Feasibility of Stepwise Design of Multitolerant Programs

ALI EBNENASIR, Michigan Technological University
SANDEEP S. KULKARNI, Michigan State University

The complexity of designing programs that simultaneously tolerate multiple classes of faults, called *multitolerant* programs, is in part due to the conflicting nature of the fault tolerance requirements that must be met by a multitolerant program when different types of faults occur. To facilitate the design of multitolerant programs, we present sound and (deterministically) complete algorithms for stepwise design of two families of multitolerant programs in a high atomicity program model, where a process can read and write all program variables in an atomic step. We illustrate that if one needs to design failsafe (respectively, nonmasking) fault tolerance for one class of faults and masking fault tolerance for another class of faults, then a multitolerant program can be designed in separate polynomial-time (in the state space of the fault-intolerant program) steps regardless of the order of addition. This result has a significant methodological implication in that designers need not be concerned about unknown fault tolerance requirements that may arise due to unanticipated types of faults. Further, we illustrate that if one needs to design failsafe fault tolerance for one class of faults and nonmasking fault tolerance for a different class of faults, then the resulting problem is NP-complete in program state space. This is a counterintuitive result in that designing failsafe and nonmasking fault tolerance for the same class of faults can be done in polynomial time. We also present sufficient conditions for polynomial-time design of *failsafe-nonmasking* multitolerance. Finally, we demonstrate the stepwise design of multitolerance for a stable disk storage system, a token ring network protocol and a repetitive agreement protocol that tolerates Byzantine and transient faults. Our automatic approach decreases the design time from days to a few hours for the token ring program that is our largest example with 200 million reachable states and 8 processes.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Error handling and recovery*; D.1.3 [**Programming Techniques**]: Automatic Programming; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-Aided Software Engineering*

General Terms: Design, Verification, Reliability

Additional Key Words and Phrases: Automatic addition of fault tolerance, multitolerance

ACM Reference Format:
Ebnenasir, A. and Kulkarni, S. S. 2011. Feasibility of stepwise design of multitolerant programs. ACM Trans. Softw. Eng. Methodol. 21, 1, Article 1 (December 2011), 49 pages.
DOI = 10.1145/2063239.2063240 http://doi.acm.org/10.1145/2063239.2063240

This article is a revised and extended version of a paper presented at the International Conference on Dependable Systems and Networks (DSN) [Kulkarni and Ebnenasir 2004].
This work was partially sponsored by NSF CCF-0950678, CNS-0914913, NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF grant EIA-0130724, AFOSR Grant FA9550-10-1-0178, a grant from Michigan State University, and the Research Excellence Fund (REF) grant from Michigan Technological University.
Authors' addresses: A. Ebnenasir, Department of Computer Science, Michigan Technological University, 1400 Townsend Drive, Houghton, MI 49931; email: aebnenas@mtu.edu; S. S. Kulkarni, Computer Science and Engineering Department, Michigan State University, East Lansing, MI 48824.

DOI 10.1145/2063239.2063240 http://doi.acm.org/10.1145/2063239.2063240

1. INTRODUCTION

The focus of this paper is on *automating* the design of *multitolerant* programs from their fault-intolerant versions, where a multitolerant program tolerates multiple classes of *faults* and provides potentially different levels of *fault tolerance* to them, where a *level* of fault tolerance represents the correctness requirements in the presence of a fault-class (e.g., ensuring safety, guaranteeing recovery or satisfying both in the presence of faults). The input to this problem includes a fault-intolerant program, multiple classes of faults, and a desired level of fault tolerance corresponding to each fault-class. The output is an automatically generated multitolerant version of the intolerant program. The significance of this problem is multifold. First, while there are several approaches (e.g., FFA [SAE 1996], FTA [Vesely 1981], FMEA [Palady 1995], HAZOP [Kletz 1999]) for determining the classes of faults in early stages of design, anticipating all classes of faults and failure modes of today's computing systems is difficult (if not impossible) due to (1) the complexity and the diversity of devices used in such systems, (2) the aging of hardware systems, and (3) the complexity of early detection of faults.[1] As such, when an unanticipated fault is detected, designers have two options; either redesign a whole new program from scratch or upgrade an existing program to capture new fault tolerance functionalities while preserving existing ones. Our objective in this article is to provide automated techniques/tools for such upgrades. Second, automating the design of multitolerant programs from their intolerant version provides separation of concerns in designing tolerance for multiple classes of faults (even if they are already known to the designer). Third, if the input program meets its specifications in the absence of faults, then its automatically generated multitolerant version is correct-by-construction, thereby eliminating the need for after-the-fact verification. We analyze the time complexity of such a design method and identify instances of the problem where different levels of fault tolerance can be added in a stepwise fashion in deterministically polynomial time. We also outline future directions for further facilitation of designing multitolerant programs. Next, we motivate and intuitively define the terms in italics. (For formal definitions, see Section 2.)

Programs. We model parallel/distributed computing systems using (nondeterministic) finite-state transition systems. Transition systems can be used to capture network protocols, multicomputer systems, and abstractions of real-world software applications. Examples of such abstractions include the inter-process synchronization mechanisms of concurrent programs. More examples can easily be found in the model checking literature [Dams et al. 2002; Holzmann 1997; Holzmann et al. 2008; Visser et al. 2003]. As such, systems that cannot accurately be captured as a transition system (e.g., mechanical systems involving several masses, springs, and dampers) are outside the scope of this article.

Specifications. We follow Alpern and Schneider [1985] in defining program specifications in terms of a *safety* specification and a *liveness* specification. Intuitively, safety states that nothing bad ever happens (e.g., it is always the case that at most one process has access to a critical section of its code), and liveness requires that something good will eventually occur (e.g., it is always the case that each process will eventually enter its critical section).

[1]Experience illustrates that even if extreme care is taken in the development of software systems, there are still some classes of unanticipated faults that could cause failures; for instance, the Apollo project [Ulsamer 1973].

Faults. In the context of this article, *a class of faults* represents the effect of a specific type of faults (e.g., crash, Byzantine, message loss, omission, soft errors, input corruption, design flaws, etc.) on a program as a set of transitions that may (non-)deterministically execute [Avizienis et al. 2004]. That is, a fault transition from a state s_0 to another state s_1 can execute if s_0 is reached, but it does not have to. Specifically, we follow Avizienis et al. [2004] in that the occurrence of faults may perturb the state of a program to an *error* state from where program execution may deviate from its specification; that is, a *failure* may occur. Since our focus is on taking an existing program and redesigning it in such a way that it exhibits specified behaviors when faults occur, we are interested in how faults affect programs rather than being concerned with the reason behind the occurrence of faults. For example, the effect of crash faults on a program process is that that process will not execute any further instructions. Other examples include message loss, failure of nodes, and Byzantine behavior by a process that sends incorrect information to other processes. If the failure due to a fault-class causes other unexpected events, we capture those events as new classes of faults as well. For example, if a message loss causes a recipient process to become unresponsive, we can model the unresponsiveness of the receiver as a fault-class different from message loss, but enabled after the occurrence of message loss.

Fault Tolerance. A *fault-intolerant* program guarantees to meet its safety and liveness specifications in the absence of faults, however, in the presence of faults (i.e., when faults occur), a fault-intolerant program provides no guarantees about how it will behave (i.e., it may or may not satisfy its specifications). Intuitively, fault tolerance refers to the ability of a system to satisfy a possibly weaker version of its specification in the presence of faults (i.e., graceful degradation [Herlihy and Wing 1991]). More precisely, we consider three levels of fault tolerance depending on the extent to which safety and liveness specifications are met when faults occur. A *failsafe* fault-tolerant program ensures that its safety specification is always satisfied; nonetheless, in the presence of faults, liveness may not be met. A *nonmasking* program guarantees recovery to states from where its safety and liveness are satisfied, but may not meet its safety during recovery. Finally, a *masking* fault-tolerant program simultaneously meets the requirements of failsafe and nonmasking fault tolerance. (See Section 2 for precise definitions of failsafe, nonmasking, and masking fault tolerance.)

Multitolerance. Software systems are often subject to several classes of faults, and they are required to provide a possibly different level of fault tolerance to each fault-class. For example, a resilient network protocol should provide masking fault tolerance to message loss, that is, ensure that there will be no duplicate messages (i.e., safety) and each message will eventually reach the receiver (i.e., liveness) even if messages get lost. However, if a more serious fault (e.g., a router failure) occurs then it may only provide nonmasking fault tolerance where the protocol eventually reconfigures itself although some properties (e.g., safety or satisfaction of pending requests) may not be satisfied during reconfiguration. A multitolerant program that tolerates both message loss and node failure could ensure that if only the former fault occurs then masking fault tolerance is provided but if the latter fault occurs then nonmasking fault tolerance is provided. The importance of such multitolerant systems can be easily observed from the fact that several algorithms for designing multitolerant programs as well as several instances of multitolerant programs can be readily found in the literature.

Most existing approaches [Anagnostou 1993; Arora and Kulkarni 1998a; Dolev and Herman 1995; Dolev and Yagel 2007; Dolev and Hoch 2007a; Malekpour 2006;

Tsang and Magill 1994] for the design of multitolerant programs are based on a *design-and-verification* method, where algorithms that tolerate multiple classes of faults are designed first and then verified to ensure correctness. The verification task is often difficult and expensive as one must mechanically prove that (i) in the absence of faults, the multitolerant program satisfies its safety and liveness; (ii) in the presence of each individual class of faults the multitolerant program provides a required level of fault tolerance; (iii) each level of fault tolerance designed for tolerating a specific fault-class does not interfere with the normal functionalities in the absence of faults; and more importantly, (iv) the fault tolerance functionalities designed for each fault-class do not interfere with the functionalities designed for other levels of fault tolerance. As such, *automatic design* of multitolerant programs that are correct by construction is an ideal goal in the context of programs where correctness is crucial (e.g., safety-critical programs). Kulkarni and Arora [2000] present a family of algorithms that automatically add a *single* level of fault tolerance (e.g., failsafe, nonmasking, masking) to programs. To facilitate the design of multitolerant programs, it would be beneficial if designers could reuse Kulkarni and Arora's algorithms [Kulkarni and Arora 2000] without being dependent on their internal implementation. An outcome of such reuse is a stepwise method for the automated design of multitolerance, where only one fault-class is considered at a time. Moreover, any improvement in time/space complexity of Kulkarni and Arora's algorithms would also improve the time/space efficiency of stepwise algorithms for the design of multitolerance. Furthermore, in each step, designers have the option to improve intermediate programs to capture other concerns (such as performance, quality of service, refactoring, etc.).

Stepwise Design of Multitolerance. In our previous work [Kulkarni and Ebnenasir 2004], we present a stepwise approach where we reuse Kulkarni and Arora's algorithms for the addition of a single fault-class [Kulkarni and Arora 2000] in the design of multitolerant programs. Nonetheless, our algorithms in Kulkarni and Ebnenasir [2004] suffer from the following drawbacks. First, they require all classes of faults to be known at the outset. As such, when unanticipated types of faults are detected, these algorithms do not have the potential to reuse existing intermediate programs that tolerate known faults (see Figure 1). Second, designers have to perform some preprocessing and intermediate processing that depend upon the order of adding different levels of fault tolerance and require some knowledge on the implementation of Kulkarni and Arora's algorithms in Kulkarni and Arora [2000]. That is, the stepwise algorithms in Kulkarni and Ebnenasir [2004] provide a *white-box* design method (see Figure 1). A stepwise approach that reuses Kulkarni and Arora's algorithms [Kulkarni and Arora 2000] as *black boxes* (see Figure 2) simplifies the reordering of design steps, thereby facilitating the design of new levels of fault tolerance upon the detection of new classes of faults.

In a black-box stepwise approach (Figure 2), we need to determine an appropriate order for the addition of different levels of fault tolerance to classes of faults $f_1, \cdots, f_k$ ($k > 1$) so that the resulting program tolerates $f_1, \cdots, f_k$. One approach for determining an appropriate order of addition is to create a decision tree for all possible permutations/orders for designing multitolerance to $f_1, \cdots, f_k$. Arora and Kulkarni [1998a] present such a method where they first design k initial programs each one tolerating a specific fault f_i ($1 \leq i \leq k$); each initial program becomes an immediate successor of the root of the decision tree. Then, for each initial program, $k-1$ subsequent levels of fault tolerance remain to be added, each level corresponding to a fault-class f_j, where $j \neq i$ ($1 \leq i, j \leq k$). The height of such a decision tree is k where a single level of fault tolerance is added in each internal node of the tree; each leaf of the decision tree represents an order of addition which may or may not result

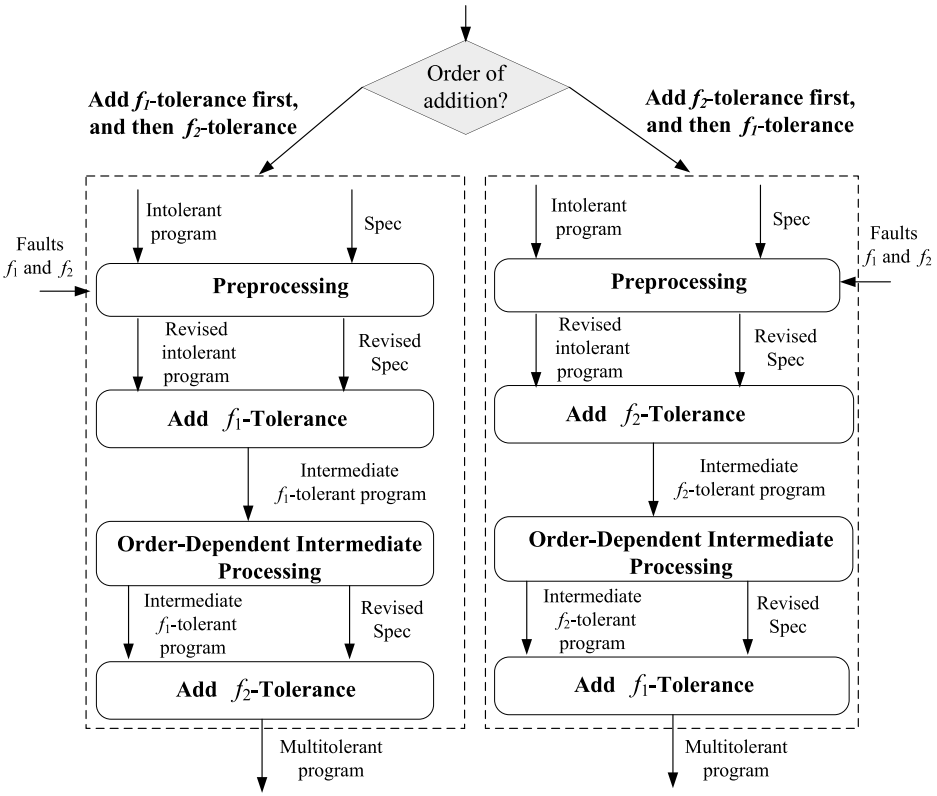

Fig. 1. The white-box approach for stepwise addition of multitolerance proposed in Kulkarni and Ebnenasir [2004]. Note that both classes of faults f_1 and f_2 must be known at the outset.

in a multitolerant program. For example, adding nonmasking fault tolerance to f_1 before adding failsafe fault tolerance to f_2 may result in an intermediate nonmasking program to which failsafe f_2-tolerance cannot be added without undermining the recovery provided by the nonmasking property. While a nondeterministic algorithm can search Arora and Kulkarni's decision tree in polynomial time to identify an appropriate order of addition, to the best of our knowledge, the design of a deterministic stepwise algorithm with polynomial-time steps is still an open problem for arbitrary inputs (i.e., faults, programs, and specifications) to the problem of multitolerance design. Moreover, we are not even aware of the existence of such a deterministic stepwise method for cases where one considers special family of programs, faults or specifications.

Contributions. With these motivations, the contributions of the article are as follows.

(1) We show that for a class of high atomicity programs, where a process can read and write all program variables in an atomic step, and a specification in the *bad transitions* (BT) model [Kulkarni and Ebnenasir 2005b], where the safety specification can be characterized in terms of bad states and bad transitions that should not occur in program computations, a deterministically sound and complete solution exists in the following two scenarios.

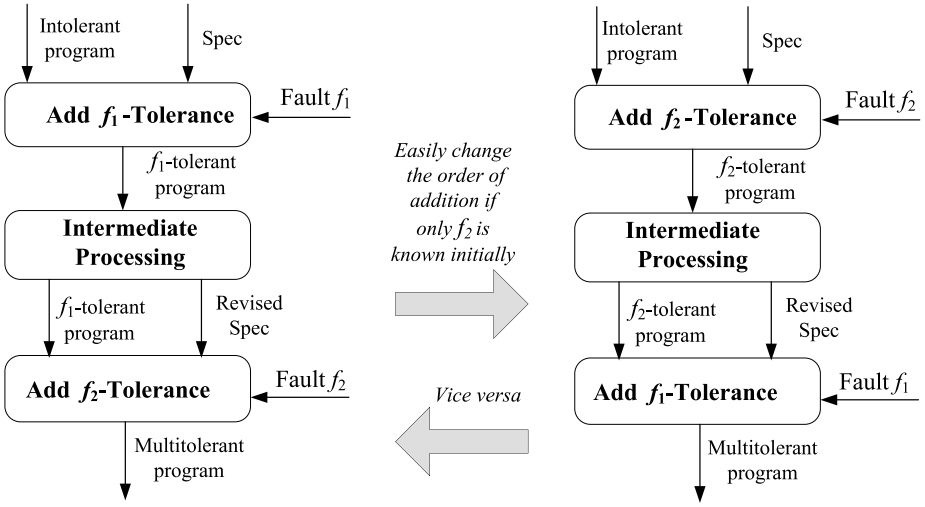

Fig. 2. A black-box approach for stepwise addition of multitolerance proposed in this paper. Either f_1 or f_2 may be unknown initially.

— Failsafe-Masking multitolerance. Failsafe fault tolerance is provided to one class of faults, and masking fault tolerance is provided to another class of faults, called failsafe-masking multitolerance. Failsafe-masking multitolerance has several applications such as network protocols that are failsafe to link/node failures and are masking to Byzantine faults [Siu et al. 1998].
— Nonmasking-Masking multitolerance. Nonmasking fault tolerance is provided to one class of faults, and masking fault tolerance is provided to another class of faults, called nonmasking-masking multitolerance. Examples of nonmasking-masking multitolerance include clock synchronization protocols [Dolev and Welch 1995; Dolev and Hoch 2007b; Ben-Or et al. 2008] that are nonmasking to transient faults and masking tolerant to Byzantine faults.

