# ADDING FAULT-TOLERANCE TO STATE MACHINE-BASED DESIGNS

SANDEEP S. KULKARNI

*Computer Science and Engineering Department*
*Michigan State University*
*East lansing MI 48824*
*E-mail: sandeep@cse.msu.edu*

ANISH ARORA

*Computer Science and Engineering Department*
*The Ohio State University*
*Columbus Ohio 43210 USA*
*E-mail: anish@cse.ohio-state.edu*

ALI EBNENASIR

*Department of Computer Science*
*Michigan Technological University*
*Houghton MI 49931 USA*
*E-mail: aebnenas@mtu.edu*

Late detection of new types of faults often results in the evolution of fault-tolerance requirements while developers have already created design artifacts. Thus, the reuse of an existing design in the development of a fault-tolerant version thereof has the potential to reduce the overall development costs. Moreover, the automation of such a reuse yields a fault-tolerant design that is correct by construction, given that the existing design is correct. To facilitate such an automation, we present an approach, where we add three levels of fault-tolerance, namely *failsafe, nonmasking*, and *masking*, to functional designs represented as state machines. Intuitively, failsafe fault-tolerance requires that safety specification is met even in the presence of faults. In the presence of faults, nonmasking fault-tolerance guarantees recovery to states from where safety and liveness specifications are satisfied. Masking fault-tolerance stipulates that (i) recovery is provided to states from where safety and liveness specifications are met, and (ii) safety specification is satisfied during such a recovery. Specifically, we present sound and complete deterministic algorithms for automated addition of (failsafe/nonmasking/masking) fault-tolerance to the functional design of concurrent programs. These polynomial-time algorithms are especially useful in model-driven development of fault-tolerant systems, where models are automatically checked and modified. We also discuss (1) the effect of distribution and safety specification model on the complexity of adding fault-tolerance, and (2) the impact of the proposed algorithms on the addition of multitolerance.

*Keywords*: Software design, Addition of fault-tolerance, Model-driven development

## 1. Introduction

We focus our attention on the problem of *redesign for fault-tolerance* where new fault-tolerance requirements arise while developers have already created some design artifacts that meet functional properties. To address this problem, we propose an approach for automated addition of fault-tolerance concerns to an existing design represented as a state machine. Such an automated addition of fault-tolerance is especially useful as it has the potential to reuse the existing design of a program while generating a fault-tolerant version thereof. More importantly, if the existing design is correct with respect to its specification, then the automatically generated fault-tolerant design will also be correct by construction.

Existing automated techniques [1–7] for the synthesis of fault-tolerant designs mostly focus on generating a design from a satisfiability proof of its specification (i.e., specification-based approach). For example, Attie and Arora and Emerson [1] present a specification-based approach for synthesizing fault-tolerant programs from their temporal logic specification. In the synthesis of reactive programs [2,7], the program/environment interaction model is based on two-player games where players take turn in making moves and one player can only affect the state of the other player through specific interface variables. Instead, in the context of adding fault-tolerance properties, faults may perturb the state of programs to any state.

We present efficient algorithms for adding fault-tolerance to a given fault-intolerant design of a concurrent program in such a way that the addition is done *solely* for the purpose of dealing with faults; i.e., we do not introduce new ways to satisfy the specification *in the absence of faults*. While our previous work [8–10] on *manual* design has shown that transforming a fault-intolerant design into a fault-tolerant design is possible and desirable, we expect that an *automated* algorithm will further help in adding fault-tolerance to existing designs. More specifically, such an automated algorithm will obviate the need for constructing the proof of correctness of the synthesized fault-tolerant design, which is often supported by theorem proving or model checking methods [11]. This advantage is especially useful when designing fault-tolerant concurrent (respectively, distributed) programs as it is well-understood that manually constructing proofs of correctness for such programs is hard. Next, we layout the context in which we model programs, specifications and levels of fault-tolerance.

**The choice of computation model.** If we consider a very general model of computation, e.g., processes that communicate via messages with unbounded message queues, then adding fault-tolerance becomes undecid-

able [12]. Hence, we need to consider simpler models of computation where the complexity of the addition problem is manageable. We consider a *high atomicity model* of computation, where a program can read and write all its variables in an atomic step. In other words, the high atomicity model allows the program to perform operations such as 'test-and-set' over program variables.

**The choice of safety specification model.** Intuitively, a safety specification stipulates that nothing bad ever happens in program computations. We model a safety specification as a set of *bad* transitions that must not occur in program computations. Our *bad transitions* (BT) model is a restricted version of Alpern and Schneider's [13] model of safety specifications, where one represents safety by a set of finite *sequences* of transitions that must not occur in program computations. We adopt the BT model for specifying the safety specification of programs since (i) the complexity of adding fault-tolerance will significantly increase to the class of NP-hard problems if one uses Alpern and Schneider's general model of safety specifications [14], and (ii) the BT model is sufficiently expressive for specifying the safety of a wide range of practical problems as we have already synthesized several fault-tolerant designs for industrial applications (e.g., altitude switch controller, cruise control, and token ring [15]) using the BT model.

**Levels of fault-tolerance.** The level of fault-tolerance identifies the extent to which the original specification is satisfied when faults occur. Although one could consider arbitrary levels of fault-tolerance, experience shows that for most programs, the fault-tolerance level falls into one of the three levels, *failsafe, nonmasking* or *masking*. These levels are based on Alpern and Schneider's [13] definition where they show that a specification can be decomposed into a safety specification and a liveness specification. More specifically, these levels are based on whether the fault-tolerant program satisfies the safety specification, the liveness specification or both.

In the presence of faults, a failsafe fault-tolerant program only satisfies its safety specification. Failsafe fault-tolerance is often used in safety-critical systems. For nonmasking fault-tolerance, we observe that satisfying the liveness specification alone is not very useful as the liveness specification does not impose any restrictions on finite computations of a program. Hence, in the presence of faults, we allow safety to be violated, and when faults stop occurring, we require recovery to states from where both safety and liveness specifications are satisfied. Nonmasking fault-tolerance is often used in networking related applications (e.g., routing, spanning tree maintenance, etc.) where maintaining safety in the presence of faults is either too expen-

sive or impossible. Finally, a masking fault-tolerant program guarantees to (i) recover to states from where both safety and liveness specifications are satisfied, and (ii) satisfy its safety specification during such a recovery. Masking fault-tolerance is the ideal level of fault-tolerance and it is used in database applications as well as in several problems (e.g., leader election, mutual exclusion, etc.) in distributed systems.

**Contributions of the chapter.** The main contributions of this chapter are as follows: We present sound and complete algorithms that solve the problem of adding failsafe/nonmasking/masking fault-tolerance to state machine-based designs of programs. The complexity of our approach is polynomial in the state space of the fault-intolerant design (see Sections 4, 5 and 6). Moreover, we discuss the role of our proposed algorithms in (i) adding fault-tolerance to the design of distributed programs, (ii) developing a software tool for adding fault-tolerance during model-driven development of software systems, and (iii) adding *multitolerance* to program designs, where a multitolerant program is subject to multiple types of faults and provides different levels of fault-tolerance corresponding to each fault-type.

**Organization of this chapter.** This chapter is organized as follows: We provide the definitions of a program design, specifications, faults, and fault-tolerance in Section 2. Using these definitions, we state the addition problem in Section 3. In Section 4, we show how to add failsafe fault-tolerance to fault-intolerant designs. In Section 5, we present an algorithm for adding nonmasking fault-tolerance to fault-intolerant designs. Subsequently, in Section 6, we solve the problem of adding masking fault-tolerance. We demonstrate our algorithms for adding fault-tolerance to the program of a parking lot gate controller. In Section 7, we discuss related work. In Section 8, we give an overview of the current results and open problems regarding automatic addition of fault-tolerance. Finally, we make concluding remarks in Section 9.

## 2. Preliminaries

In this section, for the sake of completeness, we recall the definitions of a program design, problem specifications, faults, and fault-tolerance. The design of a program is defined in terms of its state space and its transitions. Hereafter, we use the terms design and program interchangeably as we use finite state machines to represent the design of a program. The definition of specifications is adapted from Alpern and Schneider [13]. The definitions of faults and fault-tolerance are due to Arora and Gouda [16] and Arora and Kulkarni [8].

### 2.1. *Program Design Model*

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

The projection of program $p$ on state predicate $S$, denoted as $p|S$, is the program $\langle S_p, \{(s_0, s_1) : (s_0, s_1) \in \delta_p \ \wedge \ s_0, s_1 \in S\} \rangle$. In other words, $p|S$ consists of transitions of $p$ that start in $S$ and end in $S$.

