The Complexity of Adding Multitolerance

JINGSHU CHEN, Michigan State University
ALI EBNENASIR, Michigan Technological University
SANDEEP KULKARNI, Michigan State University

We focus on the problem of adding multitolerance to an existing fault-intolerant program. A multitolerant program tolerates multiple classes of faults and provides a potentially different level of fault tolerance to each of them. We consider three levels of fault tolerance, namely failsafe (i.e., satisfy safety in the presence of faults), nonmasking (i.e., recover to legitimate states after the occurrence of faults), and masking (both). For the case where the program is subject to two classes of faults, we consider six categories of multitolerant programs—FF, FN, FM, MM, MN, and NN, where F, N, and M represent failsafe, nonmasking, and masking levels of tolerance provided to each class of fault. We show that the problem of adding FF, NN, and MN multitolerance can be solved in polynomial time (in the state space of the program). However, the problem is NP-complete for adding FN, MM, and FM multitolerance. We note that the hardness of adding MM and FM multitolerance is especially atypical given that MM and FM multitolerance can be added efficiently under more restricted scenarios where multiple faults occur simultaneously in the same computation. We also present heuristics for managing the complexity of MM multitolerance. Finally, we present real-world multitolerant programs and discuss the trade-off involved in design decisions while developing such programs.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming—*Program transformation*; D.2.4 [**Software Engineering**]: Program Verification—*Formal methods*; D.2.4 [**Software Engineering**]: Program Verification—*Reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Error handling and recovery*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering*

General Terms: Design, Verification, Reliability

Additional Key Words and Phrases: Fault tolerance, multitolerance, formal methods, program synthesis

ACM Reference Format:
Jingshu Chen, Ali Ebnenasir, and Sandeep Kulkarni. 2014. The complexity of adding multitolerance. ACM Trans. Autonom. Adapt. Syst. 9, 3, Article 15 (October 2014), 33 pages.
DOI: http://dx.doi.org/10.1145/2629664

1. INTRODUCTION

One of the crucial issues in designing software systems is fault tolerance. Specifically, there are two main requirements in providing fault tolerance. The first requirement

A preliminary version of this work [Chen and Kulkarni 2010] was published in the proceedings of the 30th IEEE International Conference on Distributed Computing Systems (ICDCS) in 2010.
This research was partially sponsored by the NSF grants CNS-0914913, CNS-1329807, CNS-1318678, and CCF-1116546.
Authors' addresses: J. Chen is currently working in INRIA France as a Postdoctoral Researcher; email: chenji15@cse.msu.edu; S. Kulkarni, Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824; email: sandeep@cse.msu.edu; A. Ebnenasir, Department of Computer Science, Michigan Technological University, Houghton, MI 49931; email: aebnenas@mtu.edu.

is that the program should preserve its safety properties in the presence of faults. The second requirement is that the program *recovers* from faults so that its subsequent computation is correct—that is, meets its safety and liveness properties. Intuitively, safety states that nothing bad ever happens in program computations, and liveness stipulates that something good will eventually occur (in every program computation). When both of these requirements are met in the presence of faults, we denote the corresponding program as masking fault tolerant. Although 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. 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, Chen et al. [2009] provide nonmasking fault tolerance to memory safety bugs in Neutron, a version of the TinyOS operating system [Levis et al. 2005]. In such a case, although one could technically design a masking fault-tolerant system, it is very expensive in terms of human effort. Other examples include algorithms for clock synchronization [Li and Rus 2006; Ramanathan et al. 1990], where faults such as failure and repair of nodes, random restarts, and initial lack of synchronization 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 fault tolerance include Claesson et al. [2009], Ezhilchelvan et al. [2004], Song and Chien [2005], and Sommer and Wattenhofer [2009].

Another weaker level of fault tolerance (compared with masking fault tolerance) is failsafe. In this level, the program always satisfies its safety properties even in the presence of faults, but it may not resume satisfying its liveness properties when faults stop occurring. Failsafe fault tolerance is applied in situations where safety is much more important than liveness (e.g., safety-critical systems). Note that if liveness is satisfied, then the fault-tolerance level will be masking. Since liveness is not ensured, implementation costs of failsafe are typically lower than masking fault tolerance. Failsafe fault tolerance is also utilized in systems at component level. For example, 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 Temple [1998], where a mechanism is presented 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 systems include Lubaszewski and Courtois [1998], Schiöberg et al. [2009], Heinzmann and Zelinsky [1999], and James et al. [2009].

Since a system is often subject to multiple faults, system designers need to consider scenarios where multiple faults may occur simultaneously. That is, a fault from one class occurs before the system has recovered from a fault from another class. In such cases, system designers need to identify the desired level of tolerance when faults occur. The notion of multitolerance is motivated by this observation. Consider a system that is subject to three types of faults: f_1, f_2, and f_3. There are eight possible cases: namely, the case where no faults occur, where f_1 (respectively, f_2 and f_3) occurs, where $\{f_1, f_2\}$ (respectively, $\{f_2, f_3\}$ and $\{f_1, f_3\}$) occur, and where $\{f_1, f_2, f_3\}$ occur. In practice, the designer need not consider all of these cases explicitly. For example, if masking fault tolerance is provided to $\{f_1, f_2\}$ (i.e., masking fault tolerance is provided when f_1 and f_2 occur simultaneously), then masking fault tolerance is implied when f_1 (or f_2) occur

Table I. Complexity of Different Types of Multitolerance
The asterisk (*) denotes that the complexity differs from the restricted version considered in Ebnenasir and Kulkarni [2011].

$f_1 \parallel f_2$	Failsafe	Nonmasking	Masking
Failsafe	P	NP-complete	NP-complete*
Nonmasking	NP-complete	P	P
Masking	NP-complete*	P	NP-complete*

alone. In addition, if the designer concludes that the probability of f_1 and f_2 occurring simultaneously is negligible, then the designer can decide to provide no fault tolerance in such a situation.

Ebnenasir and Kulkarni [2011] have considered a restricted version of such multitolerant systems, where they require that if a system provides some tolerance individually to f_1 and individually to f_2, then it must provide the *minimum* of these tolerances when f_1 and f_2 occur simultaneously. For example, if masking fault tolerance is provided to f_1 and nonmasking fault tolerance is provided to f_2, then in Ebnenasir and Kulkarni [2011] it is required that nonmasking fault tolerance must be provided when both f_1 and f_2 occur simultaneously. This restriction prevents one from considering systems where the probability of simultaneous occurrence of f_1 and f_2 is negligible and hence ignored by the designer. It also does not permit modeling of scenarios where the designer intends to mask individual occurrence of f_1 and f_2 but only provide failsafe fault tolerance when both occur simultaneously. (A simple example of such a system is a triple modulo redundant (TMR) system that individually provides masking fault tolerance to a single Byzantine fault and to a single failstop fault, but it only provides a failsafe fault tolerance if both faults occur simultaneously.)

In this article, we investigate the problem of adding multitolerance to concurrent programs in a more general setting than Ebnenasir and Kulkarni [2011], where the desired behavior when multiple faults occur is given by the designer. More specifically, instead of hard coding a specific definition for what a multitolerant system should do (e.g., providing the minimum level of tolerance) when different types of faults occur simultaneously, we leave it up to the designers to determine what they expect from the multitolerant system when several faults occur. To this end, we present the following contributions:

(1) We investigate the complexity issues in automated addition of multitolerance. We consider the case where 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.) Thus, we investigate six possible combinations MM, FM, FF, FN, MN, and NN, where in each combination the first letter denotes the level of fault tolerance for f_1 and the second letter represents the level of fault tolerance to f_2. Table I summarizes the complexity of different combinations of multitolerance.
(2) We show that the problem of automated addition of MM (FM) multitolerance is NP-complete. This result (marked with an asterisk in Table I) is especially surprising given that the corresponding problems can be solved in polynomial time for the restricted version of multitolerance considered in Ebnenasir and Kulkarni [2011].
(3) We present a sound and complete algorithm for automated addition of FF multitolerance and MN multitolerance. Moreover, we present a polynomial-time heuristic for adding MM multitolerance.
(4) We investigate the relation between the restricted version of multitolerance from Ebnenasir and Kulkarni [2011] and generalized definition of multitolerance considered in this article. We also identify circumstances where the restriction from these authors improves the complexity of adding multitolerance. Specifically, we show

that for high atomicity programs, (i.e., programs in which a process can read/write all variables in an atomic step), the restriction from Ebnenasir and Kulkarni [2011] does not affect the complexity of adding MN, NN, and FN multitolerance. However, the same result does not apply for MM, FM, or FF multitolerance.

Organization of the article. Section 2 presents the formal definition of programs, specifications, faults, and fault tolerance. Section 3 formally states the problem of adding multitolerance. Section 4 presents three examples where multitolerance is used. This section also identifies examples where the restrictions of Ebnenasir and Kulkarni [2011] prevent one from designing a multitolerant program. Section 5 investigates the automated addition of FF multitolerance and then presents a sound and complete algorithm. Sections 6 and 7 present an NP-completeness proof for cases where MM and FM multitolerance are added to fault-intolerant programs. Sections 8 and 9 present sound and complete algorithms for the addition of MN and NN multitolerance. Section 10 compares the limitations and effectiveness of the restricted notion of multitolerance (presented in Ebnenasir and Kulkarni [2011]) with the more general definition of multitolerance presented in this article. We discuss the related work in Section 11. Section 12 provides a discussion on the practical significance and the limitations of the proposed approach in this work. Finally, we make concluding remarks and discuss future work in Section 13.

2. PRELIMINARIES

In this section, we give formal definitions of programs, problem specifications, faults, and fault tolerance. The programs are specified in terms of their state space and their transitions. The definition of specification is adapted from Alpern and Schneider [1985]. The definitions of faults and fault tolerance are adapted from Arora and Gouda [1993] and Kulkarni [1999].

2.1. Program

Definition 2.1 (*Program*). A *program* $\mathcal{P}$ is a tuple $\langle S_\mathcal{P}, \psi_\mathcal{P}\rangle$, where $S_\mathcal{P}$ is the *state space* (i.e., the set of all possible states), and $\psi_\mathcal{P}$ is a set of transitions, where $\psi_\mathcal{P}$ is a subset of $S_\mathcal{P} \times S_\mathcal{P}$.

Definition 2.2 (*State Predicate*). A *state predicate* S is any subset of $S_\mathcal{P}$.

Definition 2.3 (*Closure*). A state predicate S is *closed* in program $\mathcal{P} = \langle S_\mathcal{P}, \psi_\mathcal{P}\rangle$ (or briefly $\psi_\mathcal{P}$) if and only if (iff) $(\forall(s_0, s_1) \in \psi_\mathcal{P} : ((s_0 \in S) \Rightarrow (s_1 \in S)))$.

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

Definition 2.5 (*Computation Prefix*). A *computation prefix* of $\mathcal{P} = \langle S_\mathcal{P}, \psi_\mathcal{P}\rangle$ (or briefly $\psi_\mathcal{P}$) is a finite state sequence: $\langle s_0, s_1, \ldots, s_m\rangle$ s.t. $\forall j : 0 < j \leq m : (s_{j-1}, s_j) \in \psi_\mathcal{P}$.

Definition 2.6 (*Projection*). The *projection* of a set ψ of transitions on a state predicate S (denoted as $\psi|S$) is the following set of transitions: $\psi|S = \{(s_0, s_1) : (s_0, s_1) \in \psi \wedge s_0, s_1 \in S\}$.

The projection of program $\mathcal{P}$ on state predicate S (denoted as $\mathcal{P}|S$) is the program $\langle S_\mathcal{P}, \psi_\mathcal{P}|S\rangle$.

2.2. Specification

Definition 2.7 (*Safety Specification*). The *safety specification* is specified as a set of bad transitions [Kulkarni 1999]—that is, for program $\mathcal{P}$, its safety specification is a subset of $S_\mathcal{P} \times S_\mathcal{P}$.

We say that a transition (s_0, s_1) violates the safety specification *sspec* iff $(s_0, s_1) \in sspec$. A sequence $\overline{s} = \langle s_0,\ s_1, \ldots\rangle$ satisfies *sspec* iff $\forall j : 0 < j < lengthof(\overline{s}) : (s_{j-1}, s_j) \notin sspec$.

Definition 2.8 (*Liveness Specification*). A *liveness specification* is specified in terms of a set of infinite sequences.

A sequence $\overline{s} = \langle s_0,\ s_1, \ldots\rangle$ satisfies a liveness specification *lspec* iff some suffix of $\overline{s}$ is in the set of sequences specified by *lspec*. A specification *spec* for program $\mathcal{P}$ consists of a safety specification, say *sspec*, and a liveness specification, say *lspec* [Alpern and Schneider 1985]. In other words, a sequence satisfies the specification *spec* iff it satisfies the corresponding safety and liveness specification.

Remark 2.1. In the problem of adding multitolerance in Section 3, we begin with an initial program that satisfies its specification (including the liveness specification). We will show that adding multitolerance *preserves* the liveness specification. Hence, the liveness specification need not be specified explicitly.

We now define what it means for a program $\mathcal{P}$ to satisfy a specification.

Definition 2.9 (*Satisfies*). Let $\mathcal{P} = \langle S_\mathcal{P},\ \psi_\mathcal{P}\rangle$ be a program, S be a state predicate, and *spec* be a specification for $\mathcal{P}$. We write $\mathcal{P} \models_S spec$ and say that $\mathcal{P}$ *satisfies spec* from S iff (1) S is closed in $\psi_\mathcal{P}$ and (2) every computation of $\mathcal{P}$ that starts from a state in S satisfies *spec*.

Assumption 2.1. For simplicity of subsequent definitions, if $\mathcal{P}$ satisfies *spec* from S, we assume that $\mathcal{P}$ includes at least one transition from every state in S. If $\mathcal{P}$ does not include a transition from a states, we then add the transition (s, s) to $\mathcal{P}$. Note that this assumption is not restrictive in any way. It simplifies subsequent definitions, as one does not have to model terminating computations explicitly.

Definition 2.10 (*Invariant*). Let $\mathcal{P} = \langle S_\mathcal{P},\ \psi_\mathcal{P}\rangle$ be a program, S be a state predicate, and *spec* be a specification for $\mathcal{P}$. If $\mathcal{P} \models_S spec$ and $S \neq \{\}$, we say that S is an *invariant* of $\mathcal{P}$ for *spec*.

