Mechanical Verification of Automatic Synthesis of Fault-Tolerant Programs[1]

Sandeep S. Kulkarni, Borzoo Bonakdarpour, and Ali Ebnenasir

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

Abstract. Fault-tolerance is a crucial property in many systems. Thus, mechanical verification of algorithms associated with synthesis of fault-tolerant programs is desirable to ensure their correctness. In this paper, we present the mechanized verification of algorithms that automate the addition of fault-tolerance to a given fault-intolerant program using the PVS theorem prover. By this verification, not only we prove the correctness of the synthesis algorithms, but also we guarantee that any program synthesized by these algorithms is correct by construction. Towards this end, we formally define a uniform framework for formal specification and verification of fault-tolerance that consists of abstract definitions for programs, specifications, faults, and levels of fault-tolerance, so that they are independent of platform and architecture. The essence of synthesis algorithms involves fixpoint calculations. Hence, we also develop a reusable library for fixpoint calculations on finite sets in PVS.

Keywords: Fault-tolerance, PVS, Program synthesis, Program transformation, Mechanical verification, Theorem proving, Addition of fault-tolerance

1 Introduction

Fault-tolerance is a necessity in most computer systems and, hence, one needs strong assurance of fault-tolerance properties of a given system. Mechanical verification of such systems is one way to get a strong form of assurance. The related work in the literature has focused on verification of concrete fault-tolerant programs. For example, Owre et al [1] present a survey on formal verification of a fault-tolerant digital-flight control system. Mantel and Gärtner verify the correctness of a fault-tolerant broadcast protocol [2]. Qadeer and Shankar [3] mechanically verify the self-stability property of Dijkstra's mutual exclusion token

[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.

ring algorithm [4]. Kulkarni, Rushby, and Shankar [5] verify the same algorithm by exploiting the theory of detectors and correctors [6].

While the verifications performed in [1–3, 5] enable us to gain confidence in the programs being verified, it is difficult to extend these verifications to other programs. A more general approach, therefore, is to verify algorithms that generate fault-tolerant programs.

With this motivation, in this paper, we focus on the problem of *verifying algorithms that synthesize fault-tolerant programs*. With such verification, we are guaranteed that all the programs generated by the synthesis algorithms indeed satisfy their fault-tolerance requirements. Towards this end, we verify the transformation algorithms presented by Kulkarni and Arora [7,8] using the PVS theorem prover. The algorithms in [7,8], focus on the problem of transforming a given fault-intolerant program to a fault-tolerant program. To verify these algorithms, first, we model a framework for fault-tolerance in PVS. This framework consists of definitions for programs, specifications, faults, and levels of fault-tolerance. Then, we verify that the programs synthesized by the algorithms are indeed fault-tolerant. By this verification, we ensure that any program synthesized by these algorithms is also correct by construction and, hence, there is no need to verify the individual synthesized programs.

We note that the algorithms in [7,8], are the basis for their extensions to deal with simultaneous occurrence of multiple faults from different types [9] and for synthesizing distributed programs [10, 11]. Thus, the specification and verification of transformation algorithms in [7,8] is reusable in developing specification and verification of algorithms in [9–11]. Since fixpoint calculation is at the heart of the synthesis algorithms, we also develop a library for fixpoint calculations on *finite sets* in PVS. This library is reusable for other purposes that involves fixpoint calculations as well [2].

Contributions of the paper. The contributions of this paper are as follows: (1) We verify the correctness of the synthesis algorithms in [7,8]. Thus, not only we ensure their correctness but also we guarantee that any program synthesized by the algorithms is also correct by construction. (2) We provide a foundation for formal specification and verification of later research work that are extensions of [7, 8]. (3) We develop a reusable library in PVS for fixpoint calculations on finite sets.

Organization of the paper. The organization of the paper is as follows: We provide the formal definitions of programs, specifications, faults, and fault-tolerance in Section 2. Using these definitions, we formally state the problem of mechanical verification of synthesis of fault-tolerant programs in Section 3. In Section 4, first, we develop a theory for fixpoint calculations on finite sets. Then, based on the definitions in Section 2 and our fixpoint calculation library, we formally specify the synthesis algorithms proposed in [7,8] in PVS. In Section 5, we present the verification of algorithms. Finally, we make concluding remarks and discuss future work in Section 7.

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

2 Modeling a Fault-Tolerance Framework

In this section, we give formal definitions for programs, specifications, faults, and fault-tolerance. The programs are specified in terms of their state space and their transitions. The definition of specifications is adapted from Alpern and Schneider [12]. The definitions of faults and fault-tolerance are adapted from Arora and Gouda [13] and Kulkarni [6]. We also discuss how we model the definitions in PVS in an abstract way, so that they are independent of any particular program. We note that in describing this model, due to space limitation, we omit the proofs of certain simple judgements and lemmas.

2.1 Program

A program p is a finite set of transitions in its state space. In our framework, the notion of state is abstract. Hence, in PVS, we model state by an UNINTERPRETED TYPE [3]. Likewise, a transition is modeled as an ordered pair of states, which is also an uninterpreted type. We also assume that the number of states and transitions are finite. The state space of p, S_p, is the set of all possible states of p. In PVS, we model the state space by the finite *fullset* over states. We define the following JUDGEMENT to avoid getting repetitive type-checking proof obligations from the PVS type-checker:
Judgement 2.1: S_p has type of finite set.

We model program, p, by a subset of $S_p \times S_p$. A state predicate of p is a subset of S_p. In PVS, we model a state predicate, StatePred, as a finite set over states. The type Action denotes finite sets of transitions. A state predicate S is closed in the program p iff for all transitions (s_0, s_1) in p, if $s_0 \in S$ then $s_1 \in S$. Hence, we define closure 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 of p iff any pair of two consecutive states is a transition in p. We formalize this by a DEPENDENT TYPE[4] as follows:

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

where $sequence[state] : \mathbb{N} \to state$ and p is any finite set of type Action. A computation prefix is a finite sequence of states, where the first j steps are transitions in the given program:

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

[3] Uninterpreted types support abstraction by providing a means of introducing a type with a minimum of assumptions on the type and imposing almost no constraints on an implementation of the specification [14].

[4] In PVS specification language, a type may be defined in terms of an earlier defined type [14].

We deliberately model computation prefixes by infinite sequences of which only a finite part is used. This is due to the fact that using finite sequences in PVS is not very convenient and the type checker generates several proof obligations whenever finite sequences are used.

