \) such that \( \alpha \beta \in spec \). We say that \( P \) violates \( spec \) iff it is not the case that \( P \) maintains \( spec \).

We note that if \( P \) satisfies \( spec \) from \( S \) then \( P \) maintains \( spec \) from \( S \) as well, but the reverse direction does not hold. We, in particular, introduce the notion of maintains for computations that a (fault-intolerant) program cannot produce, but the computation can be extended to one that is in \( spec \) by adding recovery (see Section II-C for details).

C. Faults

The faults \( f \) that a program is subject to are systematically represented by transitions. Based on the classification of
faults from [27], this representation suffices for physical faults, process faults, message faults and improper initialization. It is not intended for program bugs (e.g. buffer overflow). However, if such bugs exhibit behavior such as component crash, it can be modeled by using this approach.

Thus, a fault for \( P = (S_P, \psi_P) \) is a subset of \( S_P \times S_P \).

**Definition 2.11:** (fault-span) A state predicate \( T \) is an \( f \)-span (read as fault-span) of \( P = (S_P, \psi_P) \) from \( S \) iff the following conditions are satisfied: (1) \( S \subseteq T \), and (2) \( T \) is closed in \( \psi_P \cup f \).

Observe that for all computations of \( \mathcal{P} \) that start from states in \( S, T \) is a boundary in the state space of \( \mathcal{P} \) up to which (but not beyond which) the states of \( \mathcal{P} \) may be perturbed by the occurrence of the transitions in \( f \). Subsequently, as we defined the computations of \( \mathcal{P} \), one can define computations of program \( P \) in the presence of faults \( f \) by simply substituting \( \psi_P \) with \( \psi_P \cup f \) in Definition 2.4.

D. Fault-Tolerance

We now define what it means for a program to be failsafe/nonmasking/masking \( f \)-tolerant (read as fault-tolerant).

**Definition 2.12:** (masking \( f \)-tolerant) A program \( \mathcal{P} \) is masking \( f \)-tolerant from \( S \) for \( \text{spec} \), iff the following conditions hold:

1) \( \mathcal{P} \models_S \text{spec} \);
2) There exists \( T \) such that:
   a) \( T \) is an \( f \)-span of \( \mathcal{P} \) from \( S \);
   b) \( (S_P, \psi_P \cup f) \) maintains \( \text{spec} \) from \( T \);
   c) Every computation of \( (S_P, \psi_P) \) that starts from a state in \( T \) eventually reaches a state of \( S \).

Hence, if program \( \mathcal{P} \) is masking \( f \)-tolerant from \( S \) for \( \text{spec} \) then \( S \) is closed in \( \psi_P \) and every computation of \( \mathcal{P} \) that starts from a state in \( S \) satisfies \( \text{spec} \) in the absence of faults. Additionally, in the presence of faults, there is a fault-span predicate \( \langle T \supseteq S \rangle \) that is closed in \( \psi_P \cup f \).

**Definition 2.13:** (failsafe \( f \)-tolerant) A program \( \mathcal{P} \) is failsafe \( f \)-tolerant from \( S \) for \( \text{spec} \), iff conditions 1, 2a and 2b in Definition 2.12 hold.

**Definition 2.14:** (nonmasking \( f \)-tolerant) A program \( \mathcal{P} \) is nonmasking \( f \)-tolerant from \( S \) for \( \text{spec} \), iff conditions 1, 2a and 2c in Definition 2.12 hold.

**Notation.** Whenever the program \( \mathcal{P} \) is clear from the context, we will omit it; thus, “\( S \) is an invariant” abbreviates “\( S \) is an invariant of \( \mathcal{P} \)”. Also, whenever the specification \( \text{spec} \) and the invariant \( S \) are clear from the context, we omit them; thus, “\( f \)-tolerant” abbreviates “\( f \)-tolerant from \( S \) for \( \text{spec} \)”.

III. Problem Statement

In this section, we first present the definition of weak multitolerance. As mentioned in Section II-D, a fault-tolerant program guarantees a desired level of fault-tolerance (i.e., failsafe/nonmasking/masking) in the presence of a specific class of faults. Now, we consider the case where the program is subject to faults from multiple fault-classes.