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

### 2.2. *Specifications and Correctness Criteria for Functional Designs*

A *specification* is a set of infinite sequences of states that is *suffix closed* and *fusion closed*. *Suffix closure* of the set means that if a state sequence $\sigma = \langle s_0, s_1, ... \rangle$ is in that set then so are all the suffixes of $\sigma$, where a suffix of $\sigma$ is a sequence $\langle s_i, s_{i+1}, ... \rangle$ for some finite $i$ such that $i \geq 0$. *Fusion closure* of the set means that if state sequences $\langle \alpha, x, \gamma \rangle$ and $\langle \beta, x, \delta \rangle$ are in that set then so are the state sequences $\langle \alpha, x, \delta \rangle$ and $\langle \beta, x, \gamma \rangle$, where $\alpha$ and $\beta$ are finite prefixes of state sequences, $\gamma$ and $\delta$ are suffixes of state sequences, and $x$ is a program state. Intuitively, fusion closure of the specification means that a design that implements the specification must execute its next transition only based on its current state; i.e., the history of a computation does not affect the next move of the program.

Following Alpern and Schneider [13], we rewrite the specification as the intersection of a *safety specification* and a *liveness specification*. For our addition algorithms, the safety specification is specified in terms of a set of bad transitions that must not occur in program computations. That is, for program $p$, its safety specification is a subset of $S_p \times S_p$. A computation violates the safety specification if it contains a bad transition. A computation satisfies the safety specification if it does not violate the safety specifica-

tion. We represent the liveness specification by a set of infinite sequences of states. A computation satisfies the liveness specification if it contains a suffix that is in the liveness specification. We show that a fault-tolerant program satisfies the liveness specification in the absence of faults iff its fault-intolerant version satisfies the liveness specification.

**Remark.** Since the specification is suffix-closed and fusion-closed, it is possible to specify a safety specification as a set of bad transitions. This result was proved earlier in [10] (see Page 26, Lemma 3.6 of [10]). Moreover, we have illustrated [14] that if one adopts Alpern and Schneider's general model of safety specification instead of our restricted model (i.e., the *bad transitions* model) then the complexity of adding fault-tolerance will significantly increase. Hence, for efficient synthesis, based on which tool support [15] can be provided, we represent safety with a set of bad transitions that must not occur in program computations.

Given a program $p$, a state predicate $S$, and a specification *spec*, we say that $p$ *refines* *spec* from $S$ iff (1) $S$ is closed in $p$, and (2) every computation of $p$ that starts in a state where $S$ is true satisfies (safety and liveness of) *spec*. If $p$ refines *spec* from $S$ and $S \neq \{\}$, we say that $S$ is an *invariant* of $p$ for *spec*.

For a finite sequence (of states) $\alpha$, we say that $\alpha$ *maintains* *spec* iff there exists a sequence of states $\beta$ such that $\alpha\beta \in$ *spec*. Likewise, we say that $\alpha$ *violates* *spec* iff it is not the case that $\alpha$ maintains *spec*. We say that $p$ maintains *spec* from $S$ iff $S$ is closed in $p$ and every computation prefix of $p$ that starts in a state in $S$ maintains *spec*.

***Notation.*** Let *spec* be a specification. We use the term *safety of spec* to mean the smallest safety specification that includes *spec*. Also, whenever the specification is clear from the context, we will omit it; thus, *S is an invariant of p* abbreviates *S is an invariant of p for* spec.

### 2.3. *Faults*

We systematically represent the faults that perturb a program by a set of transitions in program state space. We emphasize that such representation is possible notwithstanding the type of the faults (be they stuck-at, crash, fail-stop, omission, timing or Byzantine), the nature of the faults (be they permanent, transient, or intermittent), or the ability of the program to observe the effects of the faults (be they detectable or undetectable). Formally, a class of *fault* $f$ for program $p$ is a subset of $S_p \times S_p$. 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 $p \,[]\, f$. Thus, at each state where an invariant $S$ of $p$ is true, an $f$-span $T$ of $p$ from $S$ is also true. The closure of $T$ in $p \,[]\, f$ means that if any transition in $f$ (respectively, $p$) is executed in a state where $T$ is true, then $T$ is also true in the resulting state. It follows that for all computations of $p$ that start at states where $S$ is true, $T$ is a boundary in the state space of $p$ up to which (but not beyond which) the state of $p$ may be perturbed by the occurrence of the transitions in $f$.

As we defined a computation of $p$, we say that a sequence of states, $\sigma = \langle s_0, s_1, ...\rangle$ with $len(\sigma)$ states, is a computation of $p$ in the presence of $f$ iff the following three conditions are satisfied: (1) $\forall j : 0 < j < len(\sigma) : (s_{j-1}, s_j) \in (\delta_p \cup f)$, (2) if $\sigma$ is finite and terminates in state $s_l$ then there does not exist state $s$ such that $(s_l, s) \in \delta_p$, and (3) $\exists n : n \geq 0 : (\forall j : j > n : (s_{j-1}, s_j) \in \delta_p)$. The first requirement captures that in each step, either a program transition or a fault transition is executed. The second requirement captures that faults do not have to execute; i.e., if the program reaches a state where only a fault transition can be executed, it is not required that the fault transition be executed. It follows that fault transitions cannot be used to deal with deadlocked states. 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 [16–19] to ensure that eventually recovery can occur.

### 2.4. *Fault-Tolerance*

We now define what it means for a program to be fault-tolerant. We define three levels of fault-tolerance; *failsafe, nonmasking and masking*. Irrespective of the level of tolerance, in the absence of faults, a program should refine its specification from its invariant. The level of fault-tolerance characterizes the extent to which the program refines *spec* in the presence of faults. Intuitively, a failsafe fault-tolerant program ensures that in the presence of faults, the safety of *spec* is maintained. A nonmasking fault-tolerant program ensures that in the presence of faults, the program recovers to states from where *spec* is refined. A masking fault-tolerant program ensures that in the presence of faults both these properties are satisfied. Thus, we reiterate the definitions of these three levels of fault-tolerance (from [8,16]) as follows:

Program $p$ is *failsafe f-tolerant* to *spec* from $S$ iff the following two conditions hold: (1) $p$ refines *spec* from $S$, and (2) there exists $T$ such that (i) $T$ is an $f$-span of $p$ from $S$, and (ii) $p \,[]\, f$ maintains *spec* from $T$.

Program $p$ is *nonmasking f-tolerant* to *spec* from $S$ iff the following two conditions hold: (1) $p$ refines *spec* from $S$, and (2) there exists $T$ such that (i) $T$ is an $f$-span of $p$ from $S$, and (ii) every computation of $p \,[]\, f$ that starts from a state in $T$ has a state in $S$.

Program $p$ is *masking f-tolerant* to *spec* from $S$ iff the following two conditions hold: (1) $p$ refines *spec* from $S$, and (2) there exists $T$ such that (i) $T$ is an $f$-span of $p$ from $S$; (ii) $p \,[]\, f$ maintains *spec* from $T$, and (iii) every computation of $p \,[]\, f$ that starts from a state in $T$ has a state in $S$ .

***Notation.*** Henceforth, whenever the program $p$ is clear from the context, we will omit it; thus, "$S$ is an invariant" abbreviates "$S$ is an invariant of $p$" and "$f$ is a fault" abbreviates "$f$ is a fault for $p$". Also, whenever the specification *spec* and the invariant $S$ are clear from the context, we omit them; thus, "$f$-tolerant" abbreviates "$f$-tolerant for *spec* from $S$", and so on.

## 3. Problem Statement

In this section, we formally specify the problem of adding fault-tolerance to a fault-intolerant program, say $p$, in order to generate a fault-tolerant program, say $p'$. Towards this end, we first identify constraints so that $p'$ is obtained from $p$ by adding *only* fault-tolerance. Then, we discuss the soundness and the completeness issues in the context of the addition problem.

As described in Section 2, a fault-intolerant program $p$ is specified in terms of its state space $S_p$ , its transitions, $\delta_p$ , and its invariant, $S$. The safety of specification *spec* provides a set of bad transitions that should not occur in program computations. The faults, $f$, are specified in terms of state transitions. Likewise, the fault-tolerant program $p'$ is specified in terms of its state space $S_p$ , its state transitions, say $\delta'_p$ , its invariant $S'$ , its specification *spec*, and the level of fault-tolerance it provides.

