Designing Efficient Fail-Safe Multitolerant Systems

Arshad Jhumka† and Neeraj Suri‡

†Department of Computer Science University of Warwick, Coventry CV4 7AL, UK
‡ Department of Computer Science, TU - Darmstadt, Darmstadt, Germany

Abstract. In this paper, we propose a method for designing efficient fail-safe multitolerant systems. A multitolerant system is one that is able to tolerate multiple types of faults, and a fail-safe multitolerant system handles the various fault types in a fail-safe manner. Efficiency issues of interest are fault tolerance-related, and they are: (i) completeness, and (ii) accuracy. Based on earlier work, this paper makes the following contributions: (i) We develop a theory for design of efficient fail-safe multitolerance, (ii) based on the theory, we present a sound and complete algorithm that automates the addition of efficient fail-safe multitolerance, and (iii) we develop the example of an efficient fail-safe multitolerant token ring to show the viability of our approach. Our approach works for finite state systems.

Keywords: detectors, fail-safe, multitolerance, program transformation, safety specification, automation, program synthesis
Contact author: Arshad Jhumka (`arshad@dcs.warwick.ac.uk`)

1 Introduction

Fault tolerance is the ability of a program or system to satisfy its specification, even in the presence of external perturbations. Perturbations, or *faults*, are varied in nature, for example, computer intrusions, message losses, variable corruptions etc. Thus, a program intended to be deployed in such a faulty environment needs to be able to withstand the effect of these faults, i.e., we require the program to be tolerant to the faults. In other words, such a program needs to be *multitolerant.*

Alpern and Schneider showed that, in general, a specification [1] can be considered as the intersection of a *safety* specification, and a *liveness* specification. In the presence of faults, the need to satisfy both specifications is not mandatory, giving rise to different levels of fault tolerance. The more prominent of them being (i) fail-safe, (ii) non-masking and (iii) masking fault tolerance. In this paper, we focus on *fail-safe fault tolerance.* Fail-safe fault tolerance, informally, is the ability of a system to always satisfy its safety specification in the presence of faults, i.e., if the system is about to violate safety, then the system is halted. This type of fault tolerance is often used in safety critical systems, such as nuclear power plants, train control systems, where safety is more important than continuous provision of service. In practice, a backup system may be used after the original system is halted. Though a fail-safe fault-tolerant program does not have to guarantee liveness in presence of faults, some form of "controlled" liveness can be obtained [4].

Arora and Kulkarni [3] showed that there exists a class of program components, called *detectors* that is both necessary and sufficient to ensure fail-safe fault tolerance. A detector is a program component that asserts the validity of a given predicate in a given state of the program. Examples are run-time checks [9], executable assertions [5], error detection codes, comparators etc. Our focus on fail-safe fault tolerance design thus translates into focusing on the design of detectors.

Designing effective detectors is known to be a non-trivial task [9]. Composing[1] ineffective detectors with a given program will have some adverse effects, such as failures to detect erroneous states of the system, or the trigger of false alarms. To address these problems, Jhumka *et.al* [7, 6] developed a theory of detectors that identified the properties that underpin the operational effectiveness of detectors. These properties are (i) *completeness*, and (ii) *accuracy*. The completeness property of a detector is linked with a detector's ability to detect erroneous states, while the accuracy property is linked with a detector's ability to avoid mistakes (false alarms). A complete and accurate detector is termed *perfect*. The completeness and accuracy properties represent the fault tolerance *efficiency* issues on which we focus in this paper.

Our approach to the design of multitolerance is based on the well-known software engineering principle of decomposition. Instead of trying to design detectors that are to efficiently tolerate a complex fault class F, we first decompose the complex fault class F into a sequence of basic fault classes $f_1 \dots f_n$. We then design effective detectors that handle the first basic fault class f_1. Once done, we consider the next basic fault class f_2, and design effective detectors that not only handle f_2, but also do not interfere with the effectiveness of detectors that handle f_1. The idea is to incrementally design a multitolerant program such that, in any one step, effective tolerance to a new basic fault class is added, while all previous tolerances and effectiveness are preserved.

1.1 Related Work

The first work on multitolerance design was proposed by Arora and Kulkarni in [2]. However, our work differs from that of Arora and Kulkarni [2] in the following ways: (i) fault tolerance efficiency issues are at the heart of the approach proposed in this paper, unlike in [2], (ii) we present a sound and complete algorithm for automating addition of multitolerance, unlike in [2]. Later, Kulkarni and Ebnenasir [8] proposed an automated approach, as in this paper, for the addition of multitolerance. Their approach differs from that proposed in this paper in two ways: (i) they tackle the problem in a different system model (where read/write restrictions are imposed), and (ii) they do not tackle efficiency properties, as in this paper.

