Towards Reusing Formal Proofs for Verification of Fault-Tolerance[1]

Borzoo Bonakdarpour, Sandeep S. Kulkarni

Department of Computer Science and Engineering,
Michigan State University,
East Lansing, MI 48824, USA
Email: {borzoo, sandeep}@cse.msu.edu
http://www.cse.msu.edu/~{borzoo,sandeep}

Abstract. In this paper, we concentrate on mechanical verification of synthesis algorithms that add multitolerance to fault-intolerant programs using the theorem prover PVS. Multitolerance is desirable when a program is subject to different classes of faults and for each class, a different level of fault-tolerance has to be guaranteed. With this verification, we formally prove the correctness of the synthesis algorithms, which in turn shows that that any program synthesized by them is indeed *correct-by-construction*. We effectively *reuse* formal proofs of our previous work on a fixpoint theory on finite sets and a fault-tolerance theory developed for the case where programs are subject to a single class of faults. We believe manual reuse of proofs may suggest ways to automate them for verification of similar types of synthesis algorithms.

Keywords: Mechanical verification, Theorem proving, PVS, Program transformation, Program synthesis, Fault-tolerance, Multitolerance.

1 Introduction

Formal verification is considered a necessity as a means to gain confidence on correctness of systems. Gaining this confidence is more crucial when the system at hand is required to satisfy a set of high assurance properties. In such systems (e.g., mission-critical systems), fault-tolerance is a crucial part, in which a failure may lead the entire system to a catastrophic outcome. Hence, one needs strong confidence on fault-tolerance properties of a given system and, hence, formal verification of such systems (e.g., using theorem proving techniques) is inevitable.

In the literature, the focus has mostly been on verification of concrete fault-tolerant systems in two broad categories. The first category addresses formal verification of fault-tolerant synchronous/asynchronous circuits and architectures (e.g., [1–3]). The second category covers formal verification of a wide range of fault-tolerant protocols. For instance, Owre et al [4] present a survey on formal verification of a fault-tolerant digital-flight control systems. Mantel and Gärtner verify the correctness of a

[1] This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF grant EIA-0130724, and a grant from Michigan State University.

fault-tolerant broadcast protocol [5]. Qadeer and Shankar [6] mechanically verify the self-stability property of Dijkstra's mutual exclusion token ring algorithm [7]. Kulkarni, Rushby, and Shankar [8] verify the same algorithm by exploiting the theory of detectors and correctors [9].

In the aforementioned work, since the authors focus on verification of concrete systems, formal proofs of verification of one systems cannot be easily *reused* to verify the correctness of other systems. More recently, in [10, 11], we verified the correctness of the synthesis algorithms proposed in [12] that add fault-tolerance to fault-intolerant programs using the theorem prover PVS. By this verification, not only we mechanically prove the correctness of the algorithms, but also we show that any program synthesized by these algorithms is indeed correct-by-construction. More importantly, since these algorithms are the basis for their extensions to deal with occurrence of faults from different classes [13] and for synthesizing distributed [14] and real-time [15] programs, it is expected that formal specification and verification of the synthesis algorithms presented in [10, 11] are reusable in developing specification and verification of algorithms in [13–15].

In this paper, we focus on the problem of *verifying synthesis algorithms that add multitolerance* [13] *to fault-intolerant programs* using PVS [2]. Towards this end, we generalize the formal specification of our fault-tolerance theory developed in [10, 11] so that it takes the notion of different classes of faults into account. Moreover, we manually reuse the formal proofs developed for verification of the algorithms that add fault-tolerance with respect to a single class of faults to prove the correctness of algorithms in [13] that add multitolerance to programs that are subject to multiple classes of faults. We believe that manual reuse of formal proofs suggests ways to develop automated proofs for similar types of algorithms, especially those involve fixpoint calculations.

Since the core of synthesis algorithms for adding both fault-tolerance and multitolerance is fixpoint calculation, we effectively apply the fixpoint theory developed in [10]. This theory is developed for the case where domain and range of functions whose fixpoints are needed are finite sets. This specialized theory (which is reusable elsewhere) has a special interpretation in fault-tolerant computing for calculating largest invariants and smallest sets of states from where fault-tolerant requirements cannot be guaranteed.

Organization of the paper. We present formal definitions of programs, specifications, faults, and fault-tolerance in Section 2. Then, we state the problem of mechanical verification of synthesis of fault-tolerance in Section 3. In Section 4, we review our previous work on a fixpoint theory on finite sets and a fault-tolerance theory in PVS. Then, in Section 5, we present formal specification and verification of automated synthesis of multitolerance.

2 Modeling a Fault-Tolerance Framework

In this section, we present formal definitions of programs, specifications, faults, and fault-tolerance in PVS. These definitions are independent of platform and architecture.

[2] The URL `http://www.cse.msu.edu/~borzoo/pvs` contains the PVS specifications and formal proofs.