Now, we consider what it means for a fault-tolerant program $p'$ to be *derived* from $p$. As mentioned in the introduction, our derivation is based on the premise that $p'$ is obtained by adding fault-tolerance alone to $p$, i.e., we should be able to prove that $p'$ refines *spec* from $S'$ in the absence of faults by simply using that '$p$ refines *spec* from $S$'. To precisely state this requirement, we consider the relation between (1) the invariants $S$ and $S'$, and (2) the transitions $\delta_p$ and $\delta'_p$.

- If $S'$ contains states that are not in $S$ then, in the absence of faults, $p'$ will include computations that start outside $S$. However, we cannot prove that such computations are in *spec* by just using

the fact that $p$ refines *spec* from $S$ as we have no information about computations of $p$ that originate outside $S$. Therefore, we require that $S' \subseteq S$ (equivalently, $S' \Rightarrow S$).

- Regarding the transitions of $p$ and $p'$, we focus only on the transitions of $p'|S'$ and $p|S'$. If $p'|S'$ contains a transition that is not in $p|S'$, $p'$ can use this transition in order to refine *spec* in the absence of faults. Once again, we cannot prove that the resulting computation is in *spec* by just using the fact that $p$ refines *spec* from $S$. Therefore, we require that $p'|S' \subseteq p|S'$.

Using the above two requirements, we define the addition problem as follows:

**The Addition Problem**
Given $p$, $S$, *spec* and $f$ such that $p$ refines *spec* from $S$,
Identify $p'$ and $S'$ such that:
$S' \subseteq S$,
$p'|S' \subseteq p|S'$, and
$p'$ is $\mathcal{L}$ $f$-tolerant to *spec* from $S'$, where
$\mathcal{L}$ is failsafe, nonmasking or masking. □

In order to define soundness and completeness in the context of the addition problem, we define the corresponding decision problem:

**The Decision Problem**
Given $p$, $S$, *spec* and $f$ such that $p$ refines *spec* from $S$,
Does there exist $p'$ and $S'$ such that:
$S' \subseteq S$,
$p'|S' \subseteq p|S'$, and
$p'$ is $\mathcal{L}$ $f$-tolerant to *spec* from $S'$?
(where $\mathcal{L}$ is failsafe, nonmasking or masking) □

***Notations.*** Given $p$, *spec*, $S$ and $f$ as input, we say that $p'$ and $S'$ solve the addition problem for this input iff $p'$ and $S'$ satisfy the three conditions of the addition problem. We say $p'$ (respectively, $S'$) solves the addition problem iff there exists $S'$ (respectively, $p'$) such that $p'$ and $S'$ solve the addition problem.

**Soundness and completeness.** An algorithm for the addition problem is *sound* iff for any given input, its output, namely $p'$ and $S'$, solves the addition problem. An algorithm for the addition problem is *complete* iff for any given input if the answer to the decision problem is affirmative then the algorithm always finds a program $p'$ and a non-empty state predicate $S'$.

Note that our notion of completeness is *relative* to the input fault-intolerant program $p$ and its invariant $S$. That is, a complete algorithm finds a fault-tolerant version of $p$ iff a fault-tolerant program $p'$ exists that solves the addition problem for $p$.

## 4. Adding Failsafe Fault-Tolerance

In this section, we present an algorithm for the addition of failsafe fault-tolerance. We also illustrate the soundness and completeness of our algorithm. As mentioned in Section 2, we represent a safety specification as a set of bad transitions that should not occur in program computations. Given a bad transition $(s_0, s_1)$, we consider two cases: (1) $(s_0, s_1)$ is not a transition of $f$, and (2) $(s_0, s_1)$ is a transition of $f$.

For case (1), we claim that $(s_0, s_1)$ can be removed while obtaining $p'$. To see this, consider two subcases: (a) state $s_0$ is reached in the computations of $p' \,[]\, f$, and (b) state $s_0$ is never reached in any computation of $p' \,[]\, f$. In the former subcase, the transition $(s_0, s_1)$ must be removed as the safety of *spec* can be violated if $p' \,[]\, f$ ever reaches state $s_0$ and executes the transition $(s_0, s_1)$. In the latter subcase, the transition $(s_0, s_1)$ is irrelevant and, hence, can be removed.

For case (2), we cannot remove the transition $(s_0, s_1)$ as it would mean removing a fault transition. Therefore, we must ensure that $p' \,[]\, f$ never reaches the state $s_0$. In other words, for all states $s$, the transition $(s, s_0)$ must be removed in obtaining $p'$. Moreover, if any of these removed transitions, say $(s_{-1}, s_0)$, is a fault transition then we must recursively remove all transitions of the form $(s', s_{-1})$ for each state $s'$.

Using the above two cases, our algorithm for obtaining a failsafe fault-tolerant program is as follows. First, it identifies states from where the execution of a sequence of fault transitions violates safety. This is done by a (smallest) fixpoint calculation where we begin with the empty set. Then, we add state $s_0$ to this set if there exists a fault transition $(s_0, s_1)$ such that either (1) $(s_0, s_1)$ violates safety, or (2) $s_1$ is added to this set earlier. Thus, the set of *marked states* (denoted $ms$) from where faults alone can violate safety is the smallest fixpoint of the following equation:

$$X = X \cup \{s_0 :: (\exists s_1 :: (s_0, s_1) \in f \ \wedge \ (s_1 \in X \vee (s_0, s_1) \text{ violates } spec))\}$$

Then, we compute the set of *marked transitions* (denoted $mt$) that must be removed from $p$. These transitions fall in two categories: (1) transitions that reach states in $ms$, and (2) transitions that violate the safety of *spec*.

If there exist states in the invariant such that execution of one or more fault transitions from those states violates the safety of *spec*, then we recalculate the invariant by removing those states. The recalculation of the invariant is a (largest) fixpoint calculation. As shown in Theorem 4.2, any invariant $S'$ that solves the addition problem must be a subset of $S-ms$. The same theorem also shows that if $p'$ solves the addition problem then $p'|S'$ must be a subset of $p-mt$. Moreover, we show that if $p'$ and $S'$ solve the addition problem then no state in $S'$ can be a deadlock state. Hence, the invariant $S'$ equals $RemoveDeadlocks(S-ms, p-mt)$, where $RemoveDeadlocks(S,p)$ is the (largest) fixpoint of the equation:

$$X = (X \cap S) - \{s_0 : (\forall s_1 : s_1 \in X : (s_0, s_1) \notin p)\}$$

Finally, we compute the transitions of the fault-tolerant program by removing transitions of $p-mt$ that start from a state in $S'$ but reach a state outside $S'$. Thus, our algorithm is shown in Figure 1 (As mentioned in Section 2, we use a program and its transitions interchangeably.):

```
Add_failsafe(p, f : transitions, S : state predicate, spec : specification)
{
   ms := smallestfixpoint(X = X ∪ {s0 :: (∃s1 : (s0, s1) ∈ f ∧
                                  (s1 ∈ X ∨ (s0, s1) violates spec))})
   mt := {(s0, s1) : ((s1 ∈ ms) ∨ (s0, s1) violates spec) };
   S' := RemoveDeadlocks(S − ms, p−mt);
   if (S' = {}) declare no failsafe f-tolerant program p' exists;
   else p' :=EnsureClosure(p−mt, S')
}

RemoveDeadlocks(S : state predicate, p : transitions)
// Returns the largest subset of S such that computations of p
// within that subset are infinite
  { return largestfixpoint(X = (X ∩ S) −
                                  {s0 : (∀s1 : s1 ∈ X : (s0, s1) ∉ p)}) }
EnsureClosure(p : transitions, S : set of states)
  { return p−{(s0, s1) : s0 ∈ S ∧ s1 ∉ S} }
```

Fig. 1. Addition of failsafe fault-tolerance.

**Theorem 4.1** Algorithm $Add\_failsafe$ is sound. (See [20] for proof) □

**Theorem 4.2** Algorithm $Add\_failsafe$ is complete. (See [20] for proof) □

The proofs of the above theorems show the maximality of the invariant

and the transitions within them. Thus, we have

**Corollary 4.3** The invariant output by $Add\_failsafe$ is the largest invariant that solves the addition problem. □

**Corollary 4.4** Let the output of $Add\_failsafe$ be the fault-tolerant program $p'$ and its invariant $S'$. If $p''$ and $S'$ solve the addition problem then $p''|S' \subseteq p'|S'$. □

**Theorem 4.5** Algorithm $Add\_failsafe$ is in $P$. (Proof in [20]) □

**Remark.** We do not explicitly identify the fault-span in $Add\_failsafe$. If the fault-tolerant program output by $Add\_failsafe$ is $p'$ and its invariant is $S'$, we can compute the fault-span by identifying states reached in the computations of $p' \; [] \; f$ starting from a state in $S'$. However, this is not the weakest possible fault-span. The weakest possible fault-span is $true - ms$, i.e., it includes all states except those from where faults alone violate safety. Moreover, we can obtain maximal transitions within the fault-span if we add to $p'$ all transitions that (1) begin in a state outside $S'$ and (2) are not in $mt$. We leave it to the reader to check that the program obtained by adding these transitions solves the addition problem.

