MECHANICAL VERIFICATION OF AUTOMATIC SYNTHESIS OF FAULT-TOLERANT PROGRAMS

By

BORZOO BONAKDARPOUR

AN ABSTRACT OF A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science

2004

Advisor: Dr. Sandeep S. Kulkarni

ABSTRACT

Mechanical Verification of Automatic Synthesis of Fault-Tolerant Programs

By

Borzoo Bonakdarpour

Fault-tolerance is a crucial property in many computer systems. Thus, mechanical verification of algorithms associated with synthesis of fault-tolerant programs is desirable to ensure their correctness. In this thesis, 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 framework for formal specification and verification of fault-tolerance that consists of definitions of programs, specifications, faults, and levels of fault-tolerance in an abstract way, so that they are independent of platform and architecture. We also address the problem of formal verification of automatic synthesis of multitolerance, where programs are subject to multiple classes of faults. The essence of the synthesis algorithms involves fixpoint calculations. Hence, we also develop a reusable theory for fixpoint calculations on finite sets in PVS.

MECHANICAL VERIFICATION OF AUTOMATIC SYNTHESIS OF FAULT-TOLERANT PROGRAMS

By

BORZOO BONAKDARPOUR

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science

2004

ABSTRACT

Mechanical Verification of Automatic Synthesis of Fault-Tolerant Programs

By

Borzoo Bonakdarpour

Fault-tolerance is a crucial property in many computer systems. Thus, mechanical verification of algorithms associated with synthesis of fault-tolerant programs is desirable to ensure their correctness. In this thesis, 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 framework for formal specification and verification of fault-tolerance that consists of definitions of programs, specifications, faults, and levels of fault-tolerance in an abstract way, so that they are independent of platform and architecture. We also address the problem of formal verification of automatic synthesis of multitolerance, where programs are subject to multiple classes of faults. The essence of the synthesis algorithms involves fixpoint calculations. Hence, we also develop a reusable theory for fixpoint calculations on finite sets in PVS.

Dedicated to my parents in Iran and my brother for their faith, support, and encouragements. This thesis would not exist without their love.

ACKNOWLEDGEMENTS

First of all I thank my advisor, Dr. Sandeep Kulkarni, for offering me technical, financial, and moral support during the last two years. He introduced me to the area of mathematical approaches to automate program synthesis and transformation. He also introduced me to computer-assisted verification and specifically theorem proving techniques. Much of the results reported in this thesis is inspired by my discussions with him about his ideas on developing a general framework for verification of fault-tolerance properties of systems. He helped me to understand what research is and how to face and solve a new problem.

My guidance committee comprised of Dr. Sandeep Kulkarni, Dr. Betty Cheng, and Dr. Charles Ofria, and I am grateful that I had access to valuable guidance from all three of them. I express my thanks to Dr. Cheng for her critical reading and interesting questions during my defense. I express my thanks to Dr. Ofria for his valuable questions and comments on practical applications of this thesis. Also, I would like to truly thank our department's graduate director, Dr. Abdol-Hossein Esfahanian, for offering financial support through teaching assistantships for several semesters.

It has been a great pleasure to work closely with Ali Ebnenasir. He was the most accessible source for understanding the concepts of program synthesis and automatic addition of fault-tolerance. He also co-authored two papers with me. It is impossible to mention his innumerable contributions towards my work.

My special thanks go to my colleagues Laura Campbell, Mahesh Arumugam, and Sascha Konrad for their comments and proof reading of my papers. I would like to use this opportunity to thank all my friends on Michigan State University campus; because of them my stay here has been very enjoyable and memorable.

Table of Contents

List of Figures

Chapter 1

Introduction

Automated logical reasoning about computer systems, widely known as *formal methods*, has been successful in a number of domains. Proving properties of computer instructions sets is perhaps the most established application of formal methods in the resent years [1]. Another application of formal methods is in verification of critical systems, where strong assurance is required to gain more confidence in correctness of computer systems such as avionic and air traffic control systems.

For decades, mathematicians have worked on techniques for verifying that algorithms have the properties they are expected to have. While traditional mathematical techniques are valuable in this context, they are prone to human errors. Hence, many researchers have focused on computer-aided verification techniques to build frameworks for automated logical reasoning about computer systems. Thus, such computer-aided techniques for verification allow us to obtain more confidence than manual proofs.

Automated proof systems fall into two broad categories [1]. The first can be seen as an extension of testing where automated support is exploited to create tests that include all possible cases, thus providing a proof of correctness. *Model checkers* fall in this category. The second can be seen as a formalization of mathematics where logic is used to characterize mathematical reasoning, and automated formal support is used to aid the creation and checking of proofs. *Theorem provers* fall in this category. In this thesis, we focus on using theorem proving techniques to reason about properties of automatic synthesis of fault-tolerant programs.

1.1 Automated Reasoning About Fault-Tolerance

Fault-tolerance is the ability of programs to provide a desirable level of functionality in the presence of faults [2]. It 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, Rushby, Shankar, and Henke [3] propose a concrete mechanical verification for a fault-tolerant digital-flight control system. Mantel and Gärtner verify the correctness of a fault-tolerant broadcast protocol [4]. Qadeer and Shankar [5] mechanically verify the self-stability property of Dijkstra's mutual exclusion token ring algorithm [6]. Kulkarni, Rushby, and Shankar [7] verify the same algorithm by exploiting the theory of detectors and correctors [2].

While the verifications performed in [3–5, 7] 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 thesis, 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, in [8, 9], and Kulkarni and Ebnenasir in [10], using the PVS theorem prover. The algorithms in [8,9], focus on the problem of transforming a given fault-intolerant program to a fault-tolerant program. The algorithms in [10], focus on the problem of transforming a given fault-intolerant program to a multitolerant program, where the program is subject to multiple classes of faults. To verify these algorithms, we, first, 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 [8,9], are the basis for their extensions to deal with simultaneous occurrence of multiple faults from different types [10] and for distributed programs [11,12]. Thus, the verification of transformation algorithms in [8,9] is helpful in verification of algorithms in [10–12]. Since fixpoint calculation is at the heart of the synthesis algorithms, we also develop a theory for fixpoint calculations on *finite sets* in PVS. This theory is reusable for other formalizations that involve fixpoint calculations over finite sets as well.

1.2 Thesis Contributions

In this thesis, we concentrate on mechanical verification of automatic synthesis of fault-tolerant and multitolerant program. The contributions of this thesis are as follows:

- We provide a foundation and a formal framework for modeling fault-tolerance, where programs are subject to a single class of faults, and multitolerance, where programs are subject to multiple classes of faults. This framework is designed abstractly in such a way that it is independent of specific programs and platforms.

- We develop a theory in PVS for fixpoint calculations over finite sets. This theory is expected to be reusable elsewhere.

- We verify the correctness of the synthesis algorithms in [8, 9] for automatic addition of fault-tolerance, so that not only we ensure their correctness, but also we guarantee that any program synthesized by the algorithms is also correct by construction.

- We verify the correctness of the synthesis algorithms in [10] for automatic addition of multitolerance, so that not only we ensure their correctness, but also we guarantee that any program synthesized by the algorithms is also correct by construction.

1.3 Organization of the Thesis

The organization of the thesis is as follows: in Chapter 2, first, we present an introduction to PVS. Then, we provide the formal definitions of programs, specifications, faults, and fault-tolerance. Then, we formally state the problem of mechanical verification of synthesis of fault-tolerant programs. In Chapter 3, we develop and verify a theory for fixpoint calculations over finite sets. In Chapter 4, based on the formal framework developed in Chapter 2 and the fixpoint calculation theory, we formalize and verify the synthesis algorithms proposed in [8] in PVS. In Chapter 5, first, we state the transformation problem for synthesizing multitolerant programs. Then, we present how we extend our formal framework, so that it can deal with simultaneous occurrence of faults from multiple classes. Based on the extended framework, then, we mechanically verify the algorithms for synthesizing multitolerant programs. In Chapter 6, we present the related work on the problem of program synthesis and mechanical verification of fault-tolerance. In Chapter 7, we discuss about different aspects of the problem of verification of synthesis of fault-tolerant programs and we answer the questions have been raised on this work. Finally, we make concluding remarks and discuss future work in Chapter 8.

Chapter 2

Modeling a Framework for Fault-Tolerance and Problem Statement

In this chapter, we describe how we develop a general framework to model fault-tolerance [1]. More specifically, we give formal definitions of elements of a fault-tolerance framework that are programs, specifications, faults, and fault-tolerance. Based on the definition of the above elements, we formalize the synthesis algorithms in [8] in Chapter 4. Note that the definitions must be extendable, so that we can reuse them as a basis to formalize the definitions regarding multiple classes of faults and multitolerance in Chapter 5. In addition, we also discus how we model the definitions in PVS in an abstract level, so that they are independent of any particular program and platform.

In our framework, programs are specified in terms of their state space and their transitions. The definition of specifications is adapted from Alpern and Schneider [13]. The definitions of faults and fault-tolerance are adapted from Arora and Gouda [14]

[1] Appendix A contains the formal specification of this framework.

and Kulkarni [2]. We give the definition of multitolerance and its related concepts in Chapter 5 separately.

Before getting into details of formalization of fault-tolerance, we give an introduction to the PVS theorem prover. More specifically, we introduce the terminology that we effectively use throughout the thesis. The reader may skip Section 2.1, if (s)he is familiar with PVS.

2.1 An Introduction to PVS

In this section we give a brief introduction to PVS and its syntax and semantics from the PVS user manuals [15–17]. PVS (Prototyping and Verification System) provides an integrated environment for the development and analysis of formal specification and supports a range of activities involved in creating, analyzing, modifying, managing and documenting theories and proofs [16]. PVS runs on SUN 4 SPARC workstations using Solaris 2 or higher and PC systems running on Redhat Linux. PVS is implemented in Common Lisp, but it is not necessary to know Lisp to effectively use the system. The Emacs editors provide the interface to PVS.

2.1.1 The PVS Specification Language

The specification language of PVS is built on *higher-order logics* [17]. Thus, functions can take functions as arguments and return them as values, and quantifications can be applied to function variables. There is a set of built-in types and type-constructors, as well as the notion of subtypes. We effectively use the features of higher-order definitions in formalizing our framework for fault-tolerance.

Specifications are logically organized into parameterized importable *theories* and *datatypes*. In this thesis, every synthesis algorithm is modeled in a separate theory. Also, we have two more theories for formalizing our general framework for fault-tolerance and fixpoint calculations over finite sets. Now, we briefly review the elements and concepts of PVS language that we use in developing our specifications.

Types. Semantically, there are four kinds of types in PVS [17]:

- 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.

- Interpreted types are primarily a means for type expressions. Integers, Booleans, etc., are interpreted types.

- Dependent types may be defined in terms of an earlier defined type.

- Subtypes introduce a set of elements that is a subset of the set of elements in the super type.

Our PVS specifications are founded on *propositional logic*, *set theory*, and theory of *infinite sequences.* These are built-in theories along a rich set of lemmas and theorems in the PVS *prelude* that assist us in proving our theorems

2.1.2 The PVS Typechecker

The PVS typechecker analyzes theories for semantic consistency and adds semantic information to the internal representation built by the parser [16]. The type system of PVS is not algorithmically decidable; i.e., theorem proving may be required to establish the type-consistency of a PVS specification. Hence, in order to decide on type checking, PVS needs assistance from the developer. The proof obligations that need to be proved are called *Type-Correctness Conditions* (TCCs).

In many cases, when a PVS specification contains similar definitions, the typechecker may generate same proof obligations. *Judgments* provide a means for controlling this by allowing the developer to prove a proof obligation once and reuse the formal proof to discharge the same TCCs. This kind of judgements are called *constant judgements*. There exists another kind of judgments, *subtype judgements*, which is useful for discharging TCCs generated for subtypes. In this thesis, since we have no subtypes in our specifications, we only state and prove *constant judgements*.

2.1.3 The PVS Theorem Prover

PVS provides an interactive proof checker (prover) with the ability to store and replay proofs. PVS can be instructed to perform a single proof, or to rerun all the proofs in a given theory, all the proofs of all the lemmas used in a proof, or all the proofs in an entire specification. The prover maintains a *proof tree*, and it is the goal of the user to construct a proof tree which is complete, in the sense that all of the leaves are recognized as true. Each node of the proof tree is a *proof goal* that follows its offspring nodes by means of a proof step.

The PVS proof checker employs a *sequent* calculus. Each proof goal of proof tree is a *sequent*. Each sequent consists of a sequence of formulas called *antecedents* and a sequence of formulas called *consequents*. The intuitive interpretation of a sequent is that the conjunctions of the antecedents implies the disjunctions of the consequents:

$$(A_1 \ \wedge \ A_2 \ \wedge \ A_3 ...) \Longrightarrow (B_1 \ \vee \ B_2 \ \vee \ B_3 ...).$$

Now, we briefly describe some of the terminology that we use throughout this thesis for verification of our theorems:

- Since our specifications effectively contains recursive definitions, induction is our basic tool to reason about the properties of the recursive definitions.

- Skolemization is the way to remove universal quantifiers from consequents and existential quantifiers from antecedents.

- Instantiation is the way to remove existential quantifiers from consequents and universal quantifiers from antecedents. For instance, if we are to prove:

  $$(\forall x : f(x) = x + 1) \Longrightarrow (f(p) = p + 1)$$

  for some constant p, all we need to do is instantiating x with p.

- Case analysis of the current sequent splits the conjunctions in the consequent and the disjunctions in the antecedent to separate subgoals. For instance,

  $$(A \Longrightarrow B \wedge C) \quad \equiv \quad (A \Longrightarrow B) \wedge (A \Longrightarrow C).$$

- The rule of extensionality converts a set equivalence to a propositional equivalence. For instance,

  $$A \subseteq B \quad \equiv \quad \forall x : (A(x) \Rightarrow B(x)).$$

- We use propositional and arithmetic simplifications to simplify the sequent containing solvable propositional and arithmetic formulas.

- 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 performs many of these steps repeatedly until no further simplification is possible [15].

- 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 [15].

2.2 Program

A program p is a finite set of transitions in its state space. In a program-specific approach a state of a program p is obtained by assigning each variable of p a value from its domain. However, as we do not deal with a specific program, in our formal specification, we model the notion of state by an UNINTERPRETED TYPE in PVS (cf. Section 2.1.1).

Likewise, a transition is modeled as a pair of states, which is also defined as an uninterpreted type. The programs of our interest are finite-state programs. Hence, we assume that the number of states and transitions are finite. We model this by two ASSUMPTIONS in PVS to express that the types state and transition are finite. The state space of p, denoted by S_p, is the set of all possible states of p. In PVS, we model the state space by the *fullset* of the type state. We define the following JUDGEMENT to avoid getting repetitive type-checking proof obligations from the PVS type-checker (cf. Section 2.1.2):

Judgement 2.1: S_p is a finite set.

Proof: We discharge the judgement by considering the assumption that the type state is finite and applying the built-in **finite_full** lemma in the PVS prelude. In finite_full lemma, it is proved that the fullset of a finite type is a finite set.

The type Action denotes finite sets of transitions. We model program, p, by a subset of $S_p \times S_p$. This actually means p can be any constant subset of $S_p \times S_p$ and, hence, a program can be any set of transitions in its state space. A state predicate of p is a subset of S_p. In PVS, we model a state predicate, StatePred, by a finite set over states. This abstraction in definitions of program and state predicate is necessary to ensure that the verifications are correct for any program synthesized by the algorithms verified in chapters 4 and 5.

A state predicate S is closed in the program p if and only if 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:

$$\textit{closed}\ (S,p) = (\forall s_0, s_1 \mid (s_0, s_1) \in p : (s_0 \in S \ \Rightarrow\ s_1 \in S)).$$

A sequence of states, $\langle s_0, s_1, ... \rangle$, is a computation of p if and only if any pair of two consecutive states is a transition in p. 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 $sequence[state] : \mathbb{N} \to state$ and Z is any finite set from type of Action. A computation prefix is a sequence of states, where the first j steps are transitions in the given program:

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

Obviously, a computation prefix is a finite sequence of states. 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 $p|S$. Similar to the notion of program, we model projection of p on S by a finite set of transitions:

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

2.3 Specification

A specification is a set of infinite sequence of states that is suffix closed and fusion closed. Suffix closure of a set of infinite state sequences means that if a state sequence

σ is in that set then so are all the suffixes of σ. Fusion closure of a set of infinite state sequences means that if state sequences $\langle \alpha, x, \gamma \rangle$ and $\langle \beta, x, \delta \rangle$ are in that set then so are the state sequences $\langle \alpha, x, \delta \rangle$ and $\langle \beta, x, \gamma \rangle$, where α and β are finite prefixes of state sequences, γ and δ are suffixes of state sequences, and x is a program state.

Following Alpern and Schneider [13], the specification consists of the safety specification and the liveness specification. Since the specification is suffixed closed and fusion closed, it is always possible to specify the safety specification as a set of *bad transitions* [2]. Thus, for program p, we model its safety specification by an arbitrary subset of $S_p \times S_p$. Hence, we model the safety specification by a finite set of transitions, called *spec*. We explain the liveness issue in Section 2.4.

Given program p, state predicate S, and specification *spec*, we say that p satisfies its specification from S if and only if (1) S is closed in p, and (2) every computation of p that starts in a state in S, 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. In PVS, we model an invariant by an arbitrary state predicate.

2.4 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 $S_p \times S_p$. As faults have an arbitrary nature, in our formalization, we do not assume any relation between p and f. Thus, they may be disjoint, equal, or they may intersect. A computation of program p in the presence of faults f is an infinite sequence of states where either a transitions of p or a transition of f is taken at every step. Hence, a program in the presence of faults is the union of program transitions and fault transitions. Likewise, we model computation of program in the 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 if and only if the following two conditions are satisfied: (1) $S \subseteq T$, and (2) T is closed in $p \cup f$. Thus, in PVS, we model fault-span 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 *spec* from S if and only if two conditions hold: (1) p satisfies *spec* 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 contains a transition in *spec*. Intuitively, this type of fault-tolerance is only concerned about satisfying the safety specification. Failsafe fault-tolerance is essential for safety-critical systems such as nuclear reactors, high temperature boilers and etc.

We say p is nonmasking f-tolerant to *spec* from S if and only if the following conditions hold: (1) p satisfies *spec* from S, and (2) there exists T such that T is an f-span of p from S, and every computation of $p \cup f$ that starts from a state in T contains a state of S, since a nonmasking fault-tolerant program need not satisfy the safety specification in the presence of faults. Intuitively, this type of fault-tolerance allows the system to violate the safety specification while it is recovering to the normal behavior. This type of fault-tolerance is usually used in computer networks that can tolerate communication failures for a limited time.

We say that p is masking f-tolerant (read as fault-tolerant) to *spec* from S if and only if the following conditions hold: (1) p satisfies *spec* 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. Intuitively, in this type of fault-tolerance a computation of program in the presence of faults never violates the safety specification and once the state of the program is perturbed by faults, it eventually recovers to its normal behavior.