2.1 Program

Since we do not consider concrete systems, in our framework, the notion of *state* is abstract and, in PVS, modeled by an Uninterpreted Type. Likewise, a *transition* is modeled as an ordered pair of states, which is also an uninterpreted type. Furthermore, we assume that the number of states and transitions is *finite*.

A *program* p is a finite set of transitions in its state space. The *state space* of p, S_p, is the set of all possible states of p, which is modeled as the fullset over the type of states in PVS. The type Action denotes finite sets of transitions. Hence, p is defined as a constant of type Action. A *state predicate* of p is a subset of S_p. In PVS, we model state predicates by the type StatePred, which is sets of finite sets over states. The *closure* of a state predicate S in program p is formally specified as follows: $closed(S,p) = \forall s_0, s_1 \mid (s_0,s_1) \in p : (s_0 \in S \Rightarrow s_1 \in S)$. A sequence of states, $\langle s_0, s_1, ... \rangle$, is a *computation* with respect to a set Z of transitions iff any pair of two consecutive states is a transition in Z. We formalize this by a Dependent Type as follows:

$$Computation(Z) : TYPE =$$
$$\{c : sequence[state] \mid (\forall i \mid i \geq 0 : (c_i, c_{i+1}) \in Z)\}$$

where $sequence[state] : \mathbb{N} \rightarrow state$ and Z is any finite set of type Action. Furthermore, a computation *prefix* is formally specified as follows:

$$prefix(Z,j) : TYPE =$$
$$\{c : sequence[state] \mid (\forall i \mid i < j : (c_i, c_{i+1}) \in Z)\}$$

Due to convenience, we deliberately model computation prefixes by infinite sequences in which only a finite part is used. The *projection* of program p on state predicate S consists of transitions of p that start in S and end in S:

$$p \mid S = \{(s_0,s_1) \mid (s_0,s_1) \in p \ \wedge \ (s_0, s_1 \in S)\}.$$

2.2 Specification

Following Alpern and Schneider [16], a specification consists of a *safety specification* and a *liveness specification*. We let the the safety specification Σ_{bt} be a set of *bad transitions*, which is formally specified by a constant of type Action. Given program p, state predicate S, and specification Σ_{bt}, we say that p *satisfies* Σ_{bt} from S iff (1) S is closed in p, and (2) no computation of p that starts from a state in S contains a transition in Σ_{bt}. If this is not the case, we say that p *violates* Σ_{bt}. If p satisfies Σ_{bt} from S and $S \neq \{\}$, we say that S is an *invariant* of p. Since we do not deal with a specific program, in PVS, we model an invariant by an arbitrary state predicate that is closed in p.

2.3 Faults and Fault-Tolerance

The faults that a program is subject to are systematically represented by a finite set of transitions [9, 17], which is modeled by a constant of type Action. We model a computation of program p in presence of faults f as $c : Computation(p \cup f)$.
Also, we formally specify the *fault-span* of p from S as follows:

$$FaultSpan(T, S, p \cup f) = ((S \subseteq T) \ \wedge \ (closed(T, p \cup f))).$$

Observe that for all computations of p that start at states where S is true, T is a boundary in the state space up to which the state of p may be perturbed by the transitions in f.

We now define describe what we mean by different levels of fault-tolerance. Intuitively, a failsafe program, does not violate its safety specification in the presence of faults. Formally, we say that p is *failsafe* f-tolerant to Σ_{bt} from S iff two conditions hold: (1) p satisfies Σ_{bt} from S, and (2) there exists T such that T is a fault-span of p from S, and no computation prefix of $p \cup f$ that starts in T has a transition in Σ_{bt}. A nonmasking program may temporarily violate safety, but it must recover to the invariant by taking a bounded number of transitions. Formally, we say p is *nonmasking* f-tolerant to Σ_{bt} from S iff the following conditions hold: (1) p satisfies Σ_{bt} from S, and (2) every computation of $p \cup f$ that starts from any state of the state space contains a state of S. Intuitively, a masking program guarantees that if faults occur, the program continues to satisfy its safety specification. Moreover, we are ensured that after the occurrence of faults, the program eventually recovers to the invariant. Formally, we say that p is *masking* f-tolerant to Σ_{bt} from S iff the following conditions hold: (1) p satisfies Σ_{bt} from S, and (2) there exists T such that T is an f-span of p from S, no computation prefix of $p \cup f$ that starts in T has a transition in Σ_{bt}, and every computation of $p \cup f$ that starts from a state in T contains a state of S.

In [12], the liveness specification is modeled implicitly. That is, a failsafe fault-tolerant program does not *deadlock* in the absence of faults. On the other hand, non-masking and masking programs do not deadlock in both the absence and presence of faults.