### 4.1. *Case Study: Parking Lot Problem*

In order to illustrate the algorithms presented in this chapter, we use the parking lot problem (see Figure 2). The parking lot contains three spots for cars (marked $a$, $b$ and $c$). There are two gates to enter the parking lot. Immediately after each gate, there is a space where the customer can drop the car off (marked $l$ and $r$). If the gate is open, the customer may leave the car in this spot. The car will then be moved to one of the spots ($a$, $b$ or $c$). We assume that if the car is being moved from $l$ or $r$ to $a$, $b$ or $c$, no car can enter during this move. At the exit, there is one door. A car parked in spots $a$, $b$ or $c$ can leave through this door. However, only one car can leave the parking lot at a time. All doors are one-way, i.e., cars in spots $l$ and $r$ cannot leave and a car cannot enter in spots $a$, $b$ or $c$.

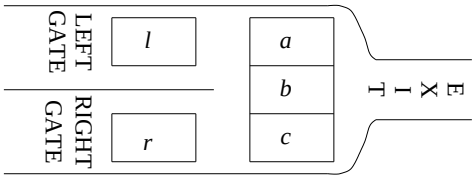


Fig. 2. Parking Lot Problem.

Thus, in the parking lot problem, there are three possible events: (1) a

car could be dropped off at the left gate and (possibly) moved to spots $a$, $b$ or $c$, (2) a car could be dropped off at the right gate and (possibly) moved to spots $a$, $b$ or $c$, or (3) a car could exit. For simplicity, we assume that each event is atomic. Also, we assume that in each spot there can be at most one car. Now, we describe the state space of the fault-intolerant program, its invariant, its safety specification, and its transitions. The fault-intolerant program, invariant, specification and faults identified in this section are used as input to the addition problem. We use the same input for $Add\_failsafe$ (this section), $Add\_nonmasking$ (in Section 5.1), and $Add\_masking$ (in Section 6.1).

**Variables and the state space of fault-intolerant program, $IP$.** To model the parking lot, we maintain three variables; $x$, $y$ and $z$. The variable $x$ denotes the number of cars that can be let in through the left gate, $y$ denotes the number of cars that can be let in through the right gate, and $z$ denotes the number of cars in the lot. Hence, the left (respectively, right) gate is open when $x$ (respectively, $y$) is positive. The domain of $x$ and $y$ is $\{0, 1, 2, 3\}$. The domain of $z$ is $\{0, 1, 2, 3, 4, 5\}$. A state of $IP$ is obtained by assigning each variable a value from its domain. The state space of $IP$, thus, contains 4x4x6 (=144) states.

**Safety specification, $spec_{IP}$.** The safety specification requires that any car in the parking lot should be able to leave. Clearly, a car in spots $a$, $b$ or $c$ can leave. However, if spots $a$, $b$ and $c$ are occupied and there is a car in spot $l$ (respectively, $r$) then that car cannot leave. Hence, it is required that the value of $z$ is at most three. Also, to model the assumption that only one event (entry/exit) can occur at a time, it is required that the value of $x$, $y$ and $z$ can change at most by 1. Thus, the safety specification rules out the following transitions:

$$spec_{IP} = \{(s_0, s_1) : z(s_0){>}3 \ \vee \ z(s_1){>}3 \ \vee \\ |x(s_1) - x(s_0)|{>}1 \ \vee \ |y(s_1) - y(s_0)|{>}1 \ \vee \ |z(s_1) - z(s_0)|{>}1\}$$

Note that the above safety specification allows $x$, $y$ and $z$ to change simultaneously. To model the requirement that if a car enters the left gate then another car cannot enter the right gate at the same time, we could strengthen the above safety specification to include a predicate of the form: 'if the value of $x$ is changed then the value of $y$ cannot change.' However, we have chosen the above specification to simplify the presentation, especially while adding masking fault-tolerance.

**Transitions $\delta_{IP}$.** For brevity, we write program transitions in terms of guarded commands [21]. A guarded command is of the form $g \longrightarrow st$,

where $g$ is a state predicate and $st$ is a statement that updates the variables in the program. The guarded command $g \longrightarrow st$ corresponds to the set of transitions $\{(s_0, s_1) : g$ is true in state $s_0$ and $s_1$ is obtained by *atomic* execution of $st$ in state $s_0\}$. The fault-intolerant program $IP$ contains four actions: The first two actions let a car enter, and the last two actions let a car exit. Upon exit, the value $x$ (or $y$) is non-deterministically increased so that a new car can enter. For simplicity, we do not model the actions corresponding to the movement of the car inside the parking lot. Thus, the transitions of the fault-intolerant program, $\delta_{IP}$, are captured by the following actions.

$$\begin{array}{lllll} IP1 :: & x{>}0 \ \wedge \ z{<}5 & \longrightarrow & x, z := x{-}1, z{+}1 \\ IP2 :: & y{>}0 \ \wedge \ z{<}5 & \longrightarrow & y, z := y{-}1, z{+}1 \\ IP3 :: & y{<}3 \ \wedge \ z{>}0 & \longrightarrow & y, z := y{+}1, z{-}1 \\ IP4 :: & x{<}3 \ \wedge \ z{>}0 & \longrightarrow & x, z := x{+}1, z{-}1 \end{array}$$

**Invariant, $S_{IP}$.** We let the invariant of the fault-intolerant program, $S_{IP}$, be $x{+}y{+}z = 3$.

Once again, this is not the only possible invariant; $x + y + z \leq 3$ is another possibility. We have chosen $S_{IP}$ as it describes several facets of the algorithm while keeping the presentation simple.

**Faults.** For our case study, we consider two faults. The first fault action, $F1$, allows a car to sneak in. We will use this fault to demonstrate an example where failsafe and masking fault-tolerance cannot be designed. The second fault action, $F2$, allows a car to sneak out. This fault action will be used to demonstrate an example where fault-tolerance can be added. Thus, the fault actions are as follows:

$$\begin{array}{lllll} F1 :: & z{<}5 & \longrightarrow & z := z{+}1 \\ F2 :: & z{>}0 & \longrightarrow & z := z{-}1 \end{array}$$

Now, given the transitions $\delta_{IP}$, faults $F1/F2$ and the specification $spec_{IP}$, we add fault-tolerance to program $IP$.

**Adding failsafe fault-tolerance to $F1$.** First, we compute the set $ms$, the set of states from where faults alone can violate safety. Towards this end, consider states $s_1, s_2, s_3$ and $s_4$ where the value of $x$ and $y$ is 0 and the value of $z$ in state $s_j$ is $j$, for $1 \leq j \leq 4$. Observe that $(s_3, s_4)$ is a transition of fault $F1$ that violates safety. It follows that $s_3$ should be included in $ms$. Also, $(s_2, s_3)$ is a transition of $F1$ that reaches $s_3$ and, hence, $s_2$ is included in $ms$. Continuing thus, state $s_1$ will also be included in $ms$. Following this

discussion, it is easy to see that starting from an arbitrary state, fault $F1$ can cause the value of $z$ to be greater than 3. Thus, $ms$ includes all states. And, since $ms$ contains all possible states, it follows that $mt$ contains all possible transitions.

Now, the first argument to RemoveDeadlocks (see Figure 1), $S_{IP} - ms$, is the empty set. Hence, RemoveDeadlocks will return the empty set. Subsequently, $Add\_failsafe$ will declare that failsafe fault-tolerance cannot be added. (This is an expected result; if the fault can continue to increase the value of $z$, it will not be possible to guarantee that $z$ will be at most 3.)

**Adding failsafe fault-tolerance to $F2$.** Once again, we compute $ms$ for $F2$. Observe that if the value of $z$ in a state is 3 or less, execution of $F2$ cannot violate safety. However, if the value of $z$ in a state is 4 or 5, execution of $F2$ violates safety. Hence, $ms$ equals the set $\{s_0 : z(s_0) > 3\}$. Thus, $S_{IP} - ms$ equals $S_{IP}$.