To illustrate the soundness and completeness of designing multitolerance in the preceding cases, we rely only on certain properties of the algorithms presented in Kulkarni and Arora [2000] for the addition of a single level of fault tolerance, and not on their implementation. As a result, we reuse the algorithms in Kulkarni and Arora [2000] as black boxes.

Investigating the design of multitolerance for high atomicity programs enables us to (i) establish impossibility results for more concrete programs, where processes have read/write restrictions with respect to program variables (see Section 2.2 for more details); (ii) use the synthesized high atomicity programs as guiding examples in the design of their concrete versions, and (iii) advance the state of knowledge towards the design of multitolerance for distributed programs (see Sections 6.2 and 6.3 for examples).

(2) Additionally, we also find a counterintuitive result that if failsafe fault tolerance is required to one class of faults and nonmasking fault tolerance is desired to another class of faults, then such a sound and deterministically complete algorithm is unlikely to exist if the complexity of adding one class of faults is to be polynomial. In particular, we show that the problem of adding failsafe fault tolerance to one class of faults and nonmasking fault tolerance to a different class of faults is NP-complete (in program state space). This result is surprising in that adding failsafe

and nonmasking fault tolerance to the *same* class of faults is polynomial [Kulkarni and Arora 2000].

(3) While the presented NP-completeness result implies that, in general, a deterministically polynomial-time stepwise method does not exist (unless $P = NP$), our NP-hardness proof enables us to (1) identify systems/specifications for which design of multitolerance can be performed in polynomial time (see Section 5 for such special cases), and (2) design polynomial-time heuristics at the expense of forfeiting the completeness. That is, if the heuristics succeed in designing a multitolerant program, then the automatically generated program is correct. However, in some cases, such heuristics may fail to generate a multitolerant program while one exists (hence the incompleteness), and (3) integrate the heuristics in an extensible repository available for developers.

(4) We demonstrate example programs synthesized by the Fault Tolerance Synthesizer (FTSyn) [Ebnenasir et al. 2008] tool, where the proposed sufficient conditions are satisfied. Specifically, we demonstrate the stepwise design of a multitolerant Stable Disk Storage (SDS) program (adapted from Bernardeschi et al. [2000]) that tolerates permanent bit-damage faults and transient faults. The significance of the multitolerant SDS program is twofold. First, the blackbox nature of our stepwise approach enables the reuse of the intermediate failsafe program that has been revised to capture a performance improvement requirement. Using our white-box approach in Kulkarni and Ebnenasir [2004], we would have to redesign the entire multitolerant program from scratch if the transient faults were unknown at the time of designing failsafe fault tolerance to damage faults. Second, Bernardeschi et al. [2000] use three replicas of the SDS system in order to mask the faults using a majority voter. They illustrate that a triple modular redundant design tolerates two fault hypotheses: (1) the case where a sector in one replica is damaged and another replica is affected by transient faults, and (2) both permanent damage and transient faults occur in the same replica. In both hypotheses, there exists a healthy majority that enables fault masking, however, the replica that is affected by transient faults may never recover; that is, it may deadlock or stay in a nonprogress cycle forever, thereby undermining the ability of the redundant system to deal with subsequent occurrences of transient faults. We demonstrate how our algorithms generate a failsafe-nonmasking multitolerant SDS program for a single replica (without any resource redundancy) that guarantees safety when bit-damage faults occur and ensures recovery from transient faults! Besides, the automatically generated multitolerant design is correct by construction; that is, there is no need for verification.

    Moreover, we have used FTSyn to design a multitolerant token passing program (see Section 6.2) that tolerates crash, state corruption and transient faults. This program has two rings where a token is circulated among 4 processes in each ring, and its state space includes 200 million states, all of them reachable due to transient faults. Using FTSyn, we decreased the design time of this program from 5 days in a manual approach to almost 7 hours. We used a version of FTSyn that we have implemented using Binary Decision Diagrams (BDDs) [Bryant 1986] on a Linux PC with an Intel Pentium IV (3.00GHz) CPU with 2 GB RAM. We also have used FTSyn in automated design of several real-world applications and classic fault-tolerant computing programs [Ebnenasir 2005] such as a cruise control system, an altitude switch controller [Ebnenasir et al. 2008], Byzantine agreement protocol, diffusing computation and alternating bit protocol. For these examples, FTSyn automatically generated correct programs in a matter of minutes. Furthermore, FTSyn automatically generated a fault-tolerant altitude switch controller that revealed human errors in the system specifications [Ebnenasir 2005].

The rest of the article is organized as follows. In Section 2, we present preliminary concepts. Then, in Section 3, we present the formal definition of multitolerance and the problem of synthesizing multitolerant programs from their fault-intolerant version. Subsequently, in Section 4, we demonstrate that, in general, stepwise design of multitolerance is NP-complete (in program state space), which constitutes an impossibility result for polynomial-time stepwise design of multitolerance unless $P = NP$. In Section 5, we investigate the feasibility of sound and complete stepwise addition of multitolerance for special cases. In Section 6, we illustrate three examples of stepwise design of multitolerant programs. We discuss related work and the practical relevance of our approach in Section 7. Finally, in Section 8, we make concluding remarks and discuss future work.

2. PRELIMINARIES

In this section, we present formal definitions of programs, problem specifications, faults, and fault tolerance. The programs are defined in terms of their set of variables, their transitions and their processes/components. The definition of specifications is adapted from Alpern and Schneider [1985]. The definitions of faults and fault tolerance are adapted from Arora and Gouda [1993] and Kulkarni [1999]. To simplify our presentation, we use a Stable Disk Storage (SDS) program as a running example.

2.1 Program

A *program* $p = \langle V_p, \delta_p, \mathcal{C}_p \rangle$ is a tuple of a finite set V_p of variables, a set of transitions δ_p and a finite set $\mathcal{C}_p$ of K processes/components, where $K \geq 1$. Each variable $v_i \in V_p$, for $1 \leq i \leq N$, has a finite nonempty domain D_i. A *state* s of p is a valuation $\langle d_1, d_2, \cdots, d_N \rangle$ of program variables $\langle v_1, v_2, \cdots, v_N \rangle$, where $d_i \in D_i$. A *transition* t is an ordered pair of states, denoted (s_0, s_1), where s_0 is the source and s_1 is the target state of t. A program *process*/*component* P_i $(1 \leq i \leq K)$ includes a set of transitions δ_i. The set δ_p of program transitions is equal to the union of the transitions of its processes; that is, $\delta_p = \cup_{i=1}^{K} \delta_i$. For a variable v and a state s, $v(s)$ denotes the value of v in s. The *state space* S_p is the set of all possible states of p. A *state predicate* of p is any subset of S_p. A state predicate S is *closed* in the program p (respectively, δ_p) iff (if and only if) $\forall s_0, s_1 : (s_0, s_1) \in \delta_p : (s_0 \in S \Rightarrow s_1 \in S)$. A sequence of states, $\sigma = \langle s_0, s_1, ... \rangle$ is a *computation* of p iff the following two conditions are satisfied: (1) if σ is infinite, then $\forall j : j > 0 : (s_{j-1}, s_j) \in \delta_p$, and (2) if σ is finite and terminates in state s_l, then $\forall j : 0 < j \leq l : (s_{j-1}, s_j) \in \delta_p$, and there does not exist state s such that $(s_l, s) \in \delta_p$. A *finite* sequence of states, $\langle s_0, s_1, ..., s_n \rangle$, is a *computation prefix* of p iff $\forall j : 0 < j \leq n : (s_{j-1}, s_j) \in \delta_p$. The projection of a program p on a nonempty state predicate S, denoted as $p|S$, is the program $\langle V_p, \{(s_0, s_1) : (s_0, s_1) \in \delta_p \ \wedge \ s_0, s_1 \in S\}, \mathcal{C}_p \rangle$. In other words, $p|S$ consists of transitions of p that start in S and end in S.

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

Example: Stable Disk Storage (SDS). The Stable Disk Storage (SDS) program (adapted from Bernardeschi et al. [2000]) includes a controller and two sectors (i.e., Sectors 0 and 1) being managed by the controller. The controller initially selects and activates a sector for a read/write operation and then issues read/write commands. After an operation is performed, the above cycle is repeated; that is, the controller selects one of the sectors and issues the next command. The SDS program has the following variables: ctrlState denotes the state of the controller and has a domain of {0,1}, where 0 represents the state where the controller wants to select a sector and 1 means that the

controller is in a state of issuing a command. The variable secNum represents which sector has been selected, and its domain is {-1,0,1}, where -1 denotes that no sector has been selected yet, 0 represents Sector 0 and 1 for Sector 1. Moreover, there are two activation signals $\mathsf{activateSec}_0$ and $\mathsf{activateSec}_1$ illustrating which sector is activated. The bit that is read/written in sector i is denoted by x_i, where $i = 0, 1$. Upon read operation, the returned bit from sector i is placed in the communication channel between that sector and the controller, denoted c_i. The same channel is used to write a value on x_i, where $i = 0, 1$. The command signal op_i demonstrates whether a read or a write operation should be performed in sector i. When op_i is 0, it means that the value of x_i is read into c_i, and a value 1 for op_i denotes that the contents of c_i is written in x_i. The domain of x_i is equal to {-1,0,1}, where -1 represents a damaged bit. The domain of c_i, op_i and $\mathsf{activateSec}_i$ is {0,1}, where $i = 0, 1$. We use Dijkstra's guarded commands language [Dijkstra 1990] as a shorthand for representing the set of program transitions. A guarded command (action) is of the form $grd \rightarrow stmt$, where grd is a state predicate and $stmt$ is a statement that updates program variables. Formally, a guarded command $grd \rightarrow stmt$ includes all program transitions $\{(s_0, s_1) : grd$ holds at s_0 and the *atomic* execution of $stmt$ at s_0 takes the program to state $s_1\}$. The guarded commands of the controller are as follows:

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

Action C_1 nondeterministically assigns 0 or 1 to secNum (denoted by the vertical bar |) representing a request received by the controller for performing an operation with either Sector 0 or Sector 1. Action C_2 changes the state of the controller to a state where a read/write command is sent to the selected sector in secNum. Action C_3 (respectively, C_5) sends a read command to Sector 0 (respectively, Sector 1), whereas C_4 (respectively, C_6) issues a write command for Sector 0 (respectively, Sector 1). Action C_7 changes the state of the controller to the sector selection mode. Since the two sectors have a similar design, we use parametric guarded commands in terms of i to represent the actions of each sector as follows ($i = 0, 1$). Notice that the value secNum is set to

-1 when the issued command is executed by the corresponding sector using one of the actions S_{i1}, S_{i2} and S_{i3}.

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

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.

2.2 Read/Write Model

We investigate the complexity of the design of multitolerance in the context of *high atomicity* concurrent programs, where each process can read/write all program variables in an atomic step. While high atomicity programs represent a restricted family of programs, the motivation behind investigating the design of multitolerance for these programs is multifold. First, if for a given program multitolerance cannot be designed in the high atomicity model, then one can conclude that multitolerance cannot be designed under additional constraints of a concrete model either. Second, our experience [Kulkarni and Ebnenasir 2003] demonstrates that designing fault tolerance in the high atomicity model provides insight for designing fault tolerance in more concrete models because the computational structure of the synthesized high atomicity programs can guide us in generating a multitolerant program in lower levels of atomicity. Third, there are correctness-preserving methods [Demirbas and Arora 2002; Marzullo et al. 1994; Nesterenko and Arora 2002] that can algorithmically refine a high atomicity program while preserving its fault tolerance properties. Fourth, the results presented in this paper provide the first step towards automatic design of multitolerant programs in more concrete models in cases where refinement methods fail. Specifically, in our previous work [Ebnenasir 2005; Kulkarni and Arora 2000], we have modeled more concrete programs by imposing read/write restrictions on program processes with respect to program variables (see Section 6 for some examples).

The effect of write restrictions is that the set of transitions of a process cannot include the transitions that update the local variables of other processes. As such, write restrictions for each process can be modeled as a set of transitions that must not be executed by that process. Read restrictions require us to *group* transitions and ensure that, during the design of multitolerance, an entire group is included or the entire group is excluded. (The idea of grouping has also appeared in previous work [Kulkarni and Arora 2000; Attie and Emerson 2001].) As an example, consider a program consisting of variables x and y and let their domain be $\{0, 1\}$. Moreover, consider a process that cannot read the variable x. We can think of the transition from the state $\langle x=0, y=0\rangle$ to the state $\langle x=0, y=1\rangle$ as an atomic *if* statement 'if x is 0 and y is 0 then set y to 1'. In this case, the process must read x. However, if we include the transition from the state $\langle x=1, y=0\rangle$ to the state $\langle x=1, y=1\rangle$ in the set of transitions of that process, then these two transitions can be thought of as 'if y is 0 then set y to 1'. In other words, the inability to read causes the transitions $(\langle x=0, y=0\rangle, \langle x=0, y=1\rangle)$ and $(\langle x=1, y=0\rangle, \langle x=1, y=1\rangle)$ to be grouped. In the set of transitions of the corresponding process, we need to include all transitions in this group or exclude all of them. Previous work [Kulkarni and Arora 2000; Kulkarni and Ebnenasir 2005a]

illustrates that adding a single level of fault tolerance under read restrictions is significantly harder than adding the same level of fault tolerance in the high atomicity model.

2.3 Specification

Following Alpern and Schneider [1985], we let the program specification *spec* be a set of infinite sequences of states. We assume that this set is suffix-closed and fusion-closed. Suffix closure of the set means that if a state sequence σ is in that set then so are all the suffixes of σ. Fusion closure of the set means that if state sequences $\langle \alpha, s, \gamma \rangle$ and $\langle \beta, s, \delta \rangle$ are in that set then so are the state sequences $\langle \alpha, s, \delta \rangle$ and $\langle \beta, s, \gamma \rangle$, where α and β are finite prefixes of state sequences, γ and δ are suffixes of state sequences, and s is a program state.

We say a computation $\sigma = \langle s_0, s_1, \cdots \rangle$ satisfies (does not violate) *spec* iff $\sigma \in spec$. Given a program p, a state predicate S, and a specification *spec*, we say that p satisfies *spec* from S iff (1) S is closed in p, and (2) every computation of p that starts in a state in S satisfies *spec*. If p satisfies *spec* from S and $S \neq \{\}$, we say that S is an invariant of p for *spec*. (Note that program p may have multiple invariants for *spec*.)

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

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

Based on Alpern and Schneider [1985], a specification *spec* can be expressed as an intersection of a safety specification and a liveness specification, each of which is also a set of infinite sequences of states. Such infinite sequences of states cannot be used as an input to a synthesis routine, especially if we are interested in identifying the complexity of the synthesis algorithm. Hence, we need to identify an equivalent (but concise) finite representation. We discuss this next.

Representation of safety during synthesis. For a suffix-closed and fusion-closed specification, the safety specification can be characterized by a *finite* set of bad transitions (see Page 26, Lemma 3.6 of Kulkarni [1999]) that must not appear in program computations. That is, for program p, its safety specification can be characterized by a subset of $\{(s_0, s_1) : (s_0, s_1) \in S_p \times S_p\}$. The set of infinite sequences representing the safety specification and the set of bad transitions are equivalent in that (1) given a set bad_{tr} of bad transitions, it corresponds to the infinite state sequences where no sequence contains any transition from bad_{tr}, and (2) given a specification *spec* in terms of a set of infinite sequences, the set of bad transitions bad_{tr} includes those transitions that do not appear in any sequence in *spec*. Since the set of bad transitions provides a concise representation for safety specification, we use that as an input to our algorithms.

Remark. If fusion or suffix closure is not provided then safety specification can be characterized in terms of finite-length prefixes [Alpern and Schneider 1985]. We have shown [Kulkarni and Ebnenasir 2005b] that if one adopts such a general model of

safety specification instead of our restricted model (i.e., the *bad transitions* model) then the complexity of synthesis significantly increases from polynomial (in program state space) to NP-hard. Hence, for efficient synthesis, based on which tool support [Ebnenasir et al. 2008] can be provided, we represent safety with a set of bad transitions that must not occur in program computations.

Representation of liveness during synthesis. From Alpern and Schneider [1985], a specification, *spec*, is a liveness specification if and only if for any finite sequence of states α, α maintains *spec*. Our synthesis algorithms do not need liveness specification during synthesis. This is due to the fact that if the fault-intolerant program satisfies its liveness specification then, in the absence of faults, the fault-tolerant program also satisfies it.

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

Example: the specification of the SDS program. Intuitively, the safety specification of the SDS program requires that (i) a read operation on x_i should return the last value written on x_i; (ii) If the value of a bit is damaged, then reading it returns 0; (iii) after a write operation on x_i, the condition $x_i = c_i$ must hold, and (iv) a write operation on a damaged bit has no effect; leaves the value of that bit unchanged. 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.)

$$\begin{aligned}\text{safety}_{SDS} = \ \{(s_0, s_1) \mid\ & ((op_i(s_0) = 0) \wedge (c_i(s_1) \neq x_i(s_0))) \ \vee \\ & ((x_i(s_0) = -1) \wedge (op_i(s_0) = 0) \wedge (c_i(s_1) \neq 0)) \ \vee \\ & ((op_i(s_0) = 1) \wedge (x_i(s_0) \neq -1) \wedge (x_i(s_1) \neq c_i(s_1))) \ \vee \\ & ((op_i(s_0) = 1) \wedge (x_i(s_0) = -1) \wedge (x_i(s_0) \neq x_i(s_1)) \}.\end{aligned}$$

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} captures 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., ctrlState= 1), then a sector must have been selected (i.e., secNum$\neq -1$). Moreover, if sector i has been activated, then the sector number matches with the activation command.

2.4 Faults

The faults that a program is subject to are systematically represented by transitions. A class of faults f for a program $p = \langle V_p, \delta_p, \mathcal{C}_p \rangle$ is a subset of $\{(s_0, s_1) : (s_0, s_1) \in S_p \times S_p\}$. We use $p[]f$ to denote the transitions obtained by taking the union of the transitions in p and the transitions in f. We say that a state predicate T is an f-span (read as fault-span) of p from S iff the following two conditions are satisfied: (1) $S \subseteq T$, and (2) T is closed in $p[]f$. Observe that for all computations of p that start in S, T is a boundary in the state space of p to which (but not beyond which) the state of p may be perturbed by the occurrence of f transitions.