The projection of program p on state predicate S consists of transitions of p that start in S and end in S, denoted as $p \mid S$. Similar to the notion of program, we model projection of p on S by a finite set of transitions: $p \mid S = \{(s_0, s_1) \mid (s_0, s_1) \in p \ \wedge \ (s_0, s_1 \in S)\}$.

2.2 Specification

The specification consists of a safety specification and a liveness specification. The safety specification is specified as a set of *bad transitions*. Thus, for program p, its safety specification is a subset of $S_p \times S_p$. Hence, we can model the safety specification by a finite set of transitions, called *spec*. We explain the liveness issue in Section 2.3.

Given program p, state predicate S, and specification *spec*, we say that p satisfies its specification from S iff (1) S is closed in p, and (2) every computation of p that starts in a state where S is true, does not contain a transition in *spec*. If p does not satisfy its specification from S, we say p violates its specification. If p satisfies specification 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. A class of fault f for program p is a subset of $S_p \times S_p$. A computation of program p in presence of faults f is an infinite sequence of states where either a transition of p or a transition of f occurs at every step. Hence, we model computation of program in presence of faults as $c : Computation(p \ \cup \ f)$.

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) $S \Rightarrow T$, and (2) T is closed in $p \ \cup \ f$. Thus, we model fault-span in PVS 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 of p up to which (but not beyond which) the state of p may be perturbed by the transitions in f. Hence, we define the different levels of fault-tolerance based on the behavior of the fault-tolerant program in its fault-span.

We say that p is failsafe f-tolerant (read as fault-tolerant) to its specification from S iff two conditions hold: (1) p satisfies its specification from S, and (2) there exists T such that T is an f-span of p from S, and no prefix of a computation of $p \ \cup \ f$ that starts in T has a transition in *spec*.

We say that p is masking f-tolerant (read as fault-tolerant) to its specification from S iff the following conditions hold: (1) p satisfies its specification from S, and (2) there exists T such that T is an f-span of p from S, no prefix of a computation of $p \cup f$ that starts in T has a transition in *spec*, and every computation of $p \cup f$ that starts from a state in T contains a state of S.

In [7,8], the liveness specification is modeled implicitly. Specifically, for fail-safe fault-tolerance, the requirement is that the fault-tolerant program does not *deadlock* in the absence of faults. And, for masking fault-tolerance, the requirement is that the fault-tolerant program does not deadlock even in the presence of faults. A program deadlocks in state s_0 iff $\forall s_1 \mid s_1 \in S : (s_0, s_1) \notin p$.

There is an additional type of tolerance in [7,8], *nonmasking*, where after the occurrence of faults, eventually the program recovers to its invariant. However, the safety specification may be violated during recovery. We omit the discussion of nonmasking tolerance, as the algorithm for this case is straightforward; it suffices to add one step recovery from all states reached in the presence of faults. However, in [15], the author presents formal specification and verification of synthesis of nonmasking fault-tolerance.

3 Problem Statement

In this section, we recall (from [7,8]) the problem of automatic synthesis of fault-tolerance. As described in Section 2, the fault-intolerant program p is specified in terms of its state space S_p, its transitions, p, and its invariant, S. The specification provides a set of bad transitions (that should not occur in program computation). The faults, f, are specified in terms of a finite set of transitions. Likewise, the fault-tolerant program p' is specified in terms of its state space S_p, its set of transitions, say p', its invariant, S', its specification, *spec*, and the type of fault-tolerance it provides.

The transformation problem is as follows (this definition will be instantiated in the obvious way depending upon the level of tolerance):

The Transformation Problem
Given p, S, *spec*, and f such that p satisfies *spec* from S.
Identify p' and S' such that:
$S' \subseteq S$
$(p'|S') \subseteq (p|S')$
p' is f-tolerant to *spec* from S'

We now explain the reasons behind the first two conditions briefly:

- If S' contains states that are not in S then, in the absence of faults, p' will include computations that start outside S and hence, p' contains new behaviors in the absence of faults. Therefore, we require that $S' \subseteq S$.
- 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 a new computation in the absence of faults and hence, we require that $p'|S' \subseteq p|S'$.

Soundness. An algorithm for the transformation problem is sound iff for any given input, its output, namely p' and S', satisfies the transformation problem.

Our goal is to mechanically verify that the proposed algorithms in [7,8] are indeed sound. In other words, based on the definitions in Section 2, we show that the algorithms in [7,8] satisfy the transformation problem.

4 Description and Specification of Synthesis Algorithms

In this section, we describe the synthesis algorithms in [7,8] and explain how we formally specify them in PVS. As mentioned in Section 2, we are interested in two levels of fault-tolerance: failsafe and masking. The essence of adding failsafe and masking fault-tolerance to a given fault-intolerant program is recalculation of the invariant of the fault-intolerant program which in turn involves calculating the fixpoint of a formula. More specifically, we calculate fixpoint of a given formula to (i) calculate the set of states from where safety may be violated by faults alone; (ii) remove the set of states from where closure of fault-span is violated by fault transitions, and (iii) remove deadlock states that occur in a given set of states.

The $\mu-$calculus theory of the PVS prelude contains general definitions of the standard fixpoint calculation, however, it is not convenient to use that theory in the context of our problem. This is due to the fact that this library focuses on infinite sets and is not specialized to account for the properties of functions used in the synthesis of fault-tolerant programs. By contrast, we find that by customizing the theory to the properties of functions used in the synthesis of fault-tolerant programs, we can simplify the verification of the synthesis algorithms. Hence, in Section 4.1, we develop a theory for fixpoint calculations on *finite sets* and we verify it in Section 5.1. This theory is expected to be reusable for other formalizations that involve fixpoint calculations on finite sets. Based on the definitions in Section 4.1, we model the synthesis algorithms for addition of failsafe and masking tolerance in sections 4.2 and 4.3 respectively.

4.1 Specification of Fixpoint Calculation for Finite Sets

In this section, we describe how we formally specify fixpoint calculation for finite sets in PVS. A fixpoint of a function $f : X \to X$ is any value $x_0 \in X$ such that $f(x_0) = x_0$. In other words, further application of f does not change its value. A function may have more than one fixpoint. The least upper bound of fixpoints is called the *smallest fixpoint* and the greatest lower bound of fixpoints is called the *largest fixpoint.* In our context, the functions whose fixpoint is calculated demonstrate certain characteristics. Hence, as described above, we focus on customizing the fixpoint theory based on these characteristics.

In the context of finite sets, domain and range of f, X, are both finite sets of finite sets. Throughout this section and in Section 5.1, the variables i, j, k range over natural numbers. The variable x is any finite set of any uninterpreted type. Variable b is any member of such finite set.