3 Problem Statement

Given are a program p with invariant S, a class of faults f, and safety specification Σ_{bt} such that p satisfies Σ_{bt} from S. The goal is to find a program p' with invariant S' such that p' is f-tolerant to Σ_{bt} from S'. Observe that:

1. If S' contains states that are not in S then, in the absence of faults, p' may include computations that start outside S. Since we require that p' satisfies Σ_{bt} from S', it implies that p' is using a new way to satisfy Σ_{bt} in the absence of faults.
2. If $p' \mid S'$ contains a transition that is not in $p \mid S'$ then p' can use this transition in order to satisfy Σ_{bt} in the absence of faults.

Thus, the synthesis problem is as follows.
Synthesis problem. Given p, S, f, and Σ_{bt} such that p satisfies Σ_{bt} from S. Identify p', S', such that:

$(C1)$ $S^{'} \subseteq S$
$(C2)$ $p^{'} \mid S^{'} \subseteq p \mid S^{'}$
$(C3)$ p' is f-tolerant to Σ_{bt} from S'. $\square$

We say that an algorithm for the synthesis problem is *sound* iff for any given input, its output, namely p' and S', satisfies the synthesis problem. Our goal is to mechanically verify that the proposed algorithms in [13] are indeed sound.

4 Review of Previous Results

In this section, we recap our results (presented in [10, 11]) on mechanical verification of synthesis algorithms proposed in [12] using PVS. More specifically, in Subsection 4.1, we review a theory for fixpoint calculations on finite sets. In subsections 4.2, 4.3, and 4.4, we reiterate our results on verification of synthesis algorithms for addition of failsafe, nonmasking, and masking fault-tolerance, respectively.

4.1 A Theory of Fixpoint Calculations

A fixpoint of a function $f : X \rightarrow X$ is any value $x_0 \in X$ such that $f(x_0) = x_0$. In the context of finite sets, both domain and range of f are finite sets of finite sets. In this section, the variables i and k range over natural numbers. The variable x is any finite set of any uninterpreted type (e.g., states) and variable b is any member of such finite set.

Largest fixpoint calculation. Let $\mathsf{DecFunc}$ be the type of functions g such that $g : \{A : \mathit{finiteset}\} \rightarrow \{B : \mathit{finiteset} \mid B \subseteq A\}$. In other words, $g(x) \subseteq x$, for all finite sets x. Let $Dec(i, x)(g)$ be a recursive function that removes the elements of the initial set x returned by the function g of type $\mathsf{DecFunc}$ at every step:

$$Dec(i, x)(g) = \begin{cases} Dec(i-1, x)(g) - g(Dec(i-1, x)(g)) & \text{if } i \neq 0; \\ x & \text{if } i = 0 \end{cases}$$

In this setting, we define the largest fixpoint as follows:

$$\mathit{LgFix}(x)(g) = \{b \mid \forall k : b \in Dec(k, x)(g))\}.$$

Theorem 4.1. $g(\mathit{LgFix}(x)(g)) = \emptyset$. □

Theorem 4.2. $\mathit{LgFix}(x)(g) = \mathit{LgFix}(\mathit{LgFix}(x)(g))(g)$. □

Smallest fixpoint calculation. Let $\mathsf{IncFunc}$ be the type of functions r such that $r : \{A : \mathit{finiteset}\} \rightarrow \{B : \mathit{finiteset} \mid A \cap B = \emptyset\}$. In other words, $x \cap r(x) = \emptyset$, for all finite sets x. Let $Inc(i, x)(r)$ be a recursive function that adds elements returned by the function r of type $\mathsf{IncFunc}$ to the initial set x at every step:

$$Inc(i, x)(r) = \begin{cases} Inc(i-1, x)(r) \cup r(Inc(i-1, x)(r)) & \text{if } i \neq 0; \\ x & \text{if } i = 0 \end{cases}$$

Similarly, we define the smallest fixpoint as follows:

$$\mathit{SmFix}(x)(r) = \{b \mid \exists k : b \in Inc(k, x)(r)\}.$$

Theorem 4.3. $r(\mathit{SmFix}(x)(r)) = \emptyset$. □

Theorem 4.4. $\mathit{SmFix}(x)(r) = \mathit{SmFix}(\mathit{SmFix}(x)(r))(r)$. □

4.2 Specification and Verification of Synthesis of Failsafe Tolerance

The essence of adding failsafe tolerance is to remove (from invariant) the states from where the safety specification may be violated via one or more fault transitions. This removal is in fact a smallest fixpoint calculation. We, then, compute the invariant of the failsafe program by removing the deadlock states which is in turn a largest fixpoint calculation. In this section, the variables s, s_0, s_1 range over states. Let ms be the set of states from where safety may be violated via one or more fault transitions. Formally,

$$ms : \mathit{StatePred} = SmFix(\mathit{msInit})(\mathit{RevReachStates}), \text{ where}$$
$$\mathit{msInit} : \mathit{StatePred} = \{s_0 \mid \exists\, s_1 \,:\, (s_0, s_1) \in f \;\wedge\; (s_0, s_1) \in \Sigma_{bt}\}$$
$$\mathit{RevReachStates}(rs : \mathit{StatePred}) : \mathit{StatePred} = \{s_0 \mid \exists\, s_1 : s_1 \in rs \wedge (s_0, s_1) \in f \wedge s_0 \notin rs\}.$$