Intuitively, a weak multitolerant program assumes that faults from two classes will not occur simultaneously (By simultaneous, we mean that a fault from one class occurs before the system has recovered from a fault from another class). Thus, while the program tolerates multiple classes of faults, it tolerates them ‘one at a time’. It follows that if the program is subject to fault classes \( f_1, f_2, \ldots, f_n \) and the program is perturbed by \( f_i \), \((0 \leq i \leq n)\) then the program provides the desired level of fault-tolerance to \( f_i \). However, if faults from \( f_j (0 \leq j \leq n, j \neq i) \) occur while the program is recovering from fault class \( f_i \) then the program may not guarantee fault-tolerance. Thus, a weak multitolerant program that tolerates fault classes \( f_1, f_2, \ldots, f_n \) provides fault-tolerance to each \( f_i (0 \leq i \leq n) \) respectively.

**Definition 3.1:** Let \( f_\delta = \{ (f_i, l_i) | 0 < i \leq n, l_i \in \{ \text{failsafe, nonmasking, masking} \} \} \) where \( n \geq 0 \).

Program \( P \) is weak multitolerant to fault set \( f_\delta \) from \( S \) for \( \text{spec} \) iff the following conditions hold:

1) (In the absence of faults) \( \mathcal{P} \models_S \text{spec} \).
2) For each \( i, 0 < i \leq n, \mathcal{P} \models l_i \) \( f_i \)-tolerant from \( S \) for \( \text{spec} \) respectively.

**Remark 3.1:** Whenever the level of fault tolerance to a given fault class is clear from the context, for brevity, we omit it.

We note that the definition of weak multitolerance can also be used to define intermediate multitolerance. For example, consider the case where masking fault-tolerance is required for both \( f_1 \) and \( f_2 \). However, if \( f_2 \) occurs while the program is recovering from \( f_1 \) then nonmasking fault-tolerance is provided. For this case, we can model such requirements by letting \( f_\delta = \{ (f_1, \text{masking}), (f_2, \text{masking}), (f_1 \cup f_2, \text{nonmasking}) \} \).

Now, using the definition of weak multitolerant programs, we identify the requirements of the problem of synthesizing a weak multitolerant program, \( \mathcal{P} \), from its fault-intolerant version, \( \mathcal{P} \). We require \( \mathcal{P} \) only add weak multitolerance and introduce no new behaviors in the absence of faults. This problem statement is the natural extension to the problem statement in [24] where fault-tolerance is added to a single class of faults. More specifically, we stipulate the following two conditions: (1) \( S' \subseteq S \); (2) \( (\mathcal{P}'|S') \subseteq (\mathcal{P}|S') \). Thus, the problem of weak multitolerance synthesis is as follows:

**Definition 3.2:** The Synthesis Problem.

Given \( \mathcal{P}, S, \text{spec} \) and \( f_\delta \): Identify \( \mathcal{P}' \) and \( S' \) such that:

- (C1) \( S' \subseteq S \);
- (C2) \( (s_0, s_1) \in \mathcal{P}' \land \delta_0 \in S' \Rightarrow (s_0, s_1) \in \mathcal{P} \), and
- (C3) \( \mathcal{P}' \) is weak multitolerant to \( f_\delta \) from \( S' \) for \( \text{spec} \).

We state the corresponding decision problem as follows:

**Definition 3.3:** The Decision Problem.

Given \( \mathcal{P}, S, \text{spec} \) and \( f_\delta \): Does there exist a program \( \mathcal{P}' \), with its invariant \( S' \) that satisfies the requirements of Definition 3.2?
IV. MM WEAK-MULTITOLERANCE

In this section, we investigate the synthesis algorithm for weak multitolerant program for the case where program is subject to two classes of faults \( f_m1 \) and \( f_m2 \) for which masking fault-tolerance is required, that is \( f_s = \{ ( f_m1,\text{ masking}) , ( f_m2,\text{ masking}) \} \) in Definition 3.1. We find a surprising result such that a MM (Masking-Masking) weak multitolerant synthesis problem is NP-complete, even though, as shown in [26], the synthesis problem of the corresponding strong multitolerant program is in P.