Whenever the specification is clear from the context, we shall omit it; thus, "S is an invariant of $\mathcal{P}$" abbreviates "S is an invariant of $\mathcal{P}$ for *spec*." Note that Definition 2.9 introduces the notion of satisfaction with respect to computations. In case of computation prefixes that are not necessarily maximal, we characterize them by determining whether they can be extended to a computation that satisfies the specification.

Definition 2.11 (*Maintains*). Program $\mathcal{P}$ *maintains spec* from S iff (1) S is closed in $\psi_\mathcal{P}$, and (2) for all computation prefixes α of $\mathcal{P}$ starting in S, there exists a computation suffix β such that $\alpha\beta \in spec$. We say that $\mathcal{P}$ *violates spec* from S iff $\mathcal{P}$ does not maintain *spec* from S.

We note that if $\mathcal{P}$ satisfies *spec* from S, then $\mathcal{P}$ maintains *spec* from S as well, but the reverse is not necessarily true. 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 2.3 for details).

2.3. Faults

Each class of fault f to which a program is subject is systematically represented by a set of transitions. Formally, a *fault class* for program $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ is a subset of $S_{\mathcal{P}} \times S_{\mathcal{P}}$. Based on the classification of faults from Laprie and Randell [2004], this representation suffices for physical faults, process faults, message faults, and improper initialization [Chen and Kulkarni 2011]. Thus, any class of faults that manifests itself by a set of (nondeterministic) transitions in program state space can be modeled in our formal setting. However, it is difficult to capture the effect of faults such as wing damage on the flight control system of an aircraft using our fault model.

Definition 2.12 (*Fault Span*). A state predicate T is an f-span (read as *fault span* for f) of $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ from S iff the following conditions are satisfied: (1) $S \subseteq T$, and (2) T is closed in $\psi_{\mathcal{P}} \cup f$.

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$\mathcal{P}$in the presence of fault f by simply substituting $\psi_{\mathcal{P}}$ with $\psi_{\mathcal{P}} \cup f$ in Definition 2.4 except the constraint (2).

Since we consider the problem of multiple classes of faults, and we would like to evaluate the fault-tolerance property when multiple faults occur in the same computation, we also extend the notion of computations in the presence of faults to computations of program $\mathcal{P}$ in the presence of multiple faults. Intuitively, in computations where faults from f_1 and f_2 occur simultaneously, every transition is either a transition of $\mathcal{P}$ or a transition of f_1 or a transition of f_2. This can also be described as a computation of $\mathcal{P}$ in the presence of $f_1 \cup f_2$, where $f_1 \cup f_2$ is another class of faults.

2.4. Fault Tolerance

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

Definition 2.13 (*Masking f-Tolerant*). A program $\mathcal{P}$ is *masking f-tolerant* from S for *spec* iff the following conditions hold:

(1) $\mathcal{P} \models_S$ *spec*; and
(2) There exists T such that
  (a) T is an f-span of $\mathcal{P}$ from S,
  (b) $\langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \cup f \rangle$ maintains *spec* from T, and
  (c) Every computation of $\langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ that starts from a state in T eventually reaches a state of S.

Thus, if program $\mathcal{P}$ is masking f-tolerant from S for *spec*, then S is closed in $\psi_{\mathcal{P}}$ and every computation of $\mathcal{P}$ that starts from a state in S satisfies *spec* in the absence of faults. Additionally, in the presence of faults, there is a fault-span predicate T ($S \subseteq T$) that is closed in $\psi_{\mathcal{P}} \cup f$.

Definition 2.14 (*Failsafe f-Tolerant*). A program $\mathcal{P}$ is *failsafe f-tolerant* from S for *spec* iff conditions (1), (2a), and (2b) in Definition 2.13 hold.

Definition 2.15 (*Nonmasking f-Tolerant*). A program $\mathcal{P}$ is *nonmasking f-tolerant* from S for *spec* iff conditions (1), (2a), and (2c) in Definition 2.13 hold.

Notation. Whenever the program $\mathcal{P}$ is clear from the context, we shall omit it; thus, "S is an invariant" abbreviates "S is an invariant of $\mathcal{P}$." In addition, whenever the specification *spec* and the invariant S are clear from the context, we omit them; thus, "f-tolerant" abbreviates "f-tolerant from S for *spec*."

3. PROBLEM STATEMENT

In this section, we first present the definition of *multitolerance*. Intuitively, in a multitolerant system, the designer considers different combinations of faults that could occur simultaneously and identifies the desired level of tolerance for that combination. For combinations that are not considered, the designer has either decided that they are unlikely to occur or that the system cannot tolerate those combination of faults. (By simultaneous, we mean that a fault from one class occurs before the system has recovered from a fault from another class). For example, consider the scenario where the system is subject to crash fault, Byzantine fault, and message loss. Furthermore, consider the case where the designer assumes that if some process is Byzantine, then the system cannot tolerate either crash or message loss. In this case, the classes of faults considered for such a program would be {crash}, {Byzantine}, {message loss}, {crash, message loss}. For each class of faults, the designer identifies the desired level of tolerance.

Based on this example, in defining a multitolerant program, we begin with fault classes $f_1, f_2, \dots f_n$ and identify the level of fault tolerance provided to each class of faults. The following definition identifies this requirement.

Definition 3.1 (*Multitolerant*). Let $f_\delta = \{\langle f_i, l_i\rangle \mid 0 < i \leq n, l_i \in \{\mathit{failsafe}, \mathit{nonmasking}, \mathit{masking}\}\}$ where $n \geq 1$. Program $\mathcal{P}$ is *multitolerant* to fault set f_δ from S for *spec* iff the following conditions hold:

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

Using the definition of multitolerant programs, we identify the requirements of the problem of synthesizing a multitolerant program $\mathcal{P}'$ with invariant S' from its fault-intolerant version $\mathcal{P}$ with invariant S. We require that $\mathcal{P}'$ only adds multitolerance and introduces no new behaviors in the *absence* of faults. This problem statement is a natural extension of the problem statement in Kulkarni et al. [2007], where fault tolerance is added to a single class of faults. More specifically, we stipulate the following two conditions: (1) $S' \subseteq S$—that is, the invariant S' of the multitolerant program $\mathcal{P}$'—is a subset of the invariant S of the given program $\mathcal{P}$; (2) $(s_0, s_1) \in \mathcal{P}' \wedge s_0 \in S' \Rightarrow (s_0, s_1) \in \mathcal{P}$. Thus, the problem of adding multitolerance is as follows.

PROBLEM 3.1 (ADDING MULTITOLERANCE). *Given $\mathcal{P}$, S, spec, and f_δ, identify $\mathcal{P}'$ and S' such that*

—*(C1) $S' \subseteq S$,*
—*(C2) $(s_0, s_1) \in \mathcal{P}' \wedge s_0 \in S' \Rightarrow (s_0, s_1) \in \mathcal{P}$, and*
—*(C3) $\mathcal{P}'$ is multitolerant to f_δ from S' for spec.*

We state the corresponding decision problem as follows.

PROBLEM 3.2 (THE MULTITOLERANCE DECISION PROBLEM). *Given $\mathcal{P}$, S, spec, and f_δ, does there exist a program $\mathcal{P}'$ with its invariant S' that satisfies the requirements of Problem 3.1?*

4. EXAMPLES

In this section, we present examples to illustrate different scenarios where distinct levels of multitolerance are provided depending on the feasibility, cost, and user requirements. The first example is a leader election protocol subject to node-leave faults and message losses. We demonstrate that providing masking fault tolerance to each fault alone (i.e., MM multitolerance) is less costly than providing masking when both faults occur simultaneously (defined in Ebnenasir and Kulkarni [2011] as a restricted version of multitolerance). The second example is a vertex coloring protocol where failsafe-nonmasking multitolerance is guaranteed. This example demonstrates that in some cases, providing any level of tolerance is impossible when multiple classes of faults occur. Finally, the third example provides masking-masking multitolerance and illustrates a case where the restrictions from Ebnenasir and Kulkarni [2011] can be satisfied if more resources are provided.

4.1. Masking-Masking Multitolerance

In this section, we present a leader election program that provides masking-masking multitolerance to two types of faults: f_{m1} and f_{m2}. The fault-type f_{m1} manifests itself as if the leader leaves, called the *leader leave fault*, and f_{m2} denotes the message loss fault. The program consists of n processes $(p_1, p_2, \ldots, p_n)$ that are organized in a connected network. Each process has a unique ID, numbered from $0, \ldots, n-1$. One of these processes is selected as the leader. When no faults occur, there are no actions that are executed. However, when the current leader leaves, other processes can compete to be the leader. When a process wants to be the leader, it starts a diffusing computation [Dijkstra and Scholten 1980] and declares itself to be the leader when the diffusing computation completes successfully. If multiple processes start diffusing computations, then the process with higher ID wins.

Program. A process initiates a diffusing computation when it receives a $\langle leave \rangle$ message from the current leader and if it is not already participating in a diffusing computation. To initiate a diffusing computation, process j sends a message $\langle j \rangle$ to all of its neighbors. It also sets its own root value and its parent to be equal to j. The root value keeps track of the initiator of a diffusing computation, and the parent value keeps track of the node that sent this diffusing computation to j. When process j receives a diffusing computation message of the form $\langle ID \rangle$ from k, it does the following. If j is already participating in a diffusing computation such that the initiator of that diffusing computation (stored in $root.j$) is higher than ID, then j ignores the new message. If the value of $root.j$ is equal to ID, then j is receiving the same diffusing computation twice. Hence, it only replies to k. Otherwise, it forwards this diffusing computation to all of its neighbors. To do so, it forwards the message $\langle ID \rangle$ and sets its own parent variable to k and its own root variable to ID. Finally, when j receives a reply message from all neighbors except the parent, j sends a reply to its parent. Moreover, if j is the initiator ($p.j == j$), then it declares itself to be the leader. Thus, the actions in this program are as follows:

```
upon receiving $\langle leave \rangle$ message from departing leader
    $\longrightarrow$                    //node $j$ sends diffusing computation message
        if $root.j == -1$
            send $\langle j \rangle$ to $nbrs.j$,
            $p.j := j$,
            $root.j := j$;
        else                //ignore
________________________________________________________________

upon receiving $\langle ID \rangle$ from $k$
    $\longrightarrow$if $root == ID$      //duplicate
            send reply to $k$ with $\langle ID \rangle$;
        else if $ID > root.j$
            $p.j := k$,
            $root.j := ID$,
            send $\langle ID \rangle$ to all neighbors except $k$;
        else     //ignore;
```

```
upon receiving reply from all neighbors except $p.j$ with $\langle root.j \rangle$
    $\longrightarrow root.j := -1$;
        if $p.j \neq j$
            send reply to $p.j$ with $\langle root.j \rangle$;
        else
            $leader.j := true$;
```

Invariant. The legitimate states of the program include those states where there is a unique leader, there are no ongoing diffusing computations to elect a leader, and the root value of every process is -1.

Fault Actions. The program is subject to two classes of faults, f_{m1} and f_{m2}, where f_{m1} denotes that the leader node leaves and notifies its neighbors, and f_{m2} represents message loss faults. Variable $channel_{i,j}$ denotes the sequence of messages in the channel between i and j. Hence, the two types of fault actions are as follows:

```
$f_{m1}$ (Leader node leave):
    $leader.j == true$
        $\longrightarrow$send $\langle leave \rangle$ to $nbrs.j$, $leader.j := false$;
$f_{m2}$ (Message loss):
    $channel_{i,j} \neq \langle\ \rangle$        //      $\langle\ \rangle$ denotes an empty channel
        $\longrightarrow$ $channel_{i,j} := tail(channel_{i,j})$;
```

Safety specification. The safety specification requires that in any state, there is at most one leader. More precisely, the program should never reach a state where:

$$spec_{mm} = (\exists j, k : j \neq k : leader.j \wedge leader.k)$$

Masking-masking multitolerance to leader node leave *and* message loss. The MM multitolerant program has the following properties:

(1) In the absence of faults, no action is executed, and there is a unique leader in the network.
(2) When fault f_{m1} occurs (i.e., the current leader leaves), one or more of its neighbors initiate a diffusing computation. Among the nodes that initiate the diffusing computation, the one with the highest ID is elected as the leader. Hence, masking fault tolerance is guaranteed when f_{m1} occurs.
(3) When fault f_{m2} causes messages to be lost, there is no effect on the number of leaders. Thus, there is a unique leader in the network. Hence, masking fault tolerance is guaranteed when f_{m2} occurs.
(4) If a message is lost during the diffusing computation, it is possible that the diffusing computation never completes, and no leader is elected. Thus, if faults f_{m1} and f_{m2} occur simultaneously, no fault tolerance is guaranteed.

Discussion. A more careful analysis of this program shows that if faults f_{m1} and f_{m2} occur together, it causes the diffusing computation to be blocked, thereby resulting in states where there is no leader. In this case, failsafe fault tolerance is provided. Thus, this is also an instance of FM multitolerance. Moreover, as one can imagine, it is possible to design a fault-tolerant program that provides masking fault tolerance to both f_{m1} and f_{m2} simultaneously. However, providing such tolerance is expensive. For example, it requires mechanisms to detect message losses (or potentially failure of a node). Additionally, it requires an overhead in terms of message retransmission. Moreover, if such faults are considered during diffusing computation, there is a need for keeping track of multiple diffusing computations initiated by the same node, such as with the use of sequence numbers. Thus, this example illustrates the case where providing multitolerance under the restrictions of Ebnenasir and Kulkarni [2011] is costly (in terms of complexity of the code, performance, etc.), and a less costly option can be provided by providing masking fault tolerance to each fault alone.

4.2. Failsafe-Nonmasking Multitolerance

In this section, we present a vertex coloring protocol that provides failsafe-nonmasking multitolerance to Byzantine fault and transient fault—that is, failsafe fault-tolerance when Byzantine fault occurs, nonmasking fault-tolerance when transient fault occurs, and no guarantees when both faults occur simultaneously.

The vertex coloring of the program is an assignment of colors to each process of the system. The goal of the program is that every process is assigned a color, and no two neighboring processes are assigned the same color. One assumption is that the degree of each process is at most d, and $d + 1$ colors are to be used.

Invariant. The legitimate states of the program are those whose colors are assigned appropriately.

Program. The program defines the following action for each node j:

$$color.j == color.k \quad \longrightarrow \quad color.j := available\_color(j);$$

In the preceding action, $color.j$ denotes the color assigned to process j, and $available\_color(j)$ returns a color not used in the locality of process j. No action is executed in the absence of faults.