Judgement 4.5. $\mathit{RevReachStates}$ is of type $\mathsf{IncFunc}$. □

Let mt be the set of program transitions whose target states are in ms or transitions that directly violate safety:

$mt : \mathit{Action} = \{(s_0, s_1) \mid (s_1 \in ms \vee (s_0, s_1) \in \Sigma_{bt})\}.$

The set of deadlock states of state predicate ds with respect to the set Z of transitions is defined as follows:

$\mathit{DeadlockStates}(Z)(ds : \mathit{StatePred}) : \mathit{StatePred} =$
$\{s_0 \mid s_0 \in ds : (\forall s_1 \mid s_1 \in ds : (s_0, s_1) \notin Z)\}.$

Judgement 4.6. $\mathit{DeadlockStates}(Z)$ is of type DecFunc. □

The invariant of a failsafe fault-tolerant program is the largest fixpoint of $S - ms$ after removing the deadlock states:

$\mathit{ConstructInvariant}(X, Z) : \mathit{StatePred} = \mathit{LgFix}(X)(\mathit{DeadlockStates}(Z))$

where X is a state predicate of type StatePred and Z is a set of transitions of type Action. The formal definition of the invariant of a failsafe program is as follows:

$S' : \mathit{StatePred} = \mathit{ConstructInvariant}(S - ms, p - mt).$

Finally, we define the set of transitions of a failsafe program by removing the transitions that violate the closure of S':

$p' : \mathit{Action} = p - mt - \{(s_0, s_1) \mid s_0 \in S' \wedge s_1 \notin S'\}.$

Theorem 4.7. $S' \subseteq S$. □

Theorem 4.8. $p'|S' \subseteq p|S'$. □

Theorem 4.9. S' is closed in p'. Formally, $\mathit{closed}(S', p')$. □

Theorem 4.10. All computations of p' that start from a state in S' are infinite. Formally,

$\mathit{DeadlockStates}(p')(S') = \emptyset.$ □

Theorem 4.11. In the presence of faults, no computation prefix of a failsafe program that starts in S' violates safety. Formally,

$\forall j : (\forall (c : \mathit{prefix}(p' \cup f, j) \mid c_0 \in S') : \forall k \mid k < j : (c_k, c_{k+1}) \notin \Sigma_{bt}).$ □

4.3 Specification and Verification of Synthesis of Nonmasking Tolerance

In order to synthesize a nonmasking program, all we need to do is adding recovery transitions that start from outside the invariant and end in the invariant. Formally,

$S' : \mathit{StatePred} = S$

$p'(T : \mathit{StatePred}, p : \mathit{Action}) : \mathit{Action} =$
$(p \mid S) \cup \{(s_0, s_1) \mid s_0 \in (T - S) \ \wedge \ s_1 \in S\}$

$p' = p'(S_p, p).$

We assume that the number of occurrences of faults in a computation is finite:

Axiom 4.12. $\forall p : (\forall c(p \cup f) : (\exists\, n \mid n \geq 0 : (\forall j \mid j \geq n : (c_j, c_{j+1}) \in p))).$ □

Theorem 4.13. $S' \subseteq S$. □

Theorem 4.14. $p'|S' \subseteq p|S'$. □

Theorem 4.15. S' is closed in p'. Formally, $\mathit{closed}(S', p')$. □

Theorem 4.16. In the presence of faults, any computation of a nonmasking program that starts from a state in the state space, eventually recovers to the invariant. Formally,

$\forall c(p' \cup f) : (\exists\, j \mid j > 0 : c_j \in S').$ □

4.4 Specification and Verification of Synthesis of Masking Tolerance

The first estimate of a masking program is a failsafe program. Synthesis of a masking program consists of a loop that keeps recalculating the set of program transitions, invariant, and fault-span of the program until it reaches a fixpoint.