We say that a sequence of states, $\sigma = \langle s_0, s_1, ... \rangle$ is a computation of p in the presence of f iff the following three conditions are satisfied: (1) if σ is infinite, then we have

$\forall j : j > 0 : (s_{j-1}, s_j) \in (\delta_p \cup f)$, (2) if σ is finite and terminates in state s_l, then $\forall j : 0 < j \leq l : (s_{j-1}, s_j) \in (\delta_p \cup f)$, and there does not exist state s such that $(s_l, s) \in \delta_p$, and (3) $\exists n : n \geq 0 : (\forall j : j > n : (s_{j-1}, s_j) \in \delta_p)$. The first requirement captures that in each step, either a program transition or a fault transition is executed. The second requirement captures that faults do not have to execute. That is, if the only transition that starts from s_l is a fault transition (s_l, s_f) then as far as the program is concerned, s_l is still a deadlock state because the program does not have control over the execution of (s_l, s_f); that is, (s_l, s_f) may or may not be fired. Finally, the third requirement captures that the number of fault occurrences in a computation is finite. This requirement is the same as that made in previous work [Arora and Gouda 1993; Arora and Kulkarni 1998c; Dijkstra 1974; Varghese 1993] to ensure that eventually recovery can occur.

Example: faults affecting the SDS program. The SDS program is subject to two classes of faults, namely permanent damage and transient faults. Permanent faults may permanently damage the contents of a bit of information x_i in a sector represented by assigning -1 to x_i (see action F_p). Transient faults may nondeterministically perturb the sector selection/activation commands issued by the controller; i.e., change the values of secNum and activateSec$_i$ (for $i = 0, 1$) arbitrarily between 0 and 1 (see actions F_{t_i}). We *model the effect of faults* on the state of SDS using the following guarded commands ($i = 0, 1$), where $F_t = (F_{t_0} \cup F_{t_1})$.

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

The notation | represents the nondeterministic assignment of one of the values separated by |.

2.5 Fault Tolerance

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

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

We say a program p is failsafe f-tolerant from S for *spec* iff p meets the requirements 1, 2, and 3. The program p is nonmasking f-tolerant from S for *spec* iff p meets the requirements 1, 2 and 4. The program p is masking f-tolerant from S for *spec* iff p meets the requirements 1, 2, 3 and 4.

Notice that we need the last requirement assuming that the program invariant is weak; that is, invariant includes the largest set of states from where the program satisfies its specifications. If designers do not start with a weak invariant, then the last requirement may be considered to be too strong. For example, consider an intolerant mutual exclusion program that, from its invariant S, ensures the mutual exclusion

property (i.e., at most one process accesses the shared data) and also ensures that each process makes progress (i.e., each process can eventually access the shared data). Suppose that this program has been designed in such a way that even if faults violate the mutual exclusion property (e.g., shared data get corrupted), the program would still satisfy its progress properties. That is, the intolerant program guarantees recovery for liveness properties even in a perturbed state outside S. Such a program provides recovery for progress properties without meeting the fourth requirement stated above. Nonetheless, if progress is the only property for which recovery needs to be designed, then S should be weakened so it includes all states from where progress is satisfied even if mutual exclusion is not.

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

Example: Fault tolerance requirements of the SDS program. When a bit is permanently damaged, the failsafe F_p-tolerant SDS program should guarantee that its safety specification $safety_{SDS}$ is always satisfied. The transient faults $F_t = F_{t_0} \cup F_{t_1}$ may nondeterministically assign values to secNum and activateSec$_i$, thereby perturbing the state of SDS outside I_{SDS}. A nonmasking F_t-tolerant SDS program recovers to I_{SDS}; nonetheless, during such recovery one of the safety requirements may be violated; for instance, the value read may not be equal to the last value written.

Other examples for nonmasking fault tolerance include network protocols that maintain a spanning tree of network nodes to facilitate network management and data dissemination [Gartner 2003]. If faults cause the spanning tree to break, the tree is reconstructed. During this reconstruction, some safety constraints may be violated, for instance, some requests remain unsatisfied. However, the system eventually recovers to a state from where a spanning tree is eventually reconstructed.

2.6 Fault Tolerance Synthesizer

In this section, we represent a brief explanation of the input-output of the Fault Tolerance Synthesizer (FTSyn) tool adapted from Ebnenasir [2005] and Ebnenasir et al. [2008]. We do not discuss the internal working of FTSyn as it is outside the scope of this paper (see Ebnenasir et al. [2008] for more details). The input to FTSyn is a text file with the following format (we denote the keywords with **bold** and the comments with *italic* fonts).

```
program progName
var // Declaring variables
    int intVarName_1, domain lowerBoundValue .. upperBoundValue;
     boolean boolVarName_1;
// Defining the processes
process processName_1
    begin
          // A list of guarded commands
          read // A list of variables readable for this process
          write // A list of writeable variables
    end
process processName_2
    begin
          // A list of guarded commands
          read // A list of variables readable for this process
```

```
        write // A list of writeable variables
    end
    ...
process processName_k
    begin
        // A list of guarded commands
        read // A list of variables readable for this process
        write // A list of writeable variables
    end
    // Defining the classes of faults
fault faultClassName_1
    begin
        // A list of guarded commands
    end

invariant // Specifying an invariant
    userDefinedStatePredicate;

specification // Specifying the safety specifications
    userDefinedPredicate;

init // Specifying some initial states
    state // A valuation to all variables
```

In order to specify a fault-intolerant program in FTSyn, we start with the keyword **program** followed by a user defined identifier. Then, we declare program variables. Currently, FTSyn can handle only integer and Boolean types. For an integer variable, we can explicitly specify a finite domain using the keyword **domain**. After variable declaration, we define a set of program processes using the keyword **process**. Each process includes a set of guarded commands and a set of read/write restrictions. In front of the **read** (respectively, **write**) keyword, we specify a set of variables that can be read (respectively, written) by that process. In the case of high atomicity programs, this list includes all program variables, whereas for low atomicity programs this list may include a proper subset of program variables. The specification of each class of faults is very similar to the specification of processes except that faults can be free from any read/write restrictions. To specify an invariant for the input program, we provide the keyword **invariant** that must follow by a state predicate representing a valid invariant. Likewise, using the keyword **specification**, designers can specify the safety specifications as predicates that represent a set of bad transitions. Finally, we should specify a list of initial states; FTSyn will use these initial states along with program/fault guarded commands to create a representation of invariant and fault-span in memory. If FTSyn succeeds in generating a fault-tolerant version of the input program, then the output is represented as a set of processes with revised (and possibly new) guarded commands.

3. PROBLEM STATEMENT

In this section, we formally define the problem of synthesizing multitolerant programs from their fault-intolerant versions. There exist several possible choices in deciding the level of fault tolerance that should be provided in the presence of multiple fault-classes. That is, several options exist in defining what level of guarantees should be provided when transitions of different classes of faults are interleaved with program

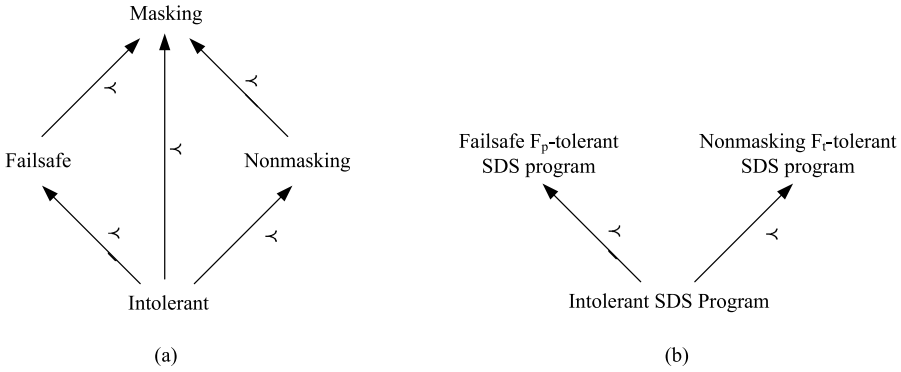

Fig. 3. (a) A *less than* relation, denoted $\prec$, imposed on levels of fault tolerance depending upon the extent to which each level satisfies program specifications (in the presence of faults). (b) Illustration of the $\prec$ relation in the context of the SDS example.

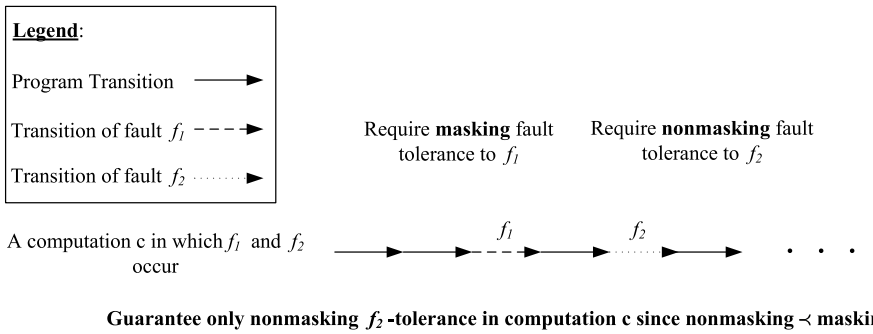

Fig. 4. Nonmasking-Masking multitolerance guarantees *at least* nonmasking f_2-tolerance in the presence of f_1 and f_2.

transitions in a computation. One possibility is to provide no guarantees when two classes of faults f_1 and f_2 appear in the same computation. With such a definition of multitolerance, the program would provide fault tolerance if faults from f_1 occur or if faults from f_2 occur. However, no guarantees will be provided if both faults occur. Another possibility is to provide the minimum level of fault tolerance when f_1 and f_2 occur. In this approach, we impose an ordering on levels of fault tolerance based on a *less than* relation (see Figure 3(a)), denoted $\prec$, which orders two levels of fault tolerance based on the level of guarantees they provide in the presence of faults. For example, failsafe fault tolerance requires only safety when faults occur, whereas masking fault tolerance requires both safety and recovery to invariant. Thus, masking fault tolerance provides more guarantees; that is, failsafe $\prec$ masking. Hence, we have failsafe $\prec$ masking, nonmasking $\prec$ masking, intolerant $\prec$ masking, intolerant $\prec$ failsafe and intolerant $\prec$ nonmasking (see Figure 3(a)). Figure 3(b) illustrates the $\prec$ relation in the context of the SDS example introduced in Section 2.

Using the relation $\prec$, we require a multitolerant program to provide the minimum level of fault tolerance required in the presence of f_1 and f_2 when f_1 and f_2 occur in the same program computation. Figure 4 illustrates this where masking fault tolerance is required to f_1 and nonmasking fault tolerance is desired to f_2, that is, the occurrence of f_2 allows violation of safety. If we were to require masking fault tolerance for cases where f_1 and f_2 occur in the same program computation, then this would mean that safety may be violated in the presence of f_2 alone, however, if f_1 occurs before/after f_2,

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

Fig. 5. Minimum level of fault tolerance (in *italic*) provided for combinations of two levels of fault tolerance.

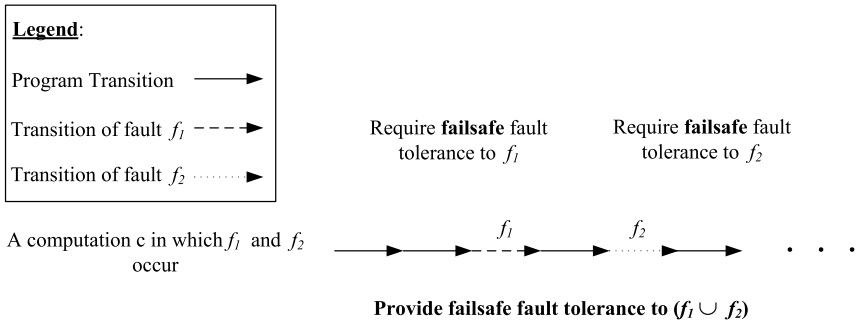

Fig. 6. If failsafe/nonmasking/masking fault tolerance is required for two different classes of faults f_1 and f_2, then one should provide failsafe/nonmasking/masking $f_1 \cup f_2$-tolerance.

then safety must be preserved. This contradicts the notion that faults are undesirable and make it harder for a program to meet its specification. Thus, the level of fault tolerance for any combination of fault-classes will be less than or equal to the level of fault tolerance provided to each class. We follow this approach to define the notion of multitolerance in this section.

We use the $\prec$ relation to determine the level of fault tolerance that should be provided when multiple classes of faults occur in the same program computation. For instance, if masking fault tolerance is required to $f1$ and failsafe (respectively, nonmasking) fault tolerance is desired to $f2$, then failsafe (respectively, nonmasking) fault tolerance should be provided for the case where $f1$ and $f2$ occur. However, if nonmasking fault tolerance is required for $f1$ and failsafe fault tolerance is desired for $f2$, then no level of fault tolerance will be guaranteed for the case where $f1$ and $f2$ occur. Figure 5 illustrates the minimum level of fault tolerance provided for different combinations of levels of fault tolerance.

When a program is subject to several classes of faults for which the same level of fault tolerance is required, the addition of multitolerance amounts to making the union of all those faults as a single fault-class and providing the desired level of fault tolerance for the union. For example, consider the situation where failsafe fault tolerance is required for two classes of faults f_1 and f_2. From the above description, failsafe fault tolerance should be provided for the fault class $f_f = f_1 \cup f_2$ (see Figure 6). Likewise, we obtain the fault-class f_n (respectively, f_m) for which nonmasking (respectively, masking) fault-tolerance is provided. Therefore, hereafter, f_f (respectively, f_n or f_m) denotes the union of all classes of faults for which failsafe (respectively, nonmasking or masking) fault-tolerance is required. We would like to note that while, in this case, we add fault tolerance to the union of fault-classes, it is feasible to apply the stepwise approach proposed in this paper to incrementally add fault tolerance to f_1 and then to f_2 (see Section 5 for details).

Now, given the transitions of a fault-intolerant program, p, its invariant, S, its specification, *spec*, and a set of classes of faults f_f, f_n, and f_m, we define what it means for a program p' (synthesized from p), with invariant S', to be multitolerant by

considering how p' behaves when (i) no faults occur, and (ii) either one of faults f_f, f_n, and f_m occurs. Observe that if faults in f_f and f_m occur in the same computation then safety must be preserved in that computation. In other words, failsafe fault-tolerance must be provided in cases where faults from f_f and/or f_m occur. If faults from f_m alone occur then masking fault-tolerance must also be provided. Thus, the set of faults to which masking fault-tolerance is provided is a subset of the set of faults to which failsafe fault-tolerance is provided. With this intuition, we require that $f_m \subseteq f_f$. Likewise, we require that $f_m \subseteq f_n$. Therefore, we define multitolerant programs as follows:

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

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

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

Now, using the definition of multitolerant programs, we identify the requirements of the problem of synthesizing a multitolerant program, p', from its fault-intolerant version, p. The problem statement is motivated by separating functional concerns from multitolerance. We ensure such a separation of concerns during the design of multitolerant programs from their fault-intolerant version by requiring that no new behaviors are introduced in the absence of faults. This problem statement is a natural extension of the problem statement in Kulkarni and Arora [2000] where fault-tolerance is added to a single class of faults.

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

The Multitolerance Synthesis Problem. Given p, S, *spec*, f_f, f_n, and f_m, identify p' and S' such that

— $S' \subseteq S$,
— $p'|S' \subseteq p|S'$, and
— p' is multitolerant to f_f, f_n, and f_m from S' for *spec*.

We state the corresponding decision problem as follows.

The Multitolerance Decision Problem. Given p, S, *spec*, f_f, f_n, and f_m:

> *Does there exist a program p', with its invariant S' that satisfy the requirements of the synthesis problem*?

Notations. Given a fault-intolerant program p, specification *spec*, invariant S and classes of faults f_f, f_n, and f_m, we say that a program p' and a predicate S' *solve the (multitolerance) synthesis problem* iff p' and S' satisfy the three requirements of the

synthesis problem. We say p' (respectively, S') solves the synthesis problem iff there exists S' (respectively, p') such that p', S' solve the synthesis problem.

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

Example: multitolerant SDS program. For demonstration purposes, we express the multitolerance synthesis problem and its corresponding decision problem in the context of the SDS example (introduced in Section 2) as follows:

The Multitolerance Synthesis Problem for the SDS Example. Given the SDS program in Section 2, its invariant I_{SDS}, its safety specification safety_{SDS}, the permanent damage fault F_p, and the transient fault F_t, identify a revised program SDS' and its invariant $I_{SDS'}$ such that

— $I_{SDS'} \subseteq I_{SDS}$,
— $\delta_{SDS'}|I_{SDS'} \subseteq \delta_{SDS}|I_{SDS'}$, and
— SDS' is failsafe-nonmasking multitolerant to F_p and F_t from $I_{SDS'}$ for safety_{SDS}. That is, SDS' is failsafe F_p-tolerant and nonmasking F_t-tolerant from $I_{SDS'}$ for safety_{SDS}.

The decision problem of designing a multitolerant SDS program is as follows.

The Multitolerance Decision Problem for the SDS Example. Given the intolerant program SDS, its invariant I_{SDS}, its safety specification safety_{SDS}, the permanent damage fault F_p, and the transient fault F_t,

> *Does there exist a program SDS', with its invariant $I_{SDS'}$ that satisfy the requirements of the synthesis problem for the SDS program?*

In Section 6, we present a failsafe-nonmasking multitolerant version of the SDS program. The multitolerant version of the SDS program satisfies the specification of the program in its invariant when no faults occur. In the presence of permanent damage faults, the multitolerant SDS program preserves the safety of the sectors at all times. If transient faults take place, then the multitolerant SDS program guarantees that it will eventually recover to its invariant. If during such a recovery, permanent faults happen, then based on Definition 3.1, the multitolerant SDS program provides no guarantees about its behavior since $F_p \cap F_t = \emptyset$, where $F_t = (F_{t_0} \cup F_{t_1})$.