The set $mt$ includes the set $spec_{IP}$ and the set of transitions that reach $ms$. It follows that $mt$ equals $spec_{IP}$. Using the values of $ms$ and $mt$, RemoveDeadlocks is called with parameters $S_{IP}$ and $\delta_{IP} - mt$. We leave it to the reader to verify that RemoveDeadlocks returns the set $S_{IP}$.

The transitions of the failsafe fault-tolerant program are obtained by calling RemoveDeadlocks. RemoveDeadlocks removes transitions of $IP$ that originate in $S_{IP}$ but reach a state outside $S_{IP}$. Since $S_{IP}$ is an invariant of $IP$, no such transitions exist. Thus, the transitions of the failsafe fault-tolerant program, $\delta_{FP}$ are as follows:

$$\delta_{FP} = \{(s_0, s_1) : (s_0, s_1) \in \delta_{IP} \ \wedge \ z(s_0) \leq 3 \ \wedge \ z(s_1) \leq 3\}$$

From the above discussion, the invariant of the failsafe fault-tolerant program is $S_{IP}$ and its transitions are those transitions of $IP$ where the value of $z$ in the initial and final state is 3 or less. In the presence of faults, the failsafe fault-tolerant program may reach a state where $x+y+z$ is less than 3. In this case, the failsafe fault-tolerant program can deadlock if it reaches a state where $x$, $y$ and $z$ are 0.

**Remark.** Note that the execution of $F2$ from states in $S_{IP}$ did not violate the safety specification. Hence, the invariant of the fault-tolerant program remained unchanged. If we had considered the fault where $z$ could be increased only if the initial value of $z$ was 3 then in that case the invariant of the fault-tolerant program would have been $S_{IP} \wedge (z < 3)$.

## 5. Adding Nonmasking Fault-Tolerance

In order to design a nonmasking $f$-tolerant program $p'$, we ensure that if $p$ is perturbed by faults $f$ then it eventually recovers to a state in $S$. To

obtain the nonmasking $f$-tolerant program, for each state $s_0$, $s_0 \notin S$, we add a transition $(s_0, s_1)$ such that $s_1 \in S$. We present our algorithm for adding nonmasking $f$-tolerant to programs in Figure 3:

```
Add_nonmasking(p, f : transitions,
        S : state predicate, spec : specification)
{
    S' := S;
    p' := (p|S) ∪ {(s0, s1) : s0 ∉ S ∧ s1 ∈ S}
}
```

Fig. 3. Addition of nonmasking fault-tolerance.

**Comment on the Add_nonmasking algorithm.** Adding nonmasking fault-tolerance has important applications in the design of resilient network protocols and reactive programs where a program provides recovery in the presence of failures. In the high atomicity model, the addition of such recovery to an existing program is straightforward since processes can read/write all program variables in an atomic step. Hence, we only need to add single-step recovery transitions from states outside the invariant to the invariant. However, we have shown [22,23] that adding nonmasking fault-tolerance to distributed programs is non-trivial and the *Add_nonmasking* algorithm provides an upper bound for checking the feasibility of adding recovery to distributed program.

**Theorem 5.1** Algorithm *Add_nonmasking* is sound and complete.

**Proof.** By construction, $p'$ and $S'$ satisfy the conditions of the addition problem. Thus, the algorithm is sound. Also, the algorithm always finds a solution to the addition problem. □

**Corollary 5.2** The invariant output by *Add_nonmasking* is the largest invariant that solves the addition problem. □

**Corollary 5.3** Let the output of *Add_nonmasking* be the fault-tolerant program $p'$ and its invariant $S'$. If $p''$ and $S'$ solve the addition problem then $p''|S' \subseteq p'|S'$. □

**Theorem 5.4** Algorithm *Add_nonmasking* is in $P$. (The proof is trivial, hence omitted.) □

***Note.*** The results of this section also hold for Alpern-Schneider's general model of safety specifications (for high atomicity programs) as the invariant is not modified and preserving safety is not required while adding recovery. However, the complexity of adding nonmasking fault-tolerance to distributed programs in Alpern-Schneider's model is still unknown.

### 5.1. *Parking Lot Problem: Adding Nonmasking Fault-Tolerance*

We consider the problem of adding nonmasking fault-tolerance to $F1$ and $F2$ identified in Section 4.1. Applying *Add_nonmasking* to program $IP$ with invariant $S_{IP}$, the invariant and the transitions of the nonmasking fault-tolerant program, say $OP$ are as follows:

$$S_{OP} = S_{IP}$$
$$\delta_{OP} = \{(s_0, s_1) : (s_0 \in S_{IP} \ \wedge \ (s_0, s_1) \in \delta_{IP}) \ \vee \ (s_0 \notin S_{IP} \ \wedge \ s_1 \in S_{IP})\}$$

Thus, in states in $S_{IP}$, the nonmasking program has the same transitions as $IP$. From states outside $S_{IP}$, $OP$ simply recovers to any state where $S_{IP}$ is true. Note that some of these recovery transitions violate safety; for example, a transition that changes the value of $x$ (respectively, $y$ or $z$) by more than one is included in these transitions. In the derivation of a masking fault-tolerant version of $IP$, we need to ensure that such transitions are not executed. In other words, it is necessary that the recovery to the invariant should occur without these transitions.

## 6. Adding Masking Fault-Tolerance

In order to design a masking $f$-tolerant program $p'$, we proceed to identify the weakest invariant $S'$ (which is stronger than $S$) and the weakest fault-span $T'$. As argued in Theorem 4.2, our first estimate of $S'$ is $S_1$ where $S_1$ = RemoveDeadlocks$(S - ms, p - mt)$. Likewise, we estimate $T'$ to be $T_1$ where $T_1 = true - ms$, i.e., $T_1$ includes all states except those in $ms$.

The calculation of the invariant and the fault-span for the masking fault-tolerant program is also a (largest) fixpoint calculation. However, due to the nesting involved in this fixpoint calculation, for simplicity, we present it operationally as a loop (Lines 5-13 in Figure 4) where we strengthen $S_1$ and $T_1$ while ensuring that if some $S'$ solves the addition problem then $S' \subseteq S_1$. This loop contains three key steps (Lines 7, 8 and 9). Line 7 is a simple statement that computes the transitions, $p_1$, that may be used if $S_1$ is the invariant and $T_1$ is the fault-span. Lines 8 and 9 are fixpoint calculations for the fault-span and the invariant respectively. The value of $p_1$ is used to compute these fixpoints. These steps are as follows:

(1) To compute the set of transitions, say $p_1$, that can be included in the masking fault-tolerant program, first, we include transitions that start from a state in $S_1$; i.e., $p|S_1$. Then, we consider transitions that start from a state in $T_1$–$S_1$. By the closure of the fault-span, these transitions

include those that reach a state in $T_1$. Finally, we remove the transitions $mt$ from this set (Line 7).

(2) We recompute the fault-span on Line 8 in Figure 4 using a (largest) fixpoint calculation. For this fixpoint calculation, we first remove states in $T_1$ from where it is not possible to reach a state in $S_1$ using the transitions of the program identified in Step 7; i.e., $p_1$. Now, if there are states, say $s_0$ and $s_1$, such that $s_0$ is in the fault-span, $s_1$ is outside the fault-span and $(s_0, s_1)$ is a fault transition then $s_0$ must be removed from the fault-span. Thus, we strengthen $T_1$ to ConstructFaultSpan($T_1 - \{s : S_1$ is not reachable from $s$ in $p_1$ $\}$, $f$), where ConstructFaultSpan($T, f$) is the (largest) fixpoint of the following equation:

$$X = (X \cap T) - \{s_0 :: (\exists s_1 :: (s_0, s_1) \in f \quad \wedge \quad s_1 \notin X)\}$$

(3) Since $S_1$ must be a subset of $T_1$, we recalculate (line 9 in Figure 4) the invariant using RemoveDeadlocks($S_1 \wedge T_1$, $p_1$) where RemoveDeadlocks itself is a (largest) fixpoint calculation.

We continue the loop on Lines 5-13 until we achieve the largest fixpoint for $S_1$. After the largest fixpoint is found, we compute the program transitions by assigning ranks to all states in $T_1$ and removing cycles outside $S_1$. We present our algorithm for adding masking fault-tolerance in Figure 4.