Before we present the formal proof, we give an intuition behind this complexity. Consider the case there exists a transition \((s_1, s_2)\) of \( f_m2 \) that violates the safety specification. We have the following two options: (i) ensure that \( s_1 \) is unreachable in the computations of \( P \cup f_m1 \), (ii) allow \( s_1 \) to be reached only while program is ‘recovering’ from \( f_m1 \). Moreover, the choice made for this state affects other similar states. In our proof, we relate the choice made between these two options to the value of boolean variables in the SAT formula. This allows us to reduce the SAT problem to the MM weak multitolerant synthesis problem.

Now in Theorem 4.1, we show the MM weak multitolerant synthesis problem is NP-complete by reducing SAT problem to it.

Theorem 4.1: The problem of synthesizing MM weak multitolerant programs from their fault-intolerant version is NP-complete.

Proof. Given a program \( P \), with its invariant \( S \), its specification \( \text{spec} \), and two classes of faults \( f_m1 \) and \( f_m2 \), since showing membership to NP is trivial, we only show that the synthesis problem identified in definition 3.2 is NP-hard when \( f_s = \{ ( f_m1,\text{ masking}) , ( f_m2,\text{ masking}) \} \).

Mapping. Now we present a polynomial-time mapping from an instance of SAT problem to a corresponding instance \( \langle P, S, \text{spec}, f_m1, f_m2 \rangle \) of the synthesis problem. An instance of the SAT problem is specified in terms of a set of literals \( x_1, x_2, \ldots, x_n \) and \( \neg x_1, \neg x_2, \ldots, \neg x_n \), of these literals, \( x_i \) and \( \neg x_i \) are complements of each other. The SAT formula is of the form \( \phi = C_1 \land C_2 \land C_3 \land \ldots \land C_k \), where each clause \( C_i \) is a disjunction of several literals. The structure of the mapped instance is such that the given SAT formula is satisfiable if and only if there exists a solution to the synthesis problem. We construct the mapped instance as follows (see Figure 1):

The invariant and state space of the fault-intolerant program, \( P \). The state space of \( P \) is as follows:

- We introduce a state \( s \). This is the only state in the invariant.
- For each propositional variable \( x_i \), \( 1 \leq r \leq n \), and its complement \( \neg x_i \) in the SAT instance, we introduce the following states: \( e_i, t_i, g_i, h_i, a_i \), and \( b_i \).
- For each clause \( C_r \), \( 1 \leq r \leq k \), we introduce states \( w_r \) and \( z_r \).
- If clause \( C_r \) includes literal \( x_i \), we introduce a state \( d_r \). If clause \( C_r \) includes literal \( \neg x_i \), we introduce a state \( d'_r \).

The transitions of the fault-intolerant program, \( P \). The transitions of \( P \) includes only a self-loop \((s, s)\).

The transitions of \( f_m1 \) and \( f_m2 \). The transitions of \( f_m1 \) and \( f_m2 \) are as follows:

- For each clause \( C_r \), we include fault transition \((s, w_r)\) in \( f_m1 \) and fault transition \((s, z_r)\) in \( f_m2 \).
- If clause \( C_r \) includes literal \( x_i \), then we include fault transition \((d_r, e_i)\) in \( f_m1 \).
- If clause \( C_r \) includes literal \( \neg x_i \), then we include transition \((d'_r, t_i)\) in \( f_m2 \).
- For each propositional variable \( x_i \) and its complement \( \neg x_i \), we include fault transition \((g_i, a_i)\) in \( f_m1 \) and \((h_i, b_i)\) in \( f_m2 \).