Fault Actions. We consider two types of faults: Byzantine faults and transient faults. Both of these faults result in changing the color of the affected process. However, the main difference between these faults is that the former is a permanent fault—that is, the affected process can change the color as often as it desires, whereas the latter is a transient fault where there is a bound on the number of times the color of some process is affected. Another difference is that the former only affects a subset of (chosen) processes, whereas the latter can affect all processes at once. To model these faults, we introduce a variable $b.j$ that denotes whether j is Byzantine, variable $count.j$ that denotes the number of times $color.j$ is affected by transient faults, and MAX that denotes the number of permitted transient faults. (Note that all of these variables are auxiliary variables, i.e., variables used in the proof but not explicitly by the program itself.) Thus, the fault actions are as follows:

```
$f_f$ (Byzantine fault):
    $b.j == true$
        $\longrightarrow color_j :=$ random(0,d); // return a random value from 0 to d;
$f_n$ (transient fault):
    $count.j < MAX$
        $\longrightarrow color.j :=$ random(0,d);
```

The preceding fault actions may corrupt the color of a process to be the same as that of one of its neighbors.

Safety specification. The safety specification requires that any two neighboring nodes that are non-Byzantine have different colors. Thus, the program should never reach a state in the state predicate $spec_{fn}$, where

$$spec_{fn} = (\exists j, k :: (k \in nbrs.j) \wedge (b.j == false) \wedge (b.k == false) \wedge (color.j == color.k)).$$

Failsafe-nonmasking multitolerance to Byzantine fault and transient fault. The program has the following properties:

(1) In the absence of faults, there is no action. Thus, the program keeps a correct color assignment to all nodes.
(2) When f_f corrupts color assignment of the Byzantine node, no non-Byzantine node is affected; hence, the safety specification is not violated. Thus, failsafe fault-tolerance is provided.
(3) When f_n causes one node to change its color transiently, safety specification may be violated at that time. Then, the recovery action will reassign its color to be the correct one, and finally the program will recover to a correct assignment to all of the nodes. Thus, nonmasking fault tolerance is provided.
(4) When both faults f_f and f_n occur simultaneously, safety specification may be violated due to the occurrence of transient fault. Thus, failsafe fault tolerance is not guaranteed. In addition, the program may not recover to a correct assignment to all nodes since the Byzantine node is corrupted permanently. Hence, nonmasking fault tolerance is also not guaranteed.

Discussion. This example illustrates the need for multitolerance. In particular, with Byzantine faults, the faults can prevent the program from recovering to a legitimate state where the colors of all nodes are assigned properly—that is, no two neighboring nodes have the same color. In this example, providing masking or nonmasking fault tolerance to Byzantine faults is impossible. Likewise, execution of the transient fault itself can violate the safety specification. Thus, providing failsafe or masking fault tolerance to transient faults is impossible. For this reason, the only possible solution is to provide failsafe fault tolerance to Byzantine faults and nonmasking fault tolerance to transient faults alone.

4.3. Masking-Masking Multitolerance

This section presents a scenario for MM multitolerance and also illustrates the difference between the notion of multitolerance considered in this article and the restricted version of multitolerance considered in Ebnenasir and Kulkarni [2011]. As discussed previously, in Ebnenasir and Kulkarni [2011] it is required that if fault tolerance is individually provided to f_1 and f_2, then tolerance must be provided to the case where both faults occur in the same computation. This example illustrates that this requirement makes it impossible to add the restricted version of multitolerance considered in Ebnenasir and Kulkarni [2011]. By contrast, it is possible to realize a more relaxed set of requirements considered in this work.

Problem description. The agreement program (AP) includes a *general* process and three *nongeneral* processes. Initially, all nongenerals are undecided. The general casts a decision, and each nongeneral copies the decision of the general and terminates (i.e., *finalizes* its decision).

Safety specification. The AP [Lamport et al. 1982] has to satisfy two safety properties, namely *agreement* and *validity*. Agreement requires that if the general is faulty, then all nonfaulty nongenerals that have finalized agree on the same decision. Validity stipulates that if the general is not faulty, then any nonfaulty nongeneral that has finalized has the same decision as that of the general.

Fault classes. The AP is subject to two classes of faults: Byzantine and fail stop. The Byzantine faults can perturb the state of *at most* one process and make it behave arbitrarily. In other words, if a process is affected by Byzantine faults (i.e., a process is Byzantine), then it can cast different decisions (i.e., lying about its decision to different processes). The fail-stop faults could cause a process to crash in a detectable fashion. A fail-stopped process does not execute any actions once it crashes. We assume that fail-stop faults only affect the nongeneral processes.

The agreement program. The set of program variables is $\{d_g, d_0, f_0, up_0, d_1, f_1, up_1, d_2, f_2, up_2\}$, where (1) d_g denotes the decision of the general, which could be 0 or 1, and d_i represents the decision of process i $(0 \leq i \leq 2)$, where the domain of d_i is $\{0, 1, \bot\}$, and $\bot$ means that process i is undecided; (2) f_i is a Boolean variable representing whether or not process i has *finalized* its decision after copying a decision from the general; and (3) up_i is also a Boolean variable that denotes whether process i has crashed in a detectable fashion (i.e., has fail stopped). The actions of process i in the AP are as follows ($\oplus$ denotes addition in modulo 3):

$$\begin{array}{lll}
A_{i1}: & d_i = \bot \ \wedge\ \neg f_i \ \wedge\ up_i & \longrightarrow \quad d_i := d_g \\
A_{i2}: & (d_i \neq \bot \ \wedge\ \neg f_i) \ \wedge\ (d_{i\oplus 1} = \bot \vee d_i = d_{i\oplus 1}) \ \wedge & \\
 & (d_{i\oplus 2} = \bot \vee d_i = d_{i\oplus 2}) \ \wedge\ (d_{i\oplus 1} \neq \bot \vee d_{i\oplus 2} \neq \bot) \ \wedge\ up_i & \longrightarrow \quad f_i := true \\
A_{i3}: & (d_i \neq \bot \ \wedge\ \neg f_i) \ \wedge\ (d_{i\oplus 1} \neq \bot) \wedge (d_{i\oplus 2} \neq \bot) \ \wedge up_i & \longrightarrow \quad d_i := majority(d_1, d_2, d_3)) \\
 & & \quad f_i := true
\end{array}$$

If process i is undecided and not crashed, then it can copy the decision of the general (see action A_{i1}). Once decided, process i can finalize its decision if at least another nongeneral has made the same decision (action A_{i2}). If all nongenerals have copied a decision from the general, then process i can finalize by setting d_i to the majority of decisions (action A_{i3}). That is, if d_i differs from the majority, then process i corrects d_i by setting it to the majority of decisions and finalizing. Otherwise, d_i is equal to the majority, and action A_{i3} finalizes the decision of process i. Notice that a crashed process can execute none of its actions because up_i becomes *false*.

Invariant. An invariant of the AP includes states in which validity and agreement are satisfied and at most one process is faulty.

Masking-masking multitolerance. The AP has the following properties:

(1) In the absence of Byzantine and fail-stop faults, the AP satisfies both validity and agreement.
(2) In the presence of Byzantine faults, if the general is Byzantine, then validity vacuously holds and agreement is guaranteed by the majority of decisions. If a nongeneral has become Byzantine, then validity holds for nonfaulty nongenerals, and agreement is vacuously satisfied. Therefore, the AP is masking fault tolerant to Byzantine faults.
(3) In the presence of fail-stop faults, one of the nongenerals stops executing (i.e., its *up* variable becomes false). Thus, the other nongenerals satisfy validity. Agreement is satisfied as well since a majority of nongenerals exists and the general is not faulty. Therefore, the AP is masking fault tolerant to fail-stop faults.
(4) When both faults occur, the program may reach a state where a nongeneral has crashed and another nongeneral has become Byzantine. If the fail-stop faults have occurred before a process makes a decision, then the guard of action A_{i3} in the other two processes is false. Since another nongeneral has become Byzantine, its decision may not be the same as the decision of the nonfaulty nongeneral. Thus, the guard of action A_{i2} of the nonfaulty nongeneral is also false. Therefore, the

entire program deadlocks—that is, masking fault tolerance cannot be guaranteed for Byzantine and fail-stop faults.

Discussion. This example demonstrates a case where it is impossible to design the restricted version of multitolerance from Ebnenasir and Kulkarni [2011]. The reason behind it is that the simultaneous occurrence of both faults may get the AP to a state where no majority of decisions exists among the nonfaulty processes. One approach for enabling recovery from such a state is to add redundancy by including an additional nongeneral process in the AP, which may not be always practical. Thus, in this case, the designer may adopt a more relaxed definition of multitolerance as considered in this article.

Although illustrated in the context of a simple example, this analysis is also applicable in several systems. For example, consider a distributed database that maintains a certain level of replication. Such a system will tolerate some classes of faults (failure of nodes, failure of links, message faults) by masking them so that the user always observes the correct state of their files. However, if several of these faults occur at once, it may compromise consistency for availability (or vice versa) by giving the user stale data during recovery. Such a system would be masking-masking-nonmasking multitolerant. If it chose consistency over availability, it is ensuring that any data given to the user is always correct but the user may be unable to obtain the desired data in the presence of simultaneous occurrence of several faults. This would be an example of a masking-masking-failsafe multitolerance. None of such systems can be modeled under the restricted version of multitolerance in Ebnenasir and Kulkarni [2011].

Another real-world example that demonstrates a case where the restricted notion of multitolerance is unnecessary includes safety-critical embedded systems [Rushby 2001]. For instance, ROBUS [Miner et al. 2002] provides a reliable bus architecture that is partitioned into fault containment regions that guarantee the independence of physical faults. Thus, in such a system, the occurrence of one type of faults will not cause the occurrence of another. This is clearly a case where only the weaker notion of multitolerance is applicable.

5. COMPLEXITY ANALYSIS OF FF MULTITOLERANCE

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

Given is a program $\mathcal{P}$, with its invariant S and its specification *spec*. Let $\mathcal{P}'$ be the synthesized program with invariant S' that is multitolerant to f_1 and f_2. By definition, $\mathcal{P}'$ must maintain *spec* from every reachable state in the computations of $\mathcal{P}' \cup f_1$ (respectively, $\mathcal{P}' \cup f_2$). To this end, in line 1 of Algorithm 1, we first identify ms_1, a set of states from where execution of one or more f_1 transitions violates safety. Clearly, $\mathcal{P}'$ 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 in 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 the 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 $\mathcal{P} - mt$ within the new invariant, S', are infinite. By the constraints of Definition 3.1 and the definition of ms_1 and ms_2, S' must be a subset of $S - ms_1 - ms_2$. Likewise, $\mathcal{P}'$ cannot include transitions that begin in S' and are in mt. Hence, the only transitions $\mathcal{P}'$ can use inside S' are a subset of $\mathcal{P} - mt$. Removal of states in $ms_1 \cup ms_2$ or transitions in mt may create some deadlock states in $S - ms_1 - ms_2$—that is, where $\mathcal{P}'$ has no outgoing transitions. Since $\mathcal{P}'$ cannot deadlock in the absence of faults, we remove such deadlock states recursively to construct S' (lines 6 and 7 of Algorithm 1). As shown in line 9, if the invariant becomes an empty set after reconstruction, we cannot find an *FF* multitolerant program $\mathcal{P}'$. If the invariant is not empty, we remove transitions that start in S' and terminate outside S' (i.e., violate the closure of S'). Notice that the removal of such transitions does not introduce any deadlock states in S' because each remaining state in S' has at least one outgoing transition to a nondeadlocked state in S'.

ALGORITHM 1: Add_FF_Weakmulti

```
   Input: $\mathcal{P}$: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 $\mathcal{P}'$ with invariant $S'$ that is multitolerant to $f_1$
           and $f_2$
1  $ms_1 := \{s_0 \; : \; \exists s_1, \; s_2, \; \dots s_n \; : (\forall j \; : \; 0 \le j < n \; : \; (s_j, \; s_{j+1}) \in f_1) \wedge \; (s_{n-1}, s_n)$ violates $spec\}$;
2  $ms_2 := \{s_0 \; : \; \exists s_1, \; s_2, \; \dots s_n \; : (\forall j \; : \; 0 \le j < n \; : \; (s_j, \; s_{j+1}) \in f_2) \wedge \; (s_{n-1}, s_n)$ violates $spec\}$;
3  $mt := \{(s_0, s_1) \; : \; ((s_1 \; \in ms_1 \cup ms_2) \; \vee \; (s_0, \; s_1)$ violates $spec)\}$;
4  $S' := S - ms_1 - ms_2$;
5  $\mathcal{P}_1 := \mathcal{P} - mt$;
6  while $\exists s_0 : s_0 \in S' : (\forall s_1 : s_1 \in S' : (s_0, s_1) \notin \mathcal{P}_1)$ do
7  |  $S' := S' - \{s_0\}$;
8  end
9  if $(S' = \{\})$ declare no multitolerant program $\mathcal{P}'$ exists return $\emptyset, \emptyset$;
10 $\mathcal{P}' := \{(s_0, \; s_1) | (s_0, s_1) \in \mathcal{P}_1, s_0 \in S', s_1 \in S'\}$;
   // $\mathcal{P}'$ only specifies transitions inside invariant $S'$;
   // $\mathcal{P}'$ can be modified to include any subset of $\{(s_0, s_1) | s_0 \notin S' \wedge (s_0, s_1) \notin mt\}$;
   /* $T$= Reachable $(S', \mathcal{P} \cup f_1)$ $\cup$ Reachable $(S', \mathcal{P} \cup f_2)$, i.e., $T$ be states reached by
      starting from $S'$ and using transitions of $\mathcal{P} \cup f_1$ (respectively, $\mathcal{P} \cup f_2$);    */
   // $\mathcal{P}'$ can include any subset of $\{(s_0, s_1) | s_0 \notin T\}$;
11 return $\mathcal{P}', S'$;
```

THEOREM 5.1. *The algorithm* Add_FF_Weakmulti *is sound and complete.*

PROOF. To show the soundness of our algorithm, we need to show that constraints $C1$, $C2$, and $C3$ of Problem 3.1 are satisfied:

(1) $S' \subseteq S$. By construction, S' is obtained by removing zero or more states in S. Thus, $C1$ is trivially satisfied.
(2) $(s_0, s_1) \in \mathcal{P}' \wedge s_0 \in S' \Rightarrow (s_0, s_1) \in \mathcal{P}$. By construction, $\mathcal{P}'$ does not have any new transitions in the absence of faults. Therefore, $C2$ is trivially satisfied.
(3) $\mathcal{P}'$ is FF multitolerant to *spec* from S'. Consider a computation c of $\mathcal{P}'$ that starts from a state in S'. From 1, c starts in a state in S, and from 2, c is a computation of $\mathcal{P}$. It follows that c satisfies *spec*. Hence, every computation of $\mathcal{P}'$ that starts from a state in S' is in *spec*—in other words, $\mathcal{P}'$ satisfies *spec* from S'. We discuss the following two cases:
    (a) *Failsafe f_1-tolerance to spec from S'*. We let the fault span T_1 be the set of states reached in any computation of $\psi_{\mathcal{P}'} \cup f_1$ that starts from a state in S'. Consider a computation prefix c of $\psi_{\mathcal{P}'} \cup f_1$ that starts from a state in T_1. From the definition of T_1 there exists a computation prefix c' of $\psi_{\mathcal{P}'} \cup f_1$ such that c is a suffix of c' and c' starts from a state in S'. If c' violates the safety of *spec*, then there exists a prefix of c', say $\langle s_0, \; s_1, \dots, \; s_n\rangle$, such that $\langle s_0, \; s_1, \dots, s_n\rangle$ violates the safety of *spec*. Let $\langle s_0, \; s_1, \dots, \; s_n\rangle$ be the smallest such prefix; it follows that

$(s_{n-1},\ s_n)$ violates the safety of *spec*, and hence $(s_{n-1},\ s_n) \in mt$. By construction, $\mathcal{P}'$ does not contain any transition in mt_1. If $(s_{n-1},\ s_n)$ is a transition of f_1, then $s_{n-1} \in ms_1$ and $(s_{n-2},\ s_{n-1}) \in mt_1$, and hence $(s_{n-2},\ s_{n-1})$ is a transition of f_1. By induction, if $\langle s_0,\ s_1, \ldots, s_n\rangle$ violates the safety of *spec*, $s_0 \in ms_1$, which is a contradiction since $s_0 \in S'$ and $S' \cap ms_1 = \emptyset$. Thus, each prefix of c' maintains *spec*. Since c is a suffix of c', each prefix of c also maintains *spec*. Thus, $\psi_{\mathcal{P}'} \cup f_1$ maintains *spec* from T_1.

(b) *Failsafe f_2-tolerance to spec from S'*. The argument is similar to part (3a).

Proof of completeness. Now we show that if an FF multitolerant program can be designed for the given fault-intolerant program, then Add_FF_Weakmulti will not declare failure. Let program $\mathcal{P}''$ and predicate S'' solve Problem 3.1. Clearly, $S'' \cap ms_1 = \emptyset$; if $s_0 \in (S'' \cap ms_1)$, then the execution of faults alone from s_0 can violate the safety of *spec*. It follows that $S'' \subseteq (S - ms_1)$. Likewise, $S'' \subseteq (S - ms_2)$. Moreover, $\mathcal{P}''|S''$ cannot include any transitions in mt; if $\mathcal{P}''|S''$ contains a transition in mt, then the execution of these transitions can violate the safety of *spec*. Thus, $\mathcal{P}''|S'' \subseteq (\mathcal{P} - mt)$. Finally, every computation of $\mathcal{P}''$ that starts in a state in S'' must be an infinite computation, if it were to be in *spec*. It follows that there exists a subset of S such that all computations of $\mathcal{P} - mt$ within that subset are infinite. Our algorithm declares that no solution for the Problem 3.1 exists only when there is no subset of $S - ms_1 - ms_2$ such that all computations of $\mathcal{P} - mt$ within that subset are infinite. It follows that our algorithm declares that no *FF* multitolerant program exists only if the answer to Problem 3.2 is false. □

Remark 5.1. Algorithm Add_FF_Weakmulti can be extended to design a multitolerant program that is subject to three or more fault classes. Toward this, we specify ms_3 for the third fault class and ms_i for the i^{th} fault class. Then, we calculate mt like line 3 in Algorithm Add_FF_Weakmulti to specify these transitions that lead to state in $\bigcup_{i=1,\ldots,n} ms_i$ (n is the number of fault classes). Besides, we need to recalculate invariant S' by removing states in $\bigcup_{i=1,\ldots,n} ms_i$. The remaining steps are similar to Algorithm Add_FF_Weakmulti.

5.1. Application of Add_FF_Weakmulti

This section presents a practical example to demonstrate how the algorithm Add_FF_Weakmulti facilitates automated synthesis of an FF multitolerant disk storage system. Specifically, we apply the Add_FF_Weakmulti algorithm in Section 5 to the fault-intolerant version of the disk storage system.

Fault-intolerant stable disk storage. The stable disk storage (SDS) program (adapted from Bernardeschi et al. [2000] and Ebnenasir and Kulkarni [2011]) has a controller that manages two sectors (i.e., Sectors 0 and 1). The controller is in a loop of selecting and activating a sector for a read/write operation. The controller and the sectors have the following state variables (which determine its state space): ctrlState $\in \{0, 1\}$ captures the state of the controller, where 0 represents the state of selecting a sector and 1 means issuing a command. The variable secNum $\in \{-1, 0, 1\}$ contains the ID of the selected sector, where –1 denotes that no sector has been selected yet, 0 represents Sector 0 and 1 for Sector 1. The controller activates the sectors by the binary variables $\mathsf{activateSec}_0$ and $\mathsf{activateSec}_1$, indicating whether a sector is activated or not. The SDS program has a variable $\mathsf{x}_i \in \{-1, 0, 1\}$ representing the bit that is read/written in sector i, where –1 represents a damaged bit and $i = 0, 1$. We represent the communication channel between the sector i and the controller by the binary variable c_i. This channel is used to read/write the bit x_i from/to sector i, where $i = 0, 1$. The controller has two command signals, denoted by the binary variable op_i, for $i = 0, 1$. A read operation on x_i is represented by $\mathsf{op}_i = 0$, and otherwise a write operation on x_i. The following guarded commands represent the set of transitions of the SDS program (adapted from Ebnenasir and Kulkarni [2011]):

$$
\begin{array}{lll}
C_1: & (\text{ctrlState} = 0) \wedge (\text{secNum} = -1) \quad \longrightarrow & \text{secNum} := 0|1; \\
C_2: & (\text{ctrlState} = 0) \wedge (\text{secNum} \neq -1) \wedge & \\
 & ((\text{activateSec}_0 = 0) \vee (\text{activateSec}_1 = 0)) \quad \longrightarrow & \text{ctrlState} := 1; \\
C_3: & (\text{ctrlState} = 1) \wedge (\text{secNum} = 0) \wedge (\text{activateSec}_0 = 0) & \\
 & \longrightarrow & \text{ctrlState} := 0; \\
 & & \text{op}_0 := 0; \\
 & & \text{activateSec}_0 := 1; \\
C_4: & (\text{ctrlState} = 1) \wedge (\text{secNum} = 0) \wedge (\text{activateSec}_0 = 0) & \\
 & \longrightarrow & \text{ctrlState} := 0; \\
 & & \text{op}_0 := 1;\ \text{c}_0 := 0|1; \\
 & & \text{activateSec}_0 := 1; \\
C_5: & (\text{ctrlState} = 1) \wedge (\text{secNum} = 1) \wedge (\text{activateSec}_1 = 0) & \\
 & \longrightarrow & \text{ctrlState} := 0; \\
 & & \text{op}_1 := 0; \\
 & & \text{activateSec}_1 := 1; \\
C_6: & (\text{ctrlState} = 1) \wedge (\text{secNum} = 1) \wedge (\text{activateSec}_1 = 0) & \\
 & \longrightarrow & \text{ctrlState} := 0; \\
 & & \text{op}_1 := 1;\ \text{c}_1 := 0|1; \\
 & & \text{activateSec}_1 := 1; \\
C_7: & (\text{ctrlState} = 1) \wedge (\text{secNum} = -1) \quad \longrightarrow & \text{ctrlState} := 0;
\end{array}
$$

When the controller receives a request for performing an operation with either Sector 0 or Sector 1, action C_1 nondeterministically assigns 0 or 1 to secNum (denoted by the vertical bar |). Then, action C_2 changes the state of the controller to the state of issuing commands. Using action C_3 (respectively, C_5), the controller sends a read command to Sector 0 (respectively, Sector 1), whereas action C_4 (respectively, C_6) issues a write command for Sector 0 (respectively, Sector 1). After the execution of actions S_{i1}, S_{i2}, and S_{i3}, where $i = 0, 1$, by the selected sector, action C_7 changes the state of the controller to the sector selection mode. Actions S_{i1} and S_{i2} illustrate how sector i performs a read operation, and action S_{i3} writes the contents of the channel c_i on x_i:

$$
\begin{array}{ll}
S_{i1}: & (\text{op}_i = 0) \wedge (\text{x}_i = 0) \wedge (\text{SecNum} = i) \wedge (\text{activateSec}_i = 1) \\
 & \qquad \longrightarrow \ \text{c}_i := 0;\ \text{secNum} := -1;\ \text{activateSec}_i := 0; \\
S_{i2}: & (\text{op}_i = 0) \wedge (\text{x}_i = 1) \wedge (\text{SecNum} = i) \wedge (\text{activateSec}_i = 1) \\
 & \qquad \longrightarrow \ \text{c}_i := 1;\ \text{secNum} := -1;\ \text{activateSec}_i := 0; \\
S_{i3}: & (\text{op}_i = 1) \wedge (\text{SecNum} = i) \wedge (\text{activateSec}_i = 1) \\
 & \qquad \longrightarrow \ \text{x}_i := \text{c}_i;\ \text{secNum} := -1;\ \text{activateSec}_i := 0;
\end{array}
$$

The specification of the stable disk storage program. Intuitively, the safety specification of the SDS program includes the following constraints: (1) a read operation on x_i should return the last value written on x_i; (2) if the value of a bit is damaged, then reading it returns 0; (3) after a write operation on x_i, the condition $\mathsf{x}_i = \mathsf{c}_i$ must hold; (4) a write operation on a damaged bit has no effect and leaves the value of that bit unchanged; and (5) during a write on a corrupted bit, the other sector must not be selected/activated. Formally, we capture the safety requirements of SDS in a parameterized form for sector i as follows ($i = 0, 1$). (Recall that we represent safety specifications as a set of bad transitions that must not appear in program computations.)

$$\begin{aligned}\text{safety}_{SDS} = \{(s_0, s_1) \mid & ((\text{op}_i(s_0) = 0) \wedge (\text{c}_i(s_1) \neq \text{x}_i(s_0))) \ \vee \\ & ((\text{x}_i(s_0) = -1) \wedge (\text{op}_i(s_0) = 0) \wedge (\text{c}_i(s_1) \neq 0)) \ \vee \\ & ((\text{op}_i(s_0) = 1) \wedge (\text{x}_i(s_0) \neq -1) \wedge (\text{x}_i(s_1) \neq \text{c}_i(s_1))\,) \ \vee \\ & ((\text{op}_i(s_0) = 1) \wedge (\text{x}_i(s_0) = -1) \wedge (\text{x}_i(s_0) \neq \text{x}_i(s_1)) \ \vee \\ & ((\text{op}_i(s_0) = 1) \wedge (\text{x}_i(s_0) = -1) \wedge (\text{secNum}(s_0) = i \wedge \text{secNum}(s_1) \neq i)\,)\,\}\end{aligned}$$

The liveness specification of the SDS program requires deadlock freedom starting from any state in the invariant I_{SDS}, where

$$\begin{aligned}I_{SDS} = \{s \mid & ((x_0(s) \neq -1) \wedge (x_1(s) \neq -1)) \wedge ((\text{ctrlState}(s) \neq 1) \vee (\text{secNum}(s) \neq -1)) \ \wedge \\ & ((\text{activateSec}_0(s) \neq 1) \vee (\text{secNum}(s) = 0)) \ \wedge \ ((\text{activateSec}_1(s) \neq 1) \vee (\text{secNum}(s) = 1))\,\}\end{aligned}$$

The invariant I_{SDS} defines the set of states where x_0 and x_1 are not damaged, and if the controller is in the state of issuing a command (i.e., $\mathsf{ctrlState} = 1$), then a sector must have been selected (i.e., $\mathsf{secNum} \neq -1$). Moreover, if sector i has been activated, then the sector number matches with the activation command.

Faults affecting the stable disk storage program. Two classes of faults perturb the SDS program, namely permanent damage and transient faults. The permanent faults (represented later by action F_p) may permanently damage the contents of a bit of information x_i in a sector by assigning –1 to x_i. The actions F_{t_i} (shown later), where $i = 0, 1$, model the effect of transient faults that may nondeterministically perturb the sector selection/activation commands—in other words, arbitrarily change the values of secNum and $\mathsf{activateSec}_i$ to 0 or 1:

$$\begin{aligned}F_p :\ & (\text{x}_i \neq -1) && \longrightarrow\ \text{x}_i := -1; \\ F_{t_i} :\ & (\text{ctrlState} = 0) \ \wedge \ (\text{SecNum} \neq -1) \ \wedge \ (\text{activateSec}_i = 1) && \\ & && \longrightarrow\ \text{secNum} := 0 \mid 1;\ \text{activateSec}_i := 0 \mid 1;\end{aligned}$$

5.1.1. Failsafe-Failsafe Multitolerant Stable Disk Storage. In this section, we demonstrate how we use the algorithm Add_FF_Weakmulti to generate an FF multitolerant version of the SDS program. We note that this is a case where failsafe fault tolerance cannot be designed if both permanent and transient faults occur. Specifically, if the transient faults occur while a write operation is being performed on a corrupted bit, then the last constraint of the safety specification safety_{SDS} is directly violated by a sequence of fault transitions (of both types). For this reason, we design an FF program that guarantees failsafe fault tolerance for each type of fault separately.