**Theorem 6.1** Algorithm *Add_masking* is sound. (See [20] for proof) □

**Theorem 6.2** Algorithm *Add_masking* is complete. (See [20] for proof) □

The proofs of the above theorems also show the maximality of the invariant and the transitions within it. Thus, we have

**Corollary 6.3** The invariant output by *Add_masking* is the largest invariant that solves the addition problem. □

**Corollary 6.4** Let the output of *Add_masking* be the fault-tolerant program $p'$ and its invariant $S'$. If $p''$ and $S'$ solve the addition problem then $p''|S' \subseteq p'|S'$. □

**Theorem 6.5** Algorithm *Add_masking* is in $P$.

**Proof.** Note that the repeat-until loop can be executed only for a polynomial number of times as in each iteration either size of $S_1$ or size of $T_1$ decreases. As in the proof of Theorem 4.5, each statement in *Add_masking* is in $P$. Thus, the algorithm *Add_masking* is in $P$. □

**Modification for stepwise synthesis.** As discussed earlier, the invariant and the fault-span computed by an algorithm that solves the addition problem should be maximal if the output of the addition algorithm is to be used as an input when fault-tolerance is added in a stepwise fash-

```
Add_masking(p, f : transitions, S : state predicate, spec : specification)
{
  ms := smallestfixpoint(X = X ∪ {s0 :: (∃s1 : (s0, s1) ∈ f ∧
                                  (s1 ∈ X ∨ (s0, s1) violates spec))})        (1)
  mt := {(s0, s1) : ((s1 ∈ ms) ∨ (s0, s1) violates spec) };                   (2)
  S1 := RemoveDeadlocks(S − ms, p − mt);                                      (3)
  T1 := true − ms;                                                            (4)

  repeat                                                                      (5)
      T2, S2 := T1, S1;                                                       (6)
      p1 := p|S1 ∪ {(s0, s1) : s0 ∉ S1 ∧ s0 ∈ T1 ∧ s1 ∈ T1} − mt;             (7)
      T1 := ConstructFaultSpan(T1 −
                              {s : S1 is not reachable from s in p1 }, f);    (8)
      S1 := RemoveDeadlocks(S1 ∧ T1, p1);                                     (9)
      if (S1 = {} ∨ T1 = {})                                                  (10)
          declare no masking f-tolerant program p′ exists;                    (11)
          exit                                                                (12)
  until (T1 = T2 ∧ S1 = S2);                                                  (13)

  For each state s : s ∈ T1 :                                                 (14)
      Rank(s) = length of the shortest computation prefix of p1               (15)
                that starts from s and ends in a state in S1;
      p′ := {(s0, s1) : ((s0, s1) ∈ p1) ∧ (s0 ∈ S1 ∨ Rank(s0) > Rank(s1))};   (16)
      S′ := S1;                                                               (17)
      T′ := T1                                                                (18)
}

ConstructFaultSpan(T : state predicate, f : transitions)
// Returns the largest subset of T that is closed in f.
{
  return largestfixpoint(X = (X ∩ T) − {s0 : (∃s1 : (s0, s1) ∈ f ∧ s1 ∉ X)})
}
```

Fig. 4. Addition of masking fault-tolerance.

ion. Corollaries 6.3 and 6.4 show the maximality of the invariant and the transitions inside it. Regarding the fault-span, these conditions are satisfied by the value of $T_1$ and $p_1$ at the end of *Add_masking*. However, to ensure that the fault-tolerant program does not remain in $T_1 - S_1$ forever, we removed certain transitions from $p_1$. This removal may be premature if the output of *Add_masking* is to be used as an input to add fault-tolerance to another fault. We have addressed [24] this problem by developing sound and complete algorithms for stepwise addition of fault-tolerance.

### 6.1. *Parking Lot Problem: Adding Masking Fault-Tolerance*

In this section, we demonstrate the addition of masking fault-tolerance in the context of the parking lot example.

**Adding masking fault-tolerance to $F1$.** As in the case of failsafe fault-tolerance, we begin with computation of $ms$, $mt$ and the first guess at the invariant of the fault-tolerant program. As discussed in Section 4.1, the invariant $S_1$ computed on Line 3 is the empty set and, hence, *Add_masking* will declare that masking fault-tolerance cannot be added. Once again this is expected; if a fault could increase the value of $z$ arbitrarily, we cannot ensure that $z$ is bounded by 3.

**Adding masking fault-tolerance to $F2$.** As in the case of failsafe fault-tolerance, the invariant $S_1$ on Line 3 will be equal to $S_{IP}$. Also, since $ms$ equals the set $(z>3)$, the value of $T_1$ on Line 4 is $z\leq 3$.

Now, we consider the first iteration of the loop on Lines 5-13. On Line 7, we compute $p_1$ as follows. First, we include a transition that originates in $S_{IP}$ (i.e., where the value of $x+y+z$ equals 3) iff it is in $\delta_{IP}$. Then, we add a transition that originates in $T_1 - S_{IP}$ (i.e., where the value $z$ is less than 3 but the sum of $x$, $y$ and $z$ is not 3) iff it reaches a state in $T_1$. Then, we remove the transitions in $mt$ (i.e., transitions where the value of $x$ (or $y$ or $z$) changes by more than 1 and transitions where the value of $z$ is more than three).

Now, observe that from each state in $T_1$, it is possible to reach a state in $S_{IP}$ by using the transitions in $p_1$. (From any state in $T_1$, we can systematically increase or decrease $x$, $y$ and $z$ by 1 so that $x+y+z$ equals 3.) Hence, the value of $T_2$ is the same as $T_1$. Subsequently, on Line 9, $S_1$ is also the same as $S_2$. Thus, the loop on Line 13 terminates.

After the loop terminates, we consider the transitions in $p_1$ and decide the rank of each state in $T_1$. Consider the state where $x=0, y=0$, and $z=0$. From this state, in one transition, we can reach a state where $x=1, y=1$ and $z=1$, and the resulting state is in $S_1$. Hence, the rank of the state where $x=0, y=0$ and $z=0$ is 1. Likewise, the rank of the states where the sum of $x$, $y$ and $z$ is in the set $\{0, 1, 2, 4, 5\}$ is 1.

For states where $x+y+z$ equals 6, if all three variables are non-zero, in one transition a state in $S_1$ is reached (by decreasing each $x$, $y$ and $z$). Hence, the rank of these states is 1. Now, consider a state, say $s$, where the value of one variable, say $x$, is 0. If $x+y+z$ equals 6 then the values of $y$ and $z$ must be 3. From $s$, it is not possible to reach $S_1$ by using one transition. However, by using two transitions, it is possible to reach a state in $S_1$. Hence, the rank of a state, where $x+y+z$ equals 6 and the value of

$x$ (respectively, $y$ or $z$) is 0, is 2.

Finally, the rank of the states where $x+y+z$ equals 7, 8 or 9, is 2.

Now, to obtain the transitions of the masking fault-tolerant program (Line 16), we use the transitions of $p_1$ where the rank decreases by 1. Thus, the transitions of the masking fault-tolerant program, $p'$ are as shown in Figure 5.

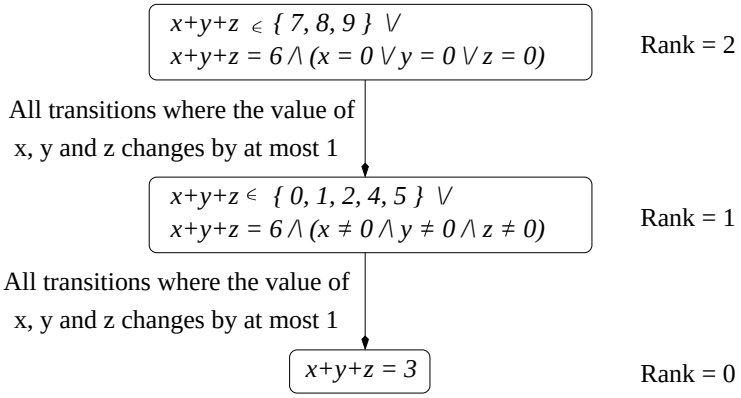


Fig. 5. Transitions of the Masking Fault-tolerant Program to Parking Lot Problem.

**Remark.** Note that all states in $T_1$ are not reached in the presence of $F2$. However, as argued in Section 3, one of the goals of our algorithms is to compute the largest possible invariant and largest possible fault-span. This is desirable if the program output by our synthesis procedure is to be used as an input to add fault-tolerance to another fault. Also, the parking lot program designed in this section is masking fault-tolerant to the fault that increments/decrements the value of $x$, $y$ and $z$ such that the value of these variables is at 3 or less.