4. IMPOSSIBILITY OF STEPWISE ADDITION

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

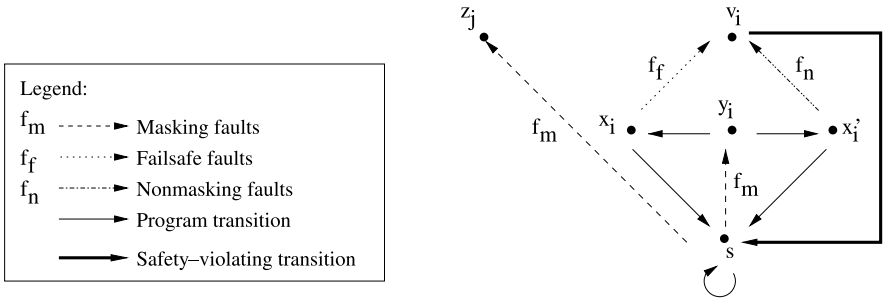

Fig. 7. The states and the transitions corresponding to a propositional variable a_i in the 3-SAT formula. The (v_i, s) transition violates safety.

NP-completeness of the stepwise design of failsafe-nonmasking (FN) multitolerance in Section 4.3.

4.1 Reducing 3-SAT to Multitolerance Synthesis

In this section, we present a polynomial-time reduction from any given instance of the 3-SAT problem to an instance of the decision problem defined in Section 3. A constructed instance of the decision problem of synthesizing multitolerance consists of the fault-intolerant program, p, its invariant, S, its specification, and three classes of faults f_f, f_n, and f_m that perturb p. The problem statement for the 3-SAT problem is as follows:

3-SAT problem. Given is a set of propositional variables, $a_1, a_2, ..., a_n$, and a Boolean formula $c = c_1 \wedge c_2 \wedge ... \wedge c_M$, where each c_j is a disjunction of exactly three literals. (A literal is a propositional variable or its negation.)

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

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

The state space and the invariant of the fault-intolerant program, p. The invariant, S, of the fault-intolerant program, p, includes only one state, say s. Corresponding to the propositional variables and disjunctions of the given 3-SAT instance, we include additional states outside the invariant. For each propositional variable a_i, we introduce the states x_i, x_i', y_i, v_i (see Figure 7). For each disjunction $c_j = (a_i \vee \neg a_k \vee a_r)$ $(1 \leq i \leq n, 1 \leq k \leq n,$ and $1 \leq r \leq n)$, we introduce a state z_j outside the invariant $(1 \leq j \leq M)$.

The transitions of the fault-intolerant program. The only transition in $p|S$ is (s, s).

The transitions of f_m. The set of fault transitions for which masking fault tolerance is required can take the program from s to y_i (corresponding to each a_i). For each disjunction c_j, we also introduce a fault transition that perturbs the program from state s to state z_j $(1 \leq j \leq M)$. Thus, f_m is equal to the set of transitions $\{(s, y_i) : 1 \leq i \leq n\} \cup \{(s, z_j) : 1 \leq j \leq M\}$.

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

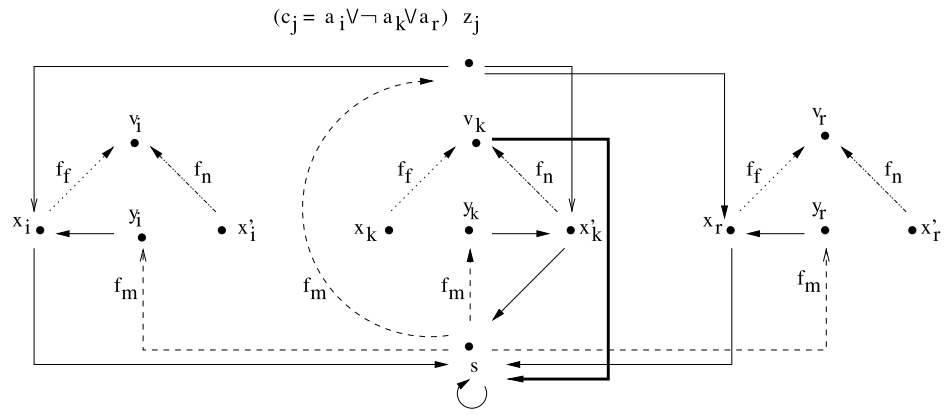

Fig. 8. The partial structure of the multitolerant program where $a_i = true, a_k = false$ and $a_r = true$. For failsafe f_f-tolerance, the deadlock states v_i and v_r are permitted as they are reachable only in computations of $p[]f_f$.

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

The safety specification of the fault-intolerant program, p. None of the fault transitions, namely f_f, f_n, and f_m identified above violates safety. In addition, for each propositional variable a_i $(1 \leq i \leq n)$, the following transitions do not violate safety (see Figure 7):

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

For each disjunction $c_j = a_i \vee \neg a_k \vee a_r$, the following transitions do not violate safety (see Figure 8):

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

The safety specification of the instance of the multitolerance problem forbids the execution of any transition except those identified above. For example, observe that, in Figure 7, the set of transitions (v_i, s), for $1 \leq i \leq n$, violates safety. Observe that the reduction presented in this section is polynomial in the size of the 3-SAT instance.

4.2 Correctness of Reduction

In this section, we show that the given instance of 3-SAT is satisfiable if and only if multitolerance can be added to the problem instance identified in Section 4.1.

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

PROOF. Since the 3-SAT formula is satisfiable, there exists an assignment of truth values to the propositional variables a_i, $1 \leq i \leq n$, such that each c_j, $1 \leq j \leq M$, is *true*. Now, we identify a multitolerant program, p', that is obtained by adding multitolerance to the fault-intolerant program p identified in Section 4.1. The invariant of p' is the same as the invariant of p (i.e., $\{s\}$). We derive the transitions of the multitolerant program p' as follows. (We illustrate a partial structure of p' where $a_i = true$, $a_k = false$, and $a_r = true$ $(1 \leq i, k, r \leq n)$ in Figure 8.) □

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

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

— *p' in the absence of faults.* $p'|S = p|S$. Thus, p' satisfies *spec* in the absence of faults.
— *Masking f_m-tolerance.* If the faults from f_m occur then the program can be perturbed to (1) y_i, $1 \leq i \leq n$, or (2) z_j, $1 \leq j \leq M$. In the first case, if a_i is *true* then there exists exactly one sequence of transitions, $\langle (y_i, x_i), (x_i, s) \rangle$, in $p' [] f_m$. Thus, any computation of $p' [] f_m$ eventually reaches a state in the invariant while preserving safety. If a_i is *false* then there exists exactly one sequence of transitions, $\langle (y_i, x'_i), (x'_i, s) \rangle$, in $p' [] f_m$. By the same argument, any computation of $p' [] f_m$ reaches a state in the invariant without violating safety.
In the second case, since c_j evaluates to *true*, one of the literals in c_j evaluates to *true*. Thus, there exists at least one transition from z_j to some state x_k (respectively, x'_k) where a_k (respectively, $\neg a_k$) is a literal in c_j and a_k (respectively, $\neg a_k$) evaluates to *true*. Moreover, the transition (z_j, x_k) is included in p' iff a_k evaluates to *true*. Thus, (z_j, x_k) is included in p' iff (x_k, s) is included in p'. Since from x_k (respectively, x'_k), there exists no other transition in $p' [] f_m$ except (x_k, s) (respectively, (x'_k, s)), every computation of p' reaches the invariant without violating safety. Thus, p' is masking f_m-tolerant.
— *Failsafe f_f-tolerance.* Based on the case considered above, if only faults from f_m occur then the program is also failsafe fault-tolerant. Hence, we consider only the case where at least one fault from $f_f - f_m$ has occurred. Faults in $f_f - f_m$ occur only in state x_i, $1 \leq i \leq n$. Program p' reaches x_i iff a_i is assigned *true* in the satisfaction of the given 3-SAT formula. Moreover, if a_i is *true* then there is no transition from v_i. Thus, after a fault transition of $f_f - f_m$ occurs, p' simply stops. Note that a failsafe program is allowed to halt outside its invariant without violating safety. Therefore, p' is failsafe f_f-tolerant.
— *Nonmasking f_n-tolerance.* Consider the case where at least one fault transition of $f_n - f_m$ has occurred. Faults in $f_n - f_m$ occur only in state x'_i, $1 \leq i \leq n$. Program p' reaches x'_i iff a_i is assigned *false* in the satisfaction of the given 3-SAT formula. Moreover, if a_i is *false* then the only transition from v_i is (v_i, s). Thus, in the presence of f_n, p' recovers to $\{s\}$.

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

PROOF. Suppose that there exists a multitolerant program p' derived from the fault-intolerant program, p, identified in Section 4.1. Since the invariant of p', S', is nonempty, $S = \{s\}$ and $S' \subseteq S$, S' must include state s. Thus, $S' = S$. Since each y_i, $1 \leq i \leq n$, is directly reachable from s by a fault from f_m, p' must provide safe recovery from y_i to s. Thus, p' must include either (y_i, x_i) or (y_i, x'_i). We make the following truth assignment as follows: If p' includes (y_i, x_i) then we assign a_i to be *true*. If p' includes (y_i, x'_i) then we assign a_i to be *false*. This way, each propositional variable in the 3-SAT formula will get at least one truth assignment. Now, we show that the truth

assignment to each propositional variable is consistent and that each disjunction in the 3-SAT formula evaluates to *true*. □

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

— *Each disjunction is true.* Let $c_j = a_i \vee \neg a_k \vee a_r$ be a disjunction in the given 3-SAT formula. Note that state z_j can be reached by the occurrence of f_m from s. Thus, p' must provide safe recovery from z_j. Since the only safe transitions from z_j are those corresponding to states x_i, x'_k and x_r, p' must include at least one of the transitions (z_j, x_i), (z_j, x'_k), or (z_j, x_r).
Now, we show that the transition included from z_j is consistent with the truth assignment of propositional variables. Consider the case where p' contains transition (z_j, x_i). Thus, p' can reach x_i in the presence of f_m alone. Moreover, let a_i be *false*. Then p' contains the transition (y_i, x'_i). Thus, x'_i can also be reached by the occurrence of f_m alone. Based on the above proof for unique assignment of truth values to propositional variables, p' cannot reach x_i and x'_i in the presence of f_m alone. Hence, if (z_j, x_i) is included in p' then a_i must have been assigned the truth value *true*; i.e., c_j becomes *true*. Likewise, if (z_j, x'_k) is included in p' then a_k must be assigned *false*. Thus, each disjunction evaluates to *true*.

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

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

4.3 NP-Completeness of Failsafe-Nonmasking (FN) Multitolerance

In order to illustrate the NP-completeness of FN multitolerance, we extend the NP-completeness proof of synthesizing multitolerance in that we replace the f_m fault transition (s, y_i) with a sequence of transitions of f_f and f_n as shown in Figure 9. Likewise, we replace fault transition (s, z_j) with a structure similar to Figure 9. Thus, y_i (respectively, z_i) is reachable by f_f faults alone and by f_n faults alone. As a result, v_i is reachable in the computations of $p'[]f_f$ and in the computations of $p'[]f_n$. Thus, to add multitolerance, safe recovery must be added from v_i to s (see Figure 7). Now, we note that with this mapping, the proofs of Lemmas 4.1 and 4.2, and Theorem 4.3 can be easily extended to show that synthesizing FN multitolerance is NP-complete.

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

5. FEASIBILITY OF STEPWISE ADDITION

While, in Section 4, we illustrate that the general case problem of designing multitolerant programs is NP-complete, in our previous work [Kulkarni and Ebnenasir 2004], we have presented sound and complete polynomial algorithms for two special cases, namely, *nonmasking-masking* (NM) multitolerance and *failsafe-masking* (FM)

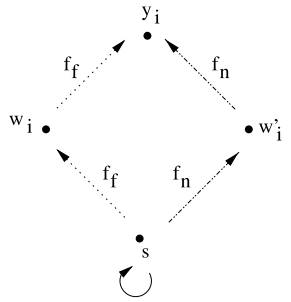

Fig. 9. A proof sketch for NP-completeness of synthesizing failsafe-nonmasking multitolerance.

multitolerance, where we add nonmasking (respectively, failsafe) fault tolerance to one fault-class and masking fault tolerance to another class of faults. Our algorithms in Kulkarni and Ebnenasir [2004] for the addition of FM and NM multitolerance reuse the existing algorithms (presented by Kulkarni and Arora [2000]) for the addition of fault tolerance to a single class of faults. While our algorithms in Kulkarni and Ebnenasir [2004] are deterministically sound and complete, there are two problems with the use of these algorithms in practice. First, these are white-box algorithms in that designers should have knowledge about the internal working of the algorithms in Kulkarni and Arora [2000]. Second, the algorithms in Kulkarni and Ebnenasir [2004] cannot be used in a stepwise fashion (see Figure 1). For instance, in the design of FM multitolerance, consider the case where only the class of faults f_f is known in early stages of development (see Figure 2). As such, we can synthesize a failsafe f_f-tolerant program p_1. Now, if we perform some correctness-preserving maintenance or quality-of-service (e.g., performance) improvements on p_1, then, upon detecting the class of faults f_m (for which masking fault tolerance is required), it is desirable to reuse p_1 in the design of the FM multitolerant program instead of the original intolerant program. To address this problem, in this section, we present a stepwise approach in which we reuse the algorithms in Kulkarni and Arora [2000] as black boxes. First, in Section 5.1, we represent the properties of the algorithms presented in Kulkarni and Arora [2000]. Then, in Sections 5.2 and 5.3, we respectively investigate the stepwise addition of NM and FM multitolerance. Finally, in Section 5.4, we present some sufficient conditions for polynomial-time addition of FN multitolerance.

5.1 Addition of Fault Tolerance to One Fault-Class

In the design of multitolerant programs, we reuse algorithms Add_Failsafe, Add_Nonmasking, and Add_Masking, presented by Kulkarni and Arora [2000]. These algorithms take a program p, its invariant S, its specification *spec*, a class of faults f, and synthesize a failsafe/nonmasking/masking f-tolerant program p' (if one exists) with a new invariant S' and an f-span T' (see Figure 10). The synthesized program p' and its invariant S' satisfy the following requirements: (1) $S' \subseteq S$; (2) $p'|S' \subseteq p|S'$, and (3) p' is failsafe/nonmasking/masking f-tolerant from S' for *spec*. We refer the readers to Kulkarni and Arora [2000] for a comprehensive explanation and the proof of correctness of these algorithms. Nonetheless, for the convenience of the readers, we represent an intuitive description of Add_Failsafe as follows (see Appendix B for a description of Add_Nonmasking, and Add_Masking).

(1) Compute the set of *offending states*, denoted OS, from where a sequence of fault transitions alone violates safety.
(2) Exclude OS from the fault span T. The new fault span is denoted $T' = T - OS$.

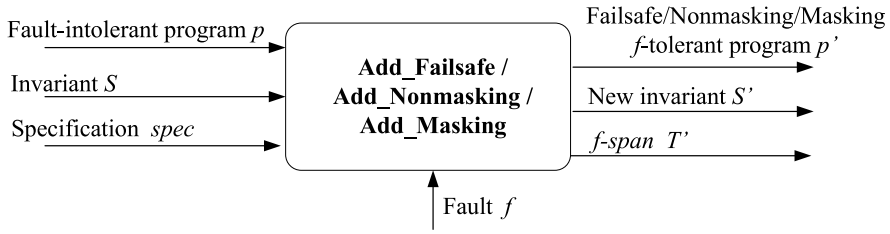

Fig. 10. Input-output of Kulkarni and Arora's algorithms [Kulkarni and Arora 2000] for adding fault tolerance to a single fault-class.

(3) Compute a set of *offending transitions*, denoted OT, that either reach an offending state or directly violate safety of *spec* starting from any state in T'.
(4) Exclude OS from the invariant S; accordingly, eliminate any deadlock states created due to the removal of offending states from S. Denote the remaining set of states by S'.
(5) If S' is empty, then declare that no failsafe fault-tolerant version of p exists and return.
(6) Otherwise, ensure the closure of S' by removing any transition that starts in S' and terminates outside S'.
(7) Return p', S', T'.

In addition to removing offending states/transitions, the Add_Masking algorithm adds new recovery transitions from deadlock states in $(T' - S')$ to S' while preserving the safety of *spec*. The Add_Nonmasking algorithm only adds recovery transitions to the invariant. In this section, we recall the relevant properties of these algorithms. We note that the description of the multitolerance algorithms and their proofs depend *only* on the properties mentioned in this section and not on the actual implementation of the algorithms in Kulkarni and Arora [2000].

For Add_Failsafe and Add_Masking, the invariant S' has the property of being the largest such possible invariant for any failsafe (respectively, masking) program obtained by adding fault tolerance to the given fault-intolerant program. More precisely, if there exists a failsafe (respectively, masking) fault-tolerant program p'', with invariant S'' that has been designed without using Add_Failsafe (respectively, Add_Masking), and the following conditions are satisfied: (1) $S'' \subseteq S$; (2) $p''|S'' \subseteq p|S''$, and (3) p'' is failsafe (respectively, masking) f-tolerant from S'' for *spec*, then S'' is a subset of S'. Moreover, if the invariant S does not include any offending states, then Add_Failsafe will not change the invariant of the fault-intolerant program. Now, let the input for Add_Failsafe be p, S, *spec* and f. Let the output of Add_Failsafe be the fault-tolerant program p' and invariant S'. We state the following properties.

Property 5.1.1. If any program p'' with invariant S'' satisfies (i) $S'' \subseteq S$; (ii) $p''|S'' \subseteq p|S''$, and (iii) p'' is failsafe f-tolerant from S'' for *spec*, then $S'' \subseteq S'$.

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

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

Fig. 11. Stepwise addition of NM multitolerance.

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

Property 5.1.3. Let the input of Add_Masking be p, S, *spec* and f. Let the output of Add_Masking be the fault-tolerant program p', invariant $S' \subseteq S$, and fault-span T'. If any program p'' with invariant S'' satisfies (i) $S'' \subseteq S$; (ii) $p''|S'' \subseteq p|S''$, (iii) p'' is masking f-tolerant from S'' for *spec*, and (iv) T'' is the fault-span used for verifying the masking fault tolerance of p'' then $S'' \subseteq S'$ and $T'' \subseteq T'$.

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

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

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

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

Based on Properties 5.1.1– 5.1.6, Kulkarni and Arora [Kulkarni and Arora 2000] show that the algorithms Add_Failsafe, Add_Nonmasking, and Add_Masking are sound and complete, that is, the output of these algorithms satisfies the three requirements for adding fault tolerance to a *single* class of faults (see Figure 10) and these algorithms can find a fault-tolerant version of the fault-intolerant program if one exists.