We follow the steps of the algorithm Add_FF_Weakmulti to observe how an FF program is generated. Notice that the occurrence of permanent faults does not directly violate safety_{SDS}; safety_{SDS} may be violated *after* permanent faults damage a bit x_i and a write operation is performed on x_i. Thus, $ms_1 = \emptyset$. Moreover, the perturbation of the variables secNum and $\mathsf{activateSec}_i$ does not violate any constraint of safety_{SDS} (i.e., $ms_2 = \emptyset$). As a result, no states are removed from S, which means that $S' = S = I_{SDS}$ and the while loop in line 6 of Add_FF_Weakmulti will not remove any states from I_{SDS}. Since ms_1 and ms_2 are empty, mt becomes equal to safety_{SDS}. Thus, in the presence of permanent faults, we need to prevent any write operation on damaged bits, which results in the following revised actions for the sectors:

$$\begin{aligned}S'_{i1} :\ & (\text{op}_i = 0) \ \wedge \ ((\text{x}_i = 0) \vee (\mathbf{x}_i = -1)) \ \wedge \ (\text{SecNum} = i) \ \wedge \ (\text{activateSec}_i = 1) \\ & \qquad \longrightarrow\ \text{c}_i := 0;\ \text{secNum} := -1;\ \text{activateSec}_i := 0; \\ S'_{i2} :\ & (\text{op}_i = 0) \ \wedge \ (\text{x}_i = 1) \ \wedge \ (\text{SecNum} = i) \ \wedge \ (\text{activateSec}_i = 1) \\ & \qquad \longrightarrow\ \text{c}_i := 1;\ \text{secNum} := -1;\ \text{activateSec}_i := 0; \\ S'_{i3} :\ & (\text{op}_i = 1) \ \wedge \ (\text{SecNum} = i) \ \wedge \ (\text{activateSec}_i = 1) \wedge (\mathbf{x}_i \neq -1) \\ & \qquad \longrightarrow\ \text{x}_i := \text{c}_i;\ \text{secNum} := -1;\ \text{activateSec}_i := 0;\end{aligned}$$

The guard of the action S_{i1} has been weakened in action S'_{i1} to include transitions originating outside I_{SDS} that return 0 if the value of x_i is –1 and a read operation has taken place.[1] The constraint ($\mathsf{x}_i \neq -1$) in action S'_{i3} guarantees that a write operation will not take place on a damaged bit. Notice that in the presence of transient faults alone, this set of revised actions along with the actions of the controller guarantee failsafe fault tolerance to transient faults. Nonetheless, transient faults may cause the program to deadlock in a state where secNum and activateSec variables do not match; for example, secNum is set to 1, representing the selection of Sector 1, but $\mathsf{activateSec}_1$ is set to 0, and $\mathsf{activateSec}_0$ is set to 1. Since failsafe fault tolerance does not require recovery from such states to I_{SDS}, it is acceptable for a failsafe program to halt outside its invariant (without violating safety). Therefore, the set of revised actions provides FF multitolerance unless both faults occur simultaneously.

6. COMPLEXITY ANALYSIS OF MM MULTITOLERANCE

In this section, we investigate Problem 3.2 for cases where we want to add multitolerance for $f_\delta = \{\langle f_{m1}, masking\rangle,\ \langle f_{m2}, masking\rangle\}$. We find a surprising result that the *MM* (*Masking-Masking*) multitolerant synthesis problem is NP-complete (in the size of the program state space), even though, under the restrictions imposed in Ebnenasir and Kulkarni [2011], this problem can be solved in P.

Before we present the formal proof, we give an intuition behind this complexity. Consider the case where there exists a transition $(s_1,\ s_2)$ of f_{m2} that violates the safety specification. We have the following two options: (1) ensure that s_1 is unreachable in the computations of $\mathcal{P} \cup f_{m2}$, and (2) 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 values of Boolean variables in the SAT formula. This allows us to reduce the SAT problem to the *MM* multitolerance synthesis problem.

THEOREM 6.1. *The problem of synthesizing MM multitolerant programs from their fault-intolerant version is NP-complete.*

PROOF. Given a program $\mathcal{P}$, with its invariant S, its specification *spec*, and two classes of faults f_{m1} and f_{m2}, we prove that the decision Problem 3.2 is NP-hard when $f_\delta = \{\langle f_{m1},\ masking\rangle,\ \langle f_{m2},\ masking\rangle\}$. Illustrating the NP membership of Problem 3.2 is straightforward and hence omitted.

Mapping. Now we present a polynomial-time mapping from an instance of the SAT problem to a corresponding instance $\langle \mathcal{P}, S, spec, f_{m1}, f_{m2}\rangle$ of Problem 3.2. An instance of the SAT problem is specified in terms of a set of literals $x_1, x_2, \ldots, x_n$ and $\neg x_1, \neg x_2, \ldots, \neg x_n$, where x_i and $\neg x_i$ are complements of each other. The SAT formula is of the form $\phi = C_1 \wedge C_2 \wedge C_3 \wedge \ldots \wedge C_k$, where each clause C_i is a disjunction of several literals. Then we show that the given SAT formula is satisfiable iff there exists a solution for the mapped instance of Problem 3.2. We construct the mapped instance as follows (Figure 1).

The invariant and state space of the fault-intolerant program, $\mathcal{P}$. The state space of $\mathcal{P}$ is as follows:

—We introduce a state s. This is the only state in the invariant S.
—For each propositional variable x_i, $1 \le i \le n$, 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, $1 \le r \le k$, we introduce states w_r and z_r.

[1]Please read the comment on line 10 of Add_FF_Weakmulti.

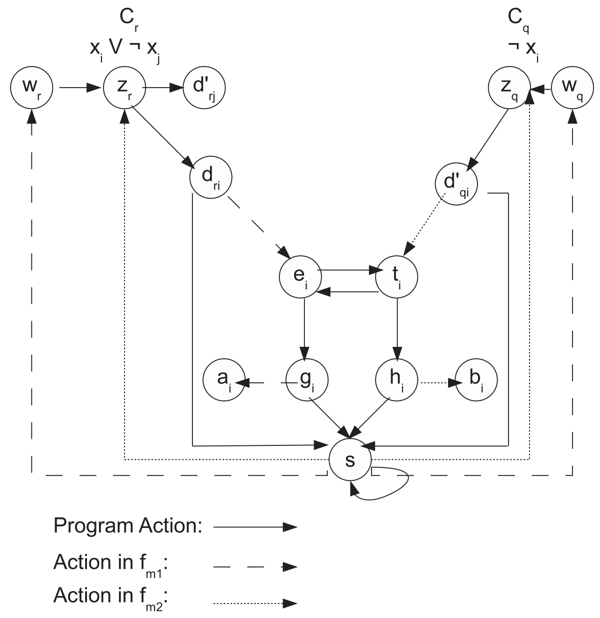

Fig. 1. The states and the transitions corresponding to a literals x_i and clauses C_r and C_q in the SAT formula.

—If clause C_r includes literal x_i, we introduce a state d_{ri}. If clause C_r includes literal $\neg x_i$, we introduce a state d'_{ri}.

The transitions of fault-intolerant program $\mathcal{P}|S$ include only one self-loop transition (s, s).

The transitions of f_{m1} and f_{m2} are as follows:

—For each clause C_r, we include the fault transition (s, w_r) in f_{m1} and the fault transition (s, z_r) in f_{m2}.
—If the clause C_r includes the literal x_i, then we include the fault transition (d_{ri}, e_i) in f_{m1}.
—If the clause C_r includes the literal $\neg x_i$, then we include the fault transition (d'_{ri}, t_i) in f_{m2}.
—For each propositional variable x_i and its complement $\neg x_i$, we include the 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 $\mathcal{P}$ is as follows:

—Transitions (g_i, a_i) and (h_i, b_i) violate safety.
—Transitions (s, s), (s, w_r), (s, z_r), (d_{ri}, e_i), and (d'_{ri}, t_i) do not violate safety.
—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_{ri}), (z_r, d'_{ri}), (e_i, t_i), (t_i, e_i), (e_i, g_i), (t_i, h_i), (g_i, s), and (h_i, s).
—All transitions except those identified previously ((z_r, w_r), (z_r, s), etc.) violate safety specification.

Reduction. Now, we show that the given SAT formula is satisfiable iff the answer to Problem 3.2 for the mapped instance is affirmative where $f_\delta = \{\langle f_{m1}, masking\rangle, \langle f_{m2}, masking\rangle\}$:

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

The invariant of $\mathcal{P}'$ is the same as the invariant of $\mathcal{P}$ (i.e., $\{s\}$). We derive the transitions of the multitolerant program $\mathcal{P}'$ 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_{ri}) and (d_{ri}, 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'_{ri}) *and* (d'_{ri}, s).

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

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

—*(In the absence of faults)* $\mathcal{P}'|S = \mathcal{P}|S$. Thus, $\mathcal{P}'$ satisfies *spec* in the absence of faults.
—*Masking f_{m1}-tolerance.* If the faults from f_{m1} occur, then the program can be perturbed to state w_r, $1 \le r \le k$. From w_r, $\mathcal{P}'$ 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, $\mathcal{P}'$ can recover to s using the two sequences of transitions, $\langle(z_r, d_{ri}), (d_{ri}, s)\rangle$, or $\langle(z_r, d_{ri}), (d_{ri}, e_i), (e_i, t_i), (t_i, h_i), (h_i, s)\rangle$. In the latter case, $\mathcal{P}'$ can recover to s using exactly one sequence of transitions, $\langle(z_r, d'_{ri}), (d'_{ri}, s)\rangle$. Note that if x_i is true, then $\mathcal{P}'$ cannot reach g_i, from where it can violate safety specification. Thus, any computation of $\mathcal{P}' \cup f_{m1}$ eventually reaches a state in the invariant. Moreover, from z_r, every computation of $\mathcal{P}' \cup f_{m1}$ does not violate the safety specification. Based on the preceding discussion, $\mathcal{P}'$ is masking tolerant to f_{m1}.
—*Masking f_{m2}-tolerance.* The argument is similar to the one showing that $\mathcal{P}'$ is masking tolerant to f_{m1}.

($\Longleftarrow$) Second, we show that if there exists a multitolerant program that solves the instance of the synthesis Problem 3.2, then the given SAT formula is satisfiable. Let $\mathcal{P}'$ be the multitolerant program derived from the fault-intolerant program $\mathcal{P}$. The invariant of $\mathcal{P}'$, 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 states added in the instance of the synthesis problem are w_r and z_r. Note that w_r can be reached from s by a transition in f_{m1}. Hence, $\mathcal{P}'$ must include the transition (w_r, z_r). Thus, z_r is reached in the computation of $\mathcal{P}' \cup f_{m1}$. Hence, $\mathcal{P}'$ must recover to s from z_r without violating *spec*. Therefore, for some i, $\mathcal{P}'$ has to have a transition of the form (z_r, d_{ri}) or (z_r, d'_{ri}). If $\mathcal{P}'$ includes (z_r, d_{ri}), then we assign x_i the truth value *true*. Likewise, if $\mathcal{P}'$ includes (z_r, d'_{ri}) for some i, then 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 values assigned to all literals are consistent—in other words, it is not the case that x_i is assigned *true* in one clause and *false* in another clause. We show this by a proof by contradiction. If x_i is assigned *true* in clause C_r and *false* in clause C_q, then $\mathcal{P}'$ includes both transitions (z_r, d_{ri}) and (z_q, d'_{qi}). Now, from d_{ri}, the program can reach e_i by the occurrence of f_{m1}

alone. Hence, the program $\mathcal{P}'$ cannot include the transition (e_i, g_i), as including this transition will allow the program to reach g_i in a computation of $\mathcal{P}' \cup f_{m1}$ and violate safety by executing (g_i, a_i). Likewise, $\mathcal{P}'$ can reach t_i by the occurrence of f_{m2} alone. Hence, $\mathcal{P}'$ cannot include the transition (t_i, h_i). If both transitions (e_i, g_i) and (t_i, h_i) are not included, then $\mathcal{P}'$ cannot recover from e_i to the state in the invariant. This contradicts the assumption that $\mathcal{P}'$ is masking f_{m1}-tolerant. Thus, the truth value assignment to all literals is consistent. □

6.1. A Heuristic for MM Multitolerance

In this section, we present a sound (but incomplete) algorithm that adds MM multitolerance to a given program $\mathcal{P}$ that is subject to two classes of faults, $f_\delta = \{\langle f_1, \mathit{masking}\rangle, \langle f_2, \mathit{masking}\rangle\}$, in polynomial time. Our algorithm Add_MM_Weakmulti takes program actions, faults, and invariant and safety specification as input and generates an MM multitolerant program. The basic idea of Add_MM_Weakmulti is to first construct the corresponding FF multitolerant program that ensures safety. Then we use the fault span of the FF multitolerant program to add recovery. Specifically, let T_1 and T_2 be the fault spans in the presence of f_1 and f_2, respectively. We ensure that every path from T_1 reaches a state in S_1, and likewise, every path from T_2 reaches a state in S_1. Additionally, we ensure that T_1 (respectively, T_2) remains closed in transitions of f_1 (respectively, f_2) during this revision process.

Given is a program $\mathcal{P}$ with its state predicate S and its specification *spec*. Let $\mathcal{P}'$ be the synthesized program with invariant S' that is multitolerant to f_1 and f_2. By definition, $\mathcal{P}'$ is masking f_1-tolerant from S' for *spec* if only f_1 occurs, and masking f_2-tolerant from S' for *spec* if only f_2 occurs. To this end, line 1 of Algorithm 2 identifies ms_1, a set of states from where execution of one or more f_1 transitions violates safety. In line 2, we identify ms_2, which is a set of states from where execution of one or more f_2 transitions violates safety. Next, we compute mt, a set of transitions that reach $ms_1 \cup ms_2$ or those that violate *spec*. By calling Algorithm 1 in line 6 of Algorithm 2, we obtain $\mathcal{P}_1$ with invariant S_1, where $\mathcal{P}_1$ is FF multitolerant to f_1 and f_2. In the loop of lines 7 through 12, we reconstruct transitions to ensure that $\langle S_1, \mathcal{P}_1 \cup f_1\rangle$ maintains *spec* from T_1 and $\langle S_1, \mathcal{P}_1 \cup f_2\rangle$ maintains *spec* from T_2 on line 9. To guarantee that from each state outside S_1 there is a path that reaches a state in S_1 and there are no cycles in states outside S_1, we update $\mathcal{P}_1$ by calling the algorithm Ensure_Recovery in lines 10 and 11. Algorithm 3 captures the details of Ensure_Recovery. Specifically, Ensure_Recovery is defined in such a way that from each state outside S_1 there is a path that reaches a state in S_1, and there are no cycles in states outside S_1. As shown in line 13 of Algorithm 2, if the invariant becomes an empty set after reconstruction, we cannot find an MM multitolerant program $\mathcal{P}'$. Details are as shown in Algorithm 2.