In [8], the liveness specification is modeled implicitly. Specifically, for failsafe 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 a state if and only if at that state there is no program transition to take. Formally, a program deadlocks in state s_0 if and only if $\forall s_1 \mid s_1 \in S : (s_0, s_1) \notin p$, where p is the set of program transitions and S is the program invariant.

In order to ensure that the synthesis algorithms for adding nonmasking and masking fault-tolerance to a given fault-intolerant program maintains the liveness specification, we assume that the number of occurrences of faults in a computation is finite. We express this assumption by an axiom in our PVS specification. Note that throughout the specification of all algorithms in this thesis, this is the only axiom used in formalizing fault-tolerance.

Axiom 2.2 : The number of occurrences of faults in a computation is finite. Formally,

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

Based on Axiom 2.2, we can infer that in any computation of program in the presence of faults, there exists a suffix of computation, which does not contain fault transitions. The lemma is helpful in the verification of addition of nonmasking and masking fault-tolerance in Chapter 4.

Lemma 2.3: In any computation of any program in the presence of faults, there exists a suffix of the computation that is fault-free. Formally,

$$\forall p : \forall c(p \cup f) : \exists(j) : \forall(n) : (suffix_j(c)_n, suffix_j(c)_{n+1}) \in p$$

where $suffix_j(c)$ is an infinite computation that starts from the jth state in c.

Proof. The proof is based on Axiom 2.2. After elimination of quantifiers by skolemization and expansion of definition of suffix, we need to prove:

$$\forall(n \mid n \geq j) : (c_n, c_{n+1}) \in p \Longrightarrow \forall m : (c_{m+j}, c_{m+j+1}) \in p.$$

After removing the universal quantifier in the consequent, we manually instantiate n with $j + m$. Then, a propositional simplification discharges the theorem.

Discussion. Throughout this chapter, we modeled all the elements of our formal framework abstractly. For instance, states and transitions are modeled as uninterpreted types, and program, faults, and the safety specification are modeled by arbitrary sets of transitions. Also, state predicate and, thus, program invariant are modeled by arbitrary subsets of the state space. This abstract modeling gives us the freedom to claim that *any programs synthesized by the algorithms in chapters 4 and 5 are correct by construction* and, hence, there is no need to verify the synthesized programs individually.

2.5 Problem Statement

In this section, we recall (from [8]) the problem of automatic synthesis of fault-tolerance and we formally state the problem of mechanical verification of synthesis of fault-tolerant programs. As described earlier in this chapter, the fault-intolerant program 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 is specified in terms of its state space S_p, its set of transitions, p', its invariant, S', its specification, *spec*, and the type of fault-tolerance it provides.

Now, we explain what it means for a fault-tolerant program p' to be derived from p. Note that this derivation is based on that p is obtained by adding fault-tolerance alone to p. Hence, we should be able to prove that in the absence of faults p' has the same behavior as p. Specifically, p' should satisfy *spec* from S' by simply using that p satisfies *spec* from S. To ensure this, we consider the relation between (1) the invariants S and S', and (2) the transitions p and p'.

- 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 $p'|S' \subseteq p|S'$.

Based on the above two requirements we state 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:

- $S' \subseteq S$

- $(p' \mid S') \subseteq (p \mid S')$

- p' is f-tolerant to *spec* from S'

In order to define soundness and completeness in the context of the transformation problem, we define the corresponding decision problem (Likewise, this definition will be instantiated in the obvious way depending upon the level of tolerance):

The Decision Problem

Given p, S, *spec*, and f such that p satisfies *spec* from S.

Does there exist p' and S' such that:

- $S' \subseteq S$
- $(p' \mid S') \subseteq (p \mid S')$
- p' is f-tolerant to *spec* from S'?

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

Completeness. An algorithm for the transformation problem is complete if and only if for any given input, if the answer to the decision problem is "*yes*" then algorithm always finds program p' and state predicate S'.

In Chapter 4, our goal is to mechanically verify that the proposed algorithms in [8] are indeed sound and complete using the PVS theorem prover. More specifically, based on the definitions in this chapter, we show that the algorithms in [8] satisfy the transformation problem. Also, we verify that the algorithms for adding *failsafe* and *nonmasking* fault-tolerance are complete.

In Chapter 5, we redefine the transformation problem for synthesis of multitolerant programs separately. Based on the new definition, we present a new problem statement for mechanical verification of automatic synthesis of multitolerance.

Chapter 3

A Theory for Fixpoint Calculations on Finite Sets

In this chapter, we present a theory for calculation of fixpoint of formulas over finite sets [1]. This theory is essential in the verification of the synthesis algorithms in chapters 4 and 5. For instance, 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 state predicate.

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 synthesis of fault-tolerant programs. By contrast, we find that by customizing the

[1] Appendix B contains the formal specification of this theory.

theory to the properties of functions used in synthesis of fault-tolerant programs, we can simplify the verification of the synthesis algorithms. Hence, in this chapter, we develop a theory for fixpoint calculations on *finite sets.* We also state and verify their properties. This library is expected to be reusable for other formalizations that involve fixpoint calculations on finite sets as well. Based on the definitions in this chapter, we model the synthesis algorithms for addition of fault-tolerance in chapters 4 and 5. Then, using the properties of smallest and largest fixpoints, verified in this chapter, we verify the correctness of the algorithms.

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. For instance, fixpoint of $f(x) = x^2 - 2x - 1$ is $x_0 = 1$. A function may have more than one fixpoint. For example, every x_0 in the domain of $f(x) = x$ is a 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 chapter, the variables i, j, k range over natural numbers. The variable x is any finite set of any uninterpreted type and variable b is any member of x.

The organization of this chapter is as follows: in Section 3.1, we present how to formalize and verify the properties of the largest fixpoint calculation. We describe the the same issues for the smallest fixpoint calculation in Section 3.2.

3.1 Calculating the Largest Fixpoint

As mentioned before, a function may have several fixpoints. The greatest lower bound is fixpoints is called the *largest fixpoint.* We recall that we study the concept of fixpoint calculations in the context of finite sets. In Section 3.1.1, we describe how we formalize the calculation of the largest fixpoint by the PVS specification language. Then, in Section 3.1.2 we verify the properties of the largest fixpoint.

3.1.1 Specification of the Largest Fixpoint Calculation

One type of functions used in synthesis of fault-tolerance is a decreasing function for which the largest fixpoint is calculated. Given is an *initial* finite set x and a function g that calculates a subset of x. The fixpoint calculation procedure removes the states in $g(x)$ from x and repeats this removal recursively until $g(x)$ becomes the empty set. This function may resemble a certain property. For example, it may calculate the set of deadlock states of a given state predicate. We formalize the procedure of calculating the largest fixpoint of a formula by first defining the type DecFunc for decreasing functions such as g such that:

$$g : \{A : finiteset\} \rightarrow \{B : finiteset \mid B \subseteq A\}.$$

In other words, for all finite sets x, $g(x) \subseteq x$ (cf. Figure 3.1).

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:

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

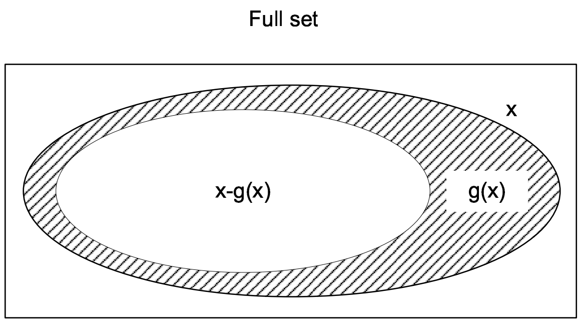

Figure 3.1: The relationship between g, x, and $x - g(x)$.

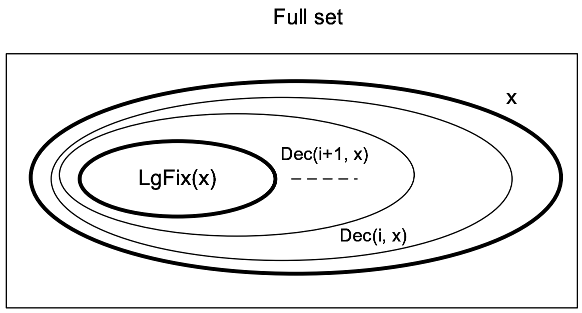

Figure 3.2: Largest fixpoint calculation.

Finally, we define the largest fixpoint as follows (cf. Figure 3.2):

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

Our goal is to prove the following properties of the largest fixpoint based on the above definitions:

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

Intuitively, the first above property means that further applications of Dec function does not change the value of the largest fixpoint. The second property means that $LgFix$ is indeed the largest fixpoint . We prove the above properties in theorems 3.7 and 3.8 in Section 3.1.2.

Remark. The above definition of fixpoint, as compared to definition of the least fixpoint in $\lambda-$calculus [18] and $\mu-$calculus, 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:

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

Moreover, if $g(LgFix(x)) = \emptyset$ then $\forall i : Dec(i, LgFix(x)) = LgFix(x)$.

3.1.2 Verification of the Properties of the Largest Fixpoint

In this section, first, we develop and prove a chain of lemmas. We use these lemmas to prove the properties of the largest fixpoint of a formula in theorems 3.7 and 3.8. We effectively use the mentioned theorems to verify the soundness and completeness of the synthesis algorithms in chapters 4 and 5.

Lemma 3.1: Any application of *Dec* function decreases the size of the initial set until it becomes the largest fixpoint. Formally,

$(j < k) \Rightarrow (Dec(k, X)(g) \subseteq Dec(j, X)(g))$

Proof. The INDUCT-AND-SIMPLIFY strategy discharges the lemma. $\square$

Lemma 3.2: Until the fixpoint is achieved, the cardinality of $Dec(j + 1, x)$ is less than or equal to $|x| - j - 1$. Formally,

$\forall j : ((g(Dec(j, x)(g)) \neq \emptyset) \quad \Longrightarrow \quad |(Dec(j + 1, x)(g)| \leq |x| - j - 1))$

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 the PVS prelude:

card_diff_subse: $\forall y, z : finiteset : ((y \subseteq z) \Longrightarrow (|z - y| = |z| - |y|))$, and nonempty_card: $\forall y : \mathit{finiteset} : (y \neq \emptyset \iff |y| > 0)$.

After instantiations, using the facts $g(x) \subseteq x$ and $g(x) \neq \emptyset$, the GRIND strategy discharges the base case. For induction step, after eliminating quantifiers, and expanding definitions, we need to prove

$$(g(\mathit{Dec}(j+1,x)(g)) \neq \emptyset \ \wedge \ |\mathit{Dec}(j+1,x)(g)| \leq |x| - j - 1) \Longrightarrow$$
$$(|\mathit{Dec}(j+1+1,x)(g)| \leq |x| - (j+1) - 1).$$

We discharge the induction step in the same way that we proved the base case. □

Lemma 3.3: If the fixpoint is reached by step j then in any subsequent steps, fixpoint will be maintained. Formally,

$$\forall j : ((g(\mathit{Dec}(j,x)(g)) = \emptyset) \Longrightarrow (\forall k \mid k \geq j : g(\mathit{Dec}(k,x)(g)) = \emptyset))$$

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:

$$(g(\mathit{Dec}(k,x)(g)) = \emptyset) \Longrightarrow (g(\mathit{Dec}(k+1,x)(g)) = \emptyset).$$

By expanding the definition of *Dec* in the consequent, $\mathit{Dec}(k+1,x)(g) = \mathit{Dec}(k,x)(g) - g(\mathit{Dec}(k,x)(g))$, and considering the antecedent we infer that $g(\mathit{Dec}(k,x)(g)) = \emptyset$, therefore $g(\mathit{Dec}(k+1,x)(g)) = g(\mathit{Dec}(k,x)(g))$, which is equal to the empty set. □

Lemma 3.4: There exists a step i such that subsequent applications of g returns the empty set. Formally,

$$\exists\ i : (\forall n \mid n \geq i : g(\mathit{Dec}(n,x)(g)) = \emptyset)$$

Proof. First, we instantiate i with $|x|$. Then, after skolemization, we need to prove $g(\mathit{Dec}(n,x)(g)) = \emptyset$. Using Lemma 3.2 and instantiating j with $|x|$, we need to show two subgoals:

Subgoal 1: $|\mathit{Dec}(|x|+1,x)(g)| > |x| - |x| - 1$, which is trivially true.

Subgoal 2: $(g(\mathit{Dec}(|x|,x)(g)) = \emptyset) \Longrightarrow (g(\mathit{Dec}(n,x)(g)) = \emptyset)$. From Lemma 3.3, 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 3.5: 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 3.4, so we proceed with the proof of the first conjunct. From Lemma 3.4, we know that the existence of j such that $\forall k \mid k \geq j : g(Dec(k,x)(g)) = \emptyset$. Using Lemma 3.4 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 3.6: For some value j, $Dec(j,x)$ will reach a fixpoint, and at this step value of $Dec(j,x)$ is equal to 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 3.5, the proof of the first conjunct is the same as proof of Lemma 3.4. To prove the second conjunct, first, we apply the rule of extensionality to convert the set equalities to boolean equalities. Then, 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 antecedent. 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 3.5, we know:

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

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

Theorem 3.7: 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 3.6, the GRIND strategy completes the proof. □

Theorem 3.8: The largest fixpoint of the largest fixpoint of a function is equal to the largest fixpoint. Formally,

$$LgFix(x)(g) = LgFix(LgFix(x)(g))(g)$$

Proof. By applying the rule of extensionality we convert the set equality to a boolean equality. Now we need to prove two subgoals:

Subgoal 1: $\forall b : (b \in LgFix(LgFix(x)(g))(g) \Rightarrow b \in LgFix(x)(g))$. After eliminating the universal quantifier, we expand the definition of the first $LgFix$ in the antecedent, which turns to $\forall k : b \in Dec(k, LgFix(x)(g))(g)$. Then after manual instantiation of k with 0, the GRIND strategy discharges the subgoal.

Subgoal 2: We need to show $\forall b : (b \in LgFix(x)(g) \Rightarrow b \in LgFix(LgFix(x)(g))(g))$. We expand the definition of the first $LgFix$ in the consequent. Now we need to prove:

$\forall b : (b \in LgFix(x)(g) \Longrightarrow \forall k : b \in Dec(k, LgFix(x)(g))(g))$.

We proceed by induction on k. In the base step, $k = 0$, we need to show:

$\forall b : (b \in LgFix(x)(g) \Longrightarrow b \in Dec(0, LgFix(x)(g))(g))$. We prove this by expanding the definition of $Dec(0, LgFix(x)(g))(g)$, which is equal to $LgFix(x)(g)$. At induction step, we need to show:

$$\forall b : ((b \in LgFix(x)(g) \ \wedge \forall j : (b \in Dec(j, LgFix(x)(g))(g)) \Longrightarrow$$
$$b \in Dec(j+1, LgFix(x)(g))(g)))$$

After eliminating the quantifiers, we expand the definition of $Dec(j+1, ...) = Dec(j, ...) - g(Dec(j, ...))$ in the consequent. From Lemma 3.3, we know:

$\forall i : (g(Dec(i, x)(g)) = \emptyset \Longrightarrow \forall k \mid k \geq i : g(Dec(k, x)(g)) = \emptyset)$.

We proceed using this lemma and we instantiate set x with $LgFix(x)(g)$ and i with 0. This generates another two subgoals:

Subgoal 2.1: $\forall k : ((g(Dec(k, LgFix(x)(g))(g)) = \emptyset) \Longrightarrow$
$(g(Dec(j, LgFix(x)(g))(g)) = \emptyset))$.

Instantiating k with j and rewriting the definitions discharges the subgoal.

Subgoal 2.2: $g(Dec(0, LgFix(x)(g))(g))) = \emptyset$. Towards this end, we expand the definition of $Dec(0, LgFix(x)(g))(g)$, which is equal to $LgFix(x)(g)$. From Theorem 3.7, we know that $g(LgFix(x)(g)) = \emptyset$. Proving this subgoal, completes the proof. $\square$

3.2 Calculating the Smallest Fixpoint

As mentioned earlier in this chapter, the least upper bound of fixpoints is called the *largest fixpoint*. In Section 3.2.1, we describe how we formalize the calculation of the largest fixpoint by the PVS specification language. Then, in Section 3.2.2 we verify the properties of the largest fixpoint.

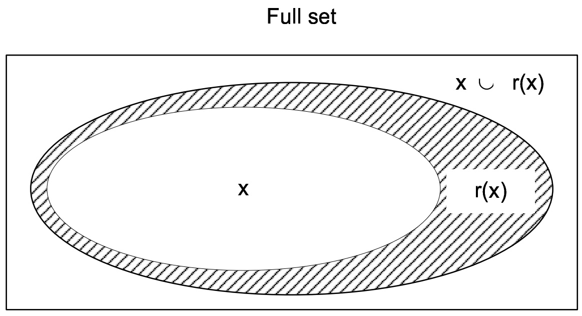

Figure 3.3: The relationship between r, x, and $x \cup r(x)$.

3.2.1 Specification of the Smallest Fixpoint Calculation

The second type of fixpoint used in fault-tolerance synthesis is an increasing function for which the smallest fixpoint is calculated. Given is an *initial* finite set x and a function r that calculates a finite set disjoint from x. Calculation of the smallest fixpoint involves adding the state in $r(x)$ to x recursively until $r(x)$ becomes the empty set. The function r may resemble a certain property. For instance, it may calculate the set of state that are reachable from x. We formalize the procedure of calculating the smallest fixpoint of a formula by first defining the type IncFunc for the increasing functions such as 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 3.3).

With such an increasing function, we define $Inc(i, x)(r)$ to formalize the recursive behavior of the smallest fixpoint calculation. $Inc(i, x)(r)$ starts adding elements to the initial set, x, that the function r of type IncFunc returns at every step:

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

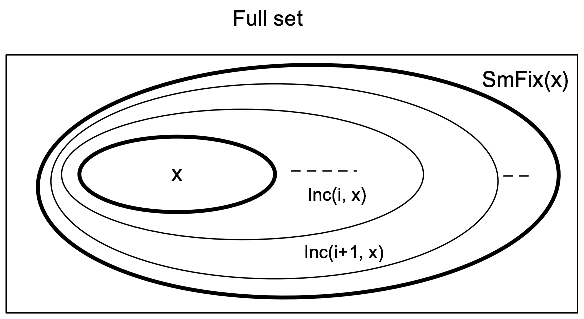

Figure 3.4: Smallest fixpoint calculation.

Finally, we define the smallest fixpoint as follows (cf. Figure 3.4):

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

Our goal is to prove the following properties of the smallest fixpoint based on the above definitions:

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

Intuitively, the first property means that $SmFix$ is indeed the smallest fixpoint. The second property means that further applications of of Inc function does not change the value of the smallest fixpoint. These properties are stated in theorems 3.15 and 3.16.