One type of functions used in synthesis of fault-tolerance is a decreasing function for which the largest fixpoint is calculated. Towards this end, we start from an *initial set* and at each step of calculation, we remove a subset of the initial set that has a certain property. Thus, the type DecFunc is the type of functions g, such that $g : \{A : \mathit{finiteset}\} \rightarrow \{B : \mathit{finiteset} \mid B \subseteq A\}$. In other words, for all finite sets x, $g(x) \subseteq x$ (cf. Figure 1-a). With such a decreasing function, we define $Dec(i, x)(g)$ to formalize the recursive behavior of the largest fixpoint calculation. $Dec(i, x)(g)$ keeps removing the elements of the initial set, x, that the function g of type DecFunc returns at every step:

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

Finally, we define the largest fixpoint as follows (cf. Figure 1-b):

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

Our goal is to prove the following property of largest fixpoint based on our definitions:

$$g(LgFix(x)(g)) = \emptyset$$

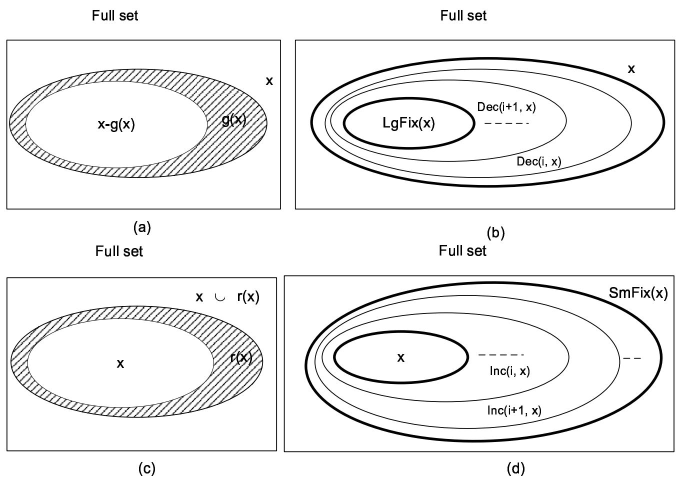

Fig. 1. (a) Relationship between g, x, and $x - g(x)$.(b) Largest fixpoint calculation. (c) Relationship between r, x, and $x \cup r(x)$. (d) Smallest fixpoint calculation

Remark. The above definition of fixpoint is somewhat non-traditional. We find that this definition assists in verification of the synthesis algorithms. For example, we apply this fixpoint calculation for removing deadlock states where

$g(x)$ denotes the deadlock states in set x. After calculating the largest fixpoint, we need to show that no deadlock states remain in the set x. Thus, we should show that $g(LgFix(x)) = \emptyset$. Moreover, if $g(LgFix(x)) = \emptyset$ then $\forall i : Dec(i, LgFix(x)) = LgFix(x)$.

The second type of fixpoint used in synthesis of fault-tolerance is an increasing function for which the smallest fixpoint is calculated. Towards this end, we start from an *initial set* and at each step, we add a set that is disjoint from the initial set. Thus, the type $\mathsf{IncFunc}$ is the type of functions r such that $r : \{A : finiteset\} \to \{B : finiteset \mid A \cap B = \emptyset\}$. In other words, for all finite sets x, $x \cap r(x) = \emptyset$ (cf. Figure 1-c). With such an increasing function, we define $Inc(i, x)(r)$ to formalize the recursive behavior of the smallest fixpoint calculation. $Inc(i, x)(r)$ keeps adding elements to the initial set, x, that the function r of type $\mathsf{IncFunc}$ returns at every step:

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

Finally, we define the smallest fixpoint as follows (cf. Figure 1-d):

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

Our goal is to prove the following property of smallest fixpoint:

$$r(SmFix(x)(r)) = \emptyset$$

4.2 Specification of the Synthesis of Failsafe Tolerance

The essence of adding failsafe tolerance is to remove the states from where safety may be violated by one or more fault transitions. We reiterate the algorithm *Add_failsafe* (from [7,8]) in Figure 2.

Throughout this section and Sections 4.3, 5.2 and 5.3, the variables x, s, s_0, s_1 range over states. The variables i, j, k, m range over natural numbers. The variable X ranges over $\mathsf{StatePred}$ and the variable Z ranges over Action. As defined in Section 3, p and p' are respectively fault-intolerant and fault-tolerant programs, S and S' are respectively invariants of fault-intolerant and fault-tolerant programs, T is fault-span, f is the finite set of faults, and *spec* is the finite set of bad transitions that represents the safety specification.

In order to construct ms, the set of states from where safety can be violated by one or more fault transitions, first, we define *msInit* as the finite set of states from where safety can be violated by a *single* fault transition. Note that $(s_0, s_1) \in spec$ means violation of the safety specification. Formally,

$$msInit : StatePred = \{s_0 \mid \exists\ s_1\ :\ (s_0, s_1) \in f\ \wedge\ (s_0, s_1) \in spec\}$$

Now, we define a function, *RevReachStates*, that calculates a state predicate from where states of another finite set, *rs*, are reachable by fault transition. Formally,

```
Add_failsafe($p, f$ : transitions, $S$ : state predicate, $spec$ : specification)
{
        $ms := smallestfixpoint(X = X \cup \{s_0 \mid (\exists s_1 :$
                        $(s_0, s_1) \in f) \wedge (s_1 \in X \vee (s_0, s_1)$ violates $spec)$ };
        $mt := \{(s_0, s_1) : ((s_1 \in ms) \vee (s_0, s_1)$ violates $spec)$ };
        $S' :=$ ConstructInvariant($S - ms, p - mt$);
        if ($S' = \{\}$) declare no failsafe $f$-tolerant program $p'$ exists;
        else $p'$ :=ConstructTransitions($p - mt, S'$)
}

ConstructInvariant($S$ : state predicate, $p$ : transitions)
// Returns the largest subset of $S$ such that computations of $p$
                        within that subset are infinite
      return $largestfixpoint(X = (X \cap S) - \{s_0 \mid (\forall s_1 : s_1 \in X : (s_0, s_1) \notin p)\}$

ConstructTransitions($p$ : transitions, $S$ : set of states)
      { return $p - \{(s_0, s_1) : s_0 \in S \wedge s_1 \notin S\}$ }
```

Fig. 2. The synthesis algorithm for adding failsafe tolerance

$$RevReachStates(rs : StatePred) : StatePred = \{s_0 \mid \exists\, s_1 : s_1 \in rs \wedge (s_0, s_1) \in f \wedge s_0 \notin rs\}$$

The following judgement helps the PVS type checker in discharging later proof obligations:
Judgement 4.1 : *RevReachStates* has type of IncFunc.