THEOREM 6.2. *The algorithm* Add_MM_Weakmulti *is sound.*

PROOF. To show the soundness of our algorithm, we need to show that constraints $C1$, $C2$, and $C3$ of Problem 3.1 are satisfied:

(1) $S_1 \subseteq S$. By the correctness of Add_FF_Weakmulti, S_1 obtained at line 6 satisfies $C1$. Since the following steps do not add any state to S_1, $C1$ is preserved by the final program $\mathcal{P}_1$.
(2) $(s_0, s_1) \in \mathcal{P}_1 \wedge s_0 \in S_1 \Rightarrow (s_0, s_1) \in \mathcal{P}$. By the correctness of Add_FF_Weakmulti, $\mathcal{P}_1$ obtained at line 6 of Algorithm 2 satisfies $C2$. Since the remaining steps do not add any transition to $\mathcal{P}_1$, $C2$ is preserved by the final program $\mathcal{P}_1$.
(3) $\mathcal{P}_1$ is MM fault-tolerant to *spec* from S_1. Consider a computation c of $\mathcal{P}_1$ that starts from a state in S_1. From part (1) of this proof, c starts in a state in S, and from part (2), c is a computation of $\mathcal{P}$. It follows that c satisfies *spec*. Hence, every computation

ALGORITHM 2: Add_MM_Weakmulti

```
Input: $\mathcal{P}$:transitions, $f_1$, $f_2$:faults of two classes that need Masking $f$-tolerance, $S$: state
       predicate, $spec$: safety specification
Output: If successful, a fault-tolerant $\mathcal{P}'$ with invariant $S'$ that is weak multitolerant to $f_1$
        and $f_2$
1  $ms_1 := \{s_0 \ : \ \exists s_1,\ s_2,\ \ldots s_n \ : (\forall j \ : \ 0 \le j < n \ : \ (s_j,\ s_{j+1}) \in f_1) \wedge \ (s_{n-1}, s_n)$ violates $spec\}$;
2  $ms_2 := \{s_0 \ : \ \exists s_1,\ s_2,\ \ldots s_n \ : (\forall j \ : \ 0 \le j < n \ : \ (s_j,\ s_{j+1}) \in f_2) \wedge \ (s_{n-1}, s_n)$ violates $spec\}$;
3  $mt := \{(s_0, s_1) : ((s_1 \in ms_1 \cup ms_2) \vee (s_0, s_1)$ violates $spec)\}$;
4  $T_1 := S - ms_1$;
5  $T_2 := S - ms_2$;
6  $\mathcal{P}_1, S_1 := Add\_FF\_Weakmulti(\mathcal{P},\ f_1,\ f_2, S, spec)$ ; // refer to Algorithm 1
7  repeat
8      $T_1' := T_1, T_2' := T_2$;
9      $\mathcal{P}_1 := \{(s_0, s_1) | (s_0 \in S_1 \Rightarrow (s_0, s_1) \in \mathcal{P}_1) \wedge (s_0 \in T_1 \Rightarrow s_1 \in T_1) \wedge (s_0 \in T_2 \Rightarrow s_1 \in T_2)\} - mt$;
10     $T_1, S_1, \mathcal{P}_1 := Ensure\_Recovery\ (\mathcal{P}_1,\ f_1, T_1, S_1)$ ; // refer to Algorithm 3
11     $T_2, S_1, \mathcal{P}_1 := Ensure\_Recovery\ (\mathcal{P}_1,\ f_2, T_2, S_1)$;
12 until $T_1' = T_1 \wedge T_2' = T_2$;
13 if $S_1 \neq \emptyset$ then return $\mathcal{P}_1, S_1$;
14 else declare failure in generating a multitolerant program $\mathcal{P}'$, return $\emptyset, \emptyset$;
```

ALGORITHM 3: Ensure_Recovery

```
Input: $\mathcal{P}$:transitions, $f$:fault actions, $T$: state predicate, $S$: state predicate
Output: $T$: state predicate, $S$: state predicate, $\mathcal{P}$:transitions
/* Goal: Find $T'$, $\mathcal{P}'$ and $S'$ such that $T' \subseteq T$ , $S' \subseteq S$, $T'$ is closed in $\mathcal{P}' \cup f$,
   every computation of $\mathcal{P}'$ from $T'$ reaches a state in $S'$.                    */
1  $S_1 := S$;
2  repeat
3      $S_1 := S$;
       /* Rank$(s_0) =$ length of the shortest computation prefix of $\mathcal{P}$ from $s_0$ to some
          state in $S$. $Rank(s_0) = \infty$ means $S$ is not reachable from $s_0$.        */
4      $T := T - \{s_0 | Rank(s_0) = \infty\}$;
5      $T := T - \{s_0 | \exists s_1 : (s_0, s_1) \in f, s_0 \in T, s_1 \notin T\}$; // Ensure $T$ is closed in $f$
6      $S := S \wedge T$;
7      while $\exists s_0 : s_0 \in S : (\forall s_1 : s_1 \in S : (s_0, s_1) \notin \mathcal{P})$ do
8          $S := S - \{s_0\}$ ;
9      end
10 until $S_1 = S$;
11 $\mathcal{P}_1 := removeCycles(\mathcal{P}, S, T)$; // returns $\mathcal{P}_1$ such that $\mathcal{P}_1 \subseteq \mathcal{P}, \mathcal{P}_1|S = \mathcal{P}|S, \mathcal{P}_1|(T - S)$
   is acyclic,
   /* and $\forall s_0 : s_0 \in T : S$ is reachable from $s_0$ in $\mathcal{P}_1$               */
   /* There are several possible implementations and any one of them is acceptable.
      One possible implementation is to rank each state based upon the shortest
      path from that state to a state inside $S$, and then remove these transitions
      that do not decrease the rank.                                                  */
12 return $\mathcal{P}_1, S, T$;
```

of $\mathcal{P}_1$ that starts from a state in S_1 is in *spec*—that is, $\mathcal{P}_1$ satisfies *spec* from S_1. Next we discuss the following two cases:

(a) *Masking f_1-tolerance to spec from S_1*. To show this, we need to show the following three properties:
 —T_1 is closed in $\mathcal{P}_1 \cup f_1$. Closure of T_1 in $\mathcal{P}_1$ is by construction. Regarding closure of T_1 in f_1, observe that there is no change in the last iteration of the loop in lines 7 through 12 of Algorithm 2. Thus, T_1 is closed in f_1.

—$\mathcal{P}_1 \cup f_1$ maintains *spec* from T_1. We let the fault span T_1 to be the set of states reached in any computation of $\mathcal{P}_1 \cup f_1$ that starts from a state in S_1. Consider a computation prefix c of $\mathcal{P}_1 \cup f_1$ that starts from a state in T_1. From the definition of T_1, there exists a computation prefix c' of $\mathcal{P}_1 \cup f_1$ such that c is a suffix of c' and c' starts from a state in S_1. If c' violates the safety of *spec*, then there exists a prefix of c', say $\langle s_0,\ s_1, \ldots,\ s_n\rangle$, such that $\langle s_0,\ s_1, \ldots,\ s_n\rangle$ violates the safety of spec. Let $\langle s_0,\ s_1, \ldots,\ s_n\rangle$ be the smallest such prefix; it follows that $(s_{n-1},\ s_n)$ violates the safety of *spec* and hence $(s_{n-1},\ s_n)\ \in mt$. By construction, $\mathcal{P}_1$ does not contain any transition in mt (see line 9 of Algorithm 2). Thus, (s_{n-1}, s_n) is a transition of f_1. If (s_{n-1}, s_n) is a transition of f_1, then $s_{n-1}\ \in ms_1$ and $(s_{n-2}, s_{n-1})\ \in mt_1$ and hence (s_{n-2}, s_{n-1}) is a transition of f_1. By induction, if $\langle s_0, s_1, \ldots, s_n\rangle$ violates the safety of *spec*, $s_0 \in ms_1$, which is a contradiction since $s_0 \in S_1$ and $S_1 \cap ms_1 = \emptyset$ (guaranteed by line 6 of Algorithm 2 since the following steps do not add any state in S_1). Thus, each prefix of c' maintains *spec*. Since c is a suffix of c', each prefix of c also maintains *spec*. Thus, $\mathcal{P}_1 \cup f_1$ maintains *spec* from T_1.

—Every computation of $\mathcal{P}$ that starts from a state in T_1 eventually reaches a state of S_1. The Ensure_Recovery algorithm only updates $\mathcal{P}_1$ by removing some transitions from $\mathcal{P}_1$ in steps 4, 5, and 10 of Algorithm 3. By construction, Ensure_Recovery removes all of the deadlock states in steps 7 and 8 recursively. In addition, the function RemoveCycles is defined in such a way that from each state outside S_1 there is a path that reaches a state in S_1, and there are no cycles in states outside S_1.

(b) Masking f_2-tolerant to *spec* from S_1. The argument is similar to part (3a). □

Remark 6.1. Note that Add_MM_Weakmulti is sound but not complete. One reason is because the function removeCycles has several possible implementations, and the choice of transitions removed in removeCycles (line 11 of Algorithm 3) for ensuring recovery in the presence of f_1 can prevent recovery in the presence of f_2. If one were to consider all possible choices of removeCycles, then the time complexity would become exponential in the state space.

Remark 6.2. Algorithm Add_MM_Weakmulti can be extended to design a multitolerant program that is subject to three or more fault classes. Toward this, we specify ms_3 for the third fault class and the corresponding ms_i for the i^{th} fault class. Then we calculate mt like line 3 in Algorithm Add_MM_Weakmulti to specify these transitions that lead to state in $\bigcup_{i=1,\ldots,n} ms_i$ (n is the number of fault classes). Moreover, we need to calculate the corresponding T_i for the i^{th} fault class. In particular, Ensure_Recovery (lines 10 and 11 of Algorithm 2) will be repeated for each fault class.

7. COMPLEXITY ANALYSIS OF FM MULTITOLERANCE

In this section, we investigate the synthesis problem for multitolerant programs for the case where the program is subject to two classes of faults, f_1 and f_2, for which failsafe and masking fault-tolerance are required, respectively—that is $f_\delta = \{\langle f_1, \mathit{failsafe}\rangle, \langle f_2, \mathit{masking}\rangle\}$ in Definition 3.1. This synthesis problem is NP-complete. This result is also surprising since the corresponding problem in Ebnenasir and Kulkarni [2011], which adds a requirement that failsafe fault tolerance be provided if both f_1 and f_2 occur simultaneously, is in P.

THEOREM 7.1. *The problem of synthesizing FM multitolerant programs from their fault-intolerant version is NP-complete.*

Proof. Given is a program $\mathcal{P}$, with its invariant S; its specification *spec*; and two classes of faults, f_1 and f_2. Since demonstrating membership to NP is trivial, we only illustrate that the synthesis problem identified in Definition 3.1 is NP-hard when $f_\delta = \{\langle f_1, \mathit{failsafe}\rangle, \langle f_2, \mathit{masking}\rangle\}$.

Mapping. We construct the mapping by changing that part of proof for Theorem 6.1 as follows:

—Replace f_{m1} fault transitions with transitions of f_2.
—Replace f_{m2} fault transitions with transitions of f_1.

Reduction. Now we show that the given SAT formula is satisfiable iff there exists a solution to the FM multitolerant synthesis problem:

($\Longrightarrow$) By the proof of Theorem 6.1, if the given SAT formula is satisfiable, then there exists a program that is masking fault tolerant to f_1 and f_{m1}. In addition, by Definitions 2.13 and 2.14, a program $\mathcal{P}$ that is masking f_1-tolerant from S for *spec* is failsafe f_1-tolerant from S for *spec*. Hence, if the given SAT formula is satisfiable, then there is a solution to the corresponding instance of the FM multitolerance synthesis.
($\Longleftarrow$) This proof is identical to the corresponding proof for Theorem 6.1, and hence we omit it. □

8. COMPLEXITY ANALYSIS OF MN MULTITOLERANCE

In this section, we study the case where a program is subject to two classes of faults, f_1 and f_2, for which masking and nonmasking fault tolerance are required, respectively—that is, $f_\delta = \{\langle f_1, \mathit{masking}\rangle, \langle f_2, \mathit{nonmasking}\rangle\}$ in Definition 3.1. We show that such an *MN* (*Masking-Nonmasking*) 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_δ 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 6, if masking fault tolerance is desired for two or more classes of faults, then the problem is NP-complete.

Given is a program $\mathcal{P}$, with its invariant S and its specification *spec*. Our objective is to synthesize a program $\mathcal{P}'$, with invariant S' that is multitolerant to f_δ. By definition, $\mathcal{P}'$ must be masking f_1-tolerant and nonmasking f_2-tolerant. The algorithm for MN multitolerance utilizes the algorithm Add_Masking (from Kulkarni and Arora [2000]) that adds masking fault tolerance to a single class of faults. Add_Masking returns the synthesized program $\mathcal{P}'$, its invariant S', and its fault span T' such that $\mathcal{P}'$ is masking fault tolerant to S', and T' is the fault span used to prove this in Definition 2.13. Algorithm 4 (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. Thus, Add_MN_Weakmulti first invokes Add_Masking on line 1 with parameters ($\mathcal{P}$, f_1, S, *spec*). As shown in line 2, if the invariant becomes an

ALGORITHM 4: Add_MN_Weakmulti

```
Input: $\mathcal{P}$:transitions, $f_{\delta 1} : \{\langle f_1, \mathit{masking}\rangle, \langle f_2, \mathit{nonmasking}\rangle\}$, $S$: state predicate, spec:
       safety specification
Output: If successful, a fault-tolerant $\mathcal{P}'$ with invariant $S'$ that is multitolerant to $f_1$
        and $f_2$
1 $\mathcal{P}_1, S', T_1$ := Add_Masking($\mathcal{P}, f_1, S, spec$);
2 if ($S' = \{\}$) declare no multitolerant program $\mathcal{P}'$ exists, return $\emptyset, \emptyset$;
3 $\mathcal{P}' := \mathcal{P}_1 | T_1 \cup \{(s_0, s_1) : (s_0, s_1) \in \mathcal{P}, s_0 \notin T_1 \wedge s_1 \in T_1\}$;
4 return $\mathcal{P}', S'$;
```

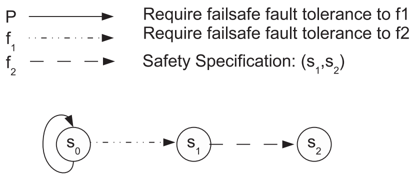
P
f_1
f_2
Require failsafe fault tolerance to f1
Require failsafe fault tolerance to f2
Safety Specification: (s_1,s_2)
s_0
s_1
s_2

Fig. 2. Feasibility of adding FF multitolerance.

empty set after reconstruction in line 1, we cannot find an MN multitolerant program $\mathcal{P}'$. If the invariant is not empty, we include additional recovery transitions to ensure recovery to T_1.