Remark. The above definition of fixpoint, as compared to definition of the least fixpoint in $\lambda-$calculus and $\mu-$calculus, is somewhat non-traditional. We find that this definition assists in verification of the synthesis algorithms. For instance, we apply the smallest fixpoint for calculating the reachable states of an initial set from where the safety specification may be directly violated, where $r(x)$ denotes the reverse reachable states of set x. After calculating the smallest fixpoint, we need to show that the smallest fixpoint contains all the states from where the safety specification can be violated in one or more steps of computation.

3.2.2 Verification of the Properties of the Smallest Fixpoint

In this section, similar to the largest fixpoint calculation, we develop and prove a chain of lemmas to verify the properties of smallest fixpoint in theorems 3.15 and 3.16. We effectively use the mentioned theorems to verify the soundness and completeness of the synthesis algorithms in chapters 4 and 5. The proof of the mentioned lemmas and theorems 3.15 and 3.16 are very similar to the corresponding lemmas and theorems in Section 3.1.2.

Lemma 3.9: Any application of Inc function decreases the size of the initial set until it becomes the smallest fixpoint. Formally,

$$(k < j) \Rightarrow (Inc(k,x)(r) \subseteq Inc(j,x)(r))$$

Proof. The INDUCT_AND_SIMLIFY discharges the lemma. $\square$

Lemma 3.10: Until the fixpoint is achieved, the cardinality of $Inc(j,x)$ is less than or equal to $\mid fullset[T] \mid - |x| - j - 1)$. Formally,

$$\forall j : (r(Inc(j,x)(r)) \neq \emptyset) \quad \Rightarrow \quad |\overline{Inc(j+1,x)(r)}| \leq \mid fullset[T] \mid - |x| - j - 1)$$

Proof. The proof is similar to the proof of Lemma 3.2 in Section 3.1.2 $\square$

Lemma 3.11: If the fixpoint is reached by step j then in any subsequent steps, fixpoint will be maintained. Formally,

$$\forall j : ((r(Inc(j,x)(r)) = \emptyset) \Rightarrow \forall k \mid k \geq j : (r(Inc(k,x)(r)) = \emptyset))$$

Proof. The proof is similar to the proof of Lemma 3.3 in Section 3.1.2. $\square$

Lemma 3.12: There exists a step i such that subsequent applications of g returns the empty set. Formally,

$$\exists\ j : (\forall (k \mid k \geq j) : (r(Inc(k,x)(r)) = \emptyset)$$

Proof. The proof is similar to the proof of Lemma 3.4 in Section 3.1.2. $\square$

Lemma 3.13: There exists a step j where fixpoint is achieved. Formally,

$$\exists\, j : (\forall k \mid k \geq j : (Inc(k,x)(r) = Inc(j,x)(r) \wedge (r(Inc(k,x)(r)) = \emptyset)))$$

Proof. The proof is similar to the proof of Lemma 3.5 in Section 3.1.2. $\square$

Lemma 3.14: For some value j, $Inc(j,x)$ will reach a fixpoint, and at this step value of $Inc(j,x)$ is equal to the smallest fixpoint. Formally,

$$\exists\, j : ((Inc(j,x)(r) = SmFix(x)(r)) \wedge (r(Inc(j,x)(r)) = \emptyset))$$

Proof. The proof is similar to the proof of Lemma 3.6 in Section 3.1.2. $\square$

Theorem 3.15: Application of function r on the largest fixpoint of a function returns the empty set. Formally,

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

Proof. The proof is similar to the proof of Theorem 3.7 in Section 3.1.2. $\square$

Theorem 3.16: The smallest fixpoint of the smallest fixpoint of a function is equal to the smallest fixpoint. Formally,

$$SmFix(x)(r) = SmFix(SmFix(x)(r))(r)$$

Proof. The proof is similar to the proof of Theorem 3.8 in Section 3.1.2. $\square$

Chapter 4

Specification and Verification of Algorithms for Synthesis of Fault-Tolerance

In this chapter, we describe the synthesis algorithms in [8] and explain how we formalize and verify them in PVS [1]. As mentioned in Chapter 2, we are interested in three levels of fault-tolerance: failsafe, nonmasking, and masking.

In this chapter, our focus is on the algorithms that synthesize fault-tolerant programs in the *high atomicity* model that are subject to *one and only one class of faults*. In this model, a program transition can read all variables as well as write all variables in one atomic step. For this model, in [8], the authors propose deterministic polynomial time algorithms to synthesize fault-tolerant programs for the three desired levels of fault-tolerance described in Chapter 2.

We will also discuss about the verification of the nondeterministic synthesis algorithm, in [8], for the programs in the *low atomicity* model where a program cannot read or write all the variables in one atomic step.

[1]Appendices C to E contain the formal specification of the algorithms.

In Section 4.1, we first describe, how to formalize and verify the algorithm for synthesis of failsafe fault-tolerant programs. Then, in Section 4.2, we present formal specification and verification of addition of nonmasking fault-tolerance. In Section 4.3, we describe how we mechanically verify the addition of masking fault-tolerance. Finally, in Section 4.4, we argue why the verification of the nondeterministic algorithm for synthesizing the programs in the low atomicity model is not quite a challenging problem.

The essence of adding failsafe and masking fault-tolerance to a given fault-intolerant program is recalculation of the invariant of 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 state predicate. The mentioned fixpoint calculations are essential parts of re-calculating the invariant and fault-span to obtain fault-tolerant programs. Once the invariant is calculated the set of program transitions can be calculated in a straight-forward manner. We effectively use the theory of fixpoint calculations developed in Chapter 3 to formalize and verify the synthesis algorithms.

Throughout this chapter, 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 2.5, p and p' are fault-intolerant and fault-tolerant programs respectively, S and S' are invariants of fault-intolerant and fault-tolerant programs respectively, 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.

4.1 Automatic Addition of Failsafe Fault-Tolerance

In this section, we present formal specification and verification of the algorithm for adding failsafe fault-tolerance proposed in [8]. Along the formal specification, we also describe the intuition behind each step of the mentioned algorithm. As for verification, we prove the correctness of both soundness and completeness of the algorithm.

4.1.1 Specification of the Synthesis of Failsafe Fault-Tolerance

Intuitively, the main feature of a failsafe program is it never violates the safety specification. However, it may not recover to its normal behavior. The essence of adding failsafe fault-tolerance to a given fault-intolerant program is to remove the states from where safety may be violated by taking one or more fault transitions. We reiterate the algorithm *Add_failsafe* (from [8, 9]) in Figure 4.1. As we describe the algorithm, we explain how to formalize it in PVS.

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 taking a *single* fault transition. Note that $(s_0, s_1) \in spec$ means (s_0, s_1) directly violates 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 through a fault transition. Formally,

```
Add_failsafe($p, f$ : transitions, $S$ : state predicate, $spec$ : specification)
{
        $ms := smallestfixpoint(X \;=\; X \cup \{s_0 \mid (\exists\, s_1 :$
                        $(s_0, s_1) \in f) \quad \wedge \quad (s_1 \in X \vee (s_0, s_1)$ 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 $p - \{(s_0, s_1) : s_0 \in S \;\;\wedge\;\; s_1 \notin S\}$ }
```

Figure 4.1: The synthesis algorithm for adding failsafe fault-tolerance

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

As mentioned earlier, we use the fixpoint theory developed in Chapter 3 to formalize the synthesis algorithms. Obviously, *RevReachStates* identifies a finite set that is disjoint from rs. Hence, *RevReachStates* has the type of IncFunc. The following judgement helps the PVS type checker to discharge later proof obligations:

Judgement 4.1 : *RevReachStates* has type of IncFunc.

Proof. The GRIND strategy simply discharges the judgement. □

We use the definition of smallest fixpoint, developed in Section 3.2.1, to define the state predicate ms. Towards this end, we consider *msInit* as the initial set, x, and *RevReachStates* as the increasing function, r:

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

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. It also removes mt from p. This removal may create deadlock states (cf. Section 2.4). The set of deadlock states in ds using program Z is defined as follows:

$$\begin{aligned} &DeadlockStates(Z)(ds : StatePred) : StatePred = \\ &\qquad \{s_0 \mid s_0 \in ds : (\forall s_1 \mid s_1 \in ds : (s_0, s_1) \notin Z)\} \end{aligned}$$

Similar to *RevReachStates*, we define the following judgement to help the PVS type checker to discharge proof obligations in the later theorems:

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

Proof. The GRIND strategy simply discharges the judgement. □

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 2.4). We define $ConstructInvariant$ using the largest fixpoint of a finite set X, that removes deadlock states of a given state predicate X:

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

We formally define define the invariant of fault-tolerant program as follows:

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

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

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

Finally, we present the definitions that we use in verification of completeness of the algorithm *Add_failsafe*. First, we define what it means for a program p'' with invariant S'' to be failsafe:

$$\begin{aligned}
&IsFailsafe(S'' : StatePred, p'' : Action) : bool = \\
&\qquad (S'' \neq \emptyset) \;\wedge\; closed(S'', p'') \;\wedge \\
&\qquad (S'' \subseteq S) \;\wedge (p'' \mid S'' \subseteq p \mid S'') \wedge \\
&\qquad (DeadlockStates(p'')(S'') = \emptyset) \wedge \\
&\qquad \forall j : (\forall c : prefix(p'' \cup f, j) \mid c_0 \in S'' : \forall k \mid k < j : (c_k, c_{k+1}) \notin spec)
\end{aligned}$$

Similar to the definition of ms, we define the set of states from where safety may be violated by taking a sequence of i fault transitions:

$$ms(i) : StatePred = Inc(i, msInit)(RevReachableStates)$$

Now, we define the set of computations that violate safety in a single step:

$$sc = \{c(f) \mid (\forall s \mid s \in msInit : ((c_0 = s) \ \wedge \ ((c_0, c_1) \in spec)))\}$$

4.1.2 Verification of the Synthesis of Failsafe Fault-Tolerance

In order to verify the correctness of the algorithm *Add_failsafe*, we prove that this algorithm is sound and complete. First, we present the proof of soundness and then we describe how to verify the completeness.

Soundness

In order to verify the soundness of *Add_failsafe*, we prove that the synthesized program p', satisfies the three conditions of the transformation problem stated in Section 2.5. More specifically, in Theorems 4.4 and 4.5, we prove that the first two conditions of the transformation problem hold. Then, in the remaining theorems, we show that the program synthesized by *Add_failsafe* is indeed failsafe fault-tolerant.

Observation 4.3: $S' \cap ms = \emptyset$

Proof. After expanding the definitions 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. □

Theorem 4.4: The first condition of the transformation problem (cf. Section 2.5) holds. Formally,

$$S' \subseteq S$$

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 definitions 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. $\square$

Theorem 4.5: The second condition of the transformation problem (cf. Section 2.5) holds. Formally,

$$p'|S' \subseteq p|S'$$

Proof. The GRIND strategy discharges the theorem. $\square$

Theorem 4.6: S' is closed in p'. Formally,

$$closed(S', p')$$

Proof. The GRIND strategy discharges the theorem. $\square$

Lemma 4.7: $\forall (s_0, s_1) : ((s_0, s_1) \in f \wedge s_1 \in ms) \Longrightarrow s_0 \in ms$

Proof. After expanding the definition of ms, we need to prove:

$$\forall (s_0, s_1) : ((s_0, s_1) \in f \ \wedge \ s_1 \in SmFix(msInit)(RevReachStates)) \Longrightarrow$$
$$s_0 \in SmFix(msInit)(RevReachStates).$$

We know that ms is the smallest fixpoint of $msInit$ by adding new reverse reachable states via fault transitions. Hence, recalculation of ms does not change its contents (cf. Theorem 3.16). Thus, if s_1 is in ms, s_0 should already be in ms as well. We proceed by applying Theorem 3.16. After instantiating x with $msInit$, and r with $RevReachStates$ in Theorem 3.16, we expand the definition of the first $SmFix$ in

$$SmFix(SmFix(msInit)(RevReachStates))(RevReachStates) =$$
$$SmFix(msInit)(RevReachStates).$$

Then, we prove the lemma by one step manual calculation of Inc to show that if s_1 is in $Inc(SmFix(...))$ then s_0 is in it as well. $\square$

Remark. Note that Lemma 4.7 is one of the instances where formalization of the fixpoint calculation in Chapter 3 is used. More specifically, using Theorem 3.16, we could show that ms contains all the states that may lead a computation to a state that violates safety and further calculation of ms does not change its contents. In other words, ms is a smallest fixpoint.

Lemma 4.8: In the absence of faults, all computations of $p - mt$ that start from a state in S' are infinite. Formally,

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

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 3.7, we instantiate x with $LgFix(S - ms)$, and g with $DeadlockStates(p - mt)$ to complete the proof. $\square$

Theorem 4.9: In the absence of faults, all computations of p' that start from a state in S' are infinite. Formally,

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

Proof. In Lemma 4.8, we showed that all computations of $p - mt$ that start from a state in S' are infinite. Now, we need to show that all 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 mechanical verification only involves a sequence of expansions and propositional simplifications. $\square$

Remark. Note that Theorem 4.9 is another instance where formalization of fixpoint calculation in Chapter 3 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 4.10: 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 quantifiers 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 4.3. 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 4.7, we know that if the terminus 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, by applying Lemma 4.7, we can infer $c_{k+1} \notin ms$. □

Theorem 4.11: Any prefix of any computation of failsafe tolerant program in the presence of faults that starts in S' does not violate safety. Formally,

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

Proof. In Lemma 4.10, 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, by definition, p' never reaches a transition that is in *spec*. Thus, a computation prefix of $p' \cup f$ that starts from a state in S' contains no transitions in *spec*. □

Theorem 4.12: For any synthesized failsafe program, there exists a fault-span. Formally,

$\exists\ T : FaultSpan(T, S', p' \cup f)$

Proof. First, we instantiate T (the fault-span) with the state space. Then, after expanding the definition of $FaultSpan$, we need to prove that S is a subset of the state space and faults does not violate the closure of the state space. Obviously, S' is a subset of state space. In addition, no transition can violate the closure of state space even in the presence of faults. Thus, after the above mentioned instantiation, the GRIND strategy discharges the theorem.

Completeness

As can be seen in Figure 4.1, algorithm *Add_failsafe* declares *failure* under two conditions. The first type of failure occurs when all the states in S are in ms as well. In other words, safety can be violated from any state of the program. The second type of failure occurs when all the states in $S - ms$ are deadlock states. With this intuition, to verify the completeness of *Add_failsafe*, we should show that if there exists a program that satisfies the transformation problem then *Add_failsafe* never declares *failure*. Let us assume this hypothetical program is p'' with the invariant S''. In theorems 4.16 and 4.19, we show that if p'' exists, *Add_failsafe* will not declare *failure* under neither of the mentioned conditions.

Lemma 4.13: For all states that are added to ms up to step i, there exists a computation of only faults that violates safety. Formally,

$\forall i : (\forall s \mid s \in ms(i) : (\exists\ c(f) \mid c_0 = s : (\exists\ k : (c_k, c_{k+1}) \in spec)))$

Proof. The proof idea for this lemma is using induction on i. In base case, $i = 0$, we should show that there exists a computation that violates safety in one step:

$$\exists\ c(f) \mid c_0 = s : (\exists\ k : (c_k, c_{k+1}) \in spec)$$

We prove the existence of such a computation by instantiating c with sc (refer to Section 4.1.1 for definition of sc). In induction step, we need to prove:

$$\begin{aligned}
&(\forall s_1 : (s_1 \in ms(j+1) \land \\
&\qquad \forall s \mid s \in ms(j) : (\exists\ c(f) \mid c_0 = s : (\exists\ k : (c_k, c_{k+1}) \in spec)))) \implies \\
&\qquad\qquad \exists\ c(f) \mid c_0 = s_1 : (\exists\ k : (c_k, c_{k+1}) \in spec))
\end{aligned}$$

The induction step intuitively means that if there exists a computation that starts from a state in ms and violates safety in j steps, there also exists a computation that starts from ms and violates safety in $j+1$ steps. To prove the induction step, we do a case analysis on the step that s_1 has been added to ms:

Case I. $\mathbf{s_1 \in ms(j)}$: This case is trivially true, because we already know that there exists a computation that starts in $ms(j)$ and violates safety.