The safety specification of the fault-intolerant program, \( P \). The safety specification is as follows:

- Transitions \((g_i, a_i)\) and \((h_i, b_i)\) violate safety specification.
- Transitions \((s, s), (s, w_r), (s, z_r), (d_r, e_i)\) and \((d'_r, t_i)\) do not violate safety specification.
- For each clause \( C_r \), each propositional variable \( x_i \) and its complement \( \neg x_i \), the following transitions do not violate safety:
  - \((w_r, z_r), (z_r, d_r), (z_r, d'_r), (e_i, t_i), (t_i, e_i), (e_i, g_i), (t_i, h_i), (g_i, s), (h_i, s)\).
- All transitions except those identified above (e.g., \((z_r, w_r), (z_r, z_r), (z_r, s)\), etc) violate safety specification.

Reduction. Now, we show that the given SAT formula is satisfiable if and only if there exists a solution to Problem 3.2 where \( f_s = \{ ( f_m1,\text{ masking}) , ( f_m2,\text{ masking}) \} \).

\( \iff \) First, we show if the given SAT formula is satisfiable, then there exists a solution that meets the requirements of the synthesis problem. Since \( \phi \) has a satisfying truth assignment, there exists an assignment of truth values to the literals \( x_i \), such that each \( C_r \) evaluates true. Now, we identify the program \( P' \), that solves MM weak multitolerant problem.

The invariant of \( P' \) is the same as the invariant of \( P \) (i.e., \( \{ s \} \)). We derive the transitions of the weak multitolerant program \( P' \) as follows:

- For each disjunction \( C_r \), we include the transition \((w_r, z_r)\).
- If \( x_i \) is assigned true:
  * We include \((e_i, t_i), (t_i, h_i), (h_i, s)\).
  * For each disjunction \( C_r \) that includes \( x_i \), we include the transitions: \((z_r, d_r)\) and \((d_r, s)\).
- If \( x_i \) is assigned false:
  * We include \((t_i, e_i), (e_i, g_i)\) and \((g_i, s)\).
For each disjunction $C_r$ that includes $\neg x_i$, we include the transitions: $(z_r, d'_r, s)$ and $(d'_r, s)$.

Thus, in the presence of $f_{m1}$ alone, $P'$ provides safe recovery to $s$ through $d_{r1}, e_i, t_i, h_i$. In the presence of $f_{m2}$ alone, $P'$ provides safe recovery to $s$ through $d'_r, t_i, e_i, g_i$.

Now, we show that $P'$ is weak multitolerant in the presence of faults $f_{m1}, f_{m2}$.

- **(In the absence of faults)** $P'|S = P|S$. Thus, $P'$ satisfies $spec$ in the absence of faults.

- **Masking f-tolerance to $f_{m1}$**. If the faults from $f_{m1}$ occur then the program can be perturbed to state $w_r$, $1 \leq r \leq k$. From $w_r$, $P'$ has only one transition that reaches $z_r$. Since $C_r$ evaluates to $true$, there exists $i$ such that either $x_i$ is a literal in $C_r$ and $x_i$ is assigned the truth value $true$ or $\neg x_i$ is a literal in $C_r$ and $x_i$ is assigned the truth value $false$. In the former case, $P'$ can recover to $s$ using the two sequences of transitions, $((z_r, d_{r1}), (d_{r1}, s))$, or $((z_r, d_{r1}), (d_{r1}, e_i), (e_i, t_i), (t_i, h_i), (h_i, s))$. In the latter case, $P'$ can recover to $s$ using exactly one sequence of transitions $((z_r, d'_r), (d'_r, s))$. Note that if $x_i$ is true then $P'$ cannot reach $g_i$ from where it can violate safety specification. Thus, any computation of $P'[\models f_{m1}$ eventually reaches a state in the invariant. Moreover, from $z_r$, every computation of $P' \cup f_{m1}$ does not violate the safety specification. Based on the above discussion, $P'$ is masking tolerant to $f_{m1}$.