## 7. Related Work

Numerous approaches exist for automatic generation of the design of a system most of which produce a design from a given specification (i.e., *specification-based* approaches). Specifically, Arora, Attie and Emerson [1] present a method for automatic synthesis of fault-tolerant programs from their temporal logic specifications. Their approach has its roots in other specification-based methods [25–28], where a design is generated from a satisfiability proof of the specification. One of the major differences between

specification-based methods and our algorithm is in our input. We begin with an existing design, whereas the specification-based methods [1,25–28] begin with a specification in some temporal logic. For this reason, we believe that our algorithms will be especially useful if a fault-intolerant design is already known or if other constraints (such as unavailability of a complete specification of the given fault-intolerant design) require that we reuse the fault-intolerant design.

In control theory, several approaches [3–6,29–31] exist for automatic synthesis of the design of a discrete-event controller from the specification of a controlled system. Our work differs from synthesizing discrete-event controllers in that (i) the computation model for synthesizing controllers is based on prioritized synchronization, whereas ours is based on interleaving, and (ii) the complexity of synthesizing a fault-tolerant design in the context of the formulation presented in this chapter is polynomial (in design state space), whereas the complexity of synthesizing controllers is NP-hard [32].

In the game-theoretic approaches, many techniques for the synthesis of controllers [7,33,34] and reactive programs [2] are based on the model of two-player games where a program makes moves in response to the moves of its environment. The program and its environment interact through a set of interface variables, and hence, the environment can only update the interface variables, whereas, in our model, faults can perturb all program variables. Moreover, in a two-player game model, players take turns and the set of states from where the first player can make a move is disjoint from the set of states from where the second player can move [7], whereas, in our work, fault-tolerance should be provided against the faults that can execute from any state.

## 8. Impact of Proposed Algorithms and Open Problems

In this section, we discuss the role of the algorithms presented in this chapter in adding fault-tolerance to the design of distributed programs and automatic addition of *multitolerance* to fault-intolerant designs, where a multitolerant program provides different levels of fault-tolerance corresponding to different types of faults. We also discuss complexity issues and open problems for future work.

**Application in adding multitolerance.** We have reused our algorithms in adding multitolerance to fault-intolerant designs [24]. For example, if a program is simultaneously subject to two classes of faults $f_1$ and $f_2$, and failsafe fault-tolerance is expected against $f_1$ and masking fault-tolerance is expected against $f_2$ then we use the $Add\_failsafe$ and the

*Add_masking* algorithms to transform the input fault-intolerant program to a multitolerant program that is failsafe $f_1$-tolerant and masking $f_2$-tolerant; i.e., *failsafe-masking* multitolerance. Using the soundness and completeness of our algorithms, we have developed sound and complete algorithms [24], where we add failsafe-masking and *nonmasking-masking* multitolerance to fault-intolerant programs. We have shown that the complexity of such addition is polynomial in program state space. However, we have shown that adding *failsafe-nonmasking* multitolerance is NP-hard! This is a surprising result in that in this chapter we showed that the simultaneous addition of failsafe *and* nonmasking fault-tolerance (i.e., masking fault-tolerance) for one fault-type can be done in polynomial time, whereas adding failsafe fault-tolerance for a fault-type $f_1$ and nonmasking fault-tolerance for a different fault-type $f_2$ is NP-hard [24].

**The effect of the safety specification model.** Given the significance of the proposed algorithms for adding fault-tolerance, it is equally important to determine the complexity of such addition of fault-tolerance in a weaker (i.e., more general) model of specification. In particular, in this chapter, we showed that if one represents a safety specification as a set of bad transitions that must not occur in program computations (i.e., bad transitions (BT) model) then the time complexity of adding failsafe, nonmasking, and masking fault-tolerance is polynomial (in program state space). The BT model is a restricted version of Alpern-Schneider's [13] general model of safety specifications, where safety is represented as a set of finite *sequences* of transitions that must not occur in program computations. We have shown [14] that if one uses Alpern-Schneider's general model of safety specification then the complexity of adding masking fault-tolerance will be NP-hard. A direct outcome of this result concerns the complexity of multitolerance in Alpern-Schneider's general model, which is NP-hard, in general, for failsafe-masking and nonmasking-masking multitolerance but unknown for failsafe-nonmasking multitolerance (see Figure 6). Hence, we argue that, for efficient automation, the focus should be on restricted models of safety specifications such as the BT model. We note that even though the BT model is not as expressive as Alpern-Schneider's general safety model, it is still sufficiently expressive to capture numerous practical problems. We have modeled the safety specification of several industrial applications (e.g., an altitude switch controller and a cruise control system [15]) using the BT model. The table in Figure 6 summarizes the effect of the safety specification model on the complexity of adding fault-tolerance and multitolerance in the high atomicity model.

| | F | N | M | F+M | N+M | F+N |
|---|---|---|---|---|---|---|
| **BT Specification Model** | P* | P* | P* | P [24] | P [24] | NPC [24] |
| **Alpern-Schenider's General Model of Safety Specification** | ? | P* | NPH [14] | NPH [14] | NPH [14] | ? |

**Legend**:
NPH: NP-Hard, but not known to be NPC
NPC: NP-Complete
F: Failsafe
F+M: Failsafe-Masking Multitolerance
P: Polynomial
N: Nonmasking
N+M: Nonmasking-Masking Multitolerance
? : Open problem
M: Masking
F+N: Failsafe-Nonmasking Multitolerance
* : Results shown in this paper
NP: Known to be in NP, but not known to be in P or to be NPC

Fig. 6. The effect of the safety specification model on the complexity of adding fault-tolerance and multitolerance. (The reference numbers refer to the papers in which the results have been appeared.)

**Adding fault-tolerance to distributed programs.** In this chapter, we addressed the problem of adding fault-tolerance in a high atomicity model, where program processes can read/write all program variables in one atomic step. We have also investigated the addition of fault-tolerance to the design of distributed programs, where processes have read/write restrictions with respect to program variables. Specifically, we illustrate that adding masking [22] and failsafe [35] fault-tolerance to distributed programs is NP-complete (in program state space). (Please see the second row, first and third columns of the table in Figure 7). To deal with the exponential complexity of the addition problem for distributed programs, Kulkarni, Arora and Chippada [36] present a set of deterministic polynomial heuristics. These heuristic are applied to provide a deterministic method for deciding about removing/retaining transitions (respectively, states) during the addition of fault-tolerance. Kulkarni and Ebnenasir [23] also present heuristics for enhancing the level of the tolerance of nonmasking distributed programs to masking fault-tolerance. Using these heuristics, Ebnenasir and Kulkarni [15] have developed an extensible software framework, called Fault-Tolerance Synthesizer (FTSyn), where developers can add fault-tolerance to (distributed) programs. We have used FTSyn for automatic addition of fault-tolerance to the design of several programs including Byzantine agreement, token rings, diffusing computation, and a simplified version of an altitude switch [15]. FTSyn has also been used in Networked Embedded Software Technology (NEST) [37]. (The source code of FTSyn is available at `http://www.cs.mtu.edu/~aebnenas/research/`

`tools/ftsyn.htm`.) To our knowledge, FTSyn is the first software tool for automatic addition of fault-tolerance to distributed programs.

Another way to reduce the complexity is to identify sufficient conditions for polynomial-time addition of fault-tolerance. Kulkarni and Ebnenasir [35] have identified a class of distributed programs and a class of specifications for which failsafe fault-tolerance can be added in polynomial time. They have also shown that programs and specifications for commonly encountered problems such as consensus, commit and agreement fall in this class. It follows that failsafe fault-tolerant programs for these problems can be designed in polynomial time. We have used these results [38] to identify heuristics that strengthen the given specification (respectively, add determinism to the given fault-intolerant program) in such a way that failsafe fault-tolerance can be added in polynomial time by using the modified specification (respectively, program). Yet another approach for dealing with the exponential complexity of adding fault-tolerance to distributed programs is to provide reuse during synthesis. Towards this end, we have presented a synthesis method [39] for adding pre-synthesized fault-tolerance components during the synthesis of different programs. We have used such components [39,40] for dealing with message loss, link/node failure, and process-restart faults.

| | F | N | M | F+M | N+M | F+N |
|---|---|---|---|---|---|---|
| **High Atomicity Model** | P* | P* | P* | P [24] | P [24] | NPC [24] |
| **Distributed Programs** | NPC [35] | NP [22] | NPC [22] | NPC [24] | NPC [24] | NPC [24] |

Fig. 7. The effect of program model on the complexity of adding fault-tolerance in the bad transition (BT) safety specification model.