Building on previous work [7, 6], our contributions in this paper are: (i) We present a theory for efficient fail-safe multitolerance design, (ii) We provide a sound, and complete algorithm that automates the addition of efficient fail-safe

[1] We will formally define this term in the next section.

multitolerance, and (iii) we present a case study of the design of a fail-safe multitolerant token ring to show the applicability of our approach.

The paper is structured as follows: Sec. 2 introduces the assumed models and terminologies. Sec. 3 defines the problem of perfect fail-safe fault tolerance. In Sec. 4 addresses the problem of adding perfect fail-safe multitolerance to programs. An example of a fully distributed, fail-safe multitolerant program for a token ring is presented in Sec. 5.

2 Preliminaries

In this section, we recall the standard formal definitions of programs, faults, fault tolerance (in particular, fail-safe fault-tolerance), and of specifications [3].

2.1 Concurrent Systems

The work assumes an *interleaved* execution semantics together with the *shared variable* communication paradigm.

2.2 Programs

A *program P* consists of a set of processes $\{p_1 \ldots p_n\}$. Each process p_i contains a finite set of actions, and a finite set of variables. Each variable stores a value of a predefined nonempty finite domain and is associated with a predefined set of initial values. In this paper, we will use two representations of a program: (i) guarded command notation, and (ii) state transition system. While formal definitions/results will be based on the transition model, the guarded command notation provides a more "visual" basis.

In the guarded command notation, an action has the form

$$\langle \text{guard} \rangle \rightarrow \langle \text{statement} \rangle$$

where the guard is a predicate over the program variables, and the statement is either the empty statement or an instantaneous value assignment to one or more variables.

The *state space S_P of a program P* is the set of all possible value assignments to variables. A *state predicate of P* is a boolean expression over the state space of P. The *set of initial states I_P* is defined by the set of all possible assignments of initial values to variables of P.

An *action ac of P is enabled in a state s* if the guard of ac evaluates to "true" in s. An action ac can be represented by a set of state pairs. Note that programs are permitted to be non-deterministic as multiple actions can be enabled in the same state.

A *computation of p* is a weakly fair (finite or infinite) sequence of states $s_0, s_1, \ldots$ such that $s_0 \in I_p$ and for each $j \geq 0$, s_{j+1} results from s_j by executing the assignment of a single action which is enabled in s_j. *Weak fairness* implies that if a program action ac is continuously enabled, ac is eventually chosen to be executed. Weak fairness implies that a computation is *maximal* with respect to program actions, i.e., if the computation is finite then no program action is enabled in the final state.

A *state s occurs in a computation* $s_0, s_1, \ldots$ iff there exists an i such that $s = s_i$. Similarly, a *transition* (s, s') *occurs in a computation* $s_0, s_1, \ldots$ iff there exists an i such that $s = s_i$ and $s' = s_{i+1}$.

In the context of this paper, programs are equivalently represented as state machines, i.e., a program is a tuple $P = (S_P, I_P, \delta_P)$, where S_P is the state space and $I_P \subseteq S_P$ is the set of initial states. Transition $(s, s') \in \delta_P$ iff ac of P is enabled in state s and execution of ac in state s results in state s'. We say that *ac induces* these transitions. State s is called the *start state* and s' the *end state* of the transition.

2.3 Specifications

A *specification* for a program P is a set of computations which is *fusion-closed*. A *specification S is fusion-closed*[2] iff the following holds for finite computations α, β, and a state s: If $\alpha = \gamma \cdot s \cdot \rho$ and $\beta = \epsilon \cdot s \cdot \sigma$ are in S, then so are computations $\gamma \cdot s \cdot \sigma$ and $\epsilon \cdot s \cdot \rho$. A *computation c of P satisfies a specification S* iff $c \in S$. A *program P satisfies a specification S* iff all possible computation of P satisfies S.

Definition 1 (Maintains). *Let P be a program, S be a specification and α be a finite computation of P. We say that α* maintains *S iff there exists a sequence of states β of P such that $\alpha \cdot \beta \in S$.*

Definition 2 (Safety specification). *A specification S of a program P is a* safety specification *iff the following condition holds: $\forall$ computation σ that violates S, $\exists$ a prefix α of σ s.t $\forall$ state sequences β, $\alpha \cdot \beta$ violates S.*

Informally, the safety specification of a program states that "something bad never happens". More formally, it defines a set of "bad" finite computation prefixes that should not be found in any computation. Thus, satisfaction of a safety specification implies that the program should not display any violating (bad) computation prefix.

2.4 Fault Models and Fault Tolerance

All standard fault models from practice which endanger a safety specification (transient or permanent faults) can be modeled as a set of added transitions. We focus on the subset of these fault models which can potentially be tolerated: We disallow faults to violate the safety specification directly. For example, in the token ring protocol, at most one process can hold the token. We allow a fault to duplicate the token, however we rule out faults that "force" a second process to hold a duplicated token, as this kind of faults cannot be tolerated. Rather, faults can change the program state (e.g., duplication of token) such that subsequent program actions execution (holding of duplicate token) violate

[2] Intuitively, fusion closure guarantees that history is available in each computation state.

the safety specification. This can be potentially tolerated by asking any process to check if some other process is already holding a token, before accepting one.