We use the definition of smallest fixpoint in Section 4.1 to define the state predicate ms. Towards this end, we instantiate the initial set with *msInit*, and the r function with *RevReachStates*:

$$ms : StatePred = SmFix(msInit)(RevReachStates)$$

Then, we define the finite set of transitions, mt, that must be removed from p. These transitions are either transitions that may lead a computation to reach a state in ms or transitions that directly violate safety:

$$mt : Action = \{(s_0, s_1) \mid (s_1 \in ms \vee (s_0, s_1) \in spec)\}$$

The algorithm *Add_failsafe* removes the set ms from the invariant of the fault-intolerant program S. However, this removal may create deadlock states. The set of deadlock states in ds of program Z is denoted as follows:

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

Judgement 4.2: $DeadlockStates(Z)$ has type of DecFunc.

We construct the invariant of the fault-tolerant program by removing the deadlock states to ensure that computations of fault-tolerant program are infinite (cf. Section 2.3). In general, we define $ConstructInvariant$ using largest fixpoint of a finite set X, that removes deadlock states of a given state predicate X:

$$ConstructInvariant(X, Z) : StatePred = LgFix(X)(DeadlockStates(Z))$$

The formal definition of the invariant of fault-tolerant program is as follows:

$$S' : StatePred = ConstructInvariant(S - ms, p - mt)$$

Finally, we construct the finite set of transitions of fault-tolerant program by removing the transitions that violate the closure of S':

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

4.3 Specification of the Synthesis of Masking Tolerance

In this section, we describe how we formally specify the addition of masking fault-tolerance to a given program p. We reiterate the algorithm *Add_masking* (from [7, 8]) in Figure 3. Note that we extensively *reuse* the formal definitions developed in Section 4.2 to model *Add_masking*.

```
Add_masking($p, f$ : transitions, $S$ : state predicate, $spec$ : specification)
{
        $ms := smallestfixpoint(X = X \cup \{s_0 \mid (\exists s_1 :$
                        $(s_0, s_1) \in f) \wedge (s_1 \in X \vee (s_0, s_1)$ violates $spec)$ };
        $mt := \{(s_0, s_1) : ((s_1 \in ms) \vee (s_0, s_1)$ violates $spec)$ };
        $S_1 :=$ ConstructInvariant($S - ms, p - mt$);
        $T_1 := true - ms$;
        repeat
                $T_2, S_2 := T_1, S_1$;
                $p_1 := p|S_2 \cup \{(s_0, s_1) : s_0 \notin S_2 \wedge s_0 \in T_2 \wedge s_1 \in T_2\} - mt$;
                $T_1 :=$ ConstructFaultSpan($T_2 - \{s : S_1$ is not reachable from $s$ in $p_1$ }, $f$);
                $S_1 :=$ ConstructInvariant($S_2 \wedge T_1, p_1$);
                if ($S_1 = \{\} \vee T_1 = \{\}$)
                        declare no masking $f$-tolerant program $p'$ exists;
                        exit
        until ($T_1 = T_2 \wedge S_1 = S_2$);

        For each state $s : s \in T_1$ :
                $Rank(s)$ = length of the shortest computation prefix of $p_1$
                        that starts from $s$ and ends in a state in $S_1$;
                $p' := \{(s_0, s_1) : ((s_0, s_1) \in p_1) \wedge (s_0 \in S_1 \vee Rank(s_0) > Rank(s_1))\}$;
                $S' := S_1$;
                $T' := T_1$
}

ConstructFaultSpan($T$ : state predicate, $f$ : transitions)
// Returns the largest subset of $T$ that is closed in $f$.
{
        return $largestfixpoint(X = (X \cap T) - \{s_0 : (\exists s_1 : (s_0, s_1) \in f \wedge s_1 \notin X)\})$
}
```

Fig. 3. The synthesis algorithm for adding masking tolerance

As mentioned in Section 2, in addition of masking fault-tolerance, the requirement for preserving the liveness properties of a program is that the fault-tolerant program does not deadlock even in the presence of faults and it should recover to the invariant after a finite number of steps while preserving safety. Hence, we assume that the number of occurrences of faults in a computation is finite by an axiom in our PVS specification. This is the only axiom used in our work.
Axiom 4.3 : $\forall p : \forall c(p \cup f) : (\exists\, n \mid n \geq 0 : (\forall j \mid j \geq n : (c_j, c_{j+1}) \in p))$.

The main difficulty in formalizing *Add_masking* algorithm is modeling the repeat-until loop (cf. Figure 3).We model the algorithm in three phases: initialization, identifying the loop invariant, and termination conditions. This loop invariant includes two properties (1) the intermediate invariant at the start of the loop is a subset of S, the invariant of the fault-intolerant program, and (2) the intersection of ms and the intermediate fault-span at the start of the loop is the empty set. Hence, in Section 5.3, to verify the algorithm, first, we show these properties for the initial guess of invariant and fault-span. Then, we show that if these properties hold at the start of an iteration, they hold at the start of the subsequent iteration as well.
Initialization: To model the part of *Add_masking* before the loop, we define S_{init} and T_{init} as follows:

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

The loop invariant: Now, we model the repeat-until loop. The value of the intermediate invariant (respectively, fault-span) at the start of the loop is S_2 (respectively, T_2). We recalculate the invariant and fault-span in the loop. Let the new values be S_1 and T_1 respectively. Now, we define S_1 and T_1 in terms of (arbitrary predicates) S_2 and T_2.