- **Masking f-tolerance to $f_{m2}$**. The argument is similar to the one showing that $P'$ is masking tolerant to $f_{m1}$.

\[ \iff \] Second, we show that if there exists a weak multitolerant program that solves the instance of the synthesis problem identified in Definition 3.2, then the given SAT formula is satisfiable. Let $P'$ be the weak multitolerant program which derived from the fault-intolerant program $P$. The invariant of $P'$, $S'$, is not empty and $S' \subseteq S$, $S'$ must include state $s$. Thus, $S'=S$. Let $C_r$ be a clause in the given SAT formula. The corresponding state added in the instance of the synthesis problem is $z_r$. Note that $w_r$ can be reached from $s$ by a transition in $f_{m1}$. Hence $P'$ must include the transition $(w_r, z_r)$. Thus $z_r$ is reached in the computation of $P' \cup f_{m1}$. Hence, $P'$ must recover to $s$ from $z_r$ without violating $spec$. Therefore, for some $i$, $P'$ has to have a transition of the form $(z_r, d_{r1})$ or $(z_r, d'_r)$. If $P'$ includes $(z_r, d_{r1})$, then the clause $C_r$ contains literal $x_i$ and we assign $x_i$ the truth value $true$. Likewise, if $P'$ includes $(z_r, d'_r)$ for some $i$, then the clause $C_r$ contains literal $\neg x_i$ and we assign $x_i$ the truth value $false$. Thus, by construction, $C_r$ evaluates to true.

Now, to complete the proof, we have to show that the truth value assigned to all literals is consistent, i.e., it is not the case that $x_i$ is assigned $true$ in one clause and $false$ in another clause. We show this by proof by contradiction. If $x_i$ is assigned $true$ in clause $C_r$ and $false$ in clause $C_q$ then $P'$ includes both transitions $(z_r, d_{r1})$ and $(z_q, d'_r)$. Now, from $d_{r1}$, the program can reach $e_i$ by the occurrence of $f_{m1}$ alone. Hence, the program $P'$ cannot include the transition $(e_i, g_i)$, as including this transition will allow the program to reach $g_i$ in a computation of $P' \cup f_{m1}$ and violate safety by executing $(g_i, a_i)$. Likewise, $P'$ can reach $t_i$ by the occurrence of $f_{m2}$ alone. Hence, $P'$ cannot include the transition $(t_i, h_i)$. If both transitions $(e_i, g_i)$ and $(t_i, h_i)$ are not included then $P'$ cannot recover from $e_i$ to the state in the invariant. This contradicts the assumption that $P'$ is masking $f_{m1}$-tolerant. Thus, the truth value assignment to all literals is consistent.

\[ \square \]
V. FM WEAK-MULTITOLERANCE

In this section, we investigate the synthesis problem for weak multitolerant programs for the case where program is subject to two classes of faults $f_1$ and $f_2$ for which respectively failsafe and masking fault-tolerance is required. That is, $f_3 = \{(f_1, \text{failsafe}), (f_2, \text{masking})\}$ in Definition 3.1. This synthesis problem is NP-complete. This result is also surprising since the corresponding problem for strong multitolerance is in P.

Theorem 5.1: The problem of synthesizing FM weak multitolerant programs from their fault-intolerant version is NP-complete.

Proof sketch. This proof is similar to that of Theorem 4.1, where the mapping from SAT formula is changed as follows: We replace $f_m1$ (respectively, $f_m2$) fault transitions with transitions of $f_2$ (respectively, $f_1$). With this change, if the SAT formula is satisfiable then $FM$ weak multitolerance can be added to the instance of the synthesis problem. This implies that $FM$ weak multitolerance can also be added. Moreover, for the reverse direction, the proof is identical to that in Theorem 4.1. For reasons of space, we omit the detailed proof.

VI. FF WEAK-MULTITOLERANCE

In this section, we investigate the synthesis problem of programs that are weak multitolerant to two classes of faults $f_1$ and $f_2$ for which respectively failsafe fault-tolerance is required. That is, $f_3 = \{(f_1, \text{failsafe}), (f_2, \text{failsafe})\}$ in Definition 3.1. We show that such a $FF$ (Failsafe-Failsafe) weak multitolerant program can be synthesized in polynomial time in the state space. Towards this end, we present a sound and complete algorithm. We note that this algorithm can be easily generalized for the case where $f_3$ includes three or more fault classes for which failsafe fault-tolerance is desired.