We defer for future work investigation of fault tolerance under the fault model where safety is directly violated.

Definition 3 (Fault model). *A* fault model F for program P and safety specification *SS is a set of transitions over the variables of P that do not violate SS, i.e., if transition (s_j, s_{j+1}) is in F and $s_0, s_1, \ldots, s_j$ is in SS, then $s_0, s_1, \ldots, s_j, s_{j+1}$ is in SS.*

We call members of F the *faults* affecting P. We say that a *fault occurs* if a fault transition is executed.

Definition 4 (Computation in the presence of faults). *A* computation of P in the presence of faults F *is a weakly P-fair sequence of states $s_0, s_1, \ldots$ such that s_0 is an initial state of P and for each $j \geq 0$, s_{j+1} results from s_j by executing a program action from P or a fault action from F*

Weakly P-fair means that only the actions of P are treated weakly fair (fault actions must not eventually occur if they are continuously enabled). In the transition system view, a fault model F adds a set of (fault) transitions to δ_P. We denote the modified transition relation by δ_P^F. We call δ_P^F the program P in presence of F. Since fault actions are not treated fairly, their occurrence is not mandatory. Note that we do not rule out faults that occur infinitely often (as long as they do not directly violate the safety property).

Earlier, we discussed that a safety specification entails keeping track of bad prefixes that should not appear in any computation. The requirement of a safety specification being fusion-closed allows us to keep track of bad *transitions*, rather than of prefixes.

Definition 5 (bad transition). *Give a program P, fault model F, and fusion-closed safety specification $SSPEC$. A transition $t \in \delta_p^F$ is* bad with respect to a safety specification SSPEC *if for all computations σ of p holds: If t occurs in σ then $\sigma \notin SSPEC$.*

This is possible as fusion-closure implies availability of history in every computation state, and the history (prefix) can be encoded into that state. Note that, under our fault model assumption, a fault transition cannot be a bad transition.

Definition 6 (Fail-safe fault-tolerance). *Given a program P with safety specification SS, and a fault model F. The program P is said to be* fail-safe F-tolerant for specification S *iff all computations of P in the presence of faults F satisfy SS.*

If F is a fault model and *SS* is a safety specification, we say that a *program P is F-intolerant for SS* iff P satisfies *SS* in the absence of F but violates *SS* in the presence of F. For brevity, we will write *fault-intolerant* instead of *F-intolerant for SS* if F and *SS* are clear from the context.

Definition 7 (Reachable transition). *A transition (s, t) of P is reachable iff there exists a computation α of P such that (s, t) occurs in α.*

Definition 8 (Reachable transition in the presence of faults). *We say that a* transition (s, t) is reachable by p in the presence of faults *iff there exists a computation α of P in presence of faults such that (s, t) occurs in α.*

3 Addition of Fail-Safe Fault Tolerance

In this section, we explain the addition of fail-safe fault tolerance to a fault-intolerant program. We first briefly review the role of detectors in ensuring fail-safe fault tolerance.

3.1 Role of Detectors in Fail-Safe Fault Tolerance

Informally, a detector[3] is a program component that detects whether a given predicate is true in a given state. Arora and Kulkarni showed in [3] that, for every action ac of a program P with safety specification SS, there exists a predicate such that execution of ac in a state where this predicate is true satisfies SS. In other words, the action ac is transformed as follows: $(g \rightarrow st) \rightarrow (d \wedge\ g \rightarrow st)$, where d is the detector implementing the predicate. In this case, we say that action ac is composed with detector d (we sometimes say that detector d is monitoring ac). We say that a program P is composed with detector d if there is an action ac of P such that ac is composed with d. We also say that a program P is composed with a set of detectors D if $\forall d \in D \exists\ ac$ of P such that ac is composed with d. If a transition (s, s') induced by ac violates SS, then such a transition is a bad transition. Thus, any computation that violates SS contains a bad transition.

Given a program P with safety specification SS expressed as a temporal logic formula, the set of bad transitions (due to fusion closure) can be computed in polynomial time by considering all transitions (s, s') where $s, s' \in S_p$. For simplicity, we assume that the safety specification is concisely expressed as a set of bad transitions. The authors of [3] also show that fail-safe fault-tolerant programs contain detectors. However, [3] did not show how to design the required detectors. To address this problem, Jhumka *et.al* [6, 7] developed a theory that underpins the design of effective (complete and accurate) detectors. We will develop the theory of multitolerance based on the theory of [6, 7], which we will briefly introduce for sake of completeness.