Note. Algorithm Add_MN_Weakmulti is the same as that from Ebnenasir and Kulkarni [2011].

THEOREM 8.1. *The algorithm* Add_MN_Weakmulti *is sound and complete.*

PROOF. By correctness of Add_Masking, $\mathcal{P}_1$ satisfies the constraints of Definition 3.1 for the case where $f_\delta = \{\langle f_1, masking\rangle\}$. Since step 3 does not add or remove any state of S or a transition from $\mathcal{P}|T$, these constraints are preserved by the final program $\mathcal{P}'$. Hence, to complete the proof of this theorem, next we show that $\mathcal{P}'$ is nonmasking f_2-tolerant. By definition of masking fault tolerance, every computation of $\mathcal{P}_1$ that starts in a state in T_1 reaches a state in S'. If f_2 perturbs the program to a state outside T_1, then the recovery transitions added in step 3 will recover the program to a state in T_1 from where it can utilize the recovery paths inside $\mathcal{P}_1$ to reach S'. Thus, $\mathcal{P}'$ is nonmasking f_2-tolerant.

Our algorithm declares that an MN multitolerant program does not exist only when Add_Masking does not find a masking f_1-tolerant program. Hence, completeness of Add_MN_Weakmulti follows from the completeness of Add_Masking. □

9. COMPLEXITY ANALYSIS OF NN MULTITOLERANCE

The algorithm Add_NN_Weakmulti for the NN (*Nonmasking-Nonmasking*) multitolerant problem is identical to Add_MN_Weakmulti, but instead of invoking Add_Masking on line 1 of Algorithm 4, we call Add_Nonmasking(from Kulkarni and Arora [2000]).

THEOREM 9.1. *The algorithm* Add_NN_Weakmulti *is sound and complete.*

PROOF. Since the proof is similar to the proof of algorithm Add_MN_Weakmulti, we omit it. □

10. COMPARISON OF FEASIBILITY OF MULTITOLERANCE

This section studies the limitations and effectiveness of the restricted notion of multitolerance presented in Ebnenasir and Kulkarni [2011] (where the minimum level of fault tolerance is provided when multiple faults occur simultaneously) versus the more general definition of multitolerance proposed in this article. We perform this comparison for all possible levels of multitolerance for two types of faults.

10.1. Feasibility Comparison of FF Multitolerance

In this section, we show that there are instances where adding FF multitolerance is feasible, although doing the same under restrictions imposed by Ebnenasir and Kulkarni [2011] (namely, failsafe fault-tolerance when both faults occur in the same computation) is not. We can show this with a simple example illustrated in Figure 2. In this example, the input is as follows. The state space of the input program is $\{s_0, s_1, s_2\}$, the input program consists of only one transition (s_0, s_0), and its invariant contains only one state s_0. The transition (s_1, s_2) violates safety. The class of fault f_1 includes only one transition (s_0, s_1), and fault f_2 includes only one transition (s_1, s_2).

Clearly, if faults f_1 and f_2 occur in the same computation, then safety can be violated from state s_0, which is the only state in the invariant. Hence, under the restrictions of Ebnenasir and Kulkarni [2011], adding FF multitolerance is not possible. However, without these restrictions, adding FF multitolerance is feasible. In fact, the program $\mathcal{P}$ itself is FF multitolerant.

10.2. Feasibility Comparison of MM and FM Multitolerance

In this section, we show that there are instances where adding MM (respectively, FM) multitolerance is feasible. However, if we consider the restricted version of multitolerance from Ebnenasir and Kulkarni [2011], where masking (respectively, failsafe) fault tolerance must be provided if faults from both classes occur in the same computation, then adding MM (respectively, FM) multitolerance is impossible. To illustrate this, we use the input obtained by mapping the SAT formula as discussed in Section 6. As shown in Section 6, if we begin with an SAT formula that is satisfiable, then the answer to the decision problem for adding MM multitolerance (Problem 3.2) is affirmative. Next we show that for this input, the answer to the decision problem for adding MM multitolerance is impossible if we add the restriction that masking fault tolerance must be provided when faults from both classes occur in the same computation. To show this, observe that in Figure 1, any recovery path to the invariant must go through either g_i or h_i from some i $(1 \le i \le n)$, where n is the number of propositional variables in the instance of the SAT problem in Section 6. If f_1 and f_2 occur in the same computation, safety will be violated when either fault transition (g_i, a_i) or (h_i, b_i) is executed.

We note that using a similar argument, we can show that there are instances where adding FM multitolerance is feasible, although it becomes infeasible if we add restrictions of Ebnenasir and Kulkarni [2011].

10.3. Feasibility Comparison of MN, NN, and FN Multitolerance

Since the algorithm that is used to synthesize the MN/NN multitolerance is the same as that in Ebnenasir and Kulkarni [2011], the synthesis problem of MN/NN multitolerance is unaffected by the restrictions imposed in Ebnenasir and Kulkarni [2011].

Finally, in FN multitolerance, no additional requirements are imposed in Ebnenasir and Kulkarni [2011] when faults from both classes occur in the same computation. This is because the minimum of failsafe and nonmasking fault tolerance is no fault tolerance. Hence, the feasibility of FN multitolerance remains unchanged when one considers the restrictions from Ebnenasir and Kulkarni [2011].

11. RELATED WORK

Automated program synthesis is studied from different perspectives. One approach (e.g., Attie et al. [2004]) focuses on synthesizing fault-tolerant programs from their specification in a temporal logic (e.g., linear temporal logic [Emerson 1990]). The term *synthesis* is also used in the context (e.g., De Niz and Rajkumar [2004], Gu and Shin [2005], Lin et al. [2004], and Hsiung and Lin [2008]) 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 dataflow, and so forth, meet the constraints of an 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 a concrete program. Thus, our approach is desirable when one needs to extend an existing system by adding fault tolerance.

Our work is closely related to the work on controller synthesis [Asarin and Maler 1999; Asarin et al. 1998; Bouyer et al. 2003; D'Souza and Madhusudan 2002] and game theory [De Alfaro et al. 2003; Faella et al. 2002; Jobstmann et al. 2005]. In most control-theoretic approaches, the supervisory control has been studied under the assumption of synchronous execution. 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 Asarin and Maler [1999], Asarin et al. [1998], De Alfaro et al. [2003], and Faella et al. [2002] are EXPTIME-complete. Furthermore, deciding the existence of a controller in Bouyer et al. [2003] and D'Souza and Madhusudan [2002] 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 our work. In addition, the high complexity of these methods is a serious barrier to making synthesis practical (e.g., tool building) for moderate-sized programs. By contrast, our approach concentrates only on specifications needed to express properties of interest, thereby decreasing the complexity of our algorithm. As a result, we have developed software tools [Ebnenasir 2007; Bonakdarpour and Kulkarni 2008] that add fault tolerance to programs with 2^{100} reachable states in less than 1 hour.

The algorithms in Kulkarni and Arora [2002] and Kulkarni and Ebnenasir [2002] have addressed the problem of adding fault tolerance to only one class of fault. Moreover, the algorithms in Ebnenasir and Kulkarni [2011] add an implicit assumption about requirements that have to be satisfied when several faults occur simultaneously. As shown in this article, there are circumstances where these implicit requirements prevent us from synthesizing the desired program even though the multitolerant system can be designed. We have illustrated this with examples in Section 4.

12. DISCUSSION

In this section, we discuss issues related to the way in which we model faults, the practical significance of our proposed work, and some limitations of our approach.

Fault model. In this article, we model the impact of faults on programs as a set of transitions (i.e., a nondeterministic finite-state machine) that perturbs the program state. Designers can identify the classes of faults dependent upon the domain of application and the requirements of the system users. To generate a fault class, first we identify the faults that may perturb the program at hand. Fault forecasting methods [Laprie and Randell 2004] can be useful to achieve this objective. Then, we formally characterize each of these faults as state perturbations. Finally, we group the faults into fault classes based on the corresponding level of tolerance required for each fault class. The desired level of tolerance is based on the user requirements and feasibility of providing that level of tolerance under system constraints/resources. As we have demonstrated in this work, by using this method of fault modeling, one can represent Byzantine, crash, message loss, input corruption, node leave, and transient faults. Moreover, previous work [Liu and Joseph 1992, 1999; Pike et al. 2004; Arora 1992; Ebnenasir and Kulkarni 2011] uses this model to capture other classes of faults such as stuck-at, omission, disk corruption, and sensor failures. However, our fault model cannot capture any types of faults that cannot be represented as a finite-state machine (e.g., impact of external disturbances on aircraft, effect of wing damage on flight control systems, mechanical systems involving several masses, springs and dampers).

Component-based design. Adding multitolerance enables a method for component-based design of multitolerant programs. Specifically, consider an existing program $\mathcal{P}$ that provides multitolerance to the fault classes $f_1, \ldots, f_{n-1}$. If designers detect a new fault class f_n after the design and implementation of $\mathcal{P}$, then it is desirable to have a revised version of $\mathcal{P}$, denoted $\mathcal{P}_c$, that provides multitolerance to $f_1, \ldots, f_{n-1}$ and f_n.

To design $\mathcal{P}_c$, developers have two options: redesign a new multitolerant program from scratch or simply design a component $\mathcal{C}$ and compose it with $\mathcal{P}$ such that the resulting composition preserves fault tolerance to faults f_i, for $1 \leq i \leq n-1$, and enables a specific level of fault tolerance when f_n occurs. In fact, the component $\mathcal{C}$ has to ensure that the computations of $\mathcal{P} \cup f_n$ meet the requirements of the desired level of fault tolerance—that is, no guarantees are provided for the computations of the composed program $\mathcal{P}_c$ in the presence of faults $f_1, \ldots, f_{n-1}$.

Practical significance. The practical impact of the proposed approach is multifold. First, the algorithms presented in this article enable a stepwise method for incremental incorporation of fault tolerance properties. Such a stepwise method is especially useful in the design of fault-tolerant systems because it is often difficult to anticipate all classes of faults in the early stages of design due to the complex and dynamic nature of today's distributed systems. Second, automated addition of multitolerance exploits computational redundancy before resorting to resource redundancy. Third, our algorithms can be integrated in model checkers to facilitate the detection and correction of conflicts between several levels of fault tolerance. Last but not least, although our focus in this article is on high atomicity programs, the proposed method can be used for the *design* of highly resilient network protocols where processing nodes might have read/write restriction with respect to the variables of other nodes.

For example, consider the consistent hashing protocol in Cassandra [Lakshman and Malik 2010], which is a decentralized file system used to manage huge datasets distributed on commodity servers. This protocol is used for data partitioning on a ring of N processes, where each process in the ring is assigned a value modulo N identifying its position on the ring. Then, a hashing mechanism uses the key of each data item to determine the coordinator process of that item. Thus, the set of legitimate configurations of the protocol includes states where we have unique position values in the ring. Consider the case of transient faults (e.g., soft errors) causing random bit flips in memory, thereby taking the consistent hashing ring to states where we may have duplicate position values (or some missing position values). In such cases, one can algorithmically design a self-stabilizing version of the consistent hashing protocol that will eventually converge to configurations where each ring process has a unique position value modulo N. Such convergence occurs without human intervention. In fact, in this case, a self-stabilizing leader election protocol will guarantee convergence to configurations where ring processes have distinct position values modulo N.

There are several other examples where the notion of multitolerance and the algorithms devised in this work can be used effectively. For instance, we have designed [Ebnenasir and Kulkarni 2011] a failsafe-nonmasking multitolerant disk controller that manages a two-sector disk storage system and issues necessary read/write command after selecting and activating a sector. The sectors are subject to data corruption faults, and the *sector-select* and *read/write* command lines from the controller to the sectors are subject to transient faults. The safety specifications require that a read operation returns the last written value, and reading a damaged bit returns 0. Moreover, writing a damaged bit has no effect on a sector. The invariant of the disk system includes the states where the selected sector is the same as the activated sector. The multitolerant disk system ensures that safety specifications are met in the presence of data corruption and the invariant is restored when transient faults cause random bit flips in the command lines.

Another example is a token passing system whose topology includes a set of intertwined unidirectional rings arranged in a two-level hierarchy [Ebnenasir and Kulkarni 2011]. Such intertwined token rings can be utilized to share resources in a controlled manner so that the integrity of the resource is maintained. In this example, the lower level includes a set of rings that behave similar to Dijkstra's self-stabilizing token ring

[Dijkstra 1974], and the higher-level ring includes one process from each lower-level ring. The circulation of the token in the higher-level ring indicates which ring is active in token circulation. The invariant of this system includes states where there is exactly one token in the lower-level rings. This system is subject to three types of faults as follows: (1) fail-stop faults that may cause at most one process to crash in a detectable fashion, (2) process restart faults that may cause random restarts in at most one process, and (3) transient faults that may perturb the entire system to any state in its state space by introducing multiple tokens. The multitolerant token passing system provides failsafe fault tolerance in the presence of fail-stop faults, masking tolerance when process restarts occur, and nonmasking fault tolerance when transient faults perturb the system, thereby enabling a failsafe-nonmasking-masking multitolerant system with a complicated topology.

Yet another example includes a distributed agreement protocol that is nonmasking-masking multitolerant to transient and Byzantine faults. Such an agreement protocol forms a basis of a replicated system to ensure that replicas are maintained in a consistent state. Whereas most protocols in this context focus on providing masking fault tolerance to a fail-stop failure of a replica, this protocol provides several additional guarantees. In particular, this protocol enables a round-based agreement system where a distinguished process casts a decision in each round and the rest of the processes should finalize that round by copying the decision of the distinguished process—in other words, agree on the same decision. The safety specifications of this system require that, in each round, if the distinguished process is nonfaulty, then the decision of all nonfaulty regular processes is identical to that of the distinguished process (i.e., *validity*); otherwise, all nonfaulty regular processes should agree on the same decision (i.e., *agreement*). An invariant of this system includes states from where the round-based computation continues indefinitely, and in each round, validity and agreement are satisfied. Two types of faults may perturb this system in each round: Byzantine faults may make a process behave maliciously when it casts different decision to different processes, and transient faults may nondeterministically change the decision values and cause incoordination in the round-based computation. We have designed a multitolerant version of this system that is masking fault tolerant to Byzantine faults and nonmasking tolerant to transient faults. That is, even in the presence of Byzantine faults, validity and agreement are met in each round, and if perturbed by transient faults, the system will restore its round-based behavior (while ensuring validity and agreement in each round).