Given a program $P$, with its invariant $S$, its specification $spec$, let $P'$ to be the synthesized program with invariant $S'$ that is weak multitolerant to $f_1$ and $f_2$. By definition, $P'$ must maintain $spec$ from every reachable state in the computation of $P' \cup f_1$ (respectively, $P' \cup f_2$). To this end, on Line 1, we first identify $ms_1$, a set of states from where execution of one or more $f_1$ transitions violates safety. Clearly $P'$ cannot reach a state in $ms_1$ either in the absence of faults or in the presence of $f_1$ alone. Likewise, we compute $ms_2$ on Line 2. Next, we compute $mt$ to be a set of transitions that reach $ms_1 \cup ms_2$ or those that violate $spec$. If there exist states in the invariant such that execution of one or more fault actions from those states violates $spec$, we recalculate the invariant by removing those states. In this recalculation, we ensure that all computations of $P - mt$ within the new invariant, $S'$, are infinite. By the constraints of Definition 3.2 and the definition of $ms_1$ and $ms_2$, $S'$ must be a subset of $S - ms_1 - ms_2$. Likewise, $P'$ cannot include transitions that begin in $S'$ (states that could be reached in the absence of faults) and are in $mt$. Hence, the only transitions $P'$ that can be used inside $S'$ are a subset of $P - mt$. Removal of states in $ms_1 \cup ms_2$ or transitions in $mt$ may cause some states in $S - ms_1 - ms_2$ to be deadlocked, i.e., where $P'$ has no outgoing transitions. Since $P'$ cannot deadlock the absence of faults, we remove such deadlock states recursively to construct $S'$ (Line 6). As shown in Line 7, if the invariant becomes an empty set after reconstruction, we cannot find a FF weak multitolerant program $P'$. If the invariant is not empty, we compute program transition set in Line 8. Details are discussed in Algorithm 1.

Theorem 6.1: The algorithm $Add_FF_WeakMulti$ is sound and complete.

For reasons of space, the soundness and completeness proofs are omitted.

VII. MN WEAK-MULTITOLERANCE

In this section, we present the synthesis algorithm for weak multitolerant programs for the case where program is subject to two classes of faults $f_1$ and $f_2$ for which respectively masking and nonmasking fault-tolerance is required, that is $f_3 = \{(f_1, \text{masking}), (f_2, \text{nonmasking})\}$ in Definition 3.1. We show that such a $MN$ (Masking-Nonmasking) weak multitolerant program can be synthesized in polynomial time in the state space. This sound and complete algorithm also can be easily generalized for the case where $f_3$ includes one class of faults for which masking fault tolerance is desired and two or more fault classes for which nonmasking fault tolerance is desired. Note that from the results in Section IV, if masking fault tolerance is desired for two or more classes of faults, then the problem is NP-complete.

Given a program $P$, with its invariant $S$, its specification $spec$, our goal is to synthesize a program $P'$, with invariant $S'$ that is weak multitolerant to $f_3$. By definition, $P'$ must be masking $f_1$-tolerant. $P'$ must also be nonmasking $f_2$-tolerant. The algorithm for $MN$ weak multitolerance utilizes the algorithm $Add_Masking$ (from [23]) that adds masking fault-tolerance to a single class of faults. $Add_Masking$ returns the synthesized program $P'$, its invariant $S'$ and its fault-span $T'$ such that $P'$ is masking fault-tolerant to $S'$ and the fault-span used to prove this in Definition 2.12 is $T'$. The algorithm $Add_MN_WeakMulti$ only relies on the correctness (i.e., soundness and completeness) of $Add_Masking$. It does not rely on the actual implementation of $Add_Masking$. (For consideration of space, we omit this algorithm in this paper. Reader can find details in [23]). Thus, $Add_MN_WeakMulti$ first invokes $Add_Masking$ on Line 1 with parameters $(P, f_1, S, spec)$. As shown in Line 2, if the invariant becomes an empty set after reconstruction in Line 1, we cannot find an $MN$ weak multitolerant program $P'$. If the invariant is not empty, we recompute program transition set in Line 3. The details are defined in Algorithm 2:
Algorithm 1: Add_FF_Weakmulti