3.2 Transformation Problem for Addition of Fail-Safe Fault Tolerance

The problem of adding fail-safe fault tolerance is formalized as follows:

Definition 9 (Fail-safe fault tolerance addition). *Let SS be a safety specification, F a fault model, and P an F-intolerant program for SS. The fail-safe transformation problem is defined as follows: Identify a program P' such that:*

1. *P' satisfies SS in the presence of F.*
2. *In the absence of F, every computation of P' is a computation of P.*
3. *In the absence of F, every computation of P is a computation of P'.*

A program p' that satisfies the above conditions is said to solve the fail-safe transformation problem for p. The second and third conditions imply that the

[3] For a more formal introduction, we refer the reader to [3].

detectors need to be transparent in the absence of faults, and should not add other ways of satisfying SS.

In the next section, we present a theory of detectors, based upon which, we provide an algorithm that synthesizes a program p' from a fault-intolerant program p, such that p' solves the fail-safe transformation problem.

3.3 A Theory of Detectors

The detector theory of [6, 7] is based on the concept of SS-inconsistency, where SS is the safety specification of a program P. The intuition behind the inconsistency is that if a given computation of P in the presence of faults violates the safety specification SS, then some "erroneous" transition has occurred in the computation, i.e., inconsistent with SS.

Definition 10 (*SS*-inconsistent transitions). *Given a fault-intolerant program P with safety specification SS, fault model F, and a computation α of P in the presence of F. A transition (s, s') is SS-*inconsistent for P w.r.t. α *iff*

- *there exists a prefix α' of α such that α' violates SS,*
- *(s, s') occurs in α', i.e., $\alpha' = \sigma \cdot s \cdot s' \cdot \beta$,*
- *all transitions in $s \cdot s' \cdot \beta$ are in δ_p, and*
- *$\sigma \cdot s$ maintains SS.*

Fig. 1 illustrates Definition 10. It shows the state transition relation of a program in the presence of faults (the transition (s_3, s_4) is introduced by F). The safety specification SS identifies a bad transition (s_6, s_7) which should be avoided. In the presence of faults, this transition becomes reachable and hence the program if F-intolerant since it exhibits a computation α_1 violating SS. In this computation, the three transitions following the fault transition match Definition 10 and hence are SS-inconsistent w.r.t. α_1 in the presence of F. Note that an SS-inconsistent transition is only reachable in the presence of faults.

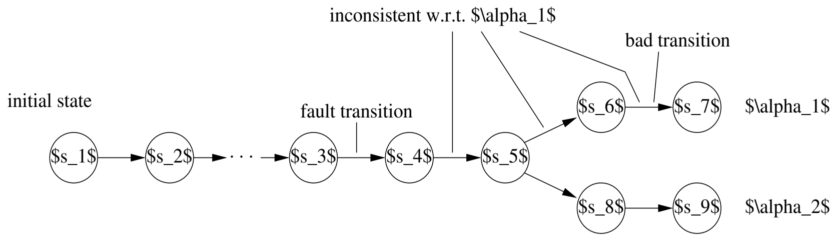

Fig. 1. Graphical explanation of SS-consistency.

Intuitively, an SS-inconsistent transition for a given program computation is a program transition where the subsequent execution of a sequence of program transitions causes the computation to violate the safety specification. In a sense, SS-inconsistent transitions lead the program computation on the "wrong path".

Now we define SS-inconsistency independent of a particular computation.

Definition 11 (*SS*-inconsistent transition for P). *Given a program P, safety specification SS, fault model F. A transition (s, s') is SS*-inconsistent for *P iff there exists a computation α of P in the presence of F such that (s, s') is SS-inconsistent for p w.r.t. α.*

In general, due to non-determinism in program execution, a transition can be *SS*-inconsistent w.r.t. a computation α_1, and not be *SS*-inconsistent w.r.t. α_2. If we cannot find a computation in the presence of faults for which a particular transition is *SS*-inconsistent then we say that this transition is *SS*-consistent.

The notion of *SS*-inconsistency is a characteristic for a computation which violates *SS* [6].

In the next section, we introduce the notion of perfect detectors using the terminology of *SS*-consistency.

3.4 Perfect Detectors

From Sec. 3.3, we observed that *SS*-inconsistent transitions are those transitions that can lead a program to violate its safety specification in the presence of faults if no precautions are taken. Perfect detectors are a means to implement these precautions. The definition of perfect detectors follows two guidelines: A detector d monitoring a given action ac of program P needs to (1) "reject" the starting states of all transitions induced by ac that are *SS*-inconsistent for P, and (2) "keep" the starting states of all induced transitions that are *SS*-consistent for P. These two properties are captured in the definition of *completeness* and *accuracy* of detectors.