THEOREM 5.1.7. *The algorithms* Add_Failsafe, Add_Nonmasking, *and* Add_Masking *are sound and complete.* (*see Kulkarni and Arora [2000] for proof.*)

5.2 Nonmasking-Masking (NM) Multitolerance

In this section, we present a sound and complete algorithm (see Figure 11) for stepwise design of NM multitolerant programs from their fault-intolerant versions that are subject to two classes of faults f_n and f_m, where $f_m \subseteq f_n$. Formally, given a program p, with its invariant S, its specification *spec*, our goal is to synthesize a program p', with invariant S' that is NM multitolerant to f_n and f_m from S' for *spec*. By definition, p' must be masking f_m-tolerant and nonmasking f_n-tolerant. Towards this end, we proceed as follows: Using the algorithm Add_Masking, we synthesize a masking f_m-tolerant program p_1, with invariant S_1, and fault-span T_m (Line 1 in Figure 11). Now, since program p_1 is masking f_m-tolerant, it provides safe recovery to its invariant, S_1,

from every state in $(T_m - S_1)$. Thus, in the presence of f_n, if p_1 is perturbed to $(T_m - S_1)$ then p_1 will satisfy the requirements of nonmasking fault tolerance (i.e., recovery to S_1). However, if f_n perturbs p_1 to a state s, where $s \notin T_m$, then recovery must be added from s. Based on Properties 5.1.5 and 5.1.6, it suffices to add recovery to T_m as provided recovery by p_1 from T_m to S_1 can be reused *even after* adding nonmasking fault tolerance. We invoke Add_Nonmasking (Line 3 in Figure 11) with T_m as an invariant of p_1.

THEOREM 5.2.1. *The* Add_Masking_Nonmasking *algorithm is sound.*

PROOF. Based on the soundness of Add_Masking (see Theorem 5.1.7), we have $S_1 \subseteq S$. The equality $S_1 = S'$ follows from the Property 5.1.5. Also, using the soundness of Add_Masking, we have $p_1|S_1 \subseteq p|S_1$ (i.e., $p_1|S' \subseteq p|S'$). In addition, based on the Property 5.1.6, we have $p_1|S' = p'|S'$. As a result, we have $p'|S' \subseteq p|S'$.

Now, we show that p' is multitolerant to f_n and f_m from S' for *spec*:

(1) *Absence of faults.* From the soundness of Add_Masking, it follows that p_1 satisfies *spec* from S_1 $(= S')$ in the absence of faults. Add_Nonmasking does not add/remove any transitions to/from $p_1|S'$ (Property 5.1.6). Thus, it follows that p' satisfies *spec* from S'.
(2) *Masking f_m-tolerance.* From the soundness of Add_Masking, p_1 is masking f_m-tolerant from $S_1(= S')$ for *spec*. Also, based on the Property 5.1.5 and 5.1.6, since $p_1|T_m = p'|T_m$, Add_Nonmasking preserves masking f_m-tolerance property of p_1. Therefore, p' is masking f_m-tolerant from S' for *spec*.
(3) *Nonmasking f_n-tolerance.* From the soundness of Add_Nonmasking, we know that p' is nonmasking f_n-tolerant from T_m for *spec*. Also, since Add_Nonmasking preserves masking f_m-tolerance property of p_1, recovery from T_m to S' is guaranteed in the presence of f_n. Therefore, p' is nonmasking f_n-tolerant from S' for *spec*. □

THEOREM 5.2.2. *The algorithm* Add_Masking_Nonmasking *is complete.*

PROOF. Add_Masking_Nonmasking declares that a multitolerant program does not exist only when Add_Masking does not find a masking f_m-tolerant program. Therefore, the completeness of Add_Masking_Nonmasking follows from the completeness of Add_Masking. □

We have illustrated [Ebnenasir and Kulkarni 2008] that nonmasking f_n-tolerance and masking f_m-tolerance can also be added in a reverse order, which has important applications where f_m is unknown at the early stages of design and any correctness-preserving revisions (e.g., for performance enhancement) performed on the nonmasking program have to be preserved while adding masking f_m-tolerance. Consider a nonmasking f_n-tolerant program p_1 with its invariant S_1 and its f_n-span T_1. Adding masking f_m-tolerance to p_1 may result in removing a set of offending states in T_1 (and S_1) from where a sequence of transitions of f_m directly violates safety. One side effect of eliminating such offending states is that the invariant S_1 may be contracted, resulting in a new invariant $S_2 \subseteq S_1$ for the masking program p_2. Moreover, the elimination of such offending states may destroy the recovery to S_1 that was already designed in p_1. To fix these issues, we invoke Add_Nonmasking again on the intermediate program p_2, its f_m-span T_2, and the fault-class f_n, thereby synthesizing a masking-nonmasking multitolerant program in three steps (i.e., add nonmasking, add masking, add nonmasking).

```
Add_Failsafe_Masking(p: transitions, f_f, f_m: fault, S: state predicate,
                                          spec: safety specification)
{
  p_1, S_1, T_1 := Add_Failsafe(p, f_f, S, spec);                          (1)
  if (S_1 = {})   declare no multitolerant program p' exists;              (2)
                  return ∅, ∅;
  spec' := spec ∪ {(s_0, s_1) : s_0 ∈ T_1 ∧ s_1 ∉ T_1};                     (3)
  p', S', T_m := Add_Masking(p_1, f_m, S_1, spec');                        (4)
  if (S' = {})    declare no multitolerant program p' exists;              (5)
                  return ∅, ∅;
  return p', S';                                                           (6)
}
```

Fig. 12. Stepwise addition of FM multitolerance.

5.3 Failsafe-Masking (FM) Multitolerance

In this section, we investigate the stepwise addition of Failsafe-Masking (FM) multitolerance to high atomicity programs that tolerate two classes of faults f_f and f_m for which failsafe and masking fault tolerance are respectively required, where $f_m \subseteq f_f$. We start by reusing the Add_Failsafe algorithm (Line 1 in Figure 12), where we add failsafe f_f-tolerance to p. The resulting program p_1 provides failsafe f_f-tolerance from its invariant S_1. To add masking f_m-tolerance to p_1, we use the Add_Masking algorithm. Based on Property 5.1.4, Add_Masking ensures that T_m does not include any state from where a sequence of f_m transitions alone violates safety. However, since $f_m \subseteq f_f$, in the addition of masking f_m-tolerance to p_1, the fault-span of the resulting program may include states from where transitions of $f_f - f_m$ alone violate safety. Even though such states do not belong to T_1, we need to ensure that during the addition of masking f_m-tolerance (Line 4 in Figure 12) they are not included in the fault-span T_m. Towards this end, we strengthen the safety specification (Line 3 in Figure 12), denoted *spec*′, by including the transitions that reach outside T_1 in the set of bad transitions. Finally, we invoke the Add_Masking algorithm to add masking f_m-tolerance to p_1 from S_1 for *spec*′.

THEOREM 5.3.1. *The* Add_Failsafe_Masking *algorithm is sound.*

PROOF. Using the soundness of Add_Failsafe, we have $S_1 \subseteq S$. Based on the Property 5.1.3, we have $S' \subseteq S_1$. Also, from the soundness of Add_Failsafe, it follows that $p_1|S_1 \subseteq p|S_1$. Since $S' \subseteq S_1$ and $p_1|S_1 \subseteq p|S_1$, we have $p_1|S' \subseteq p|S'$. From the soundness of Add_Masking, it follows that $p'|S' \subseteq p_1|S'$. Therefore, we have $p'|S' \subseteq p|S'$. □

Now, we show that p' (see Figure 12) is indeed multitolerant to f_f and f_m from S' for *spec*.

(1) *Absence of faults.* From the soundness of Add_Failsafe, it follows that p_1 satisfies *spec* from S_1 in the absence of faults. Also, based on the soundness of Add_Masking, p' satisfies *spec*′ from S' in the absence of faults. Since *spec*′ is a strengthened version of *spec* (i.e., the set of bad transitions ruled out by *spec* is a subset of the set of bad transitions ruled out by *spec*′), it follows that p' satisfies *spec* from S' in the absence of faults.
(2) *Failsafe f_f-tolerance.* From the soundness of Add_Failsafe, p_1 is failsafe f_f-tolerant from S_1 for *spec*. Thus, no computation of p_1 violates safety of specification from T_1 (respectively, from S_1). Also, based on the soundness of Add_Masking, p' does not

execute any transitions that violate *spec*′; i.e., no state outside T_1 becomes reachable due to the addition of masking f_m-tolerance. Since $S' \subseteq S_1$, no computation of p' will ever violate *spec* in the presence of f_f from S'. Therefore, p' is failsafe f_f-tolerant from S' for *spec*.

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

THEOREM 5.3.2. *The* Add_Failsafe_Masking *algorithm is complete.*

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

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

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

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

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

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

We have adopted the definition of fault-safe specifications from Kulkarni and Ebnenasir [2005a]. The examples of fault-safe specifications include important problems such as Byzantine agreement, consensus and commit (see Kulkarni and Ebnenasir [2005a]). If the specification *spec* is f_f-safe then no sequence of f_f transitions alone will violate the safety of *spec* from S. As a result, the invariant of the multitolerant program, S', will be equal to S (see Property 5.1.2). If the set of states that are reachable outside the invariant in the computations of $p[]f_f$ is disjoint from the set of states that are reachable in the computations of $p[]f_n$ then the program p can distinguish the occurrence of f_f from the occurrence of f_n by respectively detecting the disjoint state predicates $(T_f - S)$ and $(T_n - S)$. Thus, in order to guarantee FN multitolerance, program p should guarantee (i) recovery to S if p detects that fault f_n has occurred, and (ii) safety if p detects that fault f_f has occurred.

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

Fig. 13. Synthesizing failsafe-nonmasking multitolerance for mutually exclusive faults.

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

Next, we present the Add_Failsafe_Nonmasking algorithm (see Figure 13) that adds FN multitolerance to p in polynomial time if (i) faults f_f and f_n are mutually exclusive with respect to p and its invariant S, and (ii) the specification *spec* is f_f-safe.

Since *spec* is f_f-safe, Add_Failsafe does not remove any states from S, and as a result, $S_1 = S$. For this reason, this step of the algorithm is always successful; that is, Add_Failsafe always finds a failsafe f_f-tolerant program. In the next step, we reuse the Add_Nonmasking algorithm from Kulkarni and Arora [2000] to add nonmasking f_n-tolerance to p_1.

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

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

(1) *Absence of faults.* Since the equalities $S = S'$ and $p'|S' = p|S$ hold, it follows that every computation of p' starting in S' is a computation of p. Thus, p' satisfies *spec* from S'.
(2) *Failsafe f_f-tolerance.* From the soundness of Add_Failsafe, p_1 is failsafe f_f-tolerant from S_1 for *spec*. Since $S' = S_1$, p_1 is failsafe f_f-tolerant from S' for *spec*. Based on Properties 5.1.5 and 5.1.6, Add_Nonmasking does not add (respectively, remove) any transition in $p_1|S$. Also, since $(T_f - S)$ and $(T_n - S)$ are disjoint (by mutual exclusivity of f_f and f_n), Add_Nonmasking does not add any transitions to $p_1|(T_f - S)$. Hence, we have $p_1|T_f = p'|T_f$. Therefore, in the presence of f_f, p' will never execute a safety violating transition, and as a result, p' is failsafe f_f-tolerant from S' for *spec*.
(3) *Nonmasking f_n-tolerance.* Since $S' = S$ and recovery is provided from $(T_n - S)$ to S, p' is nonmasking f_n-tolerant from S' for *spec*. □

THEOREM 5.4.5. *The algorithm* Add_Failsafe_Nonmasking *has polynomial-time complexity.* (*Proof is straightforward, hence omitted.*)

6. EXAMPLES

In this section, we present three examples for stepwise addition of multitolerance. First, in Section 6.1, we present a failsafe-nonmasking multitolerant version of the SDS program (introduced in Section 2). This example illustrates how our black-box stepwise approach enables the reuse of a performance improvement change in the intermediate failsafe program in the design of the multitolerant program. Second, in

Section 6.2, we present a failsafe-nonmasking-masking multitolerant token ring program that is subject to three classes of faults. Third, we present a nonmasking-masking repetitive Byzantine agreement protocol in Section 6.3.

6.1 Failsafe-Nonmasking Multitolerant Stable Disk Storage

In this section, we demonstrate how our stepwise design method facilitates the design of a failsafe-nonmasking multitolerant version of the SDS program introduced in Section 2. Specifically, we first add failsafe fault tolerance to permanent damage faults F_p. As a result, we generate an intermediate failsafe program that does not tolerate transient faults F_{t_i}, where $i = 0, 1$. Then, we redesign the failsafe program in order to capture a write performance improvement strategy used in the design of disk storage systems [English and Stepanov 1991], called the *continuous write* strategy. In such a strategy, once a write operation is performed on a sector, the controller ensures that $k-1$ subsequent writes will be carried out on the same sector to increase the locality of write operations, where k is a fixed value. After implementing the continuous write strategy, we add nonmasking fault tolerance to transient faults to generate a multitolerant program while preserving failsafe and continuous write properties.

Sufficient conditions. We illustrate that the sufficient conditions of Theorem 5.4.4 hold for the SDS system. The specification safety$_{SDS}$ (see Section 2) is fault-safe for F_p since the occurrence of the permanent damage faults does not directly violate safety$_{SDS}$. Thus, the set of offending states is empty. Nonetheless, when permanent damage faults occur and perturb the program outside I_{SDS}, the intolerant SDS program may write on damaged bits, thereby violating safety$_{SDS}$. Such write operations constitute the set of offending transitions originating outside I_{SDS} that must not be executed by a failsafe version of the SDS program.

To demonstrate that the permanent faults and transient faults are mutually exclusive, we first specify T_{F_p} and T_{F_t} as follows:

$$
\begin{aligned}
T_{F_p} = \ & \{s \mid ((x_0(s) = -1) \vee (x_1(s) = -1)) \wedge \\
& \qquad ((\text{ctrlState}(s) \neq 1) \vee (\text{secNum}(s) \neq -1)) \ \wedge \\
& \qquad ((\text{activateSec}_0(s) \neq 1) \vee (\text{secNum}(s) = 0)) \ \wedge \\
& \qquad ((\text{activateSec}_1(s) \neq 1) \vee (\text{secNum}(s) = 1)) \} \\
T_{F_t} = \ & \{s \mid ((x_0(s) \neq -1) \wedge (x_1(s) \neq -1)) \wedge \\
& \qquad \neg \ (\ ((\text{ctrlState}(s) \neq 1) \vee (\text{secNum}(s) \neq -1)) \ \wedge \\
& \qquad \ \ ((\text{activateSec}_0(s) \neq 1) \vee (\text{secNum}(s) = 0)) \ \wedge \\
& \qquad \ \ ((\text{activateSec}_1(s) \neq 1) \vee (\text{secNum}(s) = 1)) \ ) \ \}
\end{aligned}
$$

The state predicate T_{F_p} includes all states where at least one of the variables x_0 or x_1 is equal to -1, that is, permanently damaged, and the states where the controller is not perturbed by transient faults. The state predicate T_{F_t} contains states where x_0 and x_1 are not damaged, but the controller might have been perturbed outside the invariant by transient faults. As such, the state predicates $(T_{F_p} - I_{SDS})$ and $(T_{F_t} - I_{SDS})$ are disjoint. That is, faults F_p and F_t are mutually exclusive for the SDS program and its invariant I_{SDS}.

Failsafe fault tolerance to permanent damage faults. In the first step of the algorithm in Figure 13, we generate the following failsafe program with two revised actions S'_{i1} and S'_{i3}, for $i = 0, 1$. 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. The constraint ($x_i \neq -1$) in action S'_{i3} guarantees that a write operation will not take place on a damaged bit.

$$
\begin{aligned}
S'_{i1}: \ & (op_i = 0) \ \wedge \ ((x_i = 0) \vee \mathbf{(x_i = -1)}) \ \wedge \ (SecNum = i) \ \wedge \ (activateSec_i = 1) \\
& \longrightarrow \ c_i := 0; \ secNum := -1; \ activateSec_i := 0; \\
S'_{i2}: \ & (op_i = 0) \ \wedge \ (x_i = 1) \ \wedge \ (SecNum = i) \ \wedge \ (activateSec_i = 1) \\
& \longrightarrow \ c_i := 1; \ secNum := -1; \ activateSec_i := 0; \\
S'_{i3}: \ & (op_i = 1) \ \wedge \ (SecNum = i) \ \wedge \ (activateSec_i = 1) \wedge \mathbf{(x_i \neq -1)} \\
& \longrightarrow \ x_i := c_i; \ secNum := -1; \ activateSec_i := 0;
\end{aligned}
$$

Capturing the continuous-write strategy. In order to decrease the overall access time of SDS, we redesign the failsafe F_p-tolerant program to capture a continuous-write strategy explained in the beginning of this section. For this reason, we introduce the following new variables: (1) prevOp is a variable with domain $\{0, 1\}$, where 0 denotes that the previous operation has been a read operation, and 1 represents a previous write operation; (2) prevSecNum also has a domain $\{0, 1\}$ representing which sector has done the previous operation, and (3) writeCounter keeps the number of consecutive writes that so far has been performed on a sector. We also consider a constant k whose fixed value determines how many successive writes can take place on a sector. The below actions illustrate the improved design of the controller of the failsafe SDS program, denoted as the SDS′ program. We note that we have implemented the continuous-write policy for $k = 3$.

$$
\begin{aligned}
C'_3: \ & (ctrlState = 1) \ \wedge \ (secNum = 0) \ \wedge \ (activateSec_0 = 0) \ \wedge \\
& \mathbf{((writeCounter \ \geq \ k) \ \vee \ (writeCounter = 0))} \\
& \longrightarrow \ ctrlState := 0; \\
& \quad op_0 := 0; \\
& \quad activateSec_0 := 1; \\
& \quad \mathbf{writeCounter} := 0;
\end{aligned}
$$

$$
\begin{aligned}
C_{41}: \ & (ctrlState = 1) \ \wedge \ (secNum = 0) \ \wedge \ (activateSec_0 = 0) \ \wedge \mathbf{(prevOp = 1)} \ \wedge \\
& \mathbf{(prevSecNum = 0)} \ \wedge \ \mathbf{(writeCounter < k)} \\
& \longrightarrow \ ctrlState := 0; \\
& \quad op_0 := 1; \ c_0 := 0|1; \\
& \quad activateSec_0 := 1; \\
& \quad \mathbf{writeCounter}{+}{+}; \\
C_{42}: \ & (ctrlState = 1) \ \wedge \ (secNum = 0) \ \wedge \ (activateSec_0 = 0) \ \wedge \mathbf{(prevOp = 1)} \ \wedge \\
& \mathbf{(prevSecNum = 1)} \ \wedge \ \mathbf{(writeCounter < k)} \\
& \longrightarrow \ ctrlState := 0; \\
& \quad op_1 := 1; \ c_1 := 0|1; \\
& \quad activateSec_1 := 1; \\
& \quad \mathbf{writeCounter}{+}{+}; \\
& \quad \mathbf{secNum} := 1; \\
C_{43}: \ & (ctrlState = 1) \ \wedge \ (secNum = 0) \ \wedge \ (activateSec_0 = 0) \ \wedge \\
& \mathbf{(prevOp = 0)} \ \wedge \ \mathbf{(writeCounter < k)} \\
& \longrightarrow \ ctrlState := 0; \\
& \quad op_0 := 1; \ c_0 := 0|1; \\
& \quad activateSec_0 := 1; \\
& \quad \mathbf{writeCounter}{+}{+}; \\
C_{44}: \ & (ctrlState = 1) \ \wedge \ (secNum = 0) \ \wedge \ (activateSec_0 = 0) \ \wedge \\
& \mathbf{(writeCounter \geq k)} \ \longrightarrow \ writeCounter := 0;
\end{aligned}
$$

The **bold** fonts denote new code added to the design of the controller of the failsafe program. Since actions C_1 and C_2 in the controller do not change, we have omitted them. Action C'_3 is the revised version of C_3 that issues a read command to the first sector under the additional constraint that a continuous write is not taking place. C'_3 resets the writeCounter so a continuous write can start anytime after a read. The actions C_{41}, C_{42}, C_{43} and C_{44} replace C_4, where each one of them may issue a write command to either one of the sectors depending on the previous write operation. If the controller has received a write request (represented by the fact that action C'_3 has not been executed), the previous operation has been a write on Sector 0, and the number of writes is less than k, then action C_{41} issues a write command for Sector 0. If the controller has received a write request, the previous operation has been a write on Sector 1, and the number of writes is less than k, then the controller issues a write command for Sector 1 (see Action C_{42}) instead of Sector 0 in order to implement the continuous write policy. Note that action C_{42} sets secNum and activateSec_1 to 1 in order to preserve the invariant I_{SDS}. Action C_{43} issues a write command for Sector 0 given that the previous operation has been a read operation. Action C_{44} resets the write counter if its value is greater than or equal to k. The revisions of the actions related to issuing commands for the second sector are symmetric to the revisions for the first sector. Action C'_5 replaces the action C_5 and actions C_{61}, C_{62}, C_{63} and C_{64} replace action C_6.