Case II. $\mathbf{s_1 \in ms(j+1) - ms(j)}$: To prove this case, after expanding the definitions of ms and Inc, we need to prove:

$$\begin{aligned}
&(s_1 \in RevReachableStates(Inc(j, msInit)(RevReachableStates)) \land \\
&\qquad \forall s \mid s \in ms(j) : (\exists\ c(f) \mid c_0 = s : (\exists\ k : (c_k, c_{k+1}) \in spec)) \land \\
&\qquad \neg(s_1 \in Inc(j, msInit)(RevReachableStates))) \implies \\
&\qquad\qquad\qquad \exists\ c(f) \mid c_0 = s_1 : (\exists\ k : (c_k, c_{k+1}) \in spec)
\end{aligned}$$

Now, we instantiate s with s_1 to eliminate the universal quantifier in the antecedent. By removing the universal quantifier, the existential quantifier on $c(f)$ can be removed by skolemization. Now, we need to prove:

$$\begin{aligned}
&((s_2 \in Inc(j, msInit)(RevReachableStates)) \ \land\ ((s_1, s_2) \in f) \ \land\ c'_0 = s_2 \land \\
&\qquad ((c'_k, c'_{k+1}) \in spec) \ \land\ (s_1 \notin Inc(j, msInit)(RevReachableStates))) \implies \\
&\qquad\qquad \exists\ c(f) \mid c_0 = s_1 : (\exists\ k : (c_k, c_{k+1}) \in spec))
\end{aligned}$$

This implication intuitively means that if computation c' starts from s_2 and violates safety in j steps, and (s_1, s_2) is a fault transition, then there should exist a computation that starts from s_1 and violates safety in $j+1$ steps. To prove this implication, we instantiate c with $add(s_1, c')$ where add forms a computation such that $c_0 = s_1$ and $suffix_1(c) = c'$. Obviously, c is the computation that we are looking for. The mechanical proof after instantiation of c with $add(s_1, c')$ is only a chain of automatic rewriting and propositional simplifications. $\Box$

Lemma 4.14: For all states that are added to ms, there exists a computation of only faults that violates safety. Formally,

$$\forall s \mid s \in ms : (\exists\ c(f) \mid c_0 = s : (\exists\ k : (c_k, c_{k+1}) \in spec))$$

Proof. From Lemma 4.13 we know that for all i, from any state in $ms(i)$, there exists a computation that eventually violates safety. Thus, to formally verify this lemma, we only need to do a chain of automatic skolemization, definition expansion, and instantiations. At the end, the GRIND strategy discharges the lemma. $\Box$

Lemma 4.15: The invariant of any program that satisfies the transformation problem is a subset of $S - ms$. Formally,

$$\forall S'', p'' : (IsFailsafe(S'', p'') \quad \Rightarrow \quad (S'' \subseteq S - ms))$$

Proof. After skolemization, expanding the definition of *IsFailsafe*, and applying Lemma 4.14, we need to prove:

$$\begin{aligned}
&(S'' \subseteq S \ \wedge\ S'' \neq \emptyset \wedge \\
&\forall x \mid x \in ms : (\exists\ c(f) \mid c_0 = x : \exists\ k : (c_k, c_{k+1}) \in spec) \ \wedge \\
&\qquad \forall c(p'' \cup f) \mid c_0 \in S'' : \forall j : (c_j, c_{j+1}) \notin spec) \Longrightarrow \\
&\qquad\qquad (S'' \subseteq S - ms)
\end{aligned}$$

From the second and third conjuncts in the antecedent, we can infer that any state in ms cannot be in S'', as p'' is failsafe. In other words, $S'' \cap ms = \emptyset$. Besides, S'' is nonempty. Hence, $S'' \subseteq S - ms$. To mechanically prove the Lemma based on the mentioned argument, it suffices to remove the quantifiers by skolemization and perform automatic instantiation, rewriting, and propositional simplification. $\Box$

Theorem 4.16: If there exists a program that satisfies the transformation problem, algorithm *Add_failsafe* does not declare *failure* under the condition that all states in S are in ms as well. Formally,

$\exists\ S'', p'' : (IsFailsafe(S'', p'') \ \ \Rightarrow \ \ (S - ms \neq \emptyset))$

Proof. From Lemma 4.15 we know that S'' is a subset of $S - ms$. In addition, p'' is failsafe and $S'' \neq \emptyset$. Hence, $S - ms$ cannot be empty as well. In order to mechanically verify this theorem, first we remove the existential quantifier by skolemization. Then, automatic, instantiations, rewriting, and propositional simplification discharges the theorem. $\square$

Lemma 4.17: For all transitions in mt, there exists a computation of only faults that starts with that transition and violates safety. Formally,

$\forall (t_1, t_2) \mid (t_1, t_2) \in mt : (\exists\ c(f) \mid c_0 = t_1 : (\exists\ k : (c_k, c_{k+1}) \in spec))$

Proof. The transitions in mt are the ones that are either in *spec* or their terminus are in ms. To mechanically verify this lemma, first, we perform a case analysis on the type of transition. If it is in *spec*, any computation that starts with that transition is the answer. As mentioned before, one reason that a transition is in mt is that its terminus is in ms. Thus, by applying Lemma 4.14 we can discharge the lemma, because for any computation that contains a state in ms, there exists a suffix that violates safety. $\square$

Lemma 4.18: Any failsafe program that satisfy the transformation problem is a subset of $p - mt$. Formally,

$\forall S'', p'' : (IsFailsafe(S'', p'') \ \ \Rightarrow \ \ p'' \mid S'' \subseteq p - mt)$

Proof. The proof idea is similar to the proof of Lemma 4.15; we use the fact that p'' is failsafe and based on Lemma 4.17, we show that $p'' \mid S''$ must be a subset of

$p - mt$. After skolemizing, expanding the definitions, and applying Lemma 4.17, we need to prove:

$$(\forall (t_1,t_2) \mid (t_1,t_2) \in mt : (\exists\ c(f) \mid c_0 = t_1 : (\exists\ k : (c_k, c_{k+1}) \in spec)) \ \wedge$$
$$(p'' \mid S'' \subseteq p \mid S'') \wedge$$
$$\forall c(p'' \cup f) \mid c_0 \in S'' : (\forall j : (c_j, c_{j+1}) \notin spec)) \quad \Longrightarrow$$
$$(p'' \mid S'' \subseteq p - mt)$$

From the first and third conjuncts in the antecedent, we can infer that any transition in mt cannot be in p'', as p'' is failsafe. In other words, $(p'' \mid S'') \cap mt = \emptyset$. Hence, $p'' \mid S'' \subseteq p - mt$. Based on this argument, to mechanically prove the lemma, it suffices to remove the quantifiers by skolemization and perform automatic instantiation, rewriting, and propositional simplification. □

Theorem 4.19: If there exists a program that satisfies the transformation problem, algorithm *Add_failsafe* does not declare *failure* under the condition that all the states in $S - ms$ are deadlock states. Formally,

$$\exists\ S'', p'' \mid IsFailsafe(S'', p'') : (S' \neq \emptyset)$$

Proof. From Theorem 4.16 we know that if there exists a failsafe program p'', then $S - ms$ is not equal to the empty set. Hence, it suffices to prove that if such a failsafe program exists, then all the states in $S - ms$ are not deadlock states. Towards this end, first, we apply Lemma 4.18. Then, after simplifications we need to prove:

$$((S'' \subseteq S - ms) \ \wedge \ (p'' \mid S'' \subseteq p - mt) \ \wedge$$
$$(S'' \neq \emptyset) \ \wedge \ closed(S'', p'') \ \wedge \ (p'' \mid S'' \subseteq p \mid S'') \ \wedge$$
$$(DeadlockStates(p'')(S'') \neq \emptyset)) \quad \Longrightarrow$$
$$(S' \neq \emptyset).$$

After expanding the definitions of S', *ConstructInvariant*, and *LgFix*, we need to prove:

$$((S'' \subseteq S - ms) \ \wedge \ (p'' \mid S'' \subseteq p - mt) \ \wedge$$

$$(S'' \neq \emptyset) \ \wedge \ closed(S'', p'') \ \wedge \ (p'' \mid S'' \subseteq p \mid S'') \ \wedge$$
$$(DeadlockStates(p'')(S'') \neq \emptyset)) \quad \Longrightarrow$$
$$\forall k : (Dec(k, S - ms)(DeadlockStates(p - mt)) \neq \emptyset).$$

We proceed by placing induction on k. We discharge the base case, $k = 0$, using the Grind strategy. In the induction step, we need to prove:

$$(Dec(j, (S - ms)(DeadlockStates(p - mt)) \neq \emptyset) \ \wedge$$
$$((S'' \subseteq S - ms) \ \wedge \ (p'' \mid S'' \subseteq p - mt) \ \wedge$$
$$(S'' \neq \emptyset) \ \wedge \ closed(S'', p'') \ \wedge \ (p'' \mid S'' \subseteq p \mid S'') \ \wedge$$
$$(DeadlockStates(p'')(S'') \neq \emptyset)) \quad \Longrightarrow$$
$$Dec(j + 1, (S - ms)(DeadlockStates(p - mt)) \neq \emptyset$$

To prove the induction step, first, we expand the definition of $Dec(j + 1, ...)$ by one step. Then, the Ground strategy discharges the theorem. □

4.2 Automatic Addition of Nonmasking Fault-Tolerance

In this section, we present formal specification and verification of the algorithm for adding nonmasking fault-tolerance proposed in [8]. Along the formal specification, we also describe the intuition behind each step of the mentioned algorithm. As for verification, we prove the correctness of both soundness and completeness of the algorithm.

4.2.1 Specification of the Synthesis of Nonmasking Fault-Tolerance

To synthesize a nonmasking fault-tolerant program p', we ensure that from any state in the state space, p' eventually recovers to a state in the invariant. However, during this recovery, the safety specification may be violated temporarily (cf. Section 2.4). Thus, it suffices to add one step of recovery from every state in the state space to the invariant. We reiterate the algorithm *Add_nonmasking* (from [8, 9]) in Figure 4.2.

```
Add_nonmasking($p, f$ : transitions, $S$ : state predicate, $spec$ : specification)
{
    $S' := S$;
    $p' := (p|S) \cup \{(s_0, s_1) : s_0 \notin S \wedge s_1 \in S\}$
}
```

Figure 4.2: The synthesis algorithm for adding nonmasking fault-tolerance

Modeling *Add_nonmasking* in PVS is straightforward. As safety may be temporarily violated during the recovery, the algorithm does not change the invariant of the fault-intolerant program in synthesizing the fault-tolerant program. Hence, we define the invariant of nonmasking program as follows:

$$S' : StatePred = S.$$

As mentioned earlier, the set of transitions for fault-tolerant program is the original transitions of the program plus the transitions that add recovery to the invariant in one step. In PVS, we define the set of transitions for fault-tolerant program as follows:

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

4.2.2 Verification of the Synthesis of Nonmasking Fault-Tolerance

In order to verify the correctness of algorithm *Add_nonmasking*, we prove that this algorithm is sound and complete. First, we present the proof of soundness and then we describe how to verify the completeness.

Soundness

In order to verify the soundness of *Add_nonmasking*, we need to prove two properties to show that p' is nonmasking fault-tolerant. First, we should show that the fault-tolerant program satisfies *spec* in the absence of faults. Second, we should show that in the presence of faults, if faults perturb the state of program, it eventually recovers to its invariant.

Theorem 4.20: The first condition of the transformation problem (cf. Section 2.5) holds. Formally, $S' \subseteq S$

Theorem 4.21: The second condition of the transformation problem (cf. Section 2.5) holds. Formally, $p'|S' \subseteq p|S'$

Theorem 4.22: S' is closed in p'. Formally, $closed(S', p')$

Proof. The GRIND strategy discharges the theorems 4.20, 4.21, and 4.22. □

Lemma 4.23: In the absence of faults, any computation of nonmasking program that starts from any state in the state space, eventually recovers to the invariant. Formally,

$$\forall c(p') : (\exists\ j \mid j > 0 : c_j \in S')$$

Proof. After eliminating the universal quantifier by skolemizing, we need to prove:

$$(\forall n : (c_n, c_{n+1}) \in p') \implies (\exists\ j \mid j > 0 : c_j \in S').$$

Now, if we manually instantiate n with 0 and j with 1, we should prove:

$$((c_0, c_1) \in p') \implies (c_1 \in S')$$

This implication is true, as any transition in p' ends in S'. Thus, the GRIND strategy simply discharges the lemma. $\square$

Theorem 4.24: In the presence of faults, any computation of nonmasking program that starts from any state in the state space, eventually recovers to the invariant. Formally,

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

Proof. From Lemma 2.3 in Section 2.4 we know that the number of occurrences of faults is finite and, hence, there exits a suffix of the computation that is fault-free. Using this suffix, and by applying Lemma 4.23, we can show that the computation will eventually recover to the invariant. Hence, after applying the mentioned lemmas, we need to prove:

$$\begin{aligned} &(\forall n : (c_n, c_{n+1}) \in p' \cup f\ \wedge \\ &\qquad \forall m : (suffix_j(c)_m, suffix_{j+1}(c)_{m+1}) \in p') \implies \\ &\qquad\qquad \exists\ i \mid i > 0 : c_i \in S' \end{aligned}$$

where $suffix_j(c)$ is a fault-free suffix of computation c that starts from the jth state and $suffix_j(c)_m$ is the mth state in that suffix. Now, we manually instantiate i with $j+1$ and m with 0. Obviously, $c_{j+1} \in S'$, as the transition that ends at c_{j+1} is in p'. $\square$

Theorem 4.25: For any synthesized nonmasking program, there exists a fault-span. Formally,

$$\exists\ T : FaultSpan(T, S', p' \cup f)$$

Proof. The proof of this theorem is the same as proof of Theorem 4.12. $\square$

Completeness

Add_nonmasking always finds a solution to the transformation problem. In other words, the answer to the decision problem is always *yes*. Therefore, the algorithm is complete and there is no need to formally verify the completeness.

4.3 Automatic Addition of Masking Fault-Tolerance

In this section, we present formal specification and verification of the algorithm for adding masking fault-tolerance proposed in [8]. Along the formal specification, we also describe the intuition behind each step of the algorithm. As for verification, we prove that the algorithm is sound.

4.3.1 Specification of the Synthesis of Masking Fault-Tolerance

In this section, we describe how we model the addition of masking fault-tolerance to a given program p. First, we reiterate the algorithm *Add_masking* (from [8, 9]) in Figure 4.3.

As mentioned in Chapter 2, in addition of masking, the requirement for preserving the liveness properties of a program is that the fault-tolerant program does not deadlock in the presence of faults and it should recover to the invariant after a finite number of steps while preserving safety. As mentioned in Section 2.4, we assume that the number of occurrences of faults in a computation is finite. Based on this assumption, we proved that for any computation of program in the presence of faults, there exists a suffix of the computation that is free from faults (cf. Lemma 2.3). We use this lemma to show that a masking fault-tolerant program always recovers to its

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

Figure 4.3: The synthesis algorithm for adding masking fault-tolerance

invariant.

As can be seen in Figure 4.3, the algorithm contains a loop that identifies the invariant, program transitions, and fault-span of the masking fault-tolerant program. Modeling the loop is actually the challenging part of formal specification and verification of the algorithm.

We model *Add_masking* in three phases: (1) *initialization*, (2) identifying the *loop invariant*, and (3) *termination conditions.* The initialization is straightforward and is very similar to how we modeled previous algorithms. The loop invariant includes two properties (1) the intermediate invariant, S_2, 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, T_2, at the start of the loop is the empty set. Hence, 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. Modelling the termination condition is straightforward. Now, we explain the three phases of formalization in details:

Initialization: Reasonably, our first estimate of the invariant of fault-tolerant program, S', is the invariant of its failsafe fault-tolerant version. This actually makes sense because every masking tolerant is failsafe as well. Also, the first estimate of the fault-span is the state space excluding the states from where the safety may be violated, ms. 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. Our initial estimate of the fault-span has two flaws: (1) the fault-span is not necessarily closed under

$(p - mt) \cup f$ and (2) there does not necessarily exist a path from every state in the fault-span to the invariant. To resolve the mentioned flaws, in each iteration of the loop, the algorithm, recalculates the fault-span, invariant, and program transitions until the fixpoint is reached. In other words, the algorithm ensures that for all states in the fault-span, there exists a path that contains a state in the invariant, and the fault-span is closed in program in the presence of faults. The value of the intermediate invariant (respectively, fault-span) at the beginning of the loop is S_2 (respectively, T_2). After the recalculation, let the new values of the invariant and fault-span be S_1 and T_1 respectively. We define S_1 and T_1 in terms of (arbitrary predicates) S_2 and T_2 as follows:

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. As mentioned earlier, S_2 and T_2 are two arbitrary state predicates that represent the intermediate invariant and fault-span respectively. 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, first, we 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 the intermediate invariant, S_2, is reachable. Formally,

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

Now, we define *ConstructFaultspan* as the largest subset of *TReach* that is closed in f. Formally,

$T_1 = ConstructFaultspan(TReach)$, where
$ConstructFaultspan(X : StatePred) = LgFix(X)(TNClose)$

We remark that the definition of *ConstructFaultspan* is similar to the definition of *ConstructInvariant* (cf. Section 4.1.1). The major difference is in *ConstructInvariant*, we removed deadlock states recursively until we reach a fixpoint, but in *ConstructFaultspan*, we remove the states from where closure of fault-span may be violated.

3. Since S_1 should be 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 continue the loop until we achieve the largest fixpoint for S_1 and T_1. Obviously, when the fixpoint is achieved the intermediate and ultimate invariants (respectively fault-spans) are equal. 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 the fault-span.

Finally, the algorithm removes the possible cycles from the fault-span. This removal is essential, otherwise a computation may remain in a cycle and never recovers to the invariant. We define the finite set of transitions of masking fault-tolerant program, p', as follows:

$$p' = \{(s_0, s_1) \mid ((s_0, s_1) \in p_1) \wedge ((s_0 \in S_1) \vee \mathit{rank}(s_0) > \mathit{rank}(s_1))\}, \text{ where}$$
$$\mathit{rank}(s) = \min(\mathit{length}(s, p_1, S_1)), \text{ where}$$
$$\mathit{length}(s, p, S) = \{j \mid \exists\, c(p) : (s = c_0 \wedge c_j \in S_1)\}$$

4.3.2 Verification of the Synthesis of Masking Fault-Tolerance

To verify the correctness of algorithm *Add_masking*, we prove that it is sound. Our proof idea is based on the three phases of our formalization in Section 4.3.1. More specifically, we prove that when the state of the masking fault-tolerant program is perturbed by faults, the program never violates safety and it eventually recovers to the invariant. We also prove that after the loop terminates the program has all the properties of a masking fault-tolerant program. Note that the proof idea and, in some cases, the proof tree of significant number of lemmas and theorems in this section are similar or exactly the same as lemmas and theorems we proved for *Add_failsafe* and *Add_nonmasking*. We discuss the issue of proof reusability more in Chapter 7 and 8. Now, we describe the three phases of the verification in more details.

Properties of initial values for the invariant and fault-span: As mentioned earlier, in the first phase of verification, we show that the initial values of the invariant and fault-span satisfy the loop invariant. More specifically, we prove this in Observation 4.26, and theorems 4.27 and 4.28. Their proofs are similar to proofs of Observation 4.3 and Theorem 4.4 for *Add_failsafe*. Note that many of the proofs developed for the verification of *Add_failsafe* are directly applicable to prove the lemmas and theorems in this section

Observation 4.26: There exists no state in the initial fault-span from where the safety may be violated. Formally,

$T_{init} \cap ms = \emptyset$

Theorem 4.27: Initially, the invariant is a subset of the fault-span. Formally,

$S_{init} \subseteq T_{init}$

Theorem 4.28: The initial invariant is stronger than the invariant of the fault-tolerant program (the first condition of the transformation problem). Formally,

$S_{init} \subseteq S$

We remark that we do not state a theorem for the second condition of the transformation problem in the initialization part, because the algorithm does not modify the set of program transitions before the loop.

Properties of the loop invariant: We prove that the synthesized masking fault-tolerant program satisfies the transformation problem by stating and proving a series of theorems and intermediate lemmas. Note that, in the first iteration S_2 (respectively T_2) is equal to S_{init} (respectively T_{init}). Hence, in the first iteration, the loop invariant is satisfied. Moreover, at the beginning of each iteration S_1 (respectively T_1) is assigned to S_2 (respectively T_2). Hence, we can assume that S_2

and T_2 satisfy the loop invariant at the beginning of each iteration.

So as to show the loop invariant, first, we show that if S_2 and T_2 (the intermediate invariant and fault-span) satisfy the loop invariant then so do S_1 and T_1 (cf. Theorem 4.29). Then, we state and prove additional theorems about S_1 and T_1. Proofs of Theorems 4.29-4.32 are similar to the proofs of corresponding theorems in the verification of failsafe. Hence, we omit these proofs. Note that, we effectively reuse the proofs developed for verification of *Add_failsafe* in the following theorems.