Definition 12 (Detector accuracy). *Given a program P, safety specification SS, fault model F, and a program action ac of P. A* detector d monitoring ac is *SS*-accurate for ac in P in presence of F *iff for all transitions (s, s') induced by ac holds: if (s, s') is SS-consistent for P, then $s \in d$.*

Definition 13 (Detector completeness). *Given a program P with safety specification SS, fault class F, and a program action ac of P. A* detector d monitoring action ac is *SS*-complete for ac in P in presence of F *iff for all transitions (s, s') induced by ac holds: if (s, s') is SS-inconsistent for P, then $s \notin d$.*

Definition 14 (Perfect detector). *Given a program P, safety specification SS, fault class F, and a program action ac of P. A* detector d monitoring ac is *SS*-perfect for ac in P in presence of F *iff d is both SS-complete and SS-accurate for ac in P.*

Where the specification is clear from the context we will write *accuracy* instead of *SS-accuracy* (the same holds for completeness and perfection).

Intuitively, the completeness property of a detector is related to the safety property of the program p in the sense that the detector should filter out all *SS*-inconsistent transitions for p, whereas the accuracy property relates to the liveness specification of p in the sense that the detector should not rule out *SS*-consistent transitions. This intuition is captured by the following lemmas (for proof, refer to [6, 7]).

Lemma 1 (Fault-free behavior).

Given a fault-intolerant program P and a set D of perfect detectors. Consider program P' resulting from the composition of P and D. Then the following statements hold:

1. *In the absence of faults, every computation of P' is a computation of P.*
2. *In the absence of faults, every computation of P is a computation of P' .*

Before we characterize the role of perfect detectors in presence of faults, we formally define critical actions of a program.

Definition 15 (Critical and non-critical actions). *Given a program P with safety specification SS, and fault class F. An action ac of P is said to be* critical for P w.r.t SS in presence of F *iff there exists a transition (s, s') induced by ac such that (s, s') is a bad transition that is reachable by P in presence of faults F (Definition 7) . An action is non-critical for P w.r.t SS in presence of F iff it is not critical for P w.r.t SS in presence of F.*

Lemma 2 (Behavior in the presence of faults). *Given a fault-intolerant program P with safety specification SS, and fault class F. Given also a program P' by composing each critical action ac of P w.r.t SS in presence of F with a perfect detector for ac in presence of F. Then, P' satisfies SS in presence of faults F.*

Proofs of lemmas 1 and 2 can be found in [7]. From lemmas 1 and 2, we observe that a program P' obtained by composing each critical action ac of a fault-intolerant program P with a perfect detector for ac in P in presence of faults F (which can be shown to exist [6]) solves the fail-safe fault tolerance addition problem. When a fail-safe fault-tolerant program P' satisfies the three conditions for fail-safe fault tolerance addition problem, we say that P' is *perfectly fail-safe $F-$tolerant w.r.t SS* and that P' has *efficient fail-safe F-tolerance* (since P' is a *maximal* program that satisfies SS in presence of F).

3.5 Algorithm for Adding Perfect Fail-Safe Fault Tolerance:

Having established the role of perfect detectors in fail-safe fault tolerance, in Fig. 2, we provide an algorithm that solves the fail-safe transformation problem, using perfect detectors. It takes as arguments the program P, the fault class F, and the set ss of bad program transitions encoding the safety specification (it can be shown that these are induced by critical actions of P in presence of F).

The theory (and algorithm) presented adds fail-safe fault tolerance to a single fault class. We now extend the results to handle multiple fault classes.

4 Addition of Perfect Fail-Safe Multitolerance

In this section, we consider the addition of perfect fail-safe fault tolerance for multiple fault classes. Specifically, the main question is whether perfect detectors are composable, i.e., whether the addition of two perfect detectors for two different fault classes in a program preserve each other efficiency properties (accuracy and completeness)?

```
add-perfect-fail-safe($\delta_P, \delta_F, ss$: set of bad transitions):
{ $ss_r$ := get-ssr($\delta_P, \delta_F, ss$)
return ($P' = \delta_P \setminus ss_r$)}

get-ssr($\delta_P, \delta_F, ss$: set of transitions):
{ $ss_r := \{(s,t) | (s,t) \in ss$ is reachable by $P$ in presence of $F\}$
return ($ss_r$)}
```

Fig. 2. Algorithm that solves the fail-safe fault tolerance addition problem

4.1 A Stepwise Addition Approach

The approach adopted is stepwise, as also suggested by Arora and Kulkarni in [2]. One of the problems during the design of multitolerance is that a tolerance mechanism (detector in this case) for one fault class can interfere with the tolerance mechanism for another fault class. Thus, any synthesis method or automated procedure should ensure, by construction, that no interference exists between the tolerance mechanisms for different fault classes.

First, we define a fail-safe multitolerant program.