C'_5 : $(\text{ctrlState} = 1) \wedge (\text{secNum} = 1) \wedge (\text{activateSec}_1 = 0) \wedge$
((writeCounter $\geq k$**)** $\vee$ **(writeCounter = 0))**
$\longrightarrow$ ctrlState:= 0;
$\text{op}_1 := 0$;
$\text{activateSec}_1 := 1$;
writeCounter := 0;

C_{61} : $(\text{ctrlState} = 1) \wedge (\text{secNum} = 1) \wedge (\text{activateSec}_1 = 0) \wedge$ **(prevOp = 1)** $\wedge$
(prevSecNum = 1) $\wedge$ **(writeCounter** $< k$**)**
$\longrightarrow$ ctrlState:= 0;
$\text{op}_1 := 1$; $\text{c}_1 := 0|1$;
$\text{activateSec}_1 := 1$;
writeCounter++;

C_{62} : $(\text{ctrlState} = 1) \wedge (\text{secNum} = 1) \wedge (\text{activateSec}_1 = 0) \wedge$ **(prevOp = 1)** $\wedge$
(prevSecNum = 0) $\wedge$ **(writeCounter** $< k$**)**
$\longrightarrow$ ctrlState:= 0;
$\text{op}_0 := 1$; $\text{c}_0 := 0|1$;
$\text{activateSec}_0 := 1$;
writeCounter++;
secNum := 0;

C_{63} : $(\text{ctrlState} = 1) \wedge (\text{secNum} = 1) \wedge (\text{activateSec}_1 = 0) \wedge$
(prevOp = 0) $\wedge$ **(writeCounter** $< k$**)**
$\longrightarrow$ ctrlState:= 0;
$\text{op}_1 := 1$; $\text{c}_1 := 0|1$;
$\text{activateSec}_1 := 1$;
writeCounter++;

C_{64} : $(\text{ctrlState} = 1) \wedge (\text{secNum} = 1) \wedge (\text{activateSec}_1 = 0) \wedge$
(writeCounter $\geq k$**)** $\longrightarrow$ writeCounter := 0;

Nonmasking fault tolerance to transient faults. While the SDS′ program is failsafe tolerant to permanent damage faults, the occurrence of transient faults may perturb the state

of SDS′ to an arbitrary state outside I_{SDS}. In such scenarios, after the controller selects and activates a sector, transient faults may cause the SDS′ program to deadlock in a state where secNum and activateSec variables do not match; e.g., secNum is set to 1 representing the selection of Sector 1, but activateSec_1 is set to 0, and activateSec_0 is set to 1. When transient faults stop occurring, a multitolerant SDS′ program should recover to its invariant I_{SDS}. As such, we should design nonmasking fault tolerance while preserving failsafe fault tolerance. Using the white-box approach we proposed in Kulkarni and Ebnenasir [2004], such a multitolerant program is designed from the intolerant program and not from the intermediate failsafe program, thereby destroying the continuous-write property. In this paper, we address this problem by a black-box method where after designing an intermediate fault-tolerant program (e.g., a failsafe program) and performing some maintenance or quality of service improvement (e.g., continuous-write), we design a new level of fault tolerance (e.g., nonmasking) while preserving failsafe fault tolerance and continuous-write properties. The following actions provide the necessary recovery for the SDS program:

$$
\begin{aligned}
Rec_1: \ & (\text{activateSec}_0 = 1) \wedge (\text{activateSec}_1 = 0) \wedge (\text{secNum} \neq 0) \wedge (\text{op}_0 = 0) \\
& \longrightarrow \ c_0 := x_0; \\
& \quad \text{secNum} := -1; \\
& \quad \text{activateSec}_0 := 0; \\
Rec_2: \ & (\text{activateSec}_0 = 1) \wedge (\text{activateSec}_1 = 0) \wedge (\text{secNum} \neq 0) \wedge (\text{op}_0 = 1) \\
& \longrightarrow \ x_0 := c_0; \\
& \quad \text{secNum} := -1; \\
& \quad \text{activateSec}_0 := 0; \\
& \quad \text{writeCounter++};
\end{aligned}
$$

If only Sector 0 has been activated but secNum is not 0 and a read command has been given to Sector 0, then action Rec_1 recovers SDS by reading the value of x_0 to c_0, setting secNum to -1 and correcting the value of activateSec_0 to 0. Action Rec_3 performs a similar recovery action for Sector 1. Further, actions Rec_2 and Rec_4 recover the sectors from a deadlock state where a write command has been issued.

$$
\begin{aligned}
Rec_3: \ & (\text{activateSec}_1 = 1) \wedge (\text{activateSec}_0 = 0) \wedge (\text{secNum} \neq 1) \wedge (\text{op}_1 = 0) \\
& \longrightarrow \ c_1 := x_1; \\
& \quad \text{secNum} := -1; \\
& \quad \text{activateSec}_1 := 0; \\
Rec_4: \ & (\text{activateSec}_1 = 1) \wedge (\text{activateSec}_0 = 0) \wedge (\text{secNum} \neq 1) \wedge (\text{op}_1 = 1) \\
& \longrightarrow \ x_1 := c_1; \\
& \quad \text{secNum} := -1; \\
& \quad \text{activateSec}_1 := 0; \\
& \quad \text{writeCounter++};
\end{aligned}
$$

In addition to reaching deadlock states in the presence of transient faults, the SDS′ program may reach states outside I_{SDS} from where a sequence of actions create a nonprogress cycle. For example, after the controller issues a read command for Sector 1 (i.e., secNum = 1 and $\text{activateSec}_1 = 1$), transient faults occur and set secNum to 0. Subsequently, the controller may send a read command to Sector 0 and after Sector 0 performs the read operation, this sequence gets repeated forever. That is, Sector 1 is never given a chance of read as long as a read request arrives for Sector 0. To correct such a nonprogress cycle, the constraint ($\text{activateSec}_1 = 0$) is added to action C'_3 during the addition of nonmasking tolerance to SDS′. For similar reasons, our algorithms automatically add the same constraint to the guard of action C_{41}.

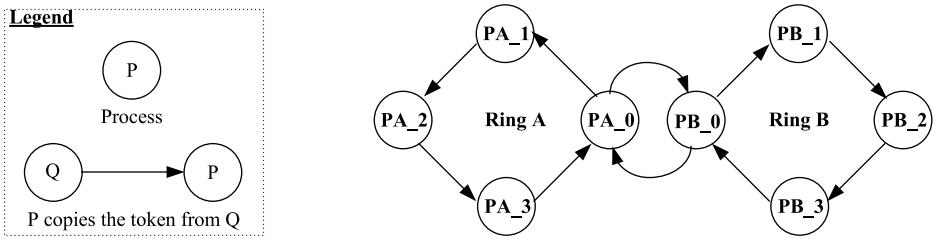

Fig. 14. The two-ring token passing program.

Likewise, the constraint ($\text{activateSec}_0 = 0$) is added to the guards of the actions C'_5 and C_{61}.

6.2 Multitolerant Token Passing

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

The Two-Ring Token Passing (TRTP) program. The TRTP program includes 8 processes located in two rings A and B (see Figure 14). In Figure 14, the arrows show the direction of token passing. Process PA_i (respectively, PB_i), $0 \leq i \leq 2$, is the predecessor of PA_{i+1} (respectively, PB_{i+1}). Process PA_3 (respectively, PB_3) is the predecessor of PA_0 (respectively, PB_0). Each process PA_i (respectively, PB_i), $0 \leq i \leq 3$, has an integer variable a_i (respectively, b_i) with the domain {-1, 0, 1, 2, 3}, where -1 represents a detectably corrupted value. Moreover, process PA_i (respectively, PB_i), $0 \leq i \leq 3$, has a Boolean variable, denoted upa_i (respectively, upb_i), that represents whether or not that process has crashed.

Process PA_i, for $1 \leq i \leq 3$, *has the token* iff $(a_{i-1} = a_i \oplus 1) \wedge upa_i \wedge (a_{i-1} \neq -1) \wedge (a_i \neq -1)$, where $\oplus$ denotes addition modulo 4. Intuitively, PA_i has the token iff a_i is one unit less a_{i-1}, PA_i has not crashed, and a_i and a_{i-1} are not detectably corrupted. Process PA_0 has the token iff $(a_0 = a_3) \wedge (b_0 = b_3) \wedge (a_0 = b_0) \wedge upa_0 \wedge (a_0 \neq -1) \wedge (a_3 \neq -1)$; that is, PA_0 has the same value as its predecessor and that value is equal to the values held by PB_0 and PB_3, PA_0 has not crashed and a_0 and a_3 are not corrupted. Process PB_0 has the token iff $(b_0 = b_3) \wedge (a_0 = a_3) \wedge ((b_0 \oplus 1) = a_0) \wedge upb_0 \wedge (b_0 \neq -1) \wedge (b_3 \neq -1)$. That is, PB_0 has the same value as its predecessor and that value is one unit less than the values held by PA_0 and PA_3, PB_0 has not crashed and b_0 and b_3 are not corrupted. Process PB_i $(1 \leq i \leq 3)$ *has the token* iff $(b_{i-1} = b_i \oplus 1) \wedge upb_i \wedge (b_{i-1} \neq -1) \wedge (b_i \neq -1)$. The TRTP program also has a Boolean variable *turn*; ring A executes only if *turn* = *true*, and if ring B executes then *turn* = *false*. Using the following actions, the processes circulate the token in rings A and B ($i = 1, 2, 3$):

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

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

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

Invariant. Consider a state s_0 where $(\forall i : 0 \leq i \leq 3 : (a_i = 0) \wedge (b_i = 0))$ and *turn* is true in s_0. The invariant of the TRTP program contains all the states that are reached

from s_0 by the execution of actions AC_i and BC_i, for $0 \leq i \leq 3$. Starting from a state s_0 where $(turn(s_0) = true) \wedge (\forall i : 0 \leq i \leq 3 : (a_i(s_0) = 0) \wedge (b_i(s_0) = 0))$, process PA_0 has the token and starts circulating the token until the program reaches the state s_1, where $(turn(s_1) = false) \wedge (\forall i : 0 \leq i \leq 3 : (a_i(s_1) = 1) \wedge (b_i(s_1) = 0))$; that is, PB_0 has the token. Process PB_0 circulates the token until the program reaches a state s_2, where $(turn(s_2) = true) \wedge (\forall i : 0 \leq i \leq 3 : (a_i(s_2) = 1) \wedge (b_i(s_2) = 1))$, process PA_0 again has the token. This way the token circulation continues in both rings. In other words, the invariant includes all states where there is *exactly one* token in both rings. The invariant $I_{TRTP} = I_{up} \ \wedge \ I_A \ \wedge \ I_B$ includes all states satisfying the following conditions:

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

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

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

The predicate I_{up} represents the set of states where no process has crashed and no variable is corrupted. The state predicate I_A (respectively, I_B) includes the states in which either all a (respectively, b) values are equal or it is the turn of ring A (respectively, B) and there is only one token in ring A (respectively, B).

Safety specification. The safety specification of TRTP stipulates that in each state *at most* one token exists. This requirement could be due to some practical constraints where, for example, TRTP is used as an underlying protocol for assuring mutual exclusion amongst a set of distributed processes and the process that has the token is allowed to access a shared resource. Additionally, no non-faulty process is allowed to copy the value of its detectably faulty predecessor.

Read/write constraints. While in the model represented in Section 2, each process can read/write all program variables in an atomic step, we consider the TRTP example under certain read/write constraints (imposed on processes with respect to the variables of other processes) in order to illustrate the applicability of our approach in the design of multitolerant programs in more concrete models. Specifically, process PA_i (respectively, PB_i), $1 \leq i \leq 3$, is allowed to read its own state and the state of its predecessor and write only a_i (respectively, b_i). Process PA_0 (respectively, PB_0) can read its own state and the state of its predecessor PA_3, PB_0 and PB_3 (respectively, PB_3, PA_0 and PA_3) and *turn*. Process PA_0 (respectively, PB_0) is permitted to write only a_0 (respectively, b_0) and *turn*.

Faults f_m. The class of faults f_m may detectably corrupt the state of only one process (i.e., set its value to -1) in one of the rings if no process is corrupted. Such faults may represent cases where a process fails and restarts.

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

The notation | represents the nondeterministic execution of only one of the assignments separated by |. The f_m-span of the TRTP program is equal to the state predicate $T_{f_m} = T_{up} \wedge T_A \wedge T'_A \wedge T_B \wedge T'_B$, where

$$
\begin{aligned}
T_{up} &= (\forall i : 0 \le i \le 3 : (upa_i \wedge upb_i)) \\
T_A &= (\exists i : 0 \le i \le 3 : (a_i = -1)) \Rightarrow ((\forall j : (0 \le j \le 3) \wedge (j \ne i) : (a_j \ne -1)) \wedge \\
&\qquad\qquad (\forall j : (0 \le j \le 3) : (b_j \ne -1))) \\
T'_A &= (\exists i : 0 \le i \le 3 : (a_i = -1)) \Rightarrow \\
&\quad (\forall i, j : (0 \le i, j \le 3) \wedge (j \ne i) : \neg(PA_i \text{ has the token}) \vee \neg(PA_j \text{ has the token})) \\
T_B &= (\exists i : 0 \le i \le 3 : (b_i = -1)) \Rightarrow ((\forall j : (0 \le j \le 3) \wedge (j \ne i) : (b_j \ne -1)) \wedge \\
&\qquad\qquad (\forall j : (0 \le j \le 3) : (a_j \ne -1))) \\
T'_B &= (\exists i : 0 \le i \le 3 : (b_i = -1)) \Rightarrow \\
&\quad (\forall i, j : (0 \le i, j \le 3) \wedge (j \ne i) : \neg(PB_i \text{ has the token}) \vee \neg(PB_j \text{ has the token})).
\end{aligned}
$$

The f_m-span T_{f_m} includes states where at most one process is corrupted in both rings. If a process is corrupted in Ring A (respectively, Ring B), then no other process is corrupted and *at most* one process has the token in Ring A (respectively, Ring B).

Notation. In the specification of the above state predicates, we have abbreviated $v(s)$ with v for brevity, where v is a program variable.

Faults f_f. In addition to corrupting the state of only one process in a specific ring (by the action FM), this class of faults may also cause a process to crash in a detectable manner; that is, set its *up* value to false. In addition to the action FM, the fault f_f includes the following action. As a result, the f_f-span, denoted T_{f_f}, is equal to the state predicate $T_A \wedge T_B \wedge T'_A \wedge T'_B$. Moreover, the set of states $T_{f_f} - I_{TRTP}$ includes states where $(\exists j : 0 \le j \le 3 : (\neg upa_j \vee \neg upb_j))$ is true.

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

Faults f_n. In addition to corrupting the state of processes by the action FM, the class of faults f_n may nondeterministically assign a value between 0 and 3 to any variable. The fault-class f_n is an undetectable transient fault that may perturb the values of a and b variables nondeterministically. Note that since f_n transitions may generate multiple tokens, it is impossible to ensure that there is only one token at all times (i.e., safety is directly violated by faults f_n).

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

The state predicate $T_A \wedge T_B \wedge (\forall j : 0 \le j \le 3 : (upa_j \wedge upb_j))$ represents f_n-span, denoted T_{f_n}. Moreover, the predicate $T_{f_n} - I_{TRTP}$ contains states where $(\forall j : 0 \le j \le 3 : (upa_j \wedge upb_j))$ holds.