Initialization. We define the initial invariant and fault-span as follows:

$S_{init} : \mathit{StatePred} = \mathit{ConstructInvariant}(S - ms, p - mt)$

$T_{init} : \mathit{StatePred} = S_p - ms.$

Theorem 4.17. $S_{init} \subseteq T_{init}$. □

Theorem 4.18. $S_{init} \subseteq S$. □

The loop invariant. Let S_2 (respectively, T_2) be an intermediate invariant (respectively, fault-span) defined as arbitrary state predicates. Let S_1 and T_1 be the recomputed invariant and fault-span in the loop. We define S_1 and T_1 in terms of state predicates S_2 and T_2 in three steps:

1. We add recovery transitions that do not violate the safety specification as follows:
   $S_2, T_2 : \mathit{StatePred}$
   $p_1 : \mathit{Action} = (p \mid S_2) \cup \{(s_0, s_1) \mid s_0 \in (T_2 - S_2) \ \wedge \ s_1 \in T_2\} - mt.$
2. Then, we recompute the largest fault-span such that faults do not violate the closure of T_1 as follows:
   $T_1 = \mathit{ConstructFaultspan}(\mathit{TReach})$, where
   $\mathit{ConstructFaultspan}(X : \mathit{StatePred}) = \mathit{LgFix}(X)(\mathit{TNClose})$
   $\mathit{TNClose}(X : \mathit{StatePred}) : \mathit{StatePred} =$
   $\{s_0 \mid \exists s_1 : s_0 \in X \ \wedge \ (s_0, s_1) \in f \ \wedge \ s_1 \notin X\}$
   $\mathit{TReach} : \mathit{StatePred} = \{s \mid s \in T_2 \wedge \mathit{reachable}(S_2, T_2, p_1, s)\}$ where
   $\mathit{reachable}(S_2, T_2, p_1, s) : \mathit{StatePred} =$
   $\exists c(p_1) : ((s \in T_2) \ \wedge \ (s = c_0) \ \wedge \ \exists j : c_j \in S_2).$
3. Since S_1 must be is a subset of T_1, we recalculate the invariant as follows:
   $S_1 : \mathit{StatePred} = \mathit{ConstructInvariant}(S_2 \cap T_1)(p_1).$

Theorem 4.19. $(S_2 \subseteq S) \Rightarrow (S_1 \subseteq S)$. □

Theorem 4.20. $S_1 \subseteq T_1$. □

Theorem 4.21. $(p_1 \mid S_2 \subseteq p \mid S_2) \Rightarrow (p_1 \mid S_1 \subseteq p \mid S_1)$. □

Theorem 4.22. $\mathit{DeadlockStates}(p_1)(S_1) = \emptyset$. □

Loop termination. We formalize the termination condition of the loop in the verification phase. More specifically, we prove that provided $(S_1 = S_2) \ \wedge \ (T_1 = T_2)$ holds, p_1 is failsafe and provides recovery from every state in fault-span.

Theorem 4.23. $(S_1 = S_2) \Rightarrow \mathit{closed}(S_1, p_1)$. □

Theorem 4.24. In the presence of faults, no computation prefix of a masking tolerant program violates safety:

$((S_1 = S_2) \ \wedge \ (T_1 = T_2)) \Rightarrow$

$\forall j : (\forall c : \mathit{prefix}(p_1 \cup f, j) \mid c_0 \in T_1 : \forall k \mid k < j : (c_k, c_{k+1}) \notin \Sigma_{bt}).$ □

Theorem 4.25. $(T_1 = T_2) \Rightarrow \mathit{closed}(T_1, p_1 \cup f)$. □

Theorem 4.26. For all states s in the fault-span T_1, there exists a computation of p_1 that starts from s and reaches the invariant S_1. Formally,

$((S_1 = S_2) \ \wedge \ (T_1 = T_2)) \Rightarrow (\forall s \mid s \in T_1 : \mathit{reachable}(S_1, T_1, p_1, s)).$ □

5 Specification and Verification of Automatic Synthesis of Multitolerance

In this section, first in Subsection 5.1, we introduce the concept of multitolerance. We also revise the synthesis problem, as we need to take *multiple* classes of faults into account. In Subsection 5.2, we first present how we generalize the definitions presented in Section 2 to make them appropriate for modeling multitolerance. Then, we present formal specification and verification of *nonmasking-masking* multitolerance. Finally, in Subsection 5.3, we present formal specification and verification of *failsafe-masking* multitolerance. We note that since addition of failsafe-nonmasking multitolerance is shown to be NP-complete [13], there does not exist a corresponding synthesis algorithm.

5.1 The Notion of Multitolerance

Let us consider the case where faults from multiple classes, say f_1 and f_2, occur in a given program computation. In [13], Kulkarni and Ebnenasir propose the requirement that the fault-tolerance provided for the class where f_1 and f_2 occur simultaneously should be equal to the minimum level of fault-tolerance provided when either f_1 or f_2 occurs. This is illustrated in the following table reiterated from [13].