Definition 16 (Fail-Safe Multitolerant Program). *Given a program P with safety specification SS, and n fault classes $F_1 \dots F_n$. A program P is said to be* fail-safe multitolerant to $F_1 \dots F_n$ for *SS iff P is fail-safe F_i-tolerant for SS for each $1 \le i \le n$. A program P is said to be* perfectly fail-safe multitolerant to $F_1 \dots F_n$ for *SS iff P is perfectly fail-safe F_i-tolerant to SS for each $1 \le i \le n$.*

The stepwise approach considers one fault class at a time, in some fixed order $F_1 \dots F_n$. The fault-intolerant program P is transformed into a perfectly fail-safe multitolerant program to fault classes $F_1 \dots F_n$. In the first step, P is augmented with detectors that will make the resulting program P_1 perfectly fail-safe fault-tolerant to F_1. Then, in the second step, P_1 is augmented with detectors that will make the resulting program P_2 perfectly fail-safe fault-tolerant to F_2, while preserving its perfect fail-safe fault tolerance to F_1. The same is repeated until all fault classes are tolerated. In other words, we want to know if perfect detectors for the various fault classes compose. This represents the main contribution (synthesis of perfect fail-safe multitolerance) of the paper. In contrast, [2, 8] focused only only on fail-safe multitolerance (which can be trivially satisfied by using the empty program), whereas this paper focuses on the non-trivial provision of *perfect* fail-safe multitolerance. We provide below the non-interference conditions that need to be satisfied by a synthesis method:

Step 1 of Non-interference Conditions: Specifically, in the first step, when the fault-intolerant program P is augmented with detectors to obtain a program P_1, the following non-interference conditions need to be verified:

1. In the absence of F_1, the detector components added to P do not interfere with P, i.e., each computation of P is in the problem specification even if it executes concurrently with the new detector components.
2. In the presence of faults F_1, each computation of the detector components is in the components' specification even if they execute concurrently with P.
3. In the presence of faults F_1, the resulting program is perfectly fail-safe F_1-tolerant.

Step 2 of Non-interference Conditions: In the second step, when the fail-safe F_1-tolerant program p_1 is augmented with detectors that will make it fail-safe F_2-tolerant program, while preserving its fail-safe F_1 tolerance, the following non-interference conditions need to be satisfied:

1. In the absence of F_1 and F_2, the new detectors for fail-safe fault tolerance to F_2 do not interfere with p_1, i.e., each computation of p_1 satisfies the problem specification even if p_1 executes concurrently with the new detectors.
2. In the presence of F_1, the new detectors for fail-safe fault tolerance to F_2 do not interfere with the fail-safe fault tolerance to F_1 of p_1, i.e., every computation of $p_1$is in the fail-safe fault-tolerance specification to F_1 even if p_1 executes concurrently with the new components.
3. In the presence of F_1, the new detectors for fail-safe fault tolerance to F_2 do not interfere with the perfect detection to F_1 of p_1.
4. In the presence of F_2, p_1 does not interfere with the new detectors that provide fail-safe fault-tolerance to F_2, i.e., every computation of the new component is in the new components specification.
5. In the presence of F_2, p_1 does not interfere with the perfect detection to F_2 provided by the new detector components.

These steps can be easily generalized to n steps. Observe that these sets of conditions specify the transformation problem for addition of perfect fail-safe multitolerance to an initially fault-intolerant program. Our next goal is to derive a sound, and complete algorithm that satisfies the various non-interference conditions during the addition of fail-safe multitolerance.

Before detailing our automated approach for addition of fail-safe multitolerance, we present a key result behind our approach.

Lemma 3 (Perfect detectors and multitolerance). *Given a fault-intolerant program P with safety specification SS, and fault classes $F_1 \ldots F_n$. Given a program P_{i-1} which is perfectly fail-safe multitolerant for SS with perfect detection to fault classes $F_1 \ldots F_{i-1}$. Given also a program P_i obtained from P_{i-1} s.t P_i is perfectly fail-safe fault-tolerant to F_i. Then, P_i is also perfectly fail-safe multitolerant to fault classes $F_1 \ldots F_{i-1}$.*

Proof sketch: We can prove this by induction over the fault sequence. The base case is trivial, while for the inductive step, since all detectors added for F_i are perfect, they reject only SS-inconsistent transitions, i.e., they do not add any transition. Hence, the new set of perfect detectors added cannot interfere with the previous detectors. Thus, perfect detection for all previous fault classes is preserved.

P_i can be obtained from P_{i-1} by composing actions that are critical in the presence of F_i with the relevant perfect detectors (Lemma 2). Lemma 3 then shows that composition with perfect detectors preserves the perfect fail-safe fault tolerance to other classes. The lemma underpins the synthesis algorithm for perfect fail-safe multitolerance. The algorithm is sound (The returned program is indeed perfectly fail-safe multitolerant to all fault classes considered) and complete (if such a perfectly fail-safe multitolerant to all fault classes considered exists, then the algorithm will find it).