**Input:** $\psi_p$-transitions, $f_1, f_2$; faults of two classes that need failsafe $f$-tolerance, $S$: state predicate, spec: safety specification

**Output:** If successful, a fault-tolerant $P'$ with invariant $S'$ that is weak multitolerant to $f_1$ and $f_2$

1. $ms_1 := \{s_0 : \exists s_1, s_2, \ldots, s_n : (\forall j : 0 \leq j < n : (s_j, s_{j+1}) \in f_1) \land (s_n-1, s_n) \text{ violates spec}\};$
2. $ms_2 := \{s_0 : \exists s_1, s_2, \ldots, s_n : (\forall j : 0 \leq j < n : (s_j, s_{j+1}) \in f_2) \land (s_n-1, s_n) \text{ violates spec}\};$
3. $mt := \{(s_0, s_1) : ((s_1 \in ms_1 \cup ms_2) \lor (s_0, s_1) \text{ violates spec})\};$
4. $S' := S - ms_1 - ms_2;$
5. $P_1 := P - mt;$
6. $S' := S - \{s_0 | s_0 \in S \cup (\forall s_1 : s_1 \in S : (s_0, s_1) \notin \psi_p)\};$
7. if ($S' = \{\}$) declare no weak multitolerant program $P'$ exists, return $\emptyset, \emptyset;$
8. $P' := P_1 \cup \{(s_0, s_1) : (s_0, s_1) \in \psi_p, s_0 \in S' \land s_1 \notin S'\};$
9. return $P', S';$

Algorithm 2: Add_MN_Weakmulti

**Input:** $\psi_p$-transitions, $f_1, f_2$; faults of two classes that need failsafe $f$-tolerance, $S$: state predicate, spec: safety specification

**Output:** If successful, a fault-tolerant $P'$ with invariant $S'$ that is weak multitolerant to $f_1$ and $f_2$

1. $P_1, S', T_1 := \text{Add_Masking}(P, f_1, S, \text{spec});$
2. if ($S' = \{\}$) declare no weak multitolerant program $P'$ exists, return $\emptyset, \emptyset;$
3. $P' := P_1 \cup \{(s_0, s_1) : (s_0, s_1) \in \psi_p, s_0 \notin T_1 \land s_1 \in T_1\};$
4. return $P', S'$;

**Theorem 7.1:** The algorithm Add_MN_Weakmulti is sound and complete.

VIII. NN WEAK-MULTITOLERANCE

The algorithm Add_NN_Weakmulti for NN (Nonmasking-Nonmasking) weak multitolerant problem is identical to Add_MN_Weakmulti, except instead of invoking Add_Masking on Line 1, we call Add_Nonmasking (from [23]).

**Theorem 8.1:** The algorithm Add_NN_Weakmulti is sound and complete.

IX. RELATED WORK

Automated program synthesis is studied from different perspectives. One approach (e.g., [5]) focuses on synthesizing fault-tolerant programs from their specification in a temporal logic (e.g., CTL, LTL, etc.). The word synthesis is also used in the context (e.g., [12] [17] [29] [19]) of transforming an abstract (such as UML) program into a concrete (such as C++) program while ensuring that the location of concrete program in memory, its data flow etc. meet the constraints of the embedded system. By contrast, our approach focuses on transformation of one abstract program into another that meets additional properties of interest. Our approach will advance the applicability of this existing work by allowing designers to add properties of interest in the abstract model and then using existing work to generate concrete program. Hence our approach is desirable when one needs extend an existing system by adding fault-tolerance.

Our work is closely related to the work on controller synthesis [3] [4] [8] [13] and game theory [11] [15] [21]. In this work, supervisory control of real-time systems has been studied under the assumption that the existing program (called a plant) and/or the given specification is deterministic. Moreover, in both game theory and controller synthesis, since highly expressive specifications are often considered, the complexity of the proposed synthesis methods is very high. For example, the synthesis problems presented in [3] [4] [11] [15] are EXPTIME-complete. Furthermore, deciding the existence of a controller in [8] [13] is 2EXPTIME-complete. In addition, these approaches do not address some of the crucial concerns of fault-tolerance (e.g., providing recovery in the presence of faults) that are considered in the our work. In addition, the high complexity of these methods is a serious barrier to actually synthesize moderate sized programs. By contrast, our approach focus only on specifications needed to express properties of interest. Hence, the complexity of our algorithm is substantially lower.