Level of Fault-Tolerance	**Failsafe**	**Nonmasking**	**Masking**
Failsafe	Failsafe	No-Tolerance	Failsafe
Nonmasking	No-Tolerance	Nonmasking	Nonmasking
Masking	failsafe	Nonmasking	Masking

In order to simplify modeling of different classes of faults, we consider the union of all the classes of faults that failsafe (respectively, nonmasking and masking) is to be provided, denoted by $f_{failsafe}$ (respectively, $f_{nonmasking}$ and $f_{masking}$). We say that a program p is *multitolerant* with respect to $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$ to Σ_{bt} from S iff the following conditions hold:

1. p satisfies Σ_{bt} from S in the absence of faults.
2. p is masking $f_{masking}$-tolerant to Σ_{bt} from S.
3. p is failsafe $(f_{failsafe} \cup f_{masking})$-tolerant to Σ_{bt} from S.
4. p is nonmasking $(f_{nonmasking} \cup f_{masking})$-tolerant to Σ_{bt} from S.

Thus, we revise the synthesis problem as follows:
Revised synthesis problem. Given p, S, Σ_{bt}, $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$ such that p satisfies Σ_{bt} from S. Identify p' and S' such that:

$(C1)$ $S^{'} \subseteq S$
$(C2)$ $p^{'} \mid S^{'} \subseteq p \mid S^{'}$
$(C3)$ p' is multitolerant wrt. $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$ to Σ_{bt} from S'.

In this section, our goal is to mechanically verify the soundness of the proposed algorithms in [13] for adding failsafe-masking and nonmasking-masking multitolerance.

5.2 Specification and Verification of Synthesis of Nonmasking-Masking Multitolerance

In order to formally specify the proposed algorithms in [13], we first define $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$ of type of Action. Then, we define $f_{nonmasking\_masking}$ and $f_{failsafe\_masking}$ as follows:

$$f_{nonmasking\_masking} : Action = f_{masking} \cup f_{nonmasking}$$
$$f_{failsafe\_masking} : Action = f_{masking} \cup f_{failsafe}$$

Since our formal framework in Section 2 is developed for the case where we deal only with one class of faults, we need to generalize some of the definitions so that they are able to express the notion of multitolerance as well. In particular, we parameterize all the definitions that are somehow related to set of faults. For instance, we redefine $msInit$, $RevReachableSet$, and ms as follows:

$msInit(anyFault : Action) : StatePred =$
$\{s_0 \mid \exists\, s_1 \;:\; (s_0, s_1) \in anyFault \;\wedge\; (s_0, s_1) \in \Sigma_{bt}\}$
$RevReachStates(anyFault : Action)(rs : StatePred) : StatePred =$
$\{s_0 \mid \exists\, s_1 : s_1 \in rs \;\wedge\; (s_0, s_1) \in anyFault \;\wedge\; s_0 \notin rs\}$
$ms(anyFault : Action) : StatePred =$
$SmFix(msInit(anyFault))(RevReachStates)$

All other definitions given in sections 2 and 4 should also be redefined in the same fashion so that they are not restricted to only one class of faults.

The core of the algorithm for adding nonmasking-masking fault-tolerance is as follows. It first adds masking fault-tolerance to p with respect to the set $f_{masking}$ of faults. Then, it adds one-step recovery to p with respect to the set $f_{nonmasking}$ of faults. We formally specify the algorithm by reusing the theories presented in Section 4. Note that add_masking (respectively, add_nonmasking and add_failsafe) is the name of the imported PVS theory for the corresponding algorithm that adds masking (respectively, nonmasking and failsafe) fault-tolerance.

$S' : StatePred = add\_masking.S_1(f_{masking})$
$T_{masking} : StatePred = add\_nonmasking.S'(T_1(f_{masking}))$
$p_1 : Action = add\_masking.p_1(f_{masking})$

Now, we model the part that adds nonmasking fault-tolerance.

$p' : Action = add\_nonmasking.p'(T_{masking}, p_1(f_{masking}))$
$T' = T_{masking}$

Notice that since a nonmasking program is not required to satisfy safety, we can simply add one-step recovery regardless of type of faults. Hence, to simplify the verification of the algorithm, we formalize the multitolerant program by $p_1(f_{masking})$ and not by $p_1(f_{nonmasking\_masking})$.

In order to verify the soundness of the algorithm, we prove that the three constraints of the revised synthesis problem hold. Indeed, all the theorems stated in Section 4 for adding masking fault-tolerance hold here with respect to the set $f_{masking}$ of faults. Also, we assume that the termination condition of the algorithm for adding masking is satisfied. While we do not present the detailed formal proofs, we present the intuitive idea of proofs so that it shows the similarities between the proofs developed in [10] and the ones for adding multitolerance. In fact, after skolemization or instantiation of the class of faults, most of the proofs are similar to the corresponding theorems presented in Section 4.

Theorem 5.1. The first condition of the synthesis problem holds. Formally,
$(S_2 \subseteq S) \Rightarrow (S' \subseteq S)$.

Proof. The proof is similar to the proof of Theorem 4.20 in the sense that it involves the same sequence of PVS commands. In particular, in both theorems, we show that the largest fixpoint of S, namely S' is a subset of S. Recall that we defined S_2 in Subsection 4.4 for specifying the loop invariant. Indeed, this proof suggests an automated strategy to prove similar theorems that involve set inclusions along with fixpoint calculations. □

Theorem 5.2. The second condition of the synthesis problem holds. Formally,

$(p' \mid S_2 \subseteq p \mid S_2) \Rightarrow (p' \mid S' \subseteq p \mid S')$.