1. We define an intermediate program p_1 as follows. We require that for a transition (s_0, s_1) in p_1, the following conditions are satisfied: (1) if $s_0 \in S_2$ then $s_1 \in S_2$, (2) if $s_0 \in T_2$ then $s_1 \in T_2$. Moreover, p_1 does not contain any transition in mt. Formally

   $S_2 : StatePred$
   $T_2 : StatePred$
   $p_1 : Action = (p \mid S_2 \cup TS) - mt$, where
   $TS : StatePred = \{(s_0, s_1) \mid s_0 \notin S_2 \ \wedge \ s_0 \in T_2 \ \wedge \ s_1 \in T_2\}$

2. To formally specify construction of T_1, we first define the finite set of states from where closure of T_2 may be violated. Formally,

   $$TNClose(X : StatePred) : StatePred = \{s_0 \mid \exists s_1 : s_0 \in X \ \wedge \ (s_0, s_1) \in f \ \wedge \ s_1 \notin X\}.$$

Then, we define the finite set of states from where S_2 is reachable. Formally,

$$TReach : StatePred = \{s \mid s \in T_2 \wedge reachable(S_2, T_2, p_1, s)\} \text{ where}$$
$$reachable(S_2, T_2, p_1, s) : StatePred =$$
$$\exists c(p_1) : ((s \in T_2) \ \wedge \ (s = c_0) \ \wedge \ \exists j : c_j \in S_2).$$

We now define $ConstructFaultspan$ as the largest subset of $TReach$ that is closed in f. Formally,
$T_1 = ConstructFaultspan(TReach)$, where
$ConstructFaultspan(X : StatePred) = LgFix(X)(TNClose)$

3. Since S_1 is a subset of T_1, we model the recalculation of invariant as follows:

$$S_1 : StatePred = ConstructInvariant(S_2 \cap T_1)(p_1)$$

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

5 Verification of the Synthesis Algorithms

In this section, we verify the soundness of the synthesis algorithms based on the formal specification in Section 4.

5.1 Verification of the Fixpoint Theory

In order to verify the soundness of the synthesis algorithms, we first prove the properties of fixpoint calculations (cf. Section 4.1) in theorems 5.6 and 5.7. Before proving those theorems we present a series of intermediate lemmas as follows:
Lemma 5.1: Until the fixpoint is achieved, the cardinality of $Dec(j+1, x)$ is less than or equal to $|x| - j - 1$. Formally,
$\forall j : ((g(Dec(j, x)(g)) \neq \emptyset) \implies |(Dec(j+1, x)(g)| \leq |x| - j - 1))$
Proof. We prove this lemma by induction on j. In the base case, $j = 0$, after eliminating the quantifiers and expanding the definitions, we need to show if $g(x)$ is nonempty then $|x - g(x)| \leq |x| - 1$. We prove this by using two predefined lemmas in PVS: $\forall y, z : finiteset : ((y \subseteq z) \implies (|z - y| = |z| - |y|))$, and $\forall y : finiteset : (y \neq \emptyset \iff |y| > 0)$. After instantiations, using the facts $g(x) \subseteq x$ and $g(x) \neq \emptyset$, the GRIND[5] strategy discharges the base case. For induction step, after eliminating quantifiers, and expanding definitions, we need to prove $(g(Dec(j+1, x)(g)) \neq \emptyset \ \wedge \ |Dec(j+1, x)(g)| \leq |x| - j - 1) \implies$

[5] The GRIND strategy performs skolemization and instantiation, propositional simplification, rewriting using lemmas as rewrite rules, definition expansion, explicit case analysis according to the case structure in the goal, and does many of these steps repeatedly until no further simplification is possible [16].

$(|Dec(j+1+1,x)(g)| \leq |x| - (j+1) - 1)$. We discharge the induction step this in the same way we proved the base case. □

Lemma 5.2: If the fixpoint is reached by step j then in any subsequent steps, fixpoint will be maintained. Formally,
$\forall j : ((g(Dec(j,x)(g)) = \emptyset) \implies (\forall k \mid k \geq j : g(Dec(k,x)(g)) = \emptyset))$
Proof. After skolemization to remove the universal quantifier, we place induction on k. The base case, $k = j = 0$, is trivially true. In the induction step, we need to prove $(g(Dec(k,x)(g)) = \emptyset) \implies (g(Dec(k+1,x)(g)) = \emptyset)$. By expanding the definition of Dec in the deducing part, $Dec(k+1,x)(g) = Dec(k,x)(g) - g(Dec(k,x)(g))$, and considering the assuming part we infer that $g(Dec(k,x)(g)) = \emptyset$, therefore $g(Dec(k+1,x)(g)) = g(Dec(k,x)(g))$, which is equal to the empty set. □

Lemma 5.3: There exists a step i such that subsequent applications of g returns the empty set. Formally,
$\exists i : (\forall n \mid n \geq i : g(Dec(n,x)(g)) = \emptyset)$
Proof. First, we instantiate i with $|x|$. Then, after skolemization, we need to prove $g(Dec(n,x)(g)) = \emptyset$. Using Lemma 5.1 and instantiating j with $|x|$, we need to show two subgoals:
Subgoal 1: $|Dec(|x|+1,x)(g)| > |x| - |x| - 1$, which is trivially true.
Subgoal 2: $(g(Dec(|x|,x)(g)) = \emptyset) \implies (g(Dec(n,x)(g)) = \emptyset)$. From Lemma 5.2, we know $\forall j : (g(Dec(j,x)(g)) = \emptyset) \implies (\forall k \mid k \geq j : g(Dec(k,x)(g)) = \emptyset)$. After automatic instantiations, we need to prove $(\forall k \mid k \geq |x| : g(Dec(k,x)(g)) = \emptyset) \implies (g(Dec(n,x)(g)) = \emptyset)$. Manual instantiation of k with n discharges the lemma. □

Lemma 5.4: There exists a step j where fixpoint is achieved. Formally,
$\exists j : (\forall k \mid k \geq j : ((Dec(k,x)(g) = Dec(j,x)(g)) \ \wedge \ (g(Dec(k,x)(g)) = \emptyset)))$
Proof. Proof of the second conjunct is exactly the same as proof of Lemma 5.3, so we proceed with the proof of the first conjunct. From Lemma 5.3, we know that the existence of j such that $\forall k \mid k \geq j : g(Dec(k,x)(g)) = \emptyset$. Using Lemma 5.3 and after skolemization, we place induction on k. In the base case, $k = j = 0$, we need to show $Dec(0,x)(g) = Dec(j,x)(g)$, which is trivially true. In induction step, we need to prove:
$$\forall i \mid i \geq j : ((Dec(i,x)(g) = Dec(j,x)(g) \wedge g(Dec(i,x)(g)) = \emptyset) \implies (Dec(i+1,x)(g) = Dec(j,x)(g)))$$
We prove this by applying the rule of extensionality and expanding $Dec(i+1,x)(g)$, which is equal to $Dec(i,x)(g) - g(Dec(i,x)(g))$. As $g(Dec(i,x)(g)) = \emptyset$, $Dec(i+1,x)(g) = Dec(i,x)(g) = Dec(j,x)(g)$ and the proof is complete. □