Theorem 4.29: $((T_2 \cap ms = \emptyset) \Rightarrow (T_1 \cap ms = \emptyset)) \ \wedge \ ((S_2 \subseteq S) \Rightarrow (S_1 \subseteq S))$

Theorem 4.30: The invariant is always a subset of the fault-span. Formally,

$S_1 \subseteq T_1$

Theorem 4.31 : If the intermediate program and invariant satisfy the second condition of the transformation problem, so do the program and S_1. Formally,

$(p_1 \mid S_2 \subseteq p \mid S_2) \Rightarrow (p_1 \mid S_1 \subseteq p \mid S_1)$

Theorem 4.32: The recalculated invariant contains no deadlock states. Formally,

$DeadlockStates(p_1)(S_1) = \emptyset$

Theorem 4.33: The recalculated fault-span is closed in f. Formally,

$closed(T_1, f)$

Proof: The proof is similar to the proof of Lemma 4.8. We know:

$T_1 = ConstructFaultSpan(...) = LgFix(...).$

Using Theorem 3.7, in the definition of $LgFix$, we instantiate X with $TReach$, and g with $TNClose$ to complete the proof. $\square$

Remark. Note that Theorem 4.33 is another instance where formalization of fixpoint calculation in Chapter 3 is used. More specifically, $closed(T_1, f)$ denotes wether the state predicate T_1 is closed in f. We repeatedly remove the states from where the closure of T_1 may be violated. Hence, once the fixpoint is reached, there are no such states.

Properties at the termination of the loop: As mentioned in Section 4.3.1, we formalize the termination condition in the verification phase; i.e, 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 4.34: Upon the loop termination, the invariant is closed in the program. Formally,

$$(S_1 = S_2) \Rightarrow \mathit{closed}(S_1, p_1)$$

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. Thus, to formally prove the theorem, we replace S_1 by S_2 to complete the proof. $\square$

Theorem 4.35: Any prefix of any computation of the masking fault-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 the proof of Theorem 4.11. $\square$

Theorem 4.36: $(T_1 = T_2) \Rightarrow \mathit{closed}(T_1, p_1 \cup f)$

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 4.33, we also know that T_1 is closed in f. Thus, using Theorem 4.33 and by replacing T_1 by T_2, we complete the proof. $\square$

Theorem 4.37: 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 apply Lemma 2.3 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 consequent, we need to prove:

$$\forall s \mid s \in T_1 : \mathit{reachable}(S_2, T_2, p_1, s).$$

After expanding the definitions of T_1, $ConstructFaultSpan$, and $LgFix$ respectively, we need to prove:

$$\forall k : (s \in \mathit{Dec}(k, \mathit{TReach})(\mathit{TClose})) \Longrightarrow \mathit{reachable}(S_2, T_2, p_1, s)$$

After instantiating k with 0, the GRIND strategy discharges the theorem. □

Theorem 4.38: For any synthesized nonmasking program, there exists a fault-span. Formally,

$$(T_1 = T_2) \Rightarrow \exists\ T : \mathit{FaultSpan}(T, S_1, p_1 \cup f)$$

Proof: From Theorem 4.30, we know that S_1 is a subset of T_1. Also, from Theorem 4.36, we know that if $T_1 = T_2$ is true, then the fault-span is closed in program in the presence of faults. Thus, for mechanical verification, it suffices to, first, instantiate T with T_1, and then, apply theorems 4.30 and 4.36. □

4.4 Adding Fault-Tolerance in the Low Atomicity Model

The three algorithms described earlier in this chapter use the notion of program state in an abstract level. In practice, the state of a program is obtained by assigning each variable a value from its domain. In the context of synthesis of real-world programs, variables of a program and restrictions on reading and writing them matter.

For example, if a program has two integer variables x and y, $[x = 3, y = 10]$ identifies a state of this program. In this manner, a predicate over the states of a program forms a set of states, so-called a state predicates. For example, $x \geq 5 \wedge y < 20$ is a state predicate.

To synthesize distributed programs, where processes of a program run on a distributed system, one should consider the read-write restrictions of variables, as it is usually impossible to read or write all the program variables in an atomic step. There has been a lot of work on synthesis of distributed programs in the literature and we briefly discuss the issue of program synthesis in Chapter 6.

In [8], the authors prove that the problem of synthesizing a masking fault-tolerant program is NP-Complete in the size of state space and, hence, there is no polynomial time algorithm to synthesize a masking tolerant program unless $P = NP$. Furthermore, the authors present a nondeterministic algorithm that guesses a solution, namely the fault-tolerant program p', invariant S', and the fault-span T'. Then, the algorithm verifies wether p' is fault-tolerant from S' and T' is a boundary up to which p' can be perturbed in the presence of faults. In other words, the nondeterministic algorithm verifies if the guessed solution satisfies the transformation problem. In [19], Kulkarni and Ebnenasir show that the problem of adding failsafe fault-tolerance in the low atomicity model is also NP-Complete in the size of state space. However, identifying the complexity of adding nonmasking fault-tolerance in the low atomicity model is still an open problem.

Formal verification of a nondeterministic algorithm is not quite a challenging problem. Once the solution is guessed, the algorithm itself verifies the solutions. Therefore, we do not include the formal specification and verification of the synthesis algorithm in the low atomicity model in this thesis.

Chapter 5

Specification and Verification of Automatic Synthesis of Multitolerance

In Chapter 4, we presented specification and verification of algorithms associated with synthesis of fault-tolerant programs that are subject to a single class of faults. Real world systems are often subject to multiple classes of faults and, hence, they need to provide appropriate level of fault-tolerance to each class of faults. In other words, in synthesizing multitolerant programs, we are looking for the answer to the following questions: How a program could handle the occurrence of faults from different classes at the same time? And how would a synthesis algorithm provide a level of fault-tolerance for each class?

In this Chapter, first, we introduce the concerns and definitions regarding multitolerance in Section 5.1. In Section 5.2, we formally state the problem of mechanical verification of automatic synthesis of multitolerant programs. In Section 5.3, first, we present how to modify the definitions in chapters 2 and 4, so that they become appropriate for modeling the synthesis of multitolerance . Then, we present formal specification and verification of *nonmasking-masking* multitolerance. Finally, in Section 5.4, we present formal specification and verification of *failsafe-masking* multitolerance[1].

The time complexity of addition of failsafe-masking and nonmasking-masking multitolerance are polynomial. However, the problem of addition of *failsafe-nonmasking* is NP-Complete and, hence, there exists no polynomial time algorithm to solve this problem unless P=NP. In [10], the authors propose a nondeterministic algorithm, but we are not interested in verification of that algorithm, as the algorithm verifies the guessed solution by itself.

Remark. We remark that in this chapter, we effectively reuse the specification and formal proofs of the framework and algorithms developed in chapters 2 and 4. This is one of the instances that our manual proof reusability shows hope for developing automated proofs (proof strategies) for future verifications of extensions of the algorithms in [8, 10].

5.1 Faults, Fault-tolerance, and Multitolerance

The notion of multitolerance was first introduced by Arora and Kulkarni [20]. The definitions of program, specification, faults, and fault-tolerance remains the same (cf. Chapter 2). Now, we define what it means when a program is multitolerant.

In Section 2.4, we gave the definition of different levels of fault-tolerance for a single class of faults. Now, we consider the case where faults from multiple classes, say f_1 and f_2, may occur in a given program computation.

There exist several possible choices in deciding the level of fault-tolerance that should be provided in the presence of multiple fault-classes. In [10], Kulkarni and Ebnenasir propose to require that the fault-tolerance provided for the class where f_1

[1]Appendix F contains the formal specification of the algorithms for synthesizing multitolerant programs.

and f_2 occur simultaneously should be equal to the minimum level of fault-tolerance provided when either f_1 or f_2 occurs. For example, if masking fault-tolerance is to be provided to f_1 and failsafe fault-tolerance is to be provided to f_2, then failsafe fault-tolerance should be provided for the case f_1 and f_2 occur simultaneously. We reiterate the following table from [10] that illustrates the minimum level of fault-tolerance provided for different combinations of levels of fault-tolerance provided to individual classes of faults:

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 $f_{masking}$). Now, given a fault-intolerant program p, its invariant S, its specification *spec*, and sets of distinct classes of faults $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$, we define what it means for a synthesized program p', with invariant S', to be multitolerant by considering how p' behaves when (i) no faults occur; (ii) only one class of faults occur, and (iii) multiple classes of faults occur. Kulkarni and Ebnenasir, in [10], define a multitolerant program as follows:

Definition. Program p' is *multitolerant* to $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$ from S' for *spec* iff the following conditions hold:

1. p' satisfies *spec* from S' in the absence of faults.

2. p' is masking $f_{masking}$-tolerant form S' for *spec*.

3. p' is failsafe $(f_{failsafe} \cup f_{masking})$-tolerant form S' for *spec*.

4. p' is nonmasking $(f_{nonmasking} \cup f_{masking})$-tolerant form S' for *spec*.

5.2 The Problem of Mechanical Verification of Automatic Synthesis of Multitolerance

In this section, we present the problem of mechanical verification of automatic synthesis of multitolerance. Based on the definition of a multitolerant program in Section 5.1, Kulkarni and Ebnenasir identify the requirements of the problem synthesizing a multitolerant program p'. The general idea of algorithms for synthesizing multitolerance proposed in [10] is based on the synthesis of fault-tolerance for single class of faults [8]. Similar to the algorithms in [8] the requirements are (1) the synthesis algorithms should not introduce any new behavior to the fault-intolerant program, and (2) the synthesis algorithms should simply *add* multitolerance. The formal definition of the transformation problem and soundness are as follows:

The Transformation Problem

Given p, S, *spec*, $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$ such that p satisfies *spec* from S. Identify p' and S' such that:

$S' \subseteq S$

$(p' \mid S') \subseteq (p \mid S')$

p' is multitolerant to $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$ from S' for *spec*

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

In this chapter, our goal is to mechanically verify that the proposed algorithms in [10] for adding failsafe-masking and nonmasking-masking multitolerance are indeed sound by the PVS theorem prover. In other words, based on the definitions in this chapter, we show that the algorithms in [10] satisfy the transformation problem.

5.3 Specification and Verification of Synthesis of Nonmasking-Masking Multitolerance

In this section, we present description, formal specification and verification of the synthesis algorithm for addition of nonmasking-masking multitolerance to fault-intolerant programs that are subject to two types of faults $f_{nonmasking}$ and $f_{masking}$. First, we reiterate the algorithm *Add_Nonmasking_Masking* from [10] in Figure 5.1.

```
Add_Nonmasking_Masking($p$: transitions, $f_{nonmasking}, f_{masking}$: fault,
                              $S$: state predicate, $spec$: safety specification)
{
  $p_1, S', T_{masking} := Add\_Masking(p, f_{masking}, S, spec)$;
  if ($S' = \{\}$)   declare no multitolerant program $p'$ exists;
                   return $\emptyset, \emptyset$;
  $p', T' := Add\_Nonmasking(p_1, f_{nonmasking} \cup f_{masking}, T_{masking}, spec)$;
  return $p', S'$;
}
```

Figure 5.1: The synthesis algorithm for adding nonmasking-masking multitolerance

In order to formalize the algorithm *Add_Nonamsking_Masking* (and *Add_Failsafe_Masking* in Section 5.4), first, we define $f_{failsafe}$, $f_{nonmasking}$, and $f_{masking}$ from the type of Action. Then, we define $f_{nonmasking\_masking}$ and $f_{failsafe\_masking}$ as follows:

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

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

Our formalization for the fault-tolerance framework developed in Chapter 2 is appropriate for the case that we deal with only one class of faults. However, in modeling synthesis of multitolerance, as we have different classes of faults, we need to redefine some of the definitions, so that they accept the type of fault. For example, in Section 4.1.1, we modeled *RevReachableSet* as follows:

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

The problem with this definition is it only represents the set that can reach rs by taking a transition in f, which is the single class of faults. To formalize the algorithms for synthesizing multitolerance, we need to redefine *RevReachableSet*, so that it accepts different classes of faults. With this introduction, we redefine *msInit*, *RevReachableSet*, and *ms* as follows:

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

All the other definitions given in chapters 2 and 4 should also be converted in the same way, so that they are not restricted to only one class of faults. We refer the reader to the appendices for formal specification of new definitions.

Now, having the elements of multitolerance framework, we can formalize the algorithm *Add_Nonamsking_Masking*. Note that most of the definitions we developed in chapters 2 and 4 appear with a parameter for the type of faults. We proceed as follows: Using the algorithm *Add_masking*, we synthesize a masking $f_{masking}$-tolerant program p_1, with invariant S', and fault-span $T_{masking}$:

$$S' : StatePred = add\_masking.S1(f_{masking})$$
$$T_{masking} : StatePred = add\_nonmasking.S'(T_1(f_{masking}))$$
$$p_1 : Action = add\_masking.p_1(f_{masking})$$

We note that *add_masking* is the name of the theory developed in Chapter 4 for the algorithm *Add_masking* in PVS. Since program p_1 is masking $f_{masking}$-tolerant, it provides recovery to its invariant S', from any state in $(T_{masking} - S')$, while preserving safety. Thus, in the presence of $f_{nonmasking\_masking}$, if p_1 is perturbed to $(T_{masking} - S')$, then p_1 satisfies the requirements of nonmasking fault-tolerance. However, if $f_{nonmasking\_masking}$ perturbs p_1 to a state that is not in $T_{masking}$, then recovery should be added from those states. Towards this end, the algorithms ensures that one step recovery is possible. Thus, using the algorithm *Add_nonamsking*, we add one step recovery from states that are not in $T_{masking}$ to the states in $T_{masking}$:

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

As mentioned in Section 4.2.1, addition of one step recovery is independent of the type of faults. Hence, to simplify the verification of *Add_Nonmasking_Masking*, we formalize the multitolerant program by $p_1(f_{masking})$ and not $p_1(f_{nonmasking\_masking})$.

In order to verify the soundness of *Add_Nonmasking_Masking*, we prove a chain of lemmas and theorems. Note that, as the fault-span of nonmasking-masking multitolerant program and masking fault-tolerant program are equal, we do not restate and reprove the lemmas and theorems that verify the properties of safety of fault-span in the presence of only $f_{masking}$. Note that, in most of the following theorems, we assume that the termination condition of the repeat-until loop in $Add\_masking$ is satisfied.

Theorem 5.1: The first condition of the transformation problem (cf. Section 5.2) holds. Formally,

$$(S_2 \subseteq S) \Rightarrow (S' \subseteq S)$$

Proof. The proof is the same as proof of Theorem 4.30. □

Theorem 5.2: The second condition of the transformation problem (cf. Section 5.2) holds. Formally,

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

Proof. The proof is the same as proof of Theorem 4.31. □

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

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

Proof. The proof is the same as proof of Theorem 4.24. □

Theorem 5.4: The invariant of the nonmasking-masking multitolerant program is a subset of the fault-span. Formally, $S' \subseteq T'$

Theorem 5.5: $(add\_masking.T_1(f_{masking}) = T_2) \quad \Rightarrow \quad closed(T', p' \cup f_{masking})$

Theorem 5.6: Upon the loop termination, the invariant is closed in the program. Formally,

$$(add\_masking.S_1(f_{masking}) = S_2) \quad \Rightarrow \quad closed(S', p')$$

Theorem 5.7: For any synthesized nonmasking-masking program, there exists a fault-span. Formally, $\exists\ T : FaultSpan(T, S', p' \cup f_{nonmasking\_masking})$

Theorem 5.8: After termination of the loop, for any state in fault-span, T', there exists a computation of p' that starts from a state in $T' - S'$ and reaches the invariant, S_1. Formally,

$$((S_1(f_{masking}) = S_2) \ \wedge \ (T_1(f_{masking}) = T_2)) \ \ \Rightarrow$$
$$\forall s \mid s \in T' : reachable(S', T', p', s)$$

Proof. The proof of theorems 5.4 to 5.8 are the same as proof of the corresponding theorems for the algorithm *add_masking* in Section 4.3.2. $\square$

5.4 Specification and Verification of Synthesis of Failsafe-Masking Multitolerance

In this section, we present description, formal specification and verification of the synthesis algorithm for addition of failsafe-masking multitolerance to fault-intolerant programs that are subject to two classes of faults $f_{failsafe}$ and $f_{masking}$. First, we reiterate the algorithm *Add_Failsafe_Masking* from [10] in Figure 5.2.

```
Add_Failsafe_Masking($p$: transitions, $f_{failsafe}, f_{masking}$: fault, $S$: state predicate,
                                                   $spec$: safety specification,)
{
  $ms := \{s_0 : \exists s_1, s_2, ...s_n : (\forall j : 0 \leq j < n : (s_j, s_{(j+1)}) \in (f_{failsafe} \cup f_{masking})) \wedge$
                                                   $(s_{(n-1)}, s_n)$ violates $spec$ };
  $mt := \{(s_0, s_1) : ((s_1 \in ms) \vee (s_0, s_1)$ violates $spec)$ };
  $p_1, S', T_{masking} := Add\_Masking(p - mt, f_{masking}, S - ms, mt)$;
  if $(S' = \{\})$   declare no multitolerant program $p'$ exists;
                return $\emptyset, \emptyset$;
  $p', T' := Add\_Failsafe(p_1, f_{failsafe} \cup f_{masking}, T_{masking} - ms, mt)$;
  return $p', S'$;
```

Figure 5.2: The synthesis algorithm for adding failsafe-masking multitolerance.

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

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

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

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

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

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

In order to verify the soundness of *Add_Failsafe_Masking*, we prove the following lemmas and theorems:

Theorem 5.10: The first condition of the transformation problem (cf. Section 5.2) holds. Formally, $(S_2 \subseteq S) \Rightarrow (S' \subseteq S)$

Theorem 5.11: The second condition of the transformation problem (cf. Section 5.2) holds. Formally, $(p' \mid S_2 \subseteq p \mid S_2) \Rightarrow (p' \mid S' \subseteq p \mid S')$

Theorem 5.12: Upon the loop termination, the invariant is closed in the program. Formally,

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

Lemma 5.13: In the presence of faults, no computation prefix of failsafe-masking multitolerant program that starts from a state in S', reaches a state in ms. Formally,

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

Theorem 5.14: Any prefix of any computation of failsafe-masking multitolerant program in the presence of faults that starts in S' does not violate safety. Formally,

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

Lemma 5.15: All computations of $p - mt$ that start from a state in T' are infinite. Formally, $DeadlockStates(p_1 - mt)(T') = \emptyset$

Theorem 5.16: All computations of p' that start from a state in T' are infinite. Formally, $DeadlockStates(p')(T') = \emptyset$

Theorem 5.17: After termination of the loop, for any state in the fault-span, T', there exists a computation of p_1 that starts from a state in $T' - S'$ and reaches the invariant, S_1. Formally,

$$((S' = S_2) \ \wedge \ (T_1(f_{masking}) = T_2)) \quad \Rightarrow$$

$$\forall s \mid s \in T' : reachable(S', T', p', s)$$

Theorem 5.18: For any synthesized failsafe-masking program, there exists a fault-span. Formally, $\exists\ T : FaultSpan(T, S', p' \cup f_{failsafe\_masking})$

Proof. Formal proof of lemmas and theorems 5.10 to 5.18 are similar to the corresponding lemmas and theorems in sections 4.1.2 and 4.3.2. There are minor differences such as the symbols and the occasions that we expand definitions. For instance, in Section 4.1.2, we proved that no computation that starts from a state in the *invariant* of failsafe fault-tolerant program S', violates safety. In Lemma 5.13 and Theorem 5.14, we prove the same thing, but for a different state predicate, the *fault-span* T', and different set of faults, $f_{failsafe\_masking}$.