Proof. The GRIND strategy discharges the theorem. □

Theorem 5.3. In the presence of $f_{nonmasking}$ faults, any computation of nonmasking-masking multitolerant program that starts from a state in the state space, eventually recovers to the invariant. Formally,

$\forall c(p' \cup f_{nonmasking}) : (\exists\, j \mid j > 0 : c_j \in S')$.

Proof. This is the only theorem that we prove to show that p' is nonmasking with respect to $f_{nonmasking}$. We reuse the proof of Theorem 4.16. In particular, we use induction to show that after occurrence of faults (cf. Axiom 4.12) the program eventually reaches a state in the invariant. The sequence of PVS prover commands suggests an automated proof for similar theorems with the following steps: (1) the basic step is using induction on the index of states in the computation (2) applying Axiom 4.12, and (3) instantiating n with the induction step variable. □

Theorem 5.4. $S' \subseteq T'$.

proof. This theorem is another instance where we need to show that one largest fixpoint is a subset of another largest fixpoint. The proof involves induction on the steps of recursive functions that compute the fixpoints. □

Theorem 5.5. $(T' = T_2) \;\Rightarrow\; closed(T', p' \cup f_{masking})$.

Theorem 5.6. The program is closed in the invariant. Formally,

$(S' = S_2) \;\Rightarrow\; closed(S', p')$.

Proof. The GRIND strategy discharges both theorems 5.5 and 5.6. In fact, it is easy to observe that since the proof only involves a sequence of definition expansions and propositional simplifications, we can intelligently take advantage of such cases to develop automated proofs.

Theorem 5.7. For all states s in the fault-span, there exists a computation of p' that starts from s and reaches the invariant S'. Formally,

$((S' = S_2) \wedge (T' = T_2)) \;\Rightarrow \forall s \mid s \in T' : \mathit{reachable}(S', T', p', s)$

Proof. The sequence of prover commands to prove this theorem is the same as that of Theorem 4.26. The essence of proof is using induction to show that if there exists a state $s_1 \in T'$ from where S' is reachable in k steps then S' is also reachable from s_0 with $k+1$ steps where (s_0, s_1) is a program transition in p'. In fact, using induction to show reachability of a state predicate from another state predicate could be the idea to develop automated proofs for similar types of theorems. □

Theorem 5.8. All computations of nonmasking-masking program p' that start from a state in S' are infinite. Formally,

$DeadlockStates(p')(S') = \emptyset$

Proof. Similar to the proof of Theorem 4.10, we prove this theorem by simply applying Theorem 4.1 and instantiating g with $\mathit{DeadlockStates}(p')$ and x with S'. Indeed, developing an automated proof only involves identifying the type of fixpoint calculation and then applying the obvious instantiations. □

Theorem 5.9. In the presence of $f_{masking}$, no computation prefix of a nonmasking-masking program violates safety. Formally,

$((S' = S_2) \ \wedge \ (T' = T_2)) \Rightarrow$
$\forall j : (\forall c : \mathit{prefix}(p_1 \cup f_{masking}, j) \mid c_0 \in T' : \forall k \mid k < j : (c_k, c_{k+1}) \notin \Sigma_{bt})$

Proof. Similar to the proof of Theorem 4.11, after applying induction, we show that no state in ms is reachable even in the presence of $f_{masking}$. Once we show that no state in ms is reachable, we can easily show that no transition in Σ_{bt} is also reachable. This proof suggests the following idea for developing an automated proof for similar types of theorems. In order to show that a computation never violates safety, first we use induction. Then we apply a lemma to inductively prove that no computation step reaches the set of unsafe states (which is a smallest fixpoint). The proof splits in two cases for set of program transitions and set of fault transitions. Both cases can be discharged using the GRIND strategy. □

5.3 Specification and Verification of Failsafe-Masking Multitolerance

The essence of the algorithm for adding failsafe-masking fault-tolerance is as follows. First, it identifies the fault-span, such that no computation of the multitolerant program p' violates safety in the presence of $f_{failsafe\_masking}$. More specifically, the algorithm identifies the states from where safety may be violated when faults in $f_{failsafe\_masking}$ occur:

$ms : \mathit{StatePred} = ms(f_{failsafe\_masking})$
$mt : \mathit{Action} = mt(f_{failsafe\_masking})$

Then, the algorithm ensures that the multitolerant program can recover to its invariant S', when the state of the program is perturbed by $f_{masking}$:

$T_{masking} : \mathit{StatePred} = T_1(f_{masking})$
$p_1 : \mathit{Action} = \mathit{add\_masking}.p_1(f_{masking})$
$S' : \mathit{StatePred} = \mathit{add\_masking}.S_1(f_{masking})$