Lemma 5.5: For some value j, $Dec(j,x)$ will reach a fixpoint, and at this step value of $Dec(j,x)$ will be the largest fixpoint. Formally,
$\exists j : (g(Dec(j,x)(g)) = \emptyset \ \wedge \ (Dec(j,x)(g) = LgFix(x)(g)))$
Proof. Similar to proof of Lemma 5.4, the proof of the first conjunct is the same

as proof of Lemma 5.3. To prove the second conjunct, first, we apply the rule of extensionality to convert the set equalities to boolean equalities. A propositional split generates two subgoals:
Subgoal 1: $\forall b \in LgFix(x)(g) : b \in Dec(j,x)(g)$. First, we expand the definition of $LgFix = \{b \mid \forall k : b \in Dec(k,x)(g)\}$ in the assuming part. Then, instantiating k with j proves the subgoal.
Subgoal 2: $\forall (b \in Dec(j,x)(g)) : b \in LgFix(x)(g)$.
To verify this subgoal, after expanding the definition of $LgFix$ and eliminating the universal quantifier by skolemization, we need to show $\forall b \in Dec(j,x)(g) : b \in Dec(k,x)(g)$. Using Lemma 5.4, we know that

$\forall i \mid i \geq j : (Dec(i,x)(g) = Dec(j,x)(g) \;\wedge\; g(Dec(i,x)(g)) = \emptyset)$.

We instantiate i with k and by propositional simplification through the GROUND[6] command, we prove this subgoal. □

Theorem 5.6: Application of function g on the largest fixpoint of a finite set returns the empty set. Formally, $g(LgFix(x)(g)) = \emptyset$
Proof. Using Lemma 5.5, the GRIND strategy completes the proof. □

Similar to largest fixpoint calculation, we prove the following theorems for verification of smallest fixpoint calculation:

Theorem 5.7: $r(SmFix(x)(r)) = \emptyset$

In order to prove Theorem 5.7, we present a series of intermediate lemmas. These lemmas and their proofs as well as proof of Theorem 5.7 are similar to the ones we presented for largest fixpoint. Therefore, for brevity, we skip the details.

5.2 Verification of the Synthesis of Failsafe Tolerance

In order to verify the soundness of *Add_failsafe* algorithm, we now prove that the synthesized program, p', satisfies the three conditions of the transformation problem stated in Section 3. More specifically, in Theorems 5.9 and 5.10, we prove the correctness of the first two conditions of the transformation problem. Then, in the remaining theorems, we show that the program synthesized by *Add_failsafe* is indeed failsafe fault-tolerant.

Observation 5.8: $S' \cap ms = \emptyset$
Proof. After expanding the definition of S', $ConstructInvariant$, and $LgFix$, we need to prove: $\forall x : (\forall k : \; x \in Dec(k, S - ms)(DeadlockStates(p - mt)) \Longrightarrow x \notin ms)$. By instantiating k with 0, propositional simplification discharges the observation. □

[6] The GROUND command invokes propositional simplification followed by arithmetic simplification and it is useful in obtaining simplified forms of the cases arising from propositional simplification [16].

Theorem 5.9: $S' \subseteq S$
Proof. Our strategy to prove this theorem is based on the fact that S' is made out of S by removing some states. After expanding the definition of S', *ConstructInvariant*, and *LgFix*, we need to prove:
$\forall k : (\forall x : (x \in Dec(k, S - ms)(DeadlockStates(p - mt)) \Longrightarrow x \in S))$.
Towards this end, first, we instantiate k with zero. Then, after expanding the definitions, we need to prove $\forall x : (x \in S - ms \Longrightarrow x \in S)$, which is trivially true. □

Theorem 5.10: $p'|S' \subseteq p|S'$
Theorem 5.11: S' is closed in p'. Formally, $closed(S', p')$
Lemma 5.12: $\forall (s_0, s_1) : ((s_0, s_1) \in f \wedge s_1 \in ms) \Longrightarrow s_0 \in ms$
Proof. The GRIND strategy discharges this lemma and theorems 5.10 and 5.11. □

Lemma 5.13: $DeadlockStates(p - mt)(S') = \emptyset$
Proof. First, we expand the definitions of S' and *ConstructInvariant*. Then, we need to prove: $DeadlockDtates(p - mt)(LgFix(S - ms)(DeadlockStates(p - mt))) = \emptyset$.
Using Theorem 5.6, we instantiate x with $LgFix(S - ms)$, and g with $DeadlockStates(p - mt)$ to complete the proof. □

Theorem 5.14: All computations of p' that start from a state in S' must be infinite. Formally, $DeadlockStates(p')(S') = \emptyset$
Proof. In Lemma 5.13, we showed that all computations of $p - mt$ that start from a state in S' are infinite. Now we need to show that all the computations of $p - mt$ after removing the transitions that violate the closure of S' are still infinite. Obviously, removal of such transitions does not have anything to do with deadlock states, because the source of a transition that violates the closure must have been removed during the removal of deadlock states. Hence, the verification is only a sequence of expansions and propositional simplifications. □

Remark. Note that Theorem 5.14 is one of the instances where formalization of the fixpoint in Section 4.1 is used. More specifically, $DeadlockStates(p')(S')$ denotes the deadlock states in S' using program p'. We repeatedly remove these deadlock states. Hence, once the fixpoint is reached, there are no deadlock states.

Lemma 5.15: In the presence of faults, no computation prefix of failsafe tolerant program that starts from a state in S', reaches a state in ms. Formally,
$\forall j : (\forall c : prefix(p' \cup f, j) \mid c_0 \in S' : \forall k \mid k < j : c_k \notin ms)$

Proof. After eliminating the universal quantifier on $c(p' \cup f)$ by skolemization, we proceed by induction on k. In the base case, $k = 0$, we need to prove $c_0 \in S' \Longrightarrow c_0 \notin ms$. The base case can be discharged using Observation 5.8. In induction step, we need to prove $(\forall n \mid n < j : (c_n, c_{n+1}) \in p' \cup f) \Longrightarrow (\forall k \mid k < j : c_k \notin ms \Rightarrow c_{k+1} \notin ms)$. From Lemma 5.12, we know that if the destination of a fault transition , (s_0, s_1), is in ms, then the source, s_0, is in ms as well. This means that if s_0 is not in ms then s_1 is not in ms either. We know that $c_k \notin ms$ and, hence, based on Lemma 5.12, $c_{k+1} \notin ms$. □