The algorithms in [23] [25] [26] have addressed the problem of adding fault-tolerance to only one class of fault. Also the algorithm in [26] is targeted toward the synthesis of programs that simultaneously tolerate multiple classes of faults whereas we address the synthesis of programs that provide the appropriate level of fault-tolerance and not to provide any tolerance if faults from one class occurs while the program is ‘recovering’ from faults from another class. The ‘weak’ multitolerance way in our work is necessary especially when it is impossible to guarantee any level of tolerance in a computation where faults from two classes
X. Discussion

In this section, we discuss several questions that are raised by our work.

The first question is how can one identify the fault classes needed to apply the algorithms in this paper? Fault-classes vary according to specific application. To generate fault-classes, first we need to identify the faults that the program may be subject to. This can be achieved by techniques in [27] for fault forecasting. Then we formally characterize each of these faults by using state perturbations. As stated in Section II, this is possible for physical faults, process faults, message faults and improper initialization. Finally, we group the faults into fault classes based on the corresponding level of tolerance. The level of tolerance is based on expectations of system users and feasibility of providing that level of tolerance under system constraints. As discussed next, level may be changed based on the results from the algorithm for adding fault tolerance.

Another question is what happens if the algorithm for weak multitolerance synthesis problem declares failure to identify a weak multitolerant program? If this happens, it means it is impossible to design the program with the requirements of weak multitolerance. One solution may be to add more redundancy. It is possible to revise the algorithm in this paper, so additional redundancy could be introduced automatically. However, we believe that addition of redundancy should be handled manually, since it requires resources and an automation algorithm cannot determine whether these resources are reasonable and available. Another possible solution may be to change the expected level of tolerance. For example, if a \( MM \) weak multitolerance is unfeasible, then system may provide \( MN \) weak multitolerance. Again, the choice of this depends upon whether the reduced level of tolerance is acceptable to system users. As stated in Section I, this involves a tradeoff between the cost and level of fault tolerance. An automation algorithm, like the one in this paper, allows designers to identify a fault-tolerant program with the given choices in terms of redundancy and level of tolerance.

We have already implemented a tool Sycraft which can provide single fault tolerance. It is based on symbolically reachable algorithm with BDDs and has been shown in [7]. Issues of single fault tolerance have been well illustrated in [23]. We plan to enrich our tool by providing heuristics and function of automated synthesis of multitolerance in the future work.

XI. Conclusion

In this paper, we addressed the problem of synthesizing weak multitolerant programs from their fault-intolerant version. The input to the synthesis problem consists of the fault-intolerant program, a set of different classes of faults that the program is subject to, and the expected level of tolerance for each class of faults. Our algorithm ensures that the synthesized program provides the appropriate level of fault-tolerance to each of those fault classes.

We discussed the problem of weak synthesis for a program which is subject to two fault-classes \( f_1 \) and \( f_2 \). Here three fault-tolerance levels are considered: failsafe, nonmasking and masking. We discussed five possible combinations \( MM, FM, FF, MN \) and \( NN \). In each combination, the first letter indicates the fault-tolerance level for \( f_1 \) and the second letter indicates the fault-tolerance level for \( f_2 \).

In analyzing the complexity of weak multitolerant synthesis, we found a surprising result that if masking fault-tolerance is desired for one class of faults and masking (or failsafe) fault-tolerance is desired for another class of faults then the problem is NP-hard. This result is counterintuitive since the corresponding problem for strong multitolerance can be solved in polynomial time. For other combinations \( FF, MN \) and \( NN \), we showed that the problem of synthesizing weak multitolerance is in \( P \). To demonstrate this, we presented a sound and complete algorithm for each combination.

REFERENCES