Adding Failsafe_Masking. We use our stepwise algorithm in Figure 12 to add failsafe f_f-tolerance and masking f_m-fault tolerance. When we apply the Add_Failsafe

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

$$
\begin{array}{lll}
AC'_0: & (a_0 = a_3) \wedge \mathbf{upa_0} \wedge (\mathbf{a_3} \neq -\mathbf{1}) \wedge turn \longrightarrow & \\
& \quad \text{if } (((a_0 = b_0) \wedge (\mathbf{b_0} \neq -\mathbf{1})) \vee \neg\mathbf{upb_0}) & a_0 := a_3 \oplus 1; \\
& \quad \text{else} & turn := false; \\
AC_{01}: & \neg\mathbf{upa_3} \wedge \mathbf{upa_0} \longrightarrow & \mathbf{upa_0} := \mathbf{false};^2 \\
AC_{02}: & \neg\mathbf{upb_0} \wedge \neg\mathbf{upb_1} \wedge \neg\mathbf{upb_2} \wedge \neg\mathbf{upb_3} \wedge \mathbf{upa_0} \wedge \neg\mathbf{turn} & \\
& \longrightarrow & \mathbf{turn} := \mathbf{true}; \\
AC'_i: & (a_{i-1} = a_i \oplus 1) \wedge \mathbf{upa_i} \wedge (\mathbf{a_{i-1}} \neq -\mathbf{1}) \longrightarrow & a_i := a_{i-1}; \\
AC_{i1}: & \neg\mathbf{upa_{i-1}} \wedge \mathbf{upa_i} \longrightarrow & \mathbf{upa_i} := \mathbf{false};^2
\end{array}
$$

Observe that the failsafe program contains new actions and new constraints in the guards of original actions. Specifically, the guards of actions AC_0 and AC_i have been strengthened in AC'_0 and AC'_i to prevent a crashed process from executing and from copying a corrupted value. Moreover, if the process PB_0 has crashed then the token continues to be circulated in ring A (see action AC'_0). If a process has not crashed but its predecessor has crashed, then that process changes its state to crashed (see actions AC_{01} and AC_{i1}).[2] If all processes in ring B have crashed and the value of *turn* is not *true*, then *turn* will be set to true to permit token circulation in ring A. Due to space constraints, we omit the revised actions $BC'_0, BC_{01}, BC_{02}, BC'_i, BC_{i1}$ ($1 \leq i \leq 3$) of processes in ring B as symmetric revisions are performed on them in the failsafe f_f-tolerant program TRTP'.

After strengthening the safety specification as prescribed in Step 3 of Figure 12, we use Add_Masking to add masking f_m-tolerance to TRTP' in order to generate the program TRTP". The guard of the action AC'_0 (respectively, BC'_0) is weakened (see actions AC''_0 and BC''_0 below) to include recovery transitions that correct the state of the process PA_0 (respectively, PB_0) once corrupted by the fault action FM. The action AC_{i2} (respectively, BC_{i2}) corrects the state of each corrupted process PA_i (respectively, PB_i), where $1 \leq i \leq 3$.

$$
\begin{array}{lll}
AC''_0: & ((a_0 = a_3) \vee (\mathbf{a_0} = -\mathbf{1})) \wedge upa_0 \wedge (a_3 \neq -1) \wedge turn \longrightarrow & \\
& \quad \text{if } (((a_0 = b_0) \wedge (b_0 \neq -1)) \vee (\mathbf{a_0} = -\mathbf{1}) \vee \neg upb_0) & a_0 := a_3 \oplus 1; \\
& \quad \text{else} & turn := false; \\
BC''_0: & ((b_0 = b_3) \vee (\mathbf{b_0} = -\mathbf{1})) \wedge upb_0 \wedge (b_3 \neq -1) \wedge \neg turn \longrightarrow & \\
& \quad \text{if } (((a_0 \neq b_0) \wedge (a_0 \neq -1)) \vee (\mathbf{b_0} = -\mathbf{1}) \vee \neg upa_0) & b_0 := b_3 \oplus 1; \\
& \quad \text{else} & turn := true; \\
AC_{i2}: & (\mathbf{a_i} = -\mathbf{1}) \wedge \mathbf{upa_i} \wedge (\mathbf{a_{i-1}} \neq -\mathbf{1}) \longrightarrow & \mathbf{a_i} := \mathbf{a_{i-1}}; \\
BC_{i2}: & (\mathbf{b_i} = -\mathbf{1}) \wedge \mathbf{upb_i} \wedge (\mathbf{b_{i-1}} \neq -\mathbf{1}) \longrightarrow & \mathbf{b_i} := \mathbf{b_{i-1}};
\end{array}
$$

Adding Nonmasking f_n-tolerance. In the final step, we add nonmasking f_n-tolerance to TRTP" to generate a failsafe-masking-nonmasking program. Note that the TRTP example meets the sufficient conditions identified in Section 5.4 for polynomial-time addition of failsafe-nonmasking multitolerance. More precisely, the occurrence of f_f may either set the value of a variable a_i (respectively, b_i) to -1, or set upa_i (respectively, upb_i) to false. Such a variable update does not introduce new tokens based

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

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

Fig. 15. The properties of the failsafe-nonmasking-masking multitolerant TRTP program.

on the definition of a token. As such, the antecedent of the definition of fault-safe specifications is false; i.e., the safety specification is f_f-safe. Moreover, the predicates $(T_{f_n} - I_{TRTP})$ and $(T_{f_f} - I_{TRTP})$ are disjoint. Therefore, using Theorem 5.4.4, the algorithm Add_Failsafe_Nonmasking generates a failsafe-nonmasking multitolerant program in polynomial time. The final multitolerant program replaces the recovery actions AC_{i2} and BC_{i2} with the new recovery actions AC_{i3} and BC_{i3}, for $1 \leq i \leq 3$. Notice that the guard of AC_{i3} (respectively, BC_{i3}) subsumes the guard of action AC_{i2} (respectively, BC_{i2}).

$$AC_{i3}: \ (\mathbf{a_{i-1}} \neq \mathbf{a_i} \oplus \mathbf{1}) \wedge (\mathbf{a_{i-1}} \neq \mathbf{a_i}) \wedge \mathbf{upa_i} \wedge (\mathbf{a_{i-1}} \neq -\mathbf{1}) \longrightarrow \mathbf{a_i} := \mathbf{a_{i-1}};$$
$$BC_{i3}: \ (\mathbf{b_{i-1}} \neq \mathbf{b_i} \oplus \mathbf{1}) \wedge (\mathbf{b_{i-1}} \neq \mathbf{b_i}) \wedge \mathbf{upb_i} \wedge (\mathbf{b_{i-1}} \neq -\mathbf{1}) \longrightarrow \mathbf{b_i} := \mathbf{b_{i-1}};$$

Observe that the final program has the following properties (see Figure 15): (1) If faults f_m occur, then there is at most one token at all times (i.e., safety) and every process is guaranteed to receive the token; (2) If faults f_f occur, then there is at most one token at all times and it is circulated amongst a *subset of processes*; (3) If faults f_n occur, then there may be multiple tokens, nonetheless, the program will eventually recover to states from where at most one token exists and every process will receive the token, and (4) If faults from $f_f \cup f_n$ occur, then the program will eventually recover to states from where at most one token exists and a *subset of processes* will receive the token. In this case, the program is the same as the self-stabilizing token ring program designed by Dijkstra [1974] if we abstract out the *up* variables.

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

6.3 Nonmasking-Masking Repetitive Byzantine Agreement

In this section, we synthesize a repetitive agreement protocol that provides masking fault tolerance to Byzantine faults and nonmasking fault tolerance to transient faults; i.e., nonmasking-masking multitolerant.

The fault-intolerant Repetitive Byzantine (RB) program. The RB program includes a general process, denoted P_g, and three nongeneral processes, denoted P_1, P_2, P_3. The program computations consist of consecutive rounds of decision making, where in each round

P_g casts a binary decision and the nongenerals copy the decision of the general and finalize their decision in the current round with an agreement on the same value. The process P_g has a decision variable, denoted d_g, with the domain $\{0, 1\}$. Each process P_i, for $1 \leq i \leq 3$, also has a decision variable, denoted d_i, with the domain $\{-1, 0, 1\}$, where -1 represents an undecided state for that process. To distinguish consecutive rounds of decision making from each other, each nongeneral process P_i uses Boolean variables sn_i and sn_old_i respectively representing the sequence number of the current round and that of the previous round. Process P_g has a Boolean variable sn_g representing the sequence number of the general. Process P_i copies its decision in an *output* decision variable d_old_i that should be read at the end of the current round. To determine whether a process is Byzantine, each process has a Boolean variable b. If all sequence numbers are equal, the general starts the next round by toggling sn_g (see action G below) and resetting all b values. Since we allow the Byzantine process to change in every round, when the general begins a new round by executing action G, all processes are assumed to be non-Byzantine. Subsequently, a fault BF_1 (described later) can cause one of the processes to become Byzantine. Thus, our definition of Byzantine faults is a generalization of that in Lamport et al. [1982], where the same process is assumed to be Byzantine at all times.

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

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

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

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

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

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

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

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

where $maj = majority(d\_old_1, d\_old_2, d\_old_3)$.

A nongeneral process outputs its decision when all nongenerals have copied a new decision from the general in the current round (see Action A'_{i1}). If the output decision of a nongeneral process P_i differs from that of the majority of nongenerals, then P_i revises its decision (see Action A_{i3}). When all nongenerals have output their decision and the decision of a nongeneral process P_i is the same as that of the majority, then P_i finalizes its current round of decision making (see Action A'_{i2}). Observe that the program RB′ consisting of actions $G, A_{i0}, A'_{i1}, A'_{i2}$ and A_{i3}, for $1 \leq i \leq 3$, is masking f_m-tolerant.

Transient faults. In addition to the class of faults f_m, the transient faults f_n perturb the state of program RB′ and change the decision values and sequence numbers by the following action (Notice that the following fault action is a parameterized action for $1 \leq i \leq 3$.):

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

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

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

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

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

The above actions guarantee that from any arbitrary state, the nonmasking-masking multitolerant program RB$''$ recovers to *roundInv*; that is, RB$''$ is self-stabilizing [Dijkstra 1974] to Byzantine and transient faults.

7. DISCUSSION

In this section, we discuss related work, the relevance and contributions of our approach in software engineering, and dependability practice. We also discuss the limitations of the approach proposed in this article.

Formalization of faults and fault tolerance. Numerous approaches formally model faults and fault tolerance in the context of transitions systems to facilitate modeling and analyzing fault tolerance. For instance, several methods use process algebra [Bernardeschi et al. 2000; Gnesi et al. 2005; Peleska 1991; Prasad 1984] to model each fault-class as a process that is composed with system processes. The resulting composition represents the behavior of the system in the presence of faults. Subsequently, they design fault tolerance functionalities and use verification techniques (e.g., model checking) to ensure design correctness. For instance, Bernardeschi et al. [2000] use observational equivalence to prove that the composition of system processes, fault processes and fault tolerance processes behaves similar to the intolerant program's behaviors in the absence of faults. Pike et al. [2004] model faults and fault tolerance in a typed-system in higher-order logic. They enable the specification and verification of fault-tolerant systems using the PVS theorem prover. Specifically, they consider a fault type to be a representation of the effect of faults on a system. Liu and Joseph's formalization [Liu and Joseph 1992, 1999] is the closest to ours where they model faults by a set of atomic actions that can nondeterministically perturb the program state. They design recovery actions and then verify that the union of program and recovery actions meets a specification representing expected program behaviors in the presence of faults. Nonetheless, their approach seems tedious if one needs to design different levels of fault tolerance to distinct classes of faults. The proposed approach in this article facilitates/automates designing different levels of fault tolerance to distinct classes of faults.

Automated design. Most existing approaches for automated design of fault-tolerant programs generate programs from formal specifications, called *specification-based* methods [Attie and Emerson 1998, 2001; Attie et al. 2004; Emerson and Clarke 1982; Liu and Joseph 1992, 1999; Manna and Wolper 1984]. For instance, Attie et al. [2004] create the synchronization skeleton of fault-tolerant programs from temporal logic specifications. Their approach is based on techniques that design programs from a satisfiability proof of their formal specifications [Attie and Emerson 1998, 2001; Emerson and Clarke 1982; Manna and Wolper 1984]. Control-theoretic approaches [Cho and Lim 1998; Lafortune and Lin 1992; Lin and Wonham 1990; Ramadge and Wonham 1989; Rohloff 2004; Rudie and Wonham 1992; Rudie et al. 2003] focus on generating

the design of a discrete-event controller from the specification of a controlled system in the presence of uncontrolled events. While control-theoretic approaches are mostly based on a prioritized synchronization computational model, our computational model is based on nondeterministic interleaving of all the actions of different processes. Game-theoretic techniques [Pnueli and Rosner 1989; Thomas 1995, 2002; Wallmeier et al. 2003] for automated design of reactive programs are mostly based on a two-player game model in which a program and its environment take turns in executing their actions. Moreover, the interaction of the program and its environment is channeled through a set of interface variables, whereas in our work, faults can nondeterministically update any program variable from any state. The specification-based approaches generally require higher time complexity [Kupferman and Vardi 2001; Pnueli and Rosner 1990] and provide limited/no reuse when an existing program is revised. By contrast, we start from an existing fault-intolerant program instead of its specification, and automatically *revise* the program to capture different levels of fault tolerance. The advantages of this *revision-based* approach are multifold. First, this approach separates the concern of fault tolerance from functional concerns. Such a separation of concerns has been found to be valuable in several settings [Arora and Kulkarni 1998b; Jeffords et al. 2009]. Second, it reuses the computations of an existing program in the design of a fault-tolerant version thereof, thereby potentially reducing development costs. Third, it provides a stepwise method for the design of multitolerance.

Separation of concerns. Our stepwise method simplifies the design of multitolerant programs by (1) separating the functional concerns from fault tolerance concerns, and (2) enabling the design of different levels of fault tolerance for multiple classes of faults one at a time. Such a separation of concerns mitigates the complexity of design because ensuring the noninterference of functional and fault tolerance concerns, and guaranteeing the noninterference of different levels of fault tolerance with each other are difficult tasks. More specifically, in an individual step of our method, designers should consider only the program behaviors in the presence of a specific known fault-class without any concern for potential conflicts between the level of fault tolerance they are currently designing and the levels of fault tolerance they will have to design in the future. Without using our approach, designers would run into two major difficulties. First, upon detecting a new fault-class, designers would have to redesign an existing system from scratch considering the newly detected fault along with previously discovered faults. Second, it would be difficult to preserve the previously designed levels of fault tolerance while designing fault tolerance for the new fault-class. Our black-box stepwise method addresses both problems.

Impact on dependability practice. The proposed design method in this article improves the current practice in the design of fault-tolerant systems in several directions. The common practice in designing fault-tolerant/multitolerant systems is *Design-and-Verification* (D&V). That is, a tolerant system is first designed manually, and then, some verification technique is used to establish its correctness. It is expensive to apply such a D&V method for today's systems because the frequency of detecting new classes of faults that were not anticipated at design time is higher due to (1) the complexity and the diversity of devices used in such systems, (2) the aging of hardware systems, and (3) the nondeterministic nature of concurrent systems. To facilitate the development of new fault tolerance functionalities *after* a new class of fault is detected, we propose the algorithms that take an existing program and automatically explore the possibility of generating a revised version of the input program that tolerates a new class of faults while preserving its existing fault tolerance functionalities.

This approach potentially reduces development/maintenance costs by exploring the computational structure of an existing program towards creating a fault-tolerant version thereof. In other words, before utilizing resource redundancy for the design of fault tolerance, we first explore the possibility of providing fault tolerance using computational redundancy. This benefit is evident in the SDS example presented in the revised version. Moreover, sometimes the new fault tolerance requirements expected for the newly detected fault is in conflict with some other existing fault tolerance functionalities. Detecting such conflicts manually is by itself a hard problem. The least our approach could do is to detect such conflicts and warn developers not to spend their time on the design of a multitolerant program that does not exist. As an additional piece of evidence, we conducted an experiment on the TRTP example in Section 6.2, where we manually designed the multitolerant program and then verified the manual design using the model checker SPIN [Holzmann 1997]. The manual design-and-verification approach took 5 days for us. Then, we used FTSyn to design the multitolerant TRTP program, which decreased the design time to almost 7 hours. Notice that this program has 200 million reachable states and 8 processes. In this experiment, we used a version of FTSyn that has been implemented using Binary Decision Diagrams (BDDs) [Bryant 1986] on a Linux PC with an Intel Pentium IV (3.00GHz) CPU with 2 GB RAM.

Moreover, our algorithms provide a theoretical foundation for a design automation engine that can be integrated with specification and modeling languages (e.g., SAL [de Moura et al. 2003] and SCR [Heitmeyer et al. 1997; Jeffords et al. 2009]) to facilitate the design of multitolerant systems. For instance, we are investigating the integration of FTSyn in the Unified Modeling Language (UML) [Rumbaugh et al. 1999] to enable designers in automated modeling and design of fault-tolerant systems at the level of UML state diagrams [Ebnenasir and Cheng 2007a, 2007b].

Methodological significance. The proposed approach in this article has important methodological implications in the *design* of fault-tolerant parallel programs (e.g., multicore programs). With the advent of multicore processors, the next major challenge in the coming decade is to facilitate parallel programming for mainstream developers. Towards this end, it is highly important that we provide necessary techniques and tools for reasoning about the behaviors of shared memory parallel programs in the presence of design flaws before we actually implement them. Our proposed approach provides a theoretical foundation for the design of highly resilient parallel programs as design flaws (e.g., the causes of race conditions) can easily be captured in our formal framework as transitions that perturb the state of a parallel program to error states. Moreover, as the examples in Section 6 illustrate, our approach can also be applied for the design of highly reliable distributed (i.e., low atomicity) programs and network protocols (e.g., distributed agreement) that constitute the underlying components of many services in the Internet and scientific computing applications.

Limitations. There are some limitations regarding our modeling framework, the input requirements of our algorithms and the scalability of our technique in tool development. First, as mentioned in the Introduction, our formal modeling captures only the systems/faults that can be represented in terms of finite-state transition systems. As such, any system that cannot be *precisely* modeled as a finite-state transition system is outside the scope of our approach. The investigation of adding multitolerance to infinite state systems remains an open problem too. Further, our formal framework does not capture any property that cannot be captured in the context of Alpern and Schneider's topological characterization of linear temporal properties (e.g., ω-regular properties over infinite trees [Manolios and Trefler 2003]).