Limitations. The approach presented in this article has some constraints in terms of the input to the synthesis algorithms and tool development. First, thus far we have investigated the problem of adding multitolerance for finite-state programs—that is, the problem of adding multitolerance to infinite-state programs is still open. Second, during the addition of multitolerance, our algorithms preserve only the properties that can be captured in the linear topological characterization of specifications by Alpern and Schneider [1985]. For instance, if the properties of the intolerant program are specified in the computation tree logic [Emerson 1990], then we do not guarantee that they will be preserved in the absence of faults. Third, the input program should be maximal. In other words, from any state, the program should have the maximum number of nondeterministic outgoing transitions. The maximality of the intolerant programs increases the chances of success in adding multiple levels of fault tolerance. For example, the action S'_{i1} in the SDS example (synthesized in Section 5.1) includes additional unreachable transitions that might be useful for the addition of new levels of fault tolerance. Fourth, our model of programs is an abstract model in that we do not add multitolerance to C/C++/Java programs. Nonetheless, we can exploit the existing model extraction techniques [Holzmann 2000; Dams et al.

2002; Visser et al. 2003] that are used in model checking where a finite model is generated from a C/C++/Java program and then multitolerance is added to the extracted model. Fifth, the time/space complexity of synthesis is a bottleneck for tool development. To tackle this challenge, we plan to reuse the tools that we have developed for the addition of a single level of fault tolerance to concurrent programs [Ebnenasir et al. 2008] for the automated design of multitolerant programs. Specifically, we have developed distributed [Ebnenasir 2007] and symbolic [Bonakdarpour and Kulkarni 2008] techniques that increase the scalability of algorithms for the addition of fault tolerance significantly (e.g., for programs with 2^{100} reachable states).

Finally, in cases where the algorithms for adding multitolerance declare failure, the design of multitolerance becomes impossible under the constraints/resources of the program at hand. One solution may be to add more redundancy. It is possible to revise the algorithms in this article, so additional redundancy could be introduced automatically. However, we believe that addition of redundancy should be handled manually, as 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 *MM* multitolerance is unfeasible, then system may provide *MN* multitolerance. Again, the choice of this depends upon whether the reduced level of tolerance is acceptable to *system users*. As stated in Section 1, this involves a trade-off between the cost and level of fault tolerance. Automation algorithms, like the ones in this article, allow designers to identify a fault-tolerant program with the given choices in terms of redundancy and level of tolerance.

13. CONCLUSION

In this work, we addressed the problem of synthesizing multitolerant programs from their fault-intolerant version—that is, *adding multitolerance*. The input to the synthesis problem consists of the fault-intolerant program, a set of different classes of faults to which the program is subject, and the expected level of tolerance for each class of faults. We consider three levels of fault tolerance: (1) a failsafe fault-tolerant program guarantees to meet its safety specifications at all times (i.e., both in the presence and in the absence of faults), (2) a nonmasking fault-tolerant program ensures recovery to a set of legitimate states from where its safety and liveness specifications are satisfied, and (3) a masking fault-tolerant program is failsafe and nonmasking at the same time. The problem of adding multitolerance is motivated by the observation that a program is often subject to multiple classes of faults and the level of tolerance provided to them is often different. The problem of adding multitolerance considers the special case where the faults are independent—in other words, occurrences of faults from multiple classes are unlikely to happen simultaneously, and hence it suffices to ensure that the program provides the required tolerance to each fault class. By contrast, the restricted notion of multitolerance considered in Ebnenasir and Kulkarni [2011] focuses on scenarios where faults from several fault classes can happen simultaneously. For this reason, we illustrated that there are several instances where adding multitolerance is feasible but adding restricted multitolerance is not feasible.

Regarding the complexity of adding multitolerance, we considered five possible combinations: MM, FM, FF, MN and NN. In each combination, the first letter indicates the fault-tolerance level for the first class of faults, denoted f_1, and the second letter indicates the fault-tolerance level for the second class of faults, f_2. We found a surprising result that if masking fault tolerance is desired for f_1 and masking (or failsafe) fault-tolerance is desired for f_2, then adding multitolerance is NP-hard (in program state space). This result is counterintuitive since the corresponding problem for adding restricted multitolerance can be solved in polynomial time. We also presented a sound heuristic for designing MM multitolerant programs. For other combinations, FF, MN and NN, we illustrated that the problem of synthesizing multitolerance is in P. To demonstrate this, we presented a sound and complete algorithm for each combination.

We also investigated the relation between restricted multitolerance and multitolerance. Specifically, we argue that if a program is multitolerant under the restricted definition in Ebnenasir and Kulkarni [2011], then it is also multitolerant under the definition considered in this work, although the reverse is not necessarily true. Moreover, we identify circumstances where solvability of adding multitolerance and restricted multitolerance differs. We show that (1) there are situations where adding FF multitolerance is feasible but adding the restricted FF multitolerance is not feasible, (2) there are instances where adding FM multitolerance is feasible but adding restricted FM multitolerance is not feasible, and (3) the synthesis problem of MN/NN multitolerance and restricted MN/NN multitolerance have the same feasibility property.

We have already implemented software tools that can add a single level of fault tolerance [Ebnenasir et al. 2008; Ebnenasir 2007; Bonakdarpour and Kulkarni 2008]. Using such tools, we have been able to add fault tolerance to programs with up to 2^{100} reachable states in less than an hour. To enable the addition of multitolerance, we will integrate the algorithms presented in this article into our existing tools.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments and suggestions.

REFERENCES

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

A. Arora. 1992. *A Foundation of Fault-Tolerant Computing*. Ph.D. Dissertation. University of Texas, Austin, TX.

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

E. Asarin and O. Maler. 1999. As soon as possible: Time optimal control for timed automata. In *Proceedings of the 2nd International Workshop on Hybrid Systems: Computation and Control*. 19–30.

E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. 1998. Controller synthesis for timed automata. In *Proceedings of the IFAC Symposium on System Structure and Control*. 469–474.

P. C. Attie, A. Arora, and E. A. Emerson. 2004. Synthesis of fault-tolerant concurrent programs. *ACM Transactions on Programming Languages and Systems* 26, 1, 125–185. DOI:http://dx.doi.org/10.1145/963778.963782

C. Bernardeschi, A. Fantechi, and L. Simoncini. 2000. Formally verifying fault tolerant system designs. *Computer Journal* 43, 3, 191–205.

B. Bonakdarpour and S. S. Kulkarni. 2008. SYCRAFT: A tool for synthesizing distributed fault-tolerant programs. In *Proceedings of the 19th International Conference on Concurrency Theory (CONCUR'08)*. 167–171.

P. Bouyer, D. D'Souza, P. Madhusudan, and A. Petit. 2003. Timed control with partial observability. In *Computer Aided Verification*. Lecture Notes in Computer Science, Vol. 2725. Springer, 180–192.

J. Chen and S. Kulkarni. 2010. Complexity analysis of weak multitolerance. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*. 398–407.

J. Chen and S. Kulkarni. 2011. Effectiveness of transition systems to model faults. In *Proceedings of the 2nd International Workshop on Logical Aspects of Fault-Tolerance (LAFT'11)*.

Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr. 2009. Surviving sensor network software faults. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, 235–246.

V. Claesson, H. Lonn, and N. Suri. 2004. An efficient TDMA start-up and restart synchronization approach for distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems* 15, 8, 725–739.

D. Dams, W. Hesse, and G. J. Holzmann. 2002. Abstracting C with abC. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)*. 515–520.

L. De Alfaro, M. Faella, T. A. Henzinger, R. Majumdar, and M. Stoelinga. 2003. The element of surprise in timed games. In *Proceedings of the International Conference on Concurrency Theory (CONCUR'03)*. 144–158.

D. De Niz and R. Rajkumar. 2004. Glue code generation: Closing the loophole in model-based development. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04), Workshop on Model-Driven Embedded Systems*. IEEE, Los Alamitos, CA.

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

E. W. Dijkstra and C. S Scholten. 1980. Termination detection for diffusing computations. *Information Processing Letters* 11, 1–4.

D. D'Souza and P. Madhusudan. 2002. Timed control synthesis for external specifications. In *Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science (STACS'02)*. 571–582.

A. Ebnenasir. 2007. Diconic addition of failsafe fault-tolerance. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. 44–53.

A. Ebnenasir and S. Kulkarni. 2011. Feasibility of stepwise design of multitolerant programs. *ACM Transactions on Software Engineering and Methodology* 21, 1, 1:1–1:49.

A. Ebnenasir, S. S. Kulkarni, and A. Arora. 2008. FTSyn: A framework for automatic synthesis of fault-tolerance. *International Journal on Software Tools for Technology Transfer* 10, 5, 455–471.

E. A. Emerson. 1990. Temporal and modal logic. In *Handbook of Theoretical Computer Science*. MIT Press, Cambridge, MA, 995–1072.

P. D. Ezhilchelvan, F. V. Brasileiro, and N. A. Speirs. 2004. A timeout-based message ordering protocol for a lightweight software implementation of TMR systems. *IEEE Transactions on Parallel and Distributed Systems* 15, 1, 53–65. DOI:http://dx.doi.org/10.1109/TPDS.2004.1264786

M. Faella, S. Torre, and A. Murano. 2002. Dense real-time games. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*. IEEE, Los Alamitos, CA, 167–176.

Z. Gu and K. G. Shin. 2005. Synthesis of real-time implementations from component-based software models. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)*. IEEE, Los Alamitos, CA, 167–176. DOI:http://dx.doi.org/10.1109/RTSS.2005.38

J. Heinzmann and A. Zelinsky. 1999. A safe-control paradigm for human–robot interaction. *Journal of Intelligent and Robotics Systems* 25, 4, 295–310. DOI:http://dx.doi.org/10.1023/A:1008135313919

G. Holzmann. 2000. Logic verification of ANSI-C code with SPIN. In *Proceedings of the 6th SPIN Workshop*. 131–147.

P.-A. Hsiung and S.-W. Lin. 2008. Automatic synthesis and verification of real-time embedded software for mobile and ubiquitous systems. *Computer Languages, Systems, and Structures* 34, 4, 153–169. DOI:http://dx.doi.org/10.1016/j.cl.2007.06.002

M. L. James, A. A. Shapiro, P. L. Springer, and H. P. Zima. 2009. Adaptive fault tolerance for scalable cluster computing in space. *International Journal of High Performance Computing Applications* 23, 3, 227–241. DOI:http://dx.doi.org/10.1177/1094342009106190

B. Jobstmann, A. Griesmayer, and R. Bloem. 2005. Program repair as a game. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*. 226–238.

S. S. Kulkarni. 1999. *Component-Based Design of Fault-Tolerance*. Ph.D. Dissertation. Ohio State University, Columbus, OH.

S. S. Kulkarni and A. Arora. 2000. Automating the addition of fault-tolerance. In *Proceedings of the 6th International Symposium of Formal Techniques in Real-Time and Fault-Tolerant Systems*. 82–93.

S. S. Kulkarni, A. Arora, and A. Ebnenasir. 2007. *Adding Fault-Tolerance to State Machine-Based Designs*. Series on Software Engineering and Knowledge Engineering, Vol. 19. Springer, 62–90.

S. S. Kulkarni and A. Ebnenasir. 2002. The complexity of adding failsafe fault-tolerance. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*. IEEE, Los Alamitos, CA, 337.

A. Lakshman and P. Malik. 2010. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2, 35–40.

L. Lamport, R. Shostak, and M. Pease. 1982. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems* 4, 3, 382–401.

J.-C. Laprie and B. Randell. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1, 11–33. DOI:http://dx.doi.org/10.1109/TDSC.2004.2

P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. 2005. TinyOS: An operating system for sensor networks. In *Ambient Intelligence.* Springer, 115–148.

Q. Li and D. Rus. 2006. Global clock synchronization in sensor networks. *IEEE Transactions on Computers* 55, 2, 214–226.

S. Lin, C. Tseng, T. Lee, and J. Fu. 2004. VERTAF: An application framework for the design and verification of embedded real-time software. *IEEE Transactions on Software Engineering* 30, 10, 656–674. DOI:http://dx.doi.org/10.1109/TSE.2004.68

Z. Liu and M. Joseph. 1992. Transformation of programs for fault-tolerance. *Formal Aspects of Computing* 4, 5, 442–469.

Z. Liu and M. Joseph. 1999. Specification and verification of fault-tolerance, timing, and scheduling. *ACM Transactions on Programming Languages and Systems* 21, 1, 46–89.

M. Lubaszewski and B. Courtois. 1998. A reliable fail-safe system. *IEEE Transactions on Computers* 47, 2, 236–241.

P. S. Miner, M. Malekpour, and W. Torres-Pomales. 2002. Conceptual design of a reliable optical BUS (ROBUS). In *Proceedings of the AIAA/IEEE Digital Avionics Systems Conference*. 1–11.

L. Pike, J. Maddalon, P. S. Miner, and A. Geser. 2004. Abstractions for fault-tolerant distributed system verification. In *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOL'04)*. 257–270.

P. Ramanathan, K. G. Shin, and R. W. Butler. 1990. Fault-tolerant clock synchronization in distributed systems. *Computer* 23, 10, 33–42.

J. M. Rushby. 2001. Bus architectures for safety-critical embedded systems. In *Proceedings of the 1st International Workshop on Embedded Software (EMSOFT'01)*. 306–323.

H. Schiöberg, R. Merz, and C. Sengul. 2009. A failsafe architecture for mesh testbeds with real users. In *Proceedings of the 2009 MobiHoc S3 workshop on MobiHoc S3 (MobiHoc S3'09)*. ACM, New York, NY, 29–32. DOI:http://dx.doi.org/10.1145/1540358.1540368

P. Sommer and R. Wattenhofer. 2009. Gradient clock synchronization in wireless sensor networks. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks (IPSN'09)*. IEEE, Los Alamitos, CA, 37–48.

H. J. Song and A. A. Chien. 2005. Feedback-based synchronization in system area networks for cluster computing. *IEEE Transactions on Parallel and Distributed Systems* 16, 10, 908–920. DOI:http://dx.doi.org/10.1109/TPDS.2005.122

C. Temple. 1998. Avoiding the babbling-idiot failure in a time-triggered communication system. In *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing (FTCS'98)*. IEEE, Los Alamitos, CA, 218–227.

W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. 2003. Model checking programs. *Journal of Automated Software Engineering* 10, 2, 203–232.

Received December 2013; revised May 2014; accepted May 2014