4.2 An Algorithm for Adding Efficient Fail-Safe Multitolerance

The algorithm for automatic synthesis of fail-safe multitolerant programs with perfect detection to all fault classes is shown in Fig. 3. The resulting program is fail-safe multitolerant to n fault classes by design (soundness).

Theorem 1 (Soundness and Completeness of *add-perfect-fail-safe-multitolerance*). *Algorithm* add-perfect-fail-safe-multitolerance *is sound and complete.*

Proof. The algorithm is sound by construction, based on Lemma 3. Completeness of the algorithm is due to our assumption of finite state (bounded) programs and by construction.

```
add-perfect-fail-safe-multitolerance($P, [F_1 \ldots F_n], ss$: set of transitions):

        {$i := 1$; $P_0 := P$
        while ($i \leq n$) do {
            $P_i :=$ add-perfect-fail-safe($P_{i-1}, F_i, ss$);
            $i := i + 1$;} od
        return($P_n$)}
```

Fig. 3. The algorithm adds fail-safe fault tolerance to n fault classes, with perfect detection to every fault class

It can also be shown that algorithm *add-perfect-fail-safe-multitolerance2* (see Fig. 4) is equivalent to algorithm *add-perfect-fail-safe-multitolerance*.

```
add-perfect-fail-safe-multitolerance2($P, [F_1 \ldots F_n], ss$: set of transitions):

        $P_n :=$ add-perfect-fail-safe($P, \bigcup_{i=1}^{n} F_i, ss$);
        return($P_n$)}
```

Fig. 4. The algorithm adds fail-safe fault tolerance to n fault classes, with perfect detection to every fault class

In the next section, we present a case study of the design of a perfect fail-safe multitolerant token ring.

5 Example of a Fail-Safe Multitolerant Token Ring

Processes $0 \ldots N$ are arranged in a ring. Process $k, 0 \leq k < N$ passes the token to process $k+1$, whereas process N passes the token to process 0. Each process k has a binary variable, $t.k$, and a process $k, k \neq N$ holds the token iff $t.k \neq t.(k+1)$, and process N holds the token iff $t.N = t.0$.

The fault-intolerant program for the token ring is as follows ($+_2$ is modulo-2 addition) :

$$\text{ITR1} :: k \neq 0 \wedge t.k \neq t.(k-1) \rightarrow t.k := t.(k-1)$$

$$\text{ITR2} :: k = 0 \wedge t.k \neq t.N +_2 1 \rightarrow t.k := t.N +_2 1$$

In the presence of faults, we do not want certain processes to take some steps. In particular, if the state of process k is corrupted, then process $k+1$ should not make any transition. The faults we consider here are general faults, such as timing, message loss or duplication, but such faults are detected by the process before any action inadvertedly accesses that state. When a fault is detected by process k, the value of $t.k$ is set to $\perp$.

Fault action: The first fault class F_1 that we consider is one that corrupts the state of a *single* process k, which can be any process.

$$F_1 :: t.k \neq \perp \wedge |\{k | t.k = \perp\}| = 0 \rightarrow t.k := \perp$$

Fail-Safe Fault Tolerance to Fault Class F_1: Running algorithm *add-perfect-fail-safe-multitolerance* will result in the following program after the first iteration.

$$\text{1-FSTR1} :: |\{k : t.k = \perp\}| \leq 1 \wedge t.(k-1) \neq \perp \wedge k \neq 0 \wedge t.k \neq t.(k-1) \rightarrow t.k := t.(k-1)$$

$$\text{1-FSTR2} :: |\{k : t.k = \perp\}| \leq 1 \wedge t.N \neq \perp \wedge k = 0 \wedge t.k \neq t.N +_2 1 \rightarrow t.k := t.N +_2 1$$

Theorem 2 (Fail-safe TR). *Program 1-FSTR is perfectly fail-safe fault-tolerant to F_1.*

Perfect Fail-Safe Fault Tolerance to Fault Classes F_1 and F_2: Second, we consider a fault class where the state of two processes k and l can be corrupted.

Fault action: The fault action that we consider is

$$F_2 :: t.k \neq \perp \wedge |\{k | t.k = \perp\}| = 1 \rightarrow t.k := \perp$$

The second iteration of algorithm *add-perfect-fail-safe-multitolerance* on program 1-FSTR will result in the following program:

$$\text{2-FSTR1} :: |\{k : t.k = \perp\}| \leq 2 \wedge t.(k-1) \neq \perp \wedge k \neq 0 \wedge t.k \neq t.(k-1) \rightarrow t.k := t.(k-1)$$

$$\text{2-FSTR2} :: |\{k : t.k = \perp\}| \leq 2 \wedge t.N \neq \perp \wedge k = 0 \wedge t.k \neq t.N +_2 1 \rightarrow t.k := t.N +_2 1$$

Theorem 3 (Fail-safe TR). *Program 2-FSTR is perfectly fail-safe fault-tolerant to F_1 and F_2.*

Fail-Safe Fault Tolerance to Fault Class $F_1 \ldots F_{N+1}$: We then consider a fault class that can corrupt the state of i $(1 \leq i \leq (N+1))$ processes.