Second, the degree of success/failure of our stepwise design of multitolerance depends upon the *maximality* of the input program and the *weakness* of its invariant. The maximality of the input program means that, from any state, the intolerant program includes the maximum number of transitions that can nondeterministically be executed while satisfying program specifications. For example, consider the multitolerant token ring program of Section 6.2. Let p_1 be the intermediate program after adding failsafe fault tolerance; p_1 is failsafe fault-tolerant to state corruption and crash faults. The exclusion of the actions AC_{01}, AC_{02} and AC_{i1} (respectively, BC_{01}, BC_{02} and BC_{i1}) from the set of transitions of p_1 would result in another failsafe program, denoted p_2. Nonetheless, if a process crashes in one of the rings, then both rings in program p_2 stop the token circulation. By contrast, program p_1 ensures that if a process in a ring is crashed, then the other ring recovers and circulates the token. Now, adding masking fault tolerance to p_1 is easier because the additional transitions captured by actions AC_{01}, AC_{02} and AC_{i1} (respectively, BC_{01}, BC_{02} and BC_{i1}) can be reused for the design of recovery to the invariant. As such, if the input program is not maximal, then the intermediate programs may not have the necessary computational redundancy that is required for the design of subsequent levels of fault tolerance, thereby resulting in the failure of stepwise design. A similar problem could arise with a strong invariant where after adding one level of fault tolerance the size of the invariant shrinks in such a way that the addition of subsequent levels of fault tolerance fails.

Third, a practical limitation of our approach (in terms of tool development) is the time/space complexity of design, which in turn affects the scalability of tools developed for the design of multitolerance. To tackle this limitation, we have developed symbolic [Bonakdarpour and Kulkarni 2007] and distributed synthesis algorithms [Ebnenasir 2007] for the addition of a single level of fault tolerance, where we have synthesized programs with 2^{100} reachable states on a cluster of a few regular PCs in a few hours. We plan to reuse the implementation of these distributed algorithms for stepwise design of multitolerance.

Finally, our notion of program in this article focuses on the abstract structure of parallel/distributed programs rather than programs written in common programming languages (e.g., C and Java). Since similar abstractions have been found useful and practical in program verification [Holzmann 1997; Visser et al. 2003], our focus is also on the design phase instead of implementations.

8. CONCLUSIONS AND FUTURE WORK

In this article, we investigated the problem of stepwise design of multitolerant programs from their fault-intolerant versions. A program that is subject to multiple classes of faults, and provides a different level of fault tolerance to each fault-class is a *multitolerant* program. We considered three levels of fault tolerance, *failsafe*, *nonmasking* and *masking*. The major contributions of this article are twofold: First, for cases where one needs to add failsafe fault tolerance to one class of faults and nonmasking fault tolerance to a *different* class of faults, we found a surprising result that this problem is NP-complete (in program state space). Since adding fault tolerance to a single class of faults is in P [Kulkarni and Arora 2000], this implies that a stepwise method, where the number of steps is constant, for failsafe-nonmasking multitolerance does not exist (unless P = NP). To deal with this NP-completeness result, we identified classes of programs, specifications and faults for which failsafe-nonmasking multitolerance can be designed in a stepwise manner (in polynomial time).

Second, we investigated the feasibility of a stepwise method that is sound and deterministically complete. Such a method is highly desirable in capturing new fault tolerance functionalities in existing programs upon detection of new classes of faults. In other words, we facilitate the upgrade of fault tolerance functionalities while

preserving existing functionalities. We presented such a sound and deterministically complete design method for *special cases* where one adds failsafe (respectively, nonmasking) fault tolerance to one class of faults and masking fault tolerance to another class of faults. More importantly, we showed that such an addition is feasible regardless of the order in which different faults are considered. This result has a significant impact for designers in that they can reuse an existing design in future additions of fault tolerance no matter what classes of faults are currently known.

Future directions. We plan to extend this work in several directions. First, we will investigate additional sufficient (or necessary) conditions for polynomial design of multitolerance. Sufficient conditions help us reduce design complexity for special cases of the multitolerance synthesis problem. Necessary conditions will identify properties that are common in programs to which multitolerance can be added efficiently. Second, we will focus on the design of heuristics that reduce design complexity by identifying classes of states/transitions that may or may not be included in a multitolerant program in cases where different fault tolerance requirements conflict. Such heuristics will be sound but incomplete in that if they generate a multitolerant program, then the synthesized program is correct (i.e., meets the requirements of the multitolerance problem defined in this paper), however, they may fail to design a multitolerant program while one exists, hence the incompleteness. Third, we will integrate the FTSyn tool in modeling environments such as SAL [de Moura et al. 2003] and SCR [Heitmeyer et al. 1997] in order to enable the developers of mission-critical systems to benefit from the automation provided by our approach. Such an integrated toolset will extend the scope of applicability of our approach as developers of different design methodologies can automatically design multitolerance. Fourth, we will focus on the application of our work in specific domains such as wireless sensor networks, data intensive systems and high performance computing. Since the functional concerns of today's systems also evolve frequently due to changes in user requirements and system reconfiguration/update, we will investigate the effect of stepwise design of evolving functional requirements (investigated in our previous work [Ebnenasir et al. 2005]) on fault tolerance concerns.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

We thank Aly Farahat for setting up the token ring example in a symbolic version of FTSyn. We also are thankful to anonymous reviewers and the editors of TOSEM for providing insightful comments and suggestions on the first draft of this article.

REFERENCES

ALPERN, B. AND SCHNEIDER, F. B. 1985. Defining liveness. *Inform. Process. Lett. 21*, 181–185.

ANAGNOSTOU, V. H. 1993. Tolerating transient and permanent failures. In *Proceedings of the 7th International Workshop on Distributed Algorithms*. Springer, 174–188.

ARORA, A. AND GOUDA, M. G. 1993. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Trans. Softw. Engin. 19,* 11, 1015–1027.

ARORA, A. AND KULKARNI, S. S. 1998a. Component based design of multitolerant systems. *IEEE Trans. Softw. Engin. 24*, 1, 63–78.

ARORA, A. AND KULKARNI, S. S. 1998b. Component based design of multitolerant systems. *IEEE Trans. Softw. Engin. 24,* 1 (January), 63–78.

ARORA, A. AND KULKARNI, S. S. 1998c. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Trans. Softw. Engin. 24,* 6, 435–450.

ATTIE, P. AND EMERSON, A. 2001. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM Trans. Program. Lang. Syst 23,* 2, 187–242.

ATTIE, P. AND EMERSON, E. 1998. Synthesis of concurrent systems with many similar processes. *ACM Trans. Program. Lang. Syst. 20,* 1, 51–115.

ATTIE, P. C., ANISH ARORA, AND EMERSON, E. A. 2004. Synthesis of fault-tolerant concurrent programs. *ACM Trans. Program. Lang. Syst. 26,* 1, 125–185.

AVIZIENIS, A., LAPRIE, J.-C., RANDELL, B., AND LANDWEHR, C. E. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Depend. Secure Comput. 1,* 1, 11–33.

BEN-OR, M., DOLEV, D., AND HOCH, E. N. 2008. Fast self-stabilizing byzantine tolerant digital clock synchronization. In *Proceedings of the 27th Annual ACM Symposium on Principles of Distributed Computing, (PODC)*. ACM, New York, NY, 385–394.

BERNARDESCHI, C., FANTECHI, A., AND SIMONCINI, L. 2000. Formally verifying fault tolerant system designs. *Comput. J. 43,* 3, 191–205.

BONAKDARPOUR, B. AND KULKARNI, S. S. 2007. Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In *Proceedings of the 27th International Conference on Distributed Computing Systems*. IEEE Computer Society, Los Alamitos, CA, 3–10.

BRYANT, R. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput. 35,* 8, 677–691.

CHO, K. AND LIM, J. 1998. Synthesis of fault-tolerant supervisor for automated manufacturing systems: A case study on photolithography process. *IEEE Trans. Robot. Automat. 14,* 2, 348–351.

DAMS, D., HESSE, W., AND HOLZMANN, G. J. 2002. Abstracting C with abC. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*. Springer, 515–520.

DE MOURA, L., OWRE, S., AND SHANKAR, N. 2003. The SAL language manual. Tech. rep. SRI-CSL-01-02, SRI International.

DEMIRBAS, M. AND ARORA, A. 2002. Convergence refinement. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*. IEEE Computer Society, Los Alamitos, CA, 589–597.

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

DIJKSTRA, E. W. 1990. *A Discipline of Programming*. Prentice-Hall.

DOLEV, D. AND HOCH, E. N. 2007a. Byzantine self-stabilizing pulse in a bounded-delay model. In *Proceedings of the International Symposium on Stabilization, Safety, and Security of Distributed Systems*. 234–252.

DOLEV, D. AND HOCH, E. N. 2007b. On self-stabilizing synchronous actions despite byzantine attacks. In *Proceedings of the 21st International Symposium on Distributed Computing (DISC)*. 193–207.

DOLEV, S. AND HERMAN, T. 1995. Superstabilizing protocols for dynamic distributed systems. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, NY, 255.

DOLEV, S. AND WELCH, J. L. 1995. Self-stabilizing clock synchronization in the presence of byzantine faults (abstract). In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 256.

DOLEV, S. AND YAGEL, R. 2007. Stabilizing trust and reputation for self-stabilizing efficient hosts in spite of byzantine guests (extended abstract). In *Proceedings of the International Symposium on Stabilization, Safety, and Security of Distributed Systems*. 266–280.

EBNENASIR, A. 2005. Automatic synthesis of fault-tolerance. Ph.D. thesis, Michigan State University.

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

EBNENASIR, A. AND CHENG, B. H. 2007a. *A Pattern-Based Approach for Modeling and Analyzing Error Recovery, Architecting Dependable Systems IV*. Lecture Notes in Computer Science, vol. 4615. 115–141.

EBNENASIR, A. AND CHENG, B. H. 2007b. Pattern-based modeling and analysis of failsafe fault-tolerance in UML. In *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium (HASE)*. IEEE Computer Society, Los Alamitos, CA, 275–282.

EBNENASIR, A. AND KULKARNI, S. S. 2008. Feasibility of stepwise addition of multitolerance to high atomicity programs. Tech. rep. CS-TR-08-03, Michigan Technological University. http://www.cs.mtu.edu/html/tr/08/08-03.pdf.

EBNENASIR, A., KULKARNI, S. S., AND BONAKDARPOUR, B. 2005. Revising UNITY programs: Possibilities and limitations. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*. 275–290.

EBNENASIR, A., KULKARNI, S. S., AND ARORA, A. 2008. FTSyn: A framework for automatic synthesis of fault-tolerance. *Int. J. Softw. Tools Techn. Transfer 10,* 5, 455–471.

EMERSON, E. AND CLARKE, E. 1982. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program. 2*, 3, 241–266.

ENGLISH, R. M. AND STEPANOV, A. A. 1991. Loge: A self-organizing disk controller. Tech. rep. HPL-91-179, Software and Systems Laboratory, Hewlett Packard.

GARTNER, F. 2003. A survey of self stabilizing spanning tree construction algorithms. Tech. rep. IC 2003/38, EPFL.

GNESI, S., LENZINI, G., AND MARTINELLI, F. 2005. Logical specification and analysis of fault tolerant systems through partial model checking. *Electron. Notes Theor. Comput. Sci. 118*, 57–70.

HEITMEYER, C. L., KIRBY, J., AND LABAW, B. G. 1997. The SCR method for formally specifying, verifying, and validating requirements: Tool support. In *Proceedings of the International Conference on Software Engineering*. 610–611.

HERLIHY, M. AND WING, J. 1991. Specifying graceful degradation. *IEEE Trans. Parall. Distrib. Syst. 2,* 1, 93–104.

HOLZMANN, G. 1997. The model checker SPIN. *IEEE Trans. Softw. Engin. 23,* 5, 279–295.

HOLZMANN, G. J., JOSHI, R., AND GROCE, A. 2008. Model driven code checking. *Automat. Softw. Engin. 15,* 3–4, 283–297.

JEFFORDS, R., HEITMEYER, C., ARCHER, M., AND LEONARD, E. 2009. A formal method for developing provably correct fault-tolerant systems using partial refinement and composition. In *Proceedings of the International Symposium on Formal Methods*.

KLETZ, T. A. 1999. *HAZOP and HAZAN: Identifying and Assessing Process Industry Standards* 4th Ed. CRC (4th).

KULKARNI, S. S. 1999. Component-based design of fault-tolerance. Ph.D. thesis, Ohio State University.

KULKARNI, S. S. AND ARORA, A. 2000. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer-Verlag, Berlin, 82–93.

KULKARNI, S. S. AND EBNENASIR, A. 2003. Enhancing the fault-tolerance of nonmasking programs. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*. IEEE Computer Society, Los Alamitos, CA, 441–449.

KULKARNI, S. S. AND EBNENASIR, A. 2004. Automated synthesis of multitolerance. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, Los Alamitos, CA, 209–218.

KULKARNI, S. S. AND EBNENASIR, A. 2005a. Complexity issues in automated synthesis of failsafe fault-tolerance. *IEEE Trans. Depend. Secure Comput. 2,* 3, 201–215.

KULKARNI, S. S. AND EBNENASIR, A. 2005b. The effect of the safety specification model on the complexity of adding masking fault-tolerance. *IEEE Trans. Depend. Secure Comput. 2,* 4, 348–355.

KUPFERMAN, O. AND VARDI, M. 2001. Synthesizing distributed systems. In *Proceedings of the 16th IEEE Symp. on Logic in Computer Science*.

LAFORTUNE, S. AND LIN, F. 1992. On tolerable and desirable behaviors in supervisory control of discrete event systems. *Discrete Event Dynamic Syst. Theory Appl. 1,* 1, 61–92.

LAMPORT, L., SHOSTAK, R., AND PEASE, M. 1982. The byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4,* 3, 382–401.

LIN, F. AND WONHAM, W. M. 1990. Decentralized control and coordination of discrete-event systems with partial observation. *IEEE Trans. Automat. Control 35,* 12, 1330–1337.

LIU, Z. AND JOSEPH, M. 1992. Transformation of programs for fault-tolerance. *Formal Aspects Comput. 4,* 5, 442–469.

LIU, Z. AND JOSEPH, M. 1999. Specification and verification of fault-tolerance, timing, and scheduling. *ACM Trans. Program. Lang. Syst. 21,* 1, 46–89.

MALEKPOUR, M. R. 2006. "a byzantine-fault tolerant self-stabilizing protocol for distributed clock synchronization systems. Tech. rep., NASA/TM-2006-214322.

MANNA, Z. AND WOLPER, P. 1984. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst. 6*, 1, 68–93.

MANOLIOS, P. AND TREFLER, R. 2003. A lattice-theoretic characterization of safety and liveness. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing (PODC'03)*. 325–333.

MARZULLO, K., SCHNEIDER, F. B., AND DEHN, J. 1994. Refinement for fault-tolerance: An aircraft hand-off protocol. Tech. rep. TR94-1417, Cornell University.

NESTERENKO, M. AND ARORA, A. 2002. Stabilization-preserving atomicity refinement. *J. Parall. Distrib. Comput. 62,* 5, 766–791.

PALADY, P. 1995. *Failure Modes and Effects Analysis*. PT Publications Inc.

PELESKA, J. 1991. Design and verification of fault tolerant systems with CSP. *Distrib. Comput. 5*, 95–106.

PIKE, L., MADDALON, J., MINER, P. S., AND GESER, A. 2004. Abstractions for fault-tolerant distributed system verification. In *Proceedings of the 17th International Conference Theorem Proving in Higher Order Logics (TPHOLs)*. 257–270.

PNUELI, A. AND ROSNER, R. 1989. On the synthesis of a reactive module. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*. ACM, New York, NY, 179–190.

PNUELI, A. AND ROSNER, R. 1990. Distributed reactive systems are hard to synthesis. In *Proceedings of the 31st IEEE Symposium on Foundation of Computer Science*. IEEE Computer Society, Los Alamitos, CA, 746–757.

PRASAD, K. V. S. 1984. Specification and proof of a simple fault tolerant system in CCS. Internal rep. CSR–178–84, University of Edinburgh.

RAMADGE, P. AND WONHAM, W. 1989. The control of discrete event systems. *Proc. IEEE 77,* 1, 81–98.

ROHLOFF, K. R. 2004. Computations on distributed discrete-event systems. Ph.D. thesis, University of Michigan, MI.

RUDIE, K. AND WONHAM, W. 1992. Think globally, act locally: Decentralized supervisory control. *IEEE Trans. Automat. Control 37,* 11, 1692–1708.

RUDIE, K., LAFORTUNE, S., AND LIN, F. 2003. Minimal communication in a distributed discrete-event systems. *IEEE Trans. Automat. Control 48,* 6, 957–975.

RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. 1999. *The Unified Modeling Language Reference Manual*. Addison-Wesley.

SAE. 1996. ARP-4761: Aerospace recommended practice: Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment 12th Ed. Society of Automotive Engineers (SAE), Warrendale PA.

SIU, H.-S., CHIN, Y.-H., AND YANG, W.-P. 1998. Byzantine agreement in the presence of mixed faults on processors and links. *IEEE Trans. Parall. Distrib. Syst. 9*, 335–345.

THOMAS, W. 1995. On the synthesis of strategies in infinite games. In *Proceedings of the Annual Symposium on Theoritical Aspects of Computer Science (STACS)*. 1–13.

THOMAS, W. 2002. Infinite games and verification (extended abstract of a tutorial). In *Proceedings of the 14th International Conference CAV*. Lecture Notes in Computer Science, vol. 2404. 58–64.

TSANG, S. AND MAGILL, E. 1994. Detecting feature interactions in the intelligent network. In *Feature Interactions in Telecommunications Systems II*, IOS Press, 236–248.
`http://www.comms.eee.strath.ac.uk/~fi/papers.html`.

ULSAMER, E. 1973. Computers key to tomorrows air force. *Air Force Mag. 56,* 7, 46–52.

VARGHESE, G. 1993. Self-stabilization by local checking and correction. Ph.D. thesis, MIT/LCS/TR-583.

VESELY, W. 1981. Fault tree handbook. US Nuclear Regulatory Committee rep. NUREG-0492, US NRC, Washington DC.

VISSER, W., HAVELUND, K., BRAT, G. P., PARK, S., AND LERDA, F. 2003. Model checking programs. *J. Automa. Softw. Engin. 10,* 2, 203–232.

WALLMEIER, N., HÜTTEN, P., AND THOMAS, W. 2003. Symbolic synthesis of finite-state controllers for request-response specifications. In *Proceedings of CIAA*. Lecture Notes in Computer Science, vol. 2759. 11–22.

Received January 2009; revised October 2009; accepted March 2010