Theorem 5.16: Any prefix of any computation of failsafe tolerant program in the presence of faults that starts in S' does not violate safety. Formally,
$\forall j : \forall (c : prefix(p' \cup f), j \mid c_0 \in S') : \forall k | k < j : (c_k, c_{k+1}) \notin spec$
Proof. In Lemma 5.15, we proved that no computation prefix of $p' \cup f$ that starts from a state in S' never reaches a state in ms. In addition, p' does not contain any transition that is in *spec*. Thus, a computation prefix of $p' \cup f$ that starts from a state in S' does not contain a transition in *spec*. □

5.3 Verification of the Synthesis of Masking Tolerance

We verify the algorithm *Add_masking* based on the three phases that we modeled the algorithm in Section 4.3. More specifically, first, we show these properties for the initial guess of invariant and fault-span. Then, we show that if these properties hold at the start of an iteration, they hold at the start of the subsequent iteration as well:

Properties of initial values for the invariant and fault-span: Similar to Observation 5.8 and Theorem 5.9, we can prove the following theorems; note that these theorems show that the initial values of the invariant and fault-span satisfy the loop invariant:
Observation 5.17: $T_{init} \cap ms = \emptyset$
Theorem 5.18: $S_{init} \subseteq T_{init}$
Theorem 5.19: $S_{init} \subseteq S$

Properties of the loop invariant: Similar to the verification of *Add_failsafe*, we prove that the synthesized masking tolerant program satisfies the transformation problem by stating and proving a series of theorems and intermediate lemmas. First, we show the loop invariant, i.e., we show that if S_2 and T_2 satisfy the loop invariant then so do S_1 and T_1 (cf. Theorem 5.20). Then, we state and prove additional theorems about S_1 and T_1. Proofs of Theorems 5.20-5.23 are similar to the proofs of corresponding theorems in the verification of failsafe. Hence, we omit these proofs.
Theorem 5.20: $((T_2 \cap ms = \emptyset) \Rightarrow (T_1 \cap ms = \emptyset)) \;\wedge\; ((S_2 \subseteq S) \Rightarrow (S_1 \subseteq S))$
Theorem 5.21: $S_1 \subseteq T_1$
Theorem 5.22 : $(p_1|S_2 \subseteq p|S_2) \Rightarrow (p_1|S_1 \subseteq p|S_1)$
Theorem 5.23: $DeadlockStates(p_1)(S_1) = \emptyset$

Theorem 5.24: The recalculated fault-span is closed in f. Formally, $closed(T_1, f)$
Proof: The proof is similar to proof of Lemma 5.13. We know that $T_1 = ConstructFaultSpan(...) = LgFix(...)$. Using Theorem 5.6, in the definition of $LgFix$, we instantiate X with $TReach$, and g with $TNClose$ to complete

the proof. □

Properties at the termination of the loop: As mentioned in Section 4.3, we prove that provided $(S_1 = S_2) \wedge (T_1 = T_2)$ is true, p_1 is failsafe and provides potential recovery from every state in fault-span.

Theorem 5.25: $(S_1 = S_2) \Rightarrow \mathit{closed}(S_1, p_1)$
Proof: Based on the fact that S_2 is closed in p_1 by construction, when $S_1 = S_2$, p_1 is closed in S_1 as well. Hence, by replacing S_1 by S_2, we complete the proof. □

Theorem 5.26: Any prefix of any computation of the masking tolerant program in the presence of faults does not violate safety. Formally, $((S_1 = S_2) \wedge (T_1 = T_2)) \Rightarrow$
$\forall j : (\forall c : \mathit{prefix}(p_1 \cup f, j) \mid c_0 \in T_1 : \forall k \mid k < j : (c_k, c_{k+1}) \notin \mathit{spec})$
Proof: Proof is similar to proof of Theorem 5.16. □

Theorem 5.27: $(T_1 = T_2) \Rightarrow \mathit{closed}(T_1, p_1 \cup f)$
Proof: Based on the fact that T_2 is closed in p_1 by construction, when $T_1 = T_2$, T_1 is closed in p_1 as well. From Theorem 5.24, we also know that $\mathit{closed}(T_1, f)$. Thus, using Theorem 5.24 and by replacing T_1 by T_2, we complete the proof. □

Theorem 5.28: After termination of the loop, for any state in fault-span, T_1, there exists a computation of p_1 that starts from that state and reaches the invariant, S_1. Formally,
$((S_1 = S_2) \wedge (T_1 = T_2)) \Rightarrow (\forall s \mid s \in T_1 : \mathit{reachable}(S_1, T_1, p_1, s))$
Proof: First, we use Axiom 4.3 to show that there exists a suffix for every computation of $p_1 \cup f$ that contains no transition in f. After replacing T_1 and S_1 by T_2 and S_2 in the deducing part, we need to prove $\forall s \mid s \in T_1 : \mathit{reachable}(S_2, T_2, p_1, s)$. By expanding the definitions of T_1, $\mathit{ConstructFaultSpan}$, and LgFix respectively, we need to prove:
$\forall k : (s \in \mathit{Dec}(k, \mathit{TReach})(\mathit{TClose})) \Longrightarrow \mathit{reachable}(S_2, T_2, p_1, s)$
By instantiation of k with 0, the GRIND strategy discharges the theorem. □

Finally, the fault-tolerant program, p' is obtained by removing cycles in p_1 that occur in states in $T_1 - S_1$. Hence, we can easily extend the theorems 5.22-5.27 to show that they hold for program p' as well. Moreover, in Theorem 5.28, the fact that the shortest path from a state in T_1 to a state in S_1 is preserved, and p' does not create deadlock states can be used to show that every computation of p' eventually reaches a state in S_1. For reasons of space, we omit the discussion of these proofs.

6 Discussion

Related work. In [17], Emerson and Clarke propose an algorithm that synthesizes a program from its temporal logic specification. Since then, other algorithms have been proposed in the literature [18–21]. In the previous work prior to [7], the input to synthesis algorithms is either an automaton or temporal logics specification and any modification in the specification requires synthesizing the new program from scratch. In contrast, the algorithms in [7] reuse the fault-intolerant program to synthesize the fault-tolerant version. This reusability helps to improve the time complexity to some extent. Thus, the algorithms proposed in [7] seem to be suitable candidates for practical implementation purposes. In [22], the authors introduce a set of heuristics for synthesizing distributed fault-tolerant programs in polynomial time. Based on the heuristics, Ebnenasir and Kulkarni have developed a tool for synthesizing fault-tolerant programs [11]. Therefore, by formal verification of the algorithms in [7], we gain more confidence on their practical implementations as well.