Chapter 6

Related Work

In this chapter, we focus on the previous work on verification of fault-tolerance properties of programs, circuits, and architectures. In Section 6.1, we briefly present the challenges in problem of program synthesis and transformation. In Section 6.2, we introduce the previous work on formal verification of fault-tolerant architectures, processors, and circuits. Then, in Section 6.3, we present related work on mechanical verification of fault-tolerance properties of programs, algorithms, and software related systems.

Throughout this thesis, our approach for verification of correctness of the algorithms in [8, 10] is by using theorem proving. Model checking is also shown to be an effective tool in verification of behavior of fault-tolerant finite state systems [21–26]. Model checking is widely used in verification of properties of domain specific fault-tolerant controllers and embedded systems. This approach is not quite related to our approach in verification. Hence, we do not get into the details of this context. In Chapter 7 we argue why we chose to use theorem proving techniques to address the problem of verification of the synthesis algorithms.

6.1 Challenges in Automatic Synthesis of Programs

In this section, we present a review of related work on the problem of automatic synthesis of reactive systems prior to [8]. As mentioned in Chapter 1, automatic synthesis of programs is an algorithmic approach for generating a new program from its specification or from another program. This problem is well studies through different approaches. In [27, 28], the authors present a survey on the problem of automatic synthesis of reactive systems. More specifically, in [27], three main approaches are presented. We now briefly introduce these approaches:

Model-Theoretic Approach

This approach is based on the extraction of a finite model from a given temporal logic[1] specification. A temporal logic specification consists of safety and liveness specifications in terms of temporal operators. Model-theoretic approach mostly addresses synthesis of *closed reactive systems* where there is no interaction between the system and its environment.

Several different algorithms have been presented in the literature for synthesis of reactive systems from their temporal logic specification. The initial work was presented by Emerson and Clark [30]. They present an algorithm for synthesis of reactive programs form CTL (Computational Tree Logic) specification. In their formalization, the safety specification is represented by *invariance formulas*; i.e., for all paths of execution, nothing bad happens. They represent the liveness properties of a program by *eventually formulas*; i.e., for all paths of execution the eventually formulas hold. The synthesis method is based on a decision procedure that verifies the satisfiability of a CTL temporal logics specification.

[1]We refer the reader to [29] for a survey on temporal logics.

Mana and Wolper [31,32] present a similar algorithm for synthesizing distributed processes that can communicate by message passing. In their approach, the program specification is given as a $PLTL$ (Propositional Linear Temporal Logics) formula.

In [33,34], Attie and Emerson present algorithmic methods, for synthesis of distributed and concurrent programs from CTL^* specification. In a more recent work, Arora, Attie, and Emerson [35] present an algorithm for synthesizing fault-tolerant program from CTL specification. Their algorithm consists of two phases. The first phase is basically performing the synthesis algorithm by Emerson and Clarke. In the second phase, they generate a fault-tolerant version of the synthesized program.

Automata-Theoretic Approach

As mentioned before, model-theoretic approaches address synthesis of closed reactive systems. By contrast, automata-theoretic approaches address synthesis of *open reactive systems* where a program has interactions with its environment.

In the automata-theoretic approach, the input of the synthesis algorithm is a tree automaton [36] that represents the specification of the system. The synthesis of an open reactive system involves converting its temporal logic specification to a tree automaton and, then, checking non-emptiness of the tree automaton; i.e., checking whether the language accepted by the tree automaton is empty or not [37–40].

In [37,38], Pnueli and Rosner show that the synthesis algorithm that generates a deterministic finite automaton that satisfies LTL specification has a doubly exponential complexity in the size of specification. In [39], Kupferman and Vardi show that the problem of synthesizing an open reactive distributed system is undecidable, if we have no assumptions about the system architecture.

Calculational Approach

In many cases, it is always desirable to *add* a property to programs rather than synthesizing a new program from new specification. This desire motivated researchers to design algorithms that can *add* a particular property to programs in addition to their old properties. In other words, in later approaches we would like to reuse the program and its invariant to calculate a new program and its new invariant.

In [41], Arora presents a new foundation for fault-tolerance. Using his formal framework, one can represent any kind of programs, faults, specifications, and fault-tolerance. In Chapter 2, we developed the formal specification of our framework based on Arora's definitions in [41].

We described the synthesis algorithms developed by Kulkarni and Arora [8] in details in Chapter 4. We also described the algorithms for automatic synthesis of multitolerance [10] in details in Chapter 5. The authors in both [8] and [10], use a calculational approach to generate new program. As mentioned before, in their method the synthesized program is correct by construction and there is no need to verify the individual programs. Furthermore, since their synthesis algorithms begin with an existing fault-intolerant program, the derived fault-tolerant program reuses the fault-intolerant version. Another advantage of their approach is, having the set of bad transitions, program, and its invariant, the algorithm can synthesize the fault-tolerant program. In other words, synthesis is possible without having the entire specification.

The issue of complexity is a serious barrier to implement the algorithms in [30, 32–35, 37–40]. In contrast, the time complexity of algorithms in [8, 10] show hope on practical implementability. More specifically, the time complexity of addition of fault-tolerance for single class of faults in the high atomicity model is polynomial and addition of failsafe and masking fault-tolerance in the low atomicity model are NP-Complete in the size of state space [8, 19]. The problem of addition of multitolerance, in general, is NP-Complete. However, the time complexity of additions of failsafe-masking and nonmasking-masking multitolerance are still polynomial. Moreover, In [19], Kulkarni and Ebnenasir identify the polynomial time boundary of the addition of failsafe fault-tolerance in the low atomicity model. In [11], Kulkarni and Ebnenasir focus on the problem of enhancing the addition of fault-tolerance of a nonmasking fault-tolerant program to masking.

To implement the synthesis algorithms, Kulkarni, Arora, and Chippada [42] develop a set of heuristics to synthesize a canonical version of the Byzantine agreement problem. Based on these heuristics, Ebnenasir and Kulkarni [12] has developed a tool for automatic synthesis of fault-tolerant programs.

We refer the reader to [27] to investigate more on the described approaches and detailed comparison.

6.2 Formal Verification of Fault-Tolerant Architectures

Hardware vendors are, perhaps, the largest community that use formal methods to model and verify their products. Researchers in this community focus on processes that show equivalence relations between a design and a specification. Simulation helps to gain more confidence, but logic correctness is a more reliable way to verify hardware. In this research area, proving the properties of computer instruction sets is perhaps the most established application of formal methods.

Fault-tolerance is usually a crucial part of safety-critical embedded systems. Thus, verification of fault-tolerance properties of hardware in embedded systems is well-studied and there has been enormous number papers and theses published in this area of research. In this section, we present the previous work on formal verification of fault-tolerant architectures, circuits, and processors.

Formal verification of fault-tolerant circuits. Kljaich, Smith, and Wojcik [43] focus on verification of logical correctness of fault-tolerance properties of digital systems. More specifically, they use theorem proving together with an extended Petri net representation to validate fault tolerance. Normal Petri net is an abstract machine that rests between Turing machines and finite state machines, in terms of decision power and modeling. *Execution* of a Petri net models nondeterministic dynamic behavior and is represented by firing transitions. The extended Petri net theory in [43] avoids the nondeterministic behavior and gives functional capability to the firing of transitions. The authors, first, give an examples of a typical 2-bit ALU to show the modeling capability of their extension to Petri net model. Then, based on their design for the 2-bit ALU, they present a fault-tolerant processor (FTP) that tolerates one or more failures. Fault-tolerance is achieved using redundant modules. Using automated reasoning implemented by PROLOG, the authors show that for possible arrangements of instructions sets, how many redundant modules are required to meet the fault-tolerance specification of the CPU.

Formal verification of synchronized fault-tolerant circuits. Verification of fault-tolerant clock synchronization circuits has been a challenging problem in the literature of hardware verification. Miner and Johnson [44] present an optimization of their previous work on formal development of a fault-tolerant synchronization circuit. The authors argue that in the previous work researchers in the formal methods community had been focused on how to verify hardware using particular verification systems and not on type of reasoning support that we need. They represent circuits as systems of stream equations, where each stream corresponds to a signal within the circuit. The signals are annotated with invariants which can be established using proof by co-induction. Miner and Johnson, exploit the invariants to verify that their formally designed circuit satisfies the localized design refinements. More specifically, first, they introduce a design hierarchy with different levels of abstraction. Then, they use PVS as their verification tool to prove the properties of their design.

NASA SPIDER project. NASA's most recent project on fault-tolerant architectures is the formally-verified fault-tolerant architecture, Scalable Processor-Independent Design for Electromagnetic Resilience (SPIDER) [45]. SPIDER is a family of fault-tolerant, reconfigurable architectures that provide mechanisms for integrating inter-dependent applications of differing criticalities. SPIDER comes with three protocols: Consistency Protocol, Diagnosis Protocol, and Synchronization Protocol. The first two protocols provide a fault-tolerant broadcast communication architecture. More specifically, the Consistency Protocol takes care of a reliable message broadcast in the presence of malicious faults, using the Byzantine Agreement problem. The Diagnosis Protocol distributes local information about the health status of nodes across the network. In [46], Miner and Geser present mechanical verification of the first two protocols. They present formal proofs in PVS that given the dynamic maximum fault assumption and a sane health status classification, the Consistency Protocol agreement, and the Diagnosis Protocol provides a sane classification of faulty nodes.

Abstractions for hardware verification. Melham [47] proposes four abstractions for formalizing and verifying hardware. He argues that such abstractions decreases the complexity and size of specification and verification. His argument is based on the fact that specifying hardware in detail and proving that the specification satisfies the requirements through formula equivalence provokes the design hierarchy. Melham suggests a satisfaction relation based on the idea of *abstraction*, rather than equivalence, is the key to make large designs tractable. He introduces four abstractions:

1. *Structural* abstraction–the suppression of information about a device's internal structure. The idea of structural abstraction is that the specification of a device should not reflect its internal construction, but its externally observable behavior.

2. *Behavioral* abstraction concerns specification that only partially define a de-

vice's behavior.

3. *Data* abstraction is well known from programming languages theory.

4. *Temporal* abstraction views a device by sequential time-dependent behavior.

Based on [47], fault-tolerance properties of hardware can be specified in structural and behavioral levels of abstraction. Examples in [47] present very top level properties of every hardware design that can be applied to specification of many systems.

6.3 Mechanical Verification of Fault-Tolerance Properties of Programs

In this section, we present the related work on formal verification of programs and algorithms that provide fault-tolerance for software systems. The related work in this section has a closer theme to our work in this thesis. In Section 6.3.1, we present related work on formal verification of fault-tolerant embedded and real-time digital systems. In Section 6.3.2, we introduce previous work on mechanical verification of algorithms that provide fault-tolerance for mutual exclusion and message passing problems. Finally, in Section 6.3.3, we present recent research work on defining abstractions for fault-tolerant computing regardless of algorithm, protocol, and hardware platform.

6.3.1 Formal Verification of Fault-Tolerant Embedded Digital Systems.

Embedded systems mostly serve in digital controllers. In many cases, such controllers are in charge of monitoring safety-critical systems that need greatest assurance. For example, *catastrophic failure of digital flight-control systems for passenger* *aircraft must be "extremely improbable"* [3]. Many researchers has focused on verification of fault-tolerant digital flight control systems in the past two decades and there has been great advances in specification and verification techniques of such systems in the literature [3, 48–52]. Clock synchronization and message passing are two major concerns in such systems. Components across distributed digital embedded systems miss clock signals due to transient and Byzantine faults. These faults can easily lead the entire system to an unsynchronized and out of order state. Hence, verification of fault-tolerant clock synchronization has being paid great attention from researchers and verification engineers. Message passing is another serious issue in embedded systems. Hence, fault-tolerant message passing and message broadcast protocols are also well-studied [3, 46]. Verification tools vary from early theorem provers such as the Boyer-Moor theorem prover and EHDM to modern ones such as PVS, HOL, and ACL2. However, most attempts address formal proof techniques for a particular fault-tolerant protocols or algorithms.

Although verification of fault-tolerant embedded systems rely on specific characteristics of different systems, there has been attempts to abstract the concepts that are common among such systems. For instance, in [49], Rushby introduces and abstraction for mechanical verification of fault-tolerant time-triggered algorithms. His approach provides a methodology that can ease the formal specification and assurance of critical fault-tolerant systems. Mantel and Gärtner [4] argue that modular reasoning about fault-tolerant systems can simplify the verification process in practice. They exercise their idea on the problem of reliable message broadcast in point-to-point networks in the presence of failures of processes.

6.3.2 Formal Verification of Fault-Tolerant Algorithms

Dijkstra's self-stabilizing token ring algorithm. A self-stabilizing algorithm is one that guarantees that the system behavior eventually recovers to a safe subset of states regardless of the initial state. Obviously, self-stabilization is a type of fault-tolerance. In [6], Dijkstra introduced his well-known self-stabilizing mutual exclusion algorithm (also known as token ring algorithm) in 1974. At the time, he assumed the algorithm is trivially correct and left the proof as an exercise for the reader. However, 12 years later, Dijkstra published a belated proof of correctness of the algorithm in [53], which was not quite trivial! In the algorithm, there are N Processes for $N > 1$ numbered 0 to $N - 1$ arranged in a unidirectional ring where each process can observe its own state and that of its predecessor. Each process i maintains a counter $v(i)$ of values from 0 to $M - 1$, where $N \leq M$. Process 0 is a distinguished process that is enabled when $v(0) = v(N - 1)$, and when enabled, it can make a transition by incrementing its counter modulo M. A nonzero process i is enabled when $v(i) \neq v(i - 1)$, and when enabled, it can update its counter value so that $v(i) = v(i - 1)$. It can be shown that despite a centralized controller the system can *self-stabilize* from an arbitrary initial state to a stable behavior where exactly one process is enabled at any state. This algorithm is useful as a solution for distributed mutual exclusion problem; i.e., the enabled process (in other words, the one that has the token) can enter its critical section. Qadeer and Shankar [5], exercise mechanical verification of self-stability property of Dijkstra's algorithm using PVS. First, they formalize the algorithm using the PVS specification language. Then, based on Dijkstra's informal proof sketch they present how to mechanically verify the self-stability property of the algorithm. However, they do not present any sort of abstraction or general design. Hence, there formal proof cannot be easily applied elsewhere.

6.3.3 Formal Verification of Abstractions for Fault-Tolerance

In previous sections in this chapter, we presented related work on mechanical verification of specific fault-tolerant circuits, protocols, algorithms, digital embedded systems, and microprocessors. In this section, we introduce the related work that attack the problem of verification of fault-tolerant systems by introducing abstractions for programs, faults, and distributed systems. Defining such abstractions make the process of verification less complex and more easy to prove.

Abstractions for fault-tolerant distributed systems. In [54], Pike et al present four kinds of abstractions for the design and analysis of fault-tolerant distributed systems. More specifically, they introduce (i) *Message Abstractions* to address the correctness of individual messages sent and received; (ii) *Fault Abstractions* to address the kinds of faults possible as well as their effect in the system; (iii) *Fault-Masking Abstractions* to address the kinds of local computations processes make to mask faults, and (iv) *Communication Abstractions* to address the kinds of data communicated and the properties required for communication to succeed in the presence of faults. The authors formalize the abstractions in PVS.

Abstractions for modeling faults, programs, and specifications. As mentioned in Section 6.2, in [47], Melham introduces four abstraction for hardware verification. Likewise, we need to introduce different levels of abstractions to make the specification and verification of fault-tolerant systems less complex. Moreover, using such abstractions, it is more likely that we can reuse formal proofs and develop proof strategies. In [7, 55, 56], the authors introduce abstractions for modeling faults, programs, specifications, and component-based design of fault-tolerant programs. In [7], Kulkarni, Rushby, and Shankar present a case-study in component-based mechanical verification of Dijkstra's self-stabilizing mutual exclusion algorithm. The authors demonstrate that decomposition of a fault-tolerant program to its components is useful to make the verification less complex and develop reusable formal proofs. More specifically, they decompose a fault-tolerant program to a fault-intolerant program and fault-tolerance components that are *detectors* and *correctors*. This decomposition facilitates the verification of a given property by focusing on the component that is responsible for satisfying it. In this sense, verification of the entire fault-tolerant program reduces to verification of the component that is in charge of satisfying a certain property. As an exercise, the authors, first, implement the theory of detectors and correctors [2] in PVS. Then, they verify the correctness of Dijkstra's token ring algorithm using the mentioned theory. Kulkarni et al verify the same algorithm that Qadeer and Shankar prove its correctness in [5], but their approach (decomposition of program into its components) is more general and abstract in the sense that their formal proofs techniques are reusable to verify other fault-tolerant program. Briefly, in [5], the authors verify three major concerns. First, they formally prove that in the absence of faults, the fault-tolerant program has the same behavior as the fault-intolerant program. Then, they verify the corrector; i.e., if faults perturb the state of program, at the resulting state, recovery to an ideal state of the fault-intolerant program is possible. Finally, they verify that there is no interference between fault-intolerant program and the corrector.

Chapter 7

Discussion

In this chapter, we discuss about different aspects of this thesis and try to answer the questions that have been raised about the verification of synthesis algorithms proposed in [8,10].

Theorem proving vs. model checking. The first issue that we would like to address is why we used theorem proving techniques to verify the synthesis algorithms. Model checking is also widely used to verify the fault-tolerance properties of many safety-critical embedded systems. However, model checking can only be used for systems that have finite and known size of state space. In our case, although the state space of the programs that the synthesis algorithms deal with are finite, the size of state space may vary. Therefore, model checking does not seem to be the right technique for verification of the synthesis algorithms.

The synthesis method. In Chapter 6, we presented related work on challenges in the problem of program synthesis and mechanical verification of fault-tolerant systems. In [30,32,33,35], the authors propose algorithms that synthesize a program from its temporal logic specification. In the previous work prior to [8], 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.

By contrast, the algorithms in [8, 10] 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 [8, 10] seem to be suitable candidates for practical implementation purposes. As an extension of the algorithms in [8], in [42], 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 [12]. Therefore, by formal verification of the algorithms in [8, 10], 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 [8, 10] make no assumptions about 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 in [11, 42] that are extensions of algorithms in [8] for any system regardless of the platform and architecture. In addition, having the formal specification and verification developed in this thesis, we can easily verify the extensions of the algorithms by reusing the specification developed in this thesis.

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. Thus, we are not required to verify individual synthesized programs.

Reusability of formal proofs. Although most of the related work on formal verification of fault-tolerance 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 4.3.2, 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 Section 4.1.2, we have shown that the algorithm for adding failsafe fault-tolerance in complete. However, in this thesis, we did not spent a great deal of attention on verification of the completeness 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 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 [8,19]. In other words, we have to apply heuristics [11, 42] 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.