Finally, if faults $f_{failsafe\_masking}$ perturb the state of the program to a state s, where $s \notin T_{masking}$ then the algorithm ensures that the safety is maintained. Towards this end, we add failsafe to p_1 with respect to the set $f_{failsafe\_masking}$ of faults from $(T_{masking} - ms)$:

$T' : \mathit{StatePred} = \mathit{ConstructInvariant}(T_{masking} - ms, p_1 - mt)$
$p' : \mathit{Action} = p_1 - mt -$
$\{(s_0, s_1) \mid ((s_0, s_1) \in (p_1 - mt)) \ \wedge \ (s_0, \in T') \ \wedge \ (s_1 \notin T')\}.$

Theorem 5.10. $(S_2 \subseteq S) \Rightarrow (S' \subseteq S)$ □

Theorem 5.11. $(p' \mid S_2 \subseteq p \mid S_2) \Rightarrow (p' \mid S' \subseteq p \mid S')$ □

Theorem 5.12. $(S' = S_2) \ \Rightarrow \ \mathit{closed}(S', p')$.

Theorem 5.13. In the absence of faults, all computations of p' that start from a state in S' are infinite. Formally, $\mathit{DeadlockStates}(p')(S') = \emptyset$.

Theorem 5.14. For any state s in the fault-span there exists a computation of p_1 that starts from s and reaches the invariant, S_1. Formally,

$((S' = S_2) \ \wedge \ (T_{masking} = T_2)) \ \Rightarrow$
$\forall s \mid s \in T' : \mathit{reachable}(S', T', p', s).$

Proof. The proof of theorems 5.10 to 5.14 is the same as the proof of corresponding theorems for adding nonmasking-masking. □

Theorem 5.15. In the presence of faults, no computation prefix of a failsafe-masking multitolerant program that starts in S' violates safety. Formally,

$(T_2 \cap ms = \emptyset) \ \Rightarrow \ \forall j : (\forall c : \mathit{prefix}(p' \cup f_{failsafe\_masking}, j) :$
$(c_0 \in S') \ \Rightarrow \ \forall k \mid k < j : (c_k, c_{k+1}) \notin \Sigma_{bt}).$

Proof. The proof idea is quite similar to that of Theorem 5.9. However, since safety may be violated by two classes of faults, namely, $f_{failsafe}$ and $f_{masking}$, the proof tree requires two case analyses for each class of faults. However, both cases can be discharged similarly using the proof of Theorem 5.9. $\square$

References

1. J. Kljaich Jr., B. T. Smith, and A. S. Wojcik. Formal verification of fault tolerance using theorem-proving techniques. *IEEE Transactions on Computers*, 38(3):366 – 376, 1989.
2. P.S. Miner and S.D. Johnson. Verification of an optimized fault-tolerant clock synchronization circuit. *Third Workshop on Designing Correct Circuits (DCC)*, 1996.
3. A. Geser and P. S. Miner. A formal correctness proof of the SPIDER diagnosis protocol. In *Theorem Proving in Higher-Order Logics (TPHOLs) – Track B Proceedings*, pages 71–86, 2002.
4. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
5. H. Mantel and F. C.Gärtner. A case study in the mechanical verification of fault-tolerance. Technical Report TUD-BS-1999-08, Department of Computer Science, Darmstadt University of Technology, 1999.
6. S. Qadeer and N. Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In *IFIP International Conference on Programming Concepts and Methods (PROCOMET)*, pages 424–443, 1998.
7. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.
8. S. S. Kulkarni, J. Rushby, and N. Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. In *The 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilization (WSS)*, pages 33–40, June 1999.
9. A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *International Conference on Distributed Computing Systems*, pages 436–443, May 1998.
10. S. S. Kulkarni, B. Bonakdarpour, and A. Ebnenasir. Mechanical verification of automatic synthesis of fault-tolerant programs. In *International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)*, pages 36–52, 2004.
11. S. S. Kulkarni, B. Bonakdarpour, and A. Ebnenasir. Mechanical verification of automatic synthesis of failsafe fault-tolerance. In *Theorem Proving in Higher-Oreder Logics (TPHOLs) Emerging Trends*, 2004.
12. S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems(FTRTFT)*, pages 82–93, 2000.
13. S. S. Kulkarni and A. Ebnenasir. Automated synthesis of multitolerance. In *International Conference on Dependable Systems and Networks (DSN)*, pages 209–219, 2004.
14. S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. *International Conference on Distributed Computing Systems (ICDCS)*, 2003.
15. B. Bonakdarpour and S. S. Kulkarni. Automatic addition of fault-tolerance to real-time programs. Technical Report MSU-CSE-06-13, Department of Computer Science and Engineering, Michigan State University, 2006.
16. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
17. 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.