Fault action: The fault action that we consider is

$$F_i :: t.k \neq \perp \wedge |\{k | t.k = \perp\}| = i - 1 \rightarrow t.k := \perp$$

The i^{th} iteration of algorithm *add-perfect-fail-safe-multitolerance* on program (i-1)-FSTR will result in the following program:

$$i\text{-FSTR1} :: |\{k|t.k = \perp\}| \leq i \wedge t.(k-1) \neq \perp \wedge k \neq 0 \wedge t.k \neq t.(k-1) \rightarrow t.k := t.(k-1)$$

$$i\text{-FSTR2} :: |\{k|t.k = \perp\}| \leq i \wedge t.N \neq \perp \wedge k = 0 \wedge t.k \neq t.N +_2 1 \rightarrow t.k := t.N +_2 1$$

Theorem 4 (Fail-safe TR). *Program i-FSTR is perfectly fail-safe fault-tolerant to F_1 to F_i for $1 \leq i \leq (N+1)$.*

From program i-FSTR, it can be easily deduced that, when $i = N+1$, $|\{k|t.k = \perp\}| \leq N+1$ is always "True" (cannot corrupt more processes than there exist), so program $N+1$-FSTR (or MFSTR - Multitolerant Fail-Safe Token Ring) simplifies to:

$$\text{MFSTR1} :: t.(k-1) \neq \perp \wedge k \neq 0 \wedge t.k \neq t.(k-1) \rightarrow t.k := t.(k-1)$$

$$\text{MFSTR2} :: t.N \neq \perp \wedge k = 0 \wedge t.k \neq t.N +_2 1 \rightarrow t.k := t.N +_2 1$$

Program MFSTR is perfectly fail-safe fault tolerant to fault classes that can corrupt the state of any number of processes (up to every process), and is identical to the fail-safe fault-tolerant token ring program presented by Arora and Kulkarni in [2]. However, our approach (and results) differs from that of [2] in two important ways. First, our approach is automated, hence proofs of correctness are obviated. Second, our intermediate programs are different, i.e., the programs tolerating less than (N+1) faults are different. In effect, all the intermediate programs in [2] are exactly the same. This is because *bad* transitions, even those that are unreachable in the presence of certain faults were removed. As a matter of contrast, for every fault class, we remove only those bad transitions that are reachable. Thus, though the overall multitolerant program is correct, the approach is not efficient as they remove more transitions than is strictly necessary. Another important consequence of our theory is that for multitolerance, a system designer knows what are sufficient conditions to achieve this. As can be observed, when a fault occurs, the system may deadlock. However, Arora and Kulkarni argued in [2] that, towards adding masking fault tolerance (both safety and liveness preserved), a stepwise approach can be adopted where first fail-safe fault tolerance is added followed by liveness properties. Hence, as future work, we are looking to automate the addition of components that add liveness to fail-safe fault-tolerant programs.

6 Summary

In this paper, we have made the following contributions: Based on previous work, (i) we have developed a theory for perfect fail-safe multitolerance, (ii) We have provided a sound, and complete algorithm that automates addition of perfect fail-safe multitolerance, while guaranteeing the non-interference conditions, and (iii) We have presented a case study of the design of a perfectly fail-safe multitolerant token ring that explains the working of our algorithm. The ability to automatically add fail-safe multitolerance to an initially fault-intolerant program is an important step in the design of fault-tolerant systems, the more so that the program is fail-safe multitolerant by design.

Acknowledgements: We wish to thank Felix Freiling for helpful discussions.

References

1. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
2. A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, Jan. 1998.
3. A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*, May 1998.
4. C. Fetzer and F. Cristian. Fail-awareness: An approach to construct fail-safe applications. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 282–291. IEEE, June 1997.
5. M. Hiller. Executable assertions for detecting data errors in embedded control systems. In *Proceedings of the International Conference on Dependable Systems and Network (DSN 2000)*, pages 24–33, 2000.
6. A. Jhumka, F. Freiling, C. Fetzer, and N. Suri. Automated synthesis of fail-safe fault-tolerance using perfect detectors. Technical report, University of Warwick, 2005.
7. A. Jhumka, M. Hiller, and N. Suri. An approach for designing and assessing detectors for dependable component-based systems. In *HASE*, pages 69–78, 2004.
8. S. Kulkarni and A. Ebnenasir. Automated synthesis of multitolerance. In *DSN*, 2004.
9. N. G. Leveson, S. S. Cha, J. C. Knight, and T. J. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering*, 16(4):432–443, 1990.