Even though the algorithms presented in this chapter add fault-tolerance to high atomicity programs, they provide an upper bound for reasoning about the possibility of adding fault-tolerance to distributed programs. For example, we have used [23] the high atomicity algorithms of this chapter in reasoning about the feasibility of enhancing the level of fault-tolerance from nonmasking to masking for distributed programs. Specifically, we search the recovery paths provided by a nonmasking fault-tolerant program from each state $s$ outside the invariant to find those paths of execution where safety specification is preserved; i.e., safe recovery paths. To find safe recovery paths in distributed programs, we have to ensure the safety of the actions of each individual process considering all possible combinations of the states of

other processes, which is a computationally expensive task. Hence, we first use the high atomicity algorithms of this chapter to investigate whether the high atomicity version of the nonmasking program provides a safe recovery path originated at $s$. If the high atomicity nonmasking program does not provide any safe recovery originated at $s$ then we infer that providing such a safe recovery for a distributed version of that programs would be impossible.

## 9. Conclusions and Future Work

In this chapter, we considered the problem of adding fault-tolerance to the design of a given fault-intolerant program represented as a finite state machine. The input to the problem was the design of a fault-intolerant program, its invariant, its safety specification and faults. The output of the problem was the design of a fault-tolerant program with a new invariant.

While solving the addition problem, we considered three commonly encountered fault-tolerance levels in a high atomicity program model, where a process could read and write all variables in an atomic step. The three levels –failsafe, nonmasking and masking– were based on the extent to which the original specification is satisfied when faults occur. We presented three polynomial-time (in the program state space) algorithms for adding fault-tolerance to a fault-intolerant program. We also illustrated that the invariant output by our algorithm is maximal and that the fault-tolerant program output by our algorithm provides maximal non-determinism inside the invariant. As argued in Section 3, this property is desirable when designing multitolerant programs where one adds fault-tolerance for multiple classes of faults in a stepwise fashion. Kulkarni and Ebnenasir [24] illustrate how they use the algorithms presented in this chapter to design sound and complete algorithms for stepwise addition of multitolerance. We also summarized the impact of the proposed algorithms on the addition of fault-tolerance to distributed programs.

In a broader perspective, we are interested in identifying the problems and formulations where the addition of fault-tolerance can be achieved efficiently (in polynomial time), and also the problems for which exponential complexity is inevitable (unless $P = NP$). By identifying such a boundary, we can determine the problems and formulations that can reap the benefits of automation and the problems for which heuristics need to be developed in order to benefit from automation. This chapter helps to make this boundary more precise in that we presented a high atomicity model of computation and a bad transition (called BT) model of safety specification for which adding three levels of fault-tolerance to existing programs

can be done in polynomial time (in program state space). Furthermore, we discussed the effect of the safety specification model on the complexity of adding fault-tolerance.

We are currently investigating the application of different state space reduction techniques in the addition of fault-tolerance. Specifically, we plan to incorporate techniques used in model checking (e.g., partial ordering, state compression) in the implementation of the software framework Fault-Tolerance Synthesizer (FTSyn) [15] so that we extend the scope of synthesis to programs with large state space.

## References

1. P. C. Attie, A. Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS). (A preliminary version of this paper appeared in Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC), 1998.)*, 26(1):125 – 185, 2004.
2. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *ACM Symposium on Principles of Programming Languages*, pages 179–190, Austin, Texas, 1989.
3. P.J. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.
4. K.H. Cho and J.T. Lim. Synthesis of fault-tolerant supervisor for automated manufacturing systems: A case study on photolithography process. *IEEE Transactions on Robotics and Automation*, 14(2):348–351, April 1998.
5. Karen Rudie, Stephane Lafortune, and Feng Lin. Minimal communication in a distributed discrete-event systems. *IEEE Transactions On Automatic Control*, 48(6), June 2003.
6. Kurt Ryan Rohloff. *Computations on distributed discrete-event systems*. PhD thesis, University of Michigan, MI, USA, 2004.
7. N. Wallmeier, P. Hütten, and Wolfgang Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In *CIAA, LNCS, Vol. 2759*, pages 11–22, 2003.
8. A. Arora and Sandeep S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *International Conference on Distributed Computing Systems*, pages 436–443, May 1998.
9. A. Arora and Sandeep S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.
10. S. S. Kulkarni. *Component-based design of fault-tolerance*. PhD thesis, Ohio State University, OH, USA, 1999.
11. Wilfried Steiner, John Rushby, Maria Sorea, and Holger Pfeifer. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. In *The International Conference on Dependable Systems*

*and Networks*, pages 189–198, Florence, Italy, June 2004. IEEE Computer Society.
12. M. J. Fischer, N. .A. Lynch, and M. S. Peterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):373–382, 1985.
13. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
14. S. S. Kulkarni and A. Ebnenasir. The effect of the safety specification model on the complexity of adding masking fault-tolerance. *IEEE Transaction on Dependable and Secure Computing*, 2(4):348–355, 2005.
15. Ali Ebnenasir and Sandeep S. Kulkarni. FTSyn: A framework for automatic synthesis of fault-tolerance. `http://www.cs.mtu.edu/~aebnenas/research/tools/ftsyn.htm`.
16. A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.
17. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.
18. A. Arora and Sandeep S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Transactions on Software Engineering*, 24(6):435–450, 1998. A preliminary version appears in the Proceedings of the Fourteenth Symposium on Reliable Distributed Systems, Bad Neuenahr, 174–185, 1995.
19. G. Varghese. *Self-stabilization by local checking and correction*. PhD thesis, MIT/LCS/TR-583, 1993.
20. Sandeep S. Kulkarni and Anish Arora. Automating the addition of fault-tolerance. Technical Report MSU-CSE-00-13, Department of Computer Science, Michigan State University, East Lansing, Michigan, June 2000.
21. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
22. S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *In Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 82–93, 2000.
23. Sandeep S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 441–449, 2003.
24. Sandeep S. Kulkarni and Ali Ebnenasir. Automated synthesis of multitolerance. In *In Proceedings of International Conference on Dependable Systems and Networks (DSN), Palazzo dei Congressi, Florence, Italy*, pages 209–218, July 2004.
25. E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synchronize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.
26. P. Attie and E. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Transactions on Programming Languages and Systems*, 20(1):51–115, 1998.
27. Z. Manna and P. Wolper. Synthesis of communicating processes from tempo-

ral logic specifications. *ACM Transactions on Programming Languages and Systems*, 6:68–93, 1984.
28. P. Attie and A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM TOPLAS (a preliminary version appeared in ACM Symposium on Principles of Distributed Computing, 1996)*, 23(2), March 2001.
29. Feng Lin and W. Murray Wonham. Decentralized control and coordination of discrete-event systems with partial observation. *IEEE Transactions On Automatic Control*, 35(12), December 1990.
30. S. Lafortune and F. Lin. On tolerable and desirable behaviors in supervisory control of discrete event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 1(1):61–92, 1992.
31. Karen Rudie and W.M. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions On Automatic Control*, 37(11):1692–1708, 1992.
32. P. Gohari and W. M. Wonham. On the complexity of supervisory control design in the RW framework. *IEEE Transactions on Systems, Man and Cybernetics, Part B*, 30(2):643–652, October 2000.
33. Wolfgang Thomas. On the synthesis of strategies in infinite games. In *STACS*, pages 1–13, 1995.
34. Wolfgang Thomas. Infinite games and verification (extended abstract of a tutorial). In *14th International Conference CAV, Copenhagen, Denmark, July 27-31, LNCS, Vol. 2404*, pages 58–64, 2002.
35. S. S. Kulkarni and A. Ebnenasir. Complexity issues in automated synthesis of failsafe fault-tolerance. *IEEE Transaction on Dependable and Secure Computing*, 2(3):201–215, July-September 2005.
36. S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of Byzantine agreement. In *Symposium on Reliable Distributed Systems*, pages 130 – 139, 2001.
37. A. Arora, M. Gouda, T. Herman, S. Kulkarni, and M. Nesternko. Self-stabilization in networked embedded software technology (NEST). Available at: http://www.dsic-web.net/meetings/urb3butp/index.html, July 2002.
38. Ali Ebnenasir and Sandeep S. Kulkarni. Efficient synthesis of failsafe fault-tolerant distributed programs. Technical Report MSU-CSE-05-13, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, April 2005.
39. Sandeep S. Kulkarni and Ali Ebnenasir. Adding fault-tolerance using pre-synthesized components. In *Fifth European Dependable Computing Conference (EDCC-5), LNCS, Vol. 3463, p. 72*, 2005.
40. Ali Ebnenasir and Sandeep S. Kulkarni. Hierarchical presynthesized components for automatic addition of fault-tolerance: A case study. In *In the extended abstracts of the ACM workshop on the Specification and Verification of Component-Based Systems (SAVCBS), Newport Beach, California*, 2004.