Advantages of mechanical verification of algorithms for the synthesis of fault-tolerant algorithms. Fault-tolerant systems are often in need of strong assurance. Mechanical verification is a reliable way to ensure that the fault-tolerance requirements of a system are met. We find that verification of algorithms for synthesis of fault-tolerance is a systematic and abstract way for formal verification of fault-tolerance.

High level of abstraction. The algorithms presented in [7] make no assumptions about the properties of the system, except that they have finite state space. This high level of abstraction enables the algorithms to be applicable to synthesize both finite state hardware and software systems. Our focus on formal verification of such abstract algorithms makes it possible to extend our work to verify other algorithms that are based on the ones in [7] for any system regardless of the platform and architecture. In addition, having the developed specification and verification in this paper, we can easily verify the extensions of the algorithms in [9, 10, 22] by reusing the specification developed in this paper.

Correctness of synthesized programs. Another advantage of verifying a synthesis algorithm rather than individual fault-tolerant programs is to guarantee that any synthesized program by the algorithm is correct by construction. This advantage makes us free from verification of individual synthesized programs.

Reusability of formal proofs. Although most of the related work on formal verification of fault-tolerance [1–3, 5] provide confidence in correctness of their concerns, reusing the formal proof of one, in verification of others is not quite convenient. Manual reusability of formal proofs is the first step to develop proof strategies. As an illustration, in Section 5.3, we showed how we manually reused the formal proofs of *Add_failsafe* to verify the soundness of *Add_masking*.

The issue of completeness. A synthesis algorithm is complete iff for any given program p with the invariant S, if there exists a solution p' with invariant S' that satisfies the transformation problem then the algorithm always finds program p' and state predicate S'. In [15], we have shown that the algorithm for adding failsafe fault-tolerance in complete. However, in this paper, we focused on verification of the soundness of the algorithms due to two reasons. First, in order to show the correctness of an algorithm, we are mostly interested in verification of the soundness of the algorithm. In our context, we need to prove that the program synthesized by the algorithm is fault-tolerant indeed; i.e., the synthesized program satisfies the requirements of the transformation problem. Second, in the low atomicity model, proving the completeness of a deterministic polynomial time synthesis algorithm is irrelevant in the sense that the problem of adding fault-tolerance to distributed programs is known to be NP-Complete [7, 23]. In other words, we have to apply heuristics [10, 22] to synthesize distributed fault-tolerant programs in polynomial time. As a result of applying such heuristics, we lose the completeness of the synthesis algorithms; i.e., if the heuristics are not applicable then the synthesis algorithm fails to synthesize a fault-tolerant distributed program although there may exist a fault-tolerant program that satisfies the requirements of the transformation problem.

7 Conclusion and Future Work

In this paper, we focused on the problem of verifying transformation algorithms that generate fault-tolerant programs that are correct by construction. We considered two types of fault-tolerance properties, *failsafe* and *masking*. We would like to note that we have also verified the algorithm for synthesizing *nonmasking* fault-tolerant programs where the program recovers to states from where its specification is satisfied although safety may be violated during recovery [15].

The algorithms verified in this paper synthesize programs in the high atomicity model, where a process can read and write all variables in an atomic step. In [7,8], authors have presented a non-deterministic algorithm for designing distributed programs. We have also verified that algorithm using PVS [15].

Since we focus on verification of the transformation algorithms, we note that our results ensure that the programs synthesized using these algorithms indeed satisfy their required fault-tolerance properties. Thus, our approach is more general than verifying a particular fault-tolerant program. Also, to verify the algorithms that synthesize failsafe and masking fault-tolerant programs, we developed a fixpoint library for finite sets. This library is expected to be applicable elsewhere.

In a broader context, the verification of the algorithms considered in this paper will assist us in verifying several other transformations. For example, in [9], the authors extend the algorithms in [7, 8] to deal with multiple classes of faults. The algorithms in [7, 8] have also been used to synthesize fault-tolerant distributed programs. As an illustration, we note that the algorithms in [10, 11, 22] that are extensions of the algorithms in [7, 8] have been used to synthesize solutions for several fault-tolerant programs including, Byzantine agreement, consensus, token ring, and alternating bit protocol. Thus, the theories developed in this paper are directly applicable to verify the transformation algorithms in [9–11, 22] as well.

Our experience shows that significant number of proofs were reused. For instance, we manually reused proofs of failsafe tolerance to verify the soundness of synthesized masking tolerant programs. We expect to reuse many of the theorems and proofs in future verifications as well. Therefore, as a future work, one can develop *proof strategies* based on our experience in reusability of proofs.

References

1. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich 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.
2. Heiko Mantel and Felix 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.
3. S. Qadeer and N. Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In David Gries and Willem-Paul de Roever, editors, *IFIP International Conference on Programming Concepts and Methods (PROCOMET '98)*, pages 424–443, Shelter Island, NY, June 1998. Chapman & Hall.
4. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.
5. S. S. Kulkarni, J. Rushby, and N. Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilization (WSS'99) Austin, Texas, USA*, pages 33–40, June 1999.
6. S. S. Kulkarni. *Component-based design of fault-tolerance.* PhD thesis, Ohio State University, 1999.
7. S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 2000.
8. S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. Technical Report MSU-CSE-00-13, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, 2001.
9. S. S. Kulkarni and A. Ebnenasir. Automated synthesis of multitolerance. IEEE Conference on Dependable and Network Systems *(DSN'04)*, 2004.
10. S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. *International Conference on Distributed Computing Systems*, 2003.
11. A. Ebnenasir and S S. Kulkarni. A framework for automatic synthesis of fault-tolerance. `http://www.cse.msu.edu/~sandeep/software/Code/synthesis-framework/`.
12. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
13. 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.
14. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference, Version 2.4.* Computer Science Laboratory, SRI International, Menlo Park, CA, December 2001. URL: http://pvs.csl.sri.com/manuals.html.
15. Borzoo Bonakdarpour. Mechanical verification of automatic synthesis of fault-tolerant programs. Master's thesis, Michigan State University, 2004.

16. N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide: Version 2.4.* Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001. URL: http://pvs.csl.sri.com/manuals.html.
17. E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesis synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.
18. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, 1984.
19. A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC)*, 1998.
20. O. Kupferman and M.Y. Vardi. Synthesizing distributed systems. In *Proc. 16th IEEE Symp. on Logic in Computer Science*, July 2001.
21. 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.
22. S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of Byzantine agreement. *Symposium on Reliable Distributed Systems*, 2001.
23. S. S. Kulkarni and A. Ebnenasir. The complexity of adding failsafe fault-tolerance. *International Conference on Distributed Computing Systems*, 2002.