Lessons learned. Formalization and verification of the synthesis algorithms and fixpoint theory presented in chapters 2 to 5 were not our first and last attempts. In fact, we tried several other ways to model the problem, but they were either not abstract enough or they were too complicated to handle. We summarize the lessons we learned as follows:

- Sometimes it is very tempting to prove the easy-looking theorems before actually verifying them manually. This temptation may lead the developer to spend a lot of time to prove an incorrect theorem. Basically, if you cannot do a proof by hand you cannot do it automatically.

- In many cases a failed proof teaches more than a successful proof, because the developer gains a better understanding of the problem and formalization through a failed proof.

- Improper formal specification usually leads us to state wrong or difficult theorems. For instance, our first attempt to formalize the largest fixpoint calculation was based on an inductive definition on the finite set itself rather the steps of fixpoint calculation. Here is our first attempt to model the largest fixpoint calculation for removing deadlock states from a given state predicate X:

  ```
  Dec (X: StatePred) : RECURSIVE StatePred =
        LET ds = DeadlockStates(X) IN
                IF ds = {} THEN X

                ELSE Dec (X - ds)
                ENDIF
  MEASURE |X|
  ```

  This formalization is probably the simplest way to model a recursive function on a finite set, but using the induction schemes for finite sets in PVS is not quite convenient and proof of very simple theorems turn to huge proof trees. On the other hand, by defining the recursive function based on the steps of fixpoint calculation rather than the finite set itself and using induction on the number of steps, which is an integer, theorems become much easier to prove. Hence, a simple formalization does not necessarily lead the developer to easier proofs.

- The development of the fixpoint theory was inspired by several implementations of specific fixpoint calculations. In fact, we did not develop the fixpoint theory before developing the synthesis algorithms! In many cases, several implementations of special cases of a problem leads the developer to develop a general theory once and instantiate it to verify the special cases.

Chapter 8

Conclusion and Future Work

In this thesis, we focused on the problem of verifying transformation algorithms that generate fault-tolerant programs that are correct by construction using PVS. We considered the programs that are subject to a single class or multiple classes of faults. For the case that a program is subject to a single class of faults we considered three types of fault-tolerance that are, *failsafe*, *nonmasking*, and *masking*. We verified soundness and completeness of synthesis algorithms for adding failsafe and nonmasking fault-tolerance. For addition of masking fault-tolerance, we only considered verification of the soundness. For the synthesis algorithms associated with addition of multitolerance, we verified the soundness of addition of *failsafe-masking* and *nonmasking-masking*.

The theory for fixpoint calculations on finite sets. The essence of adding failsafe and masking fault-tolerance as well as failsafe-masking, and nonmasking-masking multitolerance is calculating the new invariant of program, which in turn involves calculating the fixpoint of a formula. We developed a theory for fixpoint calculation on finite sets that is customized for specification and verification of fault-tolerance. More specifically, we introduced the formalization of smallest fixpoint calculation to calculate reachability of a set of sates. Moreover, we presented the formalization of largest fixpoint calculation to calculate the set deadlock states of a given state predicate. The theory developed in Chapter 3 is expected to be reusable for other formalizations that involve fixpoint calculations.

Mechanized verification of fault-tolerance and multitolerance. For the case that programs are subject to a single class of faults, we considered verification of addition of three levels of fault-tolerance. More specifically, first, we developed a formal framework for modeling fault-tolerance in PVS. This framework introduces formal definitions for programs, specifications, faults, and fault-tolerance. Then, using the framework, we formally proved that in the presence of faults, any synthesized program by the *Add_failsafe* algorithm never violates safety and in the absence of faults, the program maintains its specification. For the *Add_nonmasking* algorithm, we showed that the nonmasking fault-tolerant program provides recovery to the normal behavior when the state of the program is perturbed by faults, and in the absence of faults it maintains its specification. Finally, we verified that a masking fault-tolerant program provides safe recovery when the state of the program is perturbed by faults, and in the absence of faults it maintains its specification. In other words, we mechanically verified that the algorithms for addition of fault-tolerance in [8] are sound. We also addressed formal verification of completeness of failsafe. More specifically, we proved if there exists a failsafe program that satisfies the transformation problem (cf. Section 2.5), then the algorithm *Add_failsafe* never declares *failure*.

For the case that programs are subject to multiple classes of faults, we considered verification of addition of two types of multitolerance. More specifically, first, we extended our formal framework for fault-tolerance, so that our formal definitions can deal with multiple classes of faults. Then, using the extended framework, we formally proved that in the presence of faults, any synthesized program by the *Add_Failsafe_Masking* algorithm (i) never violates safety when the state of the program is perturbed in the presence of the class of faults that failsafe fault-tolerance is to be provided; (ii) provides safe recovery when the state of the program is perturbed in the presence of the class of faults that masking fault-tolerance is to be provided; (iii) provides failsafe fault-tolerance if faults from both types occur, and (iv) maintains its specification in the absence of faults. For the *Add_Nonmasking_Masking* algorithm, we showed that the nonmasking-masking multitolerant program (i) provides safe recovery to the normal behavior when the state of the program is perturbed by class of faults that masking fault-tolerance is to be provided; (ii) provides recovery to its invariant when the state of the program is perturbed by class of faults that nonmasking fault-tolerance is to be provided; (iii) provides nonmasking fault-tolerance if faults from both types occur, and (iv) maintains its specification in the absence of faults. In other words, we mechanically verified that the algorithms for addition of multitolerance in [10] are all sound. Verification of multitolerance is one of the instances that we simply reused both formal specification and proofs of the framework developed in Chapter 2 and the algorithms developed in Chapter 4. We expect this reusability for verification of other extensions of the algorithms [11, 42].

The algorithms verified in this thesis synthesize programs in the high atomicity model, where a process can read and write all variables in an atomic step. In [8], the authors have presented a non-deterministic algorithm for synthesizing fault-tolerant distributed programs. Moreover, in [10], the authors have introduced a non-deterministic algorithm for synthesizing failsafe-nonmasking multitolerant programs. The non-deterministic algorithms in both cases, first, guess a solutions and then verifies a solution by itself. Formal verification of such an algorithm that verifies its output by itself does not have any interesting point and, hence, we did not include them in this thesis.

Soundness vs. completeness. In this thesis, we concentrated more on verification of soundness of algorithms rather than their completeness 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 programs synthesized by an algorithm are fault-tolerant indeed. Second, proving the completeness of a non-deterministic polynomial time synthesis algorithm is irrelevant for the problem of adding fault-tolerance in the sense that they are known to be NP-Complete. In order to deal with NP-Complete problems in addition of fault-tolerance, several heuristics have been proposed [11,42]. As a result of applying such heuristics, we lose the completeness of the synthesis algorithms. Therefore, we have to only concentrate on verifying the soundness of such heuristics unless $P = NP$.

Advantages of verification of synthesis algorithms. 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.

In a broader context, the verification of the algorithms considered in this thesis will assist us in verifying several other transformations. For example, the algorithms in [8,9] have also been used to synthesize fault-tolerant distributed programs. As an illustration, we note that the algorithms in [11,12,42] that are extensions of the algorithms in [8,9] 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 thesis are directly applicable to verify the transformation algorithms in [11,12,42] as well.

Future work. As a future work, one may consider mechanical verification of heuristics developed in [42] to synthesize fault-tolerant distributed programs. Also, formal verification of the algorithms proposed in [11] for enhancing the synthesis of masking fault-tolerant distributed programs through their nonmasking version is also an interesting problem.

Our experience shows that significant number of proofs were reused. For instance, we manually reused proofs of failsafe fault-tolerance to verify the soundness of the algorithm for synthesizing masking fault-tolerant programs. We also, reused formal proofs of soundness of addition of failsafe, nonmasking, and masking fault-tolerance to verify the algorithms for addition of multitolerance. We expect to reuse many of the theorems and proofs in future verifications as well. Therefore, a future work is developing automated proofs, called *proof strategies*, based on our experience in reusability of formal proofs.

APPENDICES

Appendix A

Formal Specification of the Fault-Tolerance Framework

```
FT[state: TYPE]: THEORY
 BEGIN

  ASSUMING
   ST_is_finite: ASSUMPTION is_finite_type[state]

   TR_is_finite: ASSUMPTION is_finite_type[[state, state]]

   IMPORTING sets[state]
   IMPORTING sets_lemmas[state]
   IMPORTING finite_sets[state]
   IMPORTING sequences[state]
   IMPORTING finite_fp[state]
   IMPORTING min_nat
```

ENDASSUMING

Transition: TYPE = [state, state]

Action: TYPE = finite_set[Transition]

Computation(Z: Action): TYPE =

$\{A$: sequence[state] | $\forall$ (n: nat): $((A(n),\ A(n+1)) \in Z)\}$

prefix(Z: Action, j: nat): TYPE =

$\{c$: sequence | $\forall$ ((i: nat | $i < j$)): $((c(i),\ c(i+1)) \in Z)\}$

full_state: JUDGEMENT fullset[state] HAS_TYPE finite_set

StateSpace: finite_set = fullset[state]

S: StatePred

T: StatePred

p: Action

f: Action

sf: Action

s, s_0, s_1: VAR state

X, Y: VAR StatePred

Z: VAR Action

i, j, k: VAR nat

g: VAR DecFunc

r: VAR IncFunc

closed?(S: StatePred, P: Action): bool =

  $\forall$ (s_0, s_1): $((((s_0,\ s_1) \in P) \wedge (s_0 \in S)) \supset (s_1 \in S))$

```
proj(p: Action, S: StatePred): Action =
    {s_0, s_1 | ((s_0, s_1) ∈ p) ∧ (s_0 ∈ S) ∧ (s_1 ∈ S)}

fault_span?(T, S: StatePred, p: Action): bool =
    (∀ s_0: (s_0 ∈ S) ⊃ (s_0 ∈ T)) ∧ closed?(T, (p ∪ f))

ax1: AXIOM closed?(S, p)

ax3: AXIOM
  ∀ (c: Computation((Z ∪ f))):
    (∃ (j: nat): (∀ ((n: nat | n ≥ j)): ((c(n), c(n + 1)) ∈ Z)))

ax5: AXIOM nonempty?(S)

ms_init(anyFault: Action): StatePred =
    {s_0 | ∃ (s_1): ((s_0, s_1) ∈ anyFault) ∧ ((s_0, s_1) ∈ sf)}

reverse_reachable_states(anyFault: Action)(rs: StatePred): StatePred =
    {s_0 |
        ∃ (s_1):
            ((s_1 ∈ rs) ∧ ((s_0, s_1) ∈ anyFault) ∧ ¬ (s_0 ∈ rs))}

rs_if: JUDGEMENT reverse_reachable_states(anyFault: Action) HAS_TYPE
    IncFunc

deadlock_states(p: Action)(ds: StatePred): StatePred =
    {s_0 |
        (s_0 ∈ ds) ∧ (∀ (s_1): (s_1 ∈ ds) ⊃ ¬ ((s_0, s_1) ∈ p))}

ax6: AXIOM empty?(deadlock_states(p)(S))

dl_df: JUDGEMENT deadlock_states(p: Action) HAS_TYPE DecFunc

ms(anyFault: Action): StatePred =
    nu_fix(ms_init(anyFault))(reverse_reachable_states(anyFault))

ms(i)(anyFault: Action): StatePred =
    nu_inc(i, ms_init(anyFault))(reverse_reachable_states(anyFault))

mt(anyFault: Action): Action =
    {s_0, s_1 | ((s_1 ∈ ms(anyFault)) ∨ ((s_0, s_1) ∈ sf))}

ConstructInvariant(X: StatePred, Z: Action): StatePred =
    mu_fix(X)(deadlock_states(Z))

finite_fault: THEOREM
  ∀ (c: Computation((Z ∪ f))):
    ∃ (j: nat):
      (∀ (n: nat): ((suffix(c, j)(n), suffix(c, j)(n + 1)) ∈ Z))

END FT
```

Appendix B

Formal Specification of the Fixpoint Theory

```
finite_fp[T: TYPE]: THEORY
 BEGIN
  ASSUMING
   IMPORTING sets[T]
   IMPORTING sets_lemmas[T]
   IMPORTING finite_sets[T]
   IMPORTING mucalculus[T]
   T_is_finite: ASSUMPTION is_finite_type[T]
  ENDASSUMING


  StatePred: TYPE = finite_set[T]
  IncFunc: TYPE = [A: StatePred → {B: StatePred | disjoint?(A, B)}]
  DecFunc: TYPE = [A: StatePred → {B: StatePred | (B ⊆ A)}]
  StateSpace: finite_set = fullset[T]
```

```
s: VAR T
i, j, k: VAR nat
X: VAR StatePred
g: VAR DecFunc
r: VAR IncFunc

fullset_finite: JUDGEMENT fullset[T] HAS_TYPE finite_set

nu_inc(i: nat, X: StatePred)(r: IncFunc): RECURSIVE StatePred =
  IF i = 0 THEN X ELSE (nu_inc(i − 1, X)(r) ∪ r(nu_inc(i − 1, X)(r))) EN-
DIF
   MEASURE (λ (x: nat, y: StatePred): x)

nu_fix(X: StatePred)(r: IncFunc): StatePred =
    {s | ∃ (k: nat): (s ∈ nu_inc(k, X)(r))}

mu_dec(i: nat, X: StatePred)(g: DecFunc): RECURSIVE StatePred =
  IF i = 0 THEN X ELSE (mu_dec(i − 1, X)(g) \ g(mu_dec(i − 1, X)(g))) EN-
DIF
   MEASURE (λ (x: nat, y: StatePred): x)

mu_fix(X: StatePred)(g: DecFunc): StatePred =
    {s | ∀ (k: nat): (s ∈ mu_dec(k, X)(g))}

sfpcal(fpSet: StatePred)(r: IncFunc): RECURSIVE StatePred =
  IF nonempty?(r(fpSet)) THEN sfpcal((r(fpSet) ∪ fpSet))(r) ELSE fpSet EN-
DIF
```

```
   MEASURE card(StateSpace) − card(fpSet)

lfpcal(fpSet: StatePred)(g: DecFunc): RECURSIVE StatePred =
  IF empty?(g(fpSet)) THEN fpSet ELSE lfpcal((fpSet \ g(fpSet)))(g) ENDIF
   MEASURE card(fpSet)

mu1: LEMMA (j < k) ⊃ (mu_dec(k, X)(g) ⊆ mu_dec(j, X)(g))

mu3: LEMMA
  ∀ (j: nat):
    nonempty?(g(mu_dec(j, X)(g))) ⊃
     card(mu_dec(j + 1, X)(g)) ≤ card(X) − j − 1

mu5: LEMMA
  ∀ (j: nat):
    empty?(g(mu_dec(j, X)(g))) ⊃
     (∀ ((k: nat | k ≥ j)): empty?(g(mu_dec(k, X)(g))))

mu6: LEMMA ∃ (j: nat): ∀ ((k: nat | k ≥ j)): empty?(g(mu_dec(k, X)(g)))

mu7: LEMMA
  ∃ (j: nat):
    ∀ ((k: nat | k ≥ j)):
      mu_dec(k, X)(g) = mu_dec(j, X)(g) ∧ empty?(g(mu_dec(k, X)(g)))

mu8: LEMMA
  ∃ (j: nat):
```

$((\text{mu\_dec}(j,\ X)(g) = \text{mu\_fix}(X)(g)) \wedge \text{empty?}(g(\text{mu\_dec}(j,\ X)(g))))$

mu9: THEOREM $\text{empty?}(g(\text{mu\_fix}(X)(g)))$

mu10: THEOREM $\text{mu\_fix}(X)(g) = \text{mu\_fix}(\text{mu\_fix}(X)(g))(g)$

nu1: LEMMA $(k < j) \supset (\text{nu\_inc}(k,\ X)(r) \subseteq \text{nu\_inc}(j,\ X)(r))$

nu2_3: LEMMA

$\quad (s \in \text{nu\_inc}(j+1,\ X)(r)) \supset$

$\quad\quad ((s \in r(\text{nu\_inc}(j,\ X)(r))) \vee$

$\quad\quad\quad (\exists\ ((k\text{: nat} \mid k \leq j))\text{: } (s \in \text{nu\_inc}(k,\ X)(r))))$

nu2_4: LEMMA

$\quad (s \in \text{nu\_fix}(X)(r)) \supset$

$\quad\quad (\exists\ (j\text{: nat})\text{:}$

$\quad\quad\quad (s \in r(\text{nu\_inc}(j,\ X)(r))) \vee$

$\quad\quad\quad\quad (\exists\ ((k\text{: nat} \mid k \leq j))\text{: } (s \in \text{nu\_inc}(k,\ X)(r))))$

nu3: LEMMA

$\quad \forall\ (j\text{: nat})\text{:}$

$\quad\quad \text{nonempty?}(r(\text{nu\_inc}(j,\ X)(r))) \supset$

$\quad\quad\quad \text{card}(\overline{\text{nu\_inc}(j+1,\ X)(r)}) \leq$

$\quad\quad\quad\quad \text{card}(\text{fullset}[T]) - \text{card}(X) - j - 1$

nu5: LEMMA

$\quad \forall\ (j\text{: nat})\text{:}$

```
    empty?(r(nu_inc(j, X)(r))) ⊃
     (∀ ((k: nat | k ≥ j)): empty?(r(nu_inc(k, X)(r))))


nu6: LEMMA ∃ (j: nat): ∀ ((k: nat | k ≥ j)): empty?(r(nu_inc(k, X)(r)))


nu7: LEMMA
  ∃ (j: nat):
    ∀ ((k: nat | k ≥ j)):
      nu_inc(k, X)(r) = nu_inc(j, X)(r) ∧ empty?(r(nu_inc(k, X)(r)))


nu8: LEMMA
  ∃ (j: nat):
    ((nu_inc(j, X)(r) = nu_fix(X)(r)) ∧ empty?(r(nu_inc(j, X)(r))))


nu9: THEOREM empty?(r(nu_fix(X)(r)))


nu10: THEOREM nu_fix(nu_fix(X)(r))(r) = nu_fix(X)(r)

END finite_fp
```

Appendix C

Formal Specification of the Synthesis of Failsafe Fault-Tolerance

```
add_failsafe[state: TYPE]: THEORY
 BEGIN

  ASSUMING
   ST_is_finite: ASSUMPTION is_finite_type[state]
   TR_is_finite: ASSUMPTION is_finite_type[[state, state]]
   IMPORTING FT[state]

  ENDASSUMING

  i, j: VAR nat
  s0, s1: VAR state
```

Sp(anyFault: Action): StatePred =

  ConstructInvariant(($S \setminus$ ms(anyFault)), ($p \setminus$ mt(anyFault)))

pp(anyFault: Action): Action =

  ($p \setminus$ mt(anyFault))

    {s_0, s_1 | $\neg$ s_0 Sp(anyFault) $\wedge$ $\neg$ `member`(s_0, Sp(anyFault))}

sc:

{c: Computation(f) |

    $\forall$ ((s: state | ($s \in$ ms_init(f)))):

      (c(0) = s) $\wedge$ ((c(0), c(1)) $\in$ sf)}

sc1:

{c: Computation(f) |

    $\forall$ ((t: Transition | ($t \in (p \cap$ sf)))): c(0) = t‘2}

is_failsafe?(Spp: StatePred, ppp: Action): bool =

  nonempty?(Spp) $\wedge$

   closed?(Spp, ppp) $\wedge$

    (Spp $\subseteq S$) $\wedge$

     (proj(ppp, Spp) $\subseteq$ proj(p, Spp)) $\wedge$

      empty?(deadlock_states(ppp)(Spp)) $\wedge$

       ($\forall$ ((c: Computation((ppp $\cup f$)) | (c(0) $\in$ Spp))):

         $\forall$ (j: nat): $\neg$ (($c(j)$, $c(j+1)$) $\in$ sf))

add_failsafe_fails?: bool = empty?(Sp(f))

rs_disj: LEMMA

  $\forall$ (X: StatePred): disjoint?(X, reverse_reachable_states(f)(X))

rs_empty: LEMMA reverse_reachable_states(f)(emptyset) = emptyset

disj_Sp_ms: THEOREM disjoint?(Sp(f), ms(f))

disj_Sp_sf: THEOREM disjoint?(Sp(f), ms_init(f))

ms_f_sf: LEMMA

  $\forall$ (s_0, s_1): $((s_0,\ s_1) \in f) \wedge ((s_0,\ s_1) \in \mathrm{sf}) \supset (s_0 \in \mathrm{ms}(f))$

prop: LEMMA $(\mathrm{ms\_init}(f) \subseteq \mathrm{ms}(f))$

disj_pp_sf: LEMMA disjoint?(pp(f), sf)

prop2: LEMMA

  $\forall$ (anyFault: Action):

    $(((s_0,\ s_1) \in \mathrm{anyFault}) \wedge (s_1 \in \mathrm{ms}(\mathrm{anyFault}))) \supset$

     $(s_0 \in \mathrm{ms}(\mathrm{anyFault}))$

prop3: LEMMA

  $\forall$ (s_0, s_1):

    $(((s_0,\ s_1) \in \mathrm{sf}) \wedge ((s_0,\ s_1) \in f)) \supset (s_0 \in \mathrm{ms}(f))$

prop4: LEMMA

  $\forall$ (j: nat):

```
  $((s_0 \in \mathrm{ms}(j+1)(f)) \wedge \neg (s_0 \in \mathrm{ms}(j)(f))) \supset$
   $(\exists (s_1): ((s_0, s_1) \in f) \wedge (s_1 \in \mathrm{ms}(j)(f)))$
```

fix3: THEOREM $\mathrm{ConstructInvariant}(\mathrm{Sp}(f), (p \setminus \mathrm{mt}(f))) = \mathrm{Sp}(f)$

scond1: THEOREM $(\mathrm{Sp}(f) \subseteq S)$

scond2: THEOREM $(\mathrm{proj}(\mathrm{pp}(f), \mathrm{Sp}(f)) \subseteq \mathrm{proj}(p, \mathrm{Sp}(f)))$

scond2_1: COROLLARY $(\mathrm{pp}(f) \subseteq p)$

scond3_1: LEMMA $\mathrm{closed?}(\mathrm{Sp}(f), \mathrm{pp}(f))$

scond3_1_1: THEOREM

```
 $\forall (j: \mathrm{nat}):$
   $\forall (c: \mathrm{prefix}((\mathrm{pp}(f) \cup f), j)):$
     $(c(0) \in \mathrm{Sp}(f)) \supset (\forall ((k: \mathrm{nat} \mid k < j)): \neg (c(k) \in \mathrm{ms}(f)))$
```

scond3_1_2: THEOREM

```
 $\forall (j: \mathrm{nat}):$
   $\forall (c: \mathrm{prefix}((\mathrm{pp}(f) \cup f), j)):$
     $(c(0) \in \mathrm{Sp}(f)) \supset$
       $(\forall ((k: \mathrm{nat} \mid k < j)): \neg ((c(k), c(k+1)) \in \mathrm{sf}))$
```

scond3_2: THEOREM

```
 $\forall (c: \mathrm{Computation}(\mathrm{pp}(f))):$
   $(c(0) \in \mathrm{Sp}(f)) \supset (\forall (j: \mathrm{nat}): (c(j) \in \mathrm{Sp}(f)))$
```

scond3_3: THEOREM empty?(deadlock_states($(p \setminus \mathrm{mt}(f))$)(Sp($f$)))

scond3_3_1: THEOREM empty?(deadlock_states(pp(f))(Sp(f)))

scond3_4: THEOREM

$\forall$ (c: Computation(($\mathrm{pp}(f) \cup f$))):

$(c(0) \in \mathrm{Sp}(f)) \supset (\forall\ (j\text{: nat}): \neg\ (c(j) \in \mathrm{ms}(f)))$

scond3_5: THEOREM

$\forall$ (c: Computation(($\mathrm{pp}(f) \cup f$))):

$(c(0) \in \mathrm{Sp}(f)) \supset (\forall\ (j\text{: nat}): \neg\ ((c(j),\ c(j+1)) \in \mathrm{sf}))$

scond3_6: THEOREM

$\exists$ (T: StatePred): fault_span?(T, Sp(f), $(\mathrm{pp}(f) \cup f)$)

ccond1: LEMMA

$\forall$ ((s: state | $(s \in \mathrm{ms}(i)(f))$)):

($\exists$ ((c: Computation(f) | $c(0) = s$)):

$\exists$ (k: nat): $((c(k),\ c(k+1)) \in \mathrm{sf})$)

ccond2: LEMMA

$\forall$ ((s: state | $(s \in \mathrm{ms}(f))$)):

($\exists$ ((c: Computation(f) | $c(0) = s$)):

$\exists$ (k: nat): $((c(k),\ c(k+1)) \in \mathrm{sf})$)

ccond3: LEMMA

```
  ∀ ((t: Transition | (t ∈ mt(f)))):
    ∃ ((c: Computation(f) | c(0) = t`1)):
      ∃ (k: nat): ((c(k), c(k + 1)) ∈ sf)


ccond4: THEOREM
  ∀ (Spp: StatePred, ppp: Action):
    is_failsafe?(Spp, ppp) ⊃ (Spp ⊆ (S \ ms(f)))


ccond5: THEOREM
  ∀ (Spp: StatePred, ppp: Action):
    is_failsafe?(Spp, ppp) ⊃ (proj(ppp, Spp) ⊆ (p \ mt(f)))


ccond6: LEMMA
  (∃ (Spp: StatePred, ppp: Action): is_failsafe?(Spp, ppp)) ⊃
   nonempty?((S \ ms(f)))


completeness: THEOREM
  (∃ (Spp: StatePred, ppp: Action): is_failsafe?(Spp, ppp)) ⊃
   ¬ add_failsafe_fails?


END add_failsafe
```

Appendix D

Formal Specification of the Synthesis of Nonmasking Fault-tolerance

```
add_nonmasking[state: TYPE]: THEORY
 BEGIN

  ASSUMING
   ST_is_finite: ASSUMPTION is_finite_type[state]
   TR_is_finite: ASSUMPTION is_finite_type[[state, state]]

   IMPORTING FT[state]

  ENDASSUMING

  s_0, s_1: VAR state
```

```
Sp(anyS: StatePred): StatePred = anyS


pp(anyS: StatePred, anyp: Action): Action =
    (proj(anyp, anyS) ∪ {s_0, s_1 | ¬ (s_0 ∈ anyS) ∧ (s_1 ∈ anyS)})


scond1_1: LEMMA (Sp(S) ⊆ S)


scond1_2: LEMMA closed?(Sp(S), pp(S, p))


scond2: LEMMA (proj(pp(S, p), Sp(S)) ⊆ proj(p, Sp(S)))


scond3: LEMMA
  ∀ (c: Computation(pp(S, p))): ∃ ((j: nat | j > 0)): (c(j) ∈ Sp(S))


scond4: LEMMA
  ∀ (c: Computation((pp(S, p) ∪ f))):
    ∃ ((j: nat | j > 0)): (c(j) ∈ Sp(S))


soundness: THEOREM
  ∃ (T: StatePred): fault_span?(T, Sp(S), (pp(S, p) ∪ f))


END add_nonmasking
```

Appendix E

Formal Specification of the Synthesis of Masking Fault-tolerance

```
add_masking[state: TYPE]: THEORY
 BEGIN

  ASSUMING
   ST_is_finite: ASSUMPTION is_finite_type[state]
   TR_is_finite: ASSUMPTION is_finite_type[[state, state]]
   IMPORTING add_failsafe[state]

  ENDASSUMING

  s, s0, s1: VAR state

  S2, T2: StatePred
```

```
mask_ax: AXIOM $(S_2 \subseteq T_2)$

reachable?($S$, $T$: StatePred, $p$: Action, $s$: state): bool =
    $\exists$ ($c$: Computation($p$)):
        $(s \in T)$ $\wedge$ $s$ = $c(0)$ $\wedge$ ($\exists$ ($j$: nat): $(c(j) \in S)$)

S_init(anyFault: Action): StatePred =
    ConstructInvariant(($S \setminus$ ms(anyFault)), ($p \setminus$ mt(anyFault)))

T_init(anyFault: Action): StatePred = {$s$ | $\neg$ ($s \in$ ms(anyFault))}

TmS: Action =
    {$s_0$, $s_1$ | ($\neg$ $(s_0 \in S_2)$) $\wedge$ (($s_0 \in T_2$) $\wedge$ ($s_1 \in T_2$))}

$p_1$(anyFault: Action): Action = ((proj($p$, $S_2$) $\cup$ TmS) $\setminus$ mt(anyFault))

TmR(anyFault: Action): StatePred =
    {$s$ | ($s \in T_2$) $\wedge$ reachable?($S_2$, $T_2$, $p_1$(anyFault), $s$)}

TcL(anyFault: Action)($X$: StatePred): StatePred =
    {$s_0$ |
        $\exists$ ($s_1$):
            ($s_0 \in X$) $\wedge$ (($s_0$, $s_1$) $\in$ anyFault) $\wedge$ $\neg$ ($s_1 \in X$)}

ConstructFault_span($X$: StatePred, anyFault: Action): StatePred =
    mu_fix($X$)(TcL(anyFault))
```

T_1(anyFault: Action): StatePred =

ConstructFault_span(TmR(anyFault), anyFault)

S_1(anyFault: Action): StatePred =

ConstructInvariant(($S_2 \cap T_1$(anyFault)), p_1(anyFault))

T1_T2: LEMMA ($T_1(f) \subseteq T_2$)

S1_S2: LEMMA ($S_1(f) \subseteq S_2$)

S1_T1: THEOREM $\forall$ (f: Action): ($S_1(f) \subseteq T_1(f)$)

prop1: LEMMA disjoint?(T_2, ms(f)) $\supset$ disjoint?($T_1(f)$, ms(f))

prop2: LEMMA $\forall$ (f: Action): empty?(TcL(f)(T_1(f)))

sf_p1: LEMMA disjoint?(p_1(f), sf)

prop: LEMMA $\forall$ (f: Action): ConstructFault_span(T_1(f), f) = T_1(f)

scond1: LEMMA $\forall$ (f: Action): closed?(S_2, p_1(f))

scond2: LEMMA $\forall$ (f: Action): closed?(T_2, p_1(f))

scond3_1: THEOREM ($S_2 \subseteq S$) $\supset$ ($S_1(f) \subseteq S$)

scond3_2: THEOREM

$(\text{proj}(p_1(f),\ S_2) \subseteq \text{proj}(p,\ S_2)) \supset$

$(\text{proj}(p_1(f),\ S_1(f)) \subseteq \text{proj}(p,\ S_1(f)))$

scond4: LEMMA $\forall$ $(f\text{: Action})$: closed?$(T_1(f),\ f)$

scond4_1: THEOREM $\forall$ $(f\text{: Action})$: $(S_1(f) = S_2) \supset$ closed?$(S_1(f),\ p_1(f))$

scond4_2: THEOREM

$\forall$ $(f\text{: Action})$: $(T_1(f) = T_2) \supset$ closed?$(T_1(f),\ (p_1(f) \cup f))$

scond7: LEMMA $(T_1(f) = T_2) \supset$ closed?$(T_2,\ f)$

scond8: THEOREM

$\forall$ $(f\text{: Action})$:

$(((S_1(f) = S_2) \wedge (T_1(f) = T_2)) \supset$

$(\forall\ ((s\text{: state} \mid (s \in T_1(f))))$:

reachable?$(S_1(f),\ T_1(f),\ p_1(f),\ s)))$

scond8_2: THEOREM

$\forall$ $(f\text{: Action})$:

$(((S_1(f) = S_2) \wedge (T_1(f) = T_2)) \supset$

$(\forall\ ((s\text{: state} \mid (s \in T_1(f))))$:

reachable?$(S_1(f),\ T_1(f),\ \text{pp}(f),\ s)))$

scond3_3: THEOREM

$\forall$ $(c\text{: Computation}((p_1(f) \cup f)))$:

$(c(0) \in T_1(f)) \supset (\forall\ (j:\ \text{nat}):\ \neg\ (c(j) \in \text{ms}(f)))$

scond3_4: THEOREM

$(T_1(f)\ =\ T_2) \supset$

$(\forall\ (c:\ \text{Computation}((p_1(f) \cup f))):$

$(c(0) \in T_1(f)) \supset (\forall\ (j:\ \text{nat}):\ \neg\ ((c(j),\ c(j+1)) \in \text{sf}))$

scond3_5: THEOREM

$(T_1(f)\ =\ T_2) \supset (\exists\ (T:\ \text{StatePred}):\ \text{fault\_span?}(T,\ S_1(f),\ p_1(f)))$

scond3_6: THEOREM empty?(deadlock_states($p_1(f)$)($S_1(f)$))

scond3_8: LEMMA

$(S_1(f)\ =\ S_2\ \wedge\ T_1(f)\ =\ T_2) \supset$

$(\forall\ ((c:\ \text{Computation}((\text{pp}(f) \cup f))\ \mid\ (c(0) \in T_1(f)))):$

$\exists\ ((j:\ \text{nat}\ \mid\ j\ >\ 0)):\ (c(j) \in S_1(f)))$

scond3_9: THEOREM

$(S_1(f)\ =\ S_2\ \wedge\ T_1(f)\ =\ T_2) \supset$

$(\forall\ (j:\ \text{nat}):$

$\forall\ (c:\ \text{prefix}((p_1(f) \cup f),\ j)):$

$(c(0) \in T_1(f)) \supset$

$(\forall\ ((k:\ \text{nat}\ \mid\ k\ <\ j)):\ \neg\ (c(k) \in \text{ms}(f))))$

scond3_10: THEOREM

$(S_1(f)\ =\ S_2\ \wedge\ T_1(f)\ =\ T_2) \supset$

$(\forall\ (j:\ \text{nat}):$

```
∀ (c: prefix((pp(f) ∪ f), j)):
    (c(0) ∈ T_1(f)) ⊃
      (∀ ((k: nat | k < j)): ¬ ((c(k), c(k + 1)) ∈ sf))

END add_masking
```

Appendix F

Formal Specification of the Synthesis of Multitolerance

```
add_multift[state: TYPE]: THEORY
 BEGIN

  ASSUMING
   ST_is_finite: ASSUMPTION is_finite_type[state]
   TR_is_finite: ASSUMPTION is_finite_type[[state, state]]
   IMPORTING add_failsafe[state]
   IMPORTING add_nonmasking[state]
   IMPORTING add_masking[state]

  ENDASSUMING

  i, j: VAR nat

  s0, s1: VAR state
```

Z: VAR Action

f_failsafe: Action

f_nonmasking: Action

f_masking: Action

f_nonmasking_masking: Action = (f_masking $\cup$ f_nonmasking)

f_failsafe_masking: Action = (f_masking $\cup$ f_failsafe)

ax4: AXIOM

$\forall$ (c: Computation((Z $\cup$ f_nonmasking))):

$(\exists\ (j: \text{nat}): (\forall\ ((n: \text{nat} \mid n \geq j)): ((c(n),\ c(n+1)) \in Z)))$

Sp1: StatePred = add_masking.S_1(f_masking)

Tp1: StatePred = add_nonmasking.Sp(T_1(f_masking))

pp1: Action = add_nonmasking.pp(Tp1, p_1(f_masking))

ms: StatePred = FT.ms(f_failsafe_masking)

mt: Action = FT.mt(f_failsafe_masking)

T_masking: StatePred $=$ T_1(f_masking)

p_1: Action $=$ add_masking.p_1(f_masking)

Sp2: StatePred $=$ add_masking.S_1(f_masking)

Tp2: StatePred $=$ ConstructInvariant((T_masking $\setminus$ ms), ($p_1 \setminus$ mt))

pp2: Action $=$

```
(p1 -  mt)-
```

$(s0,\ s1)|\ ((s0,\ s1) \in (p1\ -\ mt))\ AND\ ((s0 \in Tp2)\ AND\ NOT\ (s1 \in Tp2))$

finite_fault_nonmasking: THEOREM

$\forall$ (c: Computation(($Z \cup$ f_nonmasking_masking))):

$\exists$ (j: nat):

($\forall$ (n: nat): ((suffix(c, j)(n), suffix(c, j)($n+1$)) $\in Z$))

scond3: LEMMA

$\forall$ (c: Computation(pp1)): $\exists$ ((j: nat | $j > 0$)): ($c(j) \in$ Tp1)

scond4: LEMMA

$\forall$ (c: Computation((pp1 $\cup$ f_nonmasking_masking))):

$\exists$ ((j: nat | $j > 0$)): ($c(j) \in$ Tp1)

scond1_1: THEOREM ($S_2 \subseteq S$) $\supset$ (Sp1 $\subseteq S$)

scond1_3_2: THEOREM

(proj(pp1, S_2) $\subseteq$ proj(p, S_2)) $\supset$ (proj(pp1, Sp1) $\subseteq$ proj(p, Sp1)) scond1_3_5: THEOREM $(\mathrm{Sp1} \subseteq \mathrm{Tp1})$

scond1_3_3: LEMMA

$(\text{add\_masking}.T_1(\text{f\_masking}) = T_2) \supset \text{closed?}(\mathrm{Tp1}, (\mathrm{pp1} \cup \text{f\_masking}))$

scond1_3_4: LEMMA $(\text{add\_masking}.S_1(\text{f\_masking}) = S_2) \supset \text{closed?}(\mathrm{Sp1}, \mathrm{pp1})$

scond1_8: THEOREM

$((S_1(\text{f\_masking}) = S_2) \land (T_1(\text{f\_masking}) = T_2)) \supset$

$(\forall\ ((s\text{: state} \mid (s \in \mathrm{Tp1})))\text{: reachable?}(\mathrm{Sp1}, \mathrm{Tp1}, \mathrm{pp1}, s))$

scond1_3_6: THEOREM

$\exists\ (T\text{: StatePred})\text{: fault\_span?}(T, \mathrm{Sp1}, (\mathrm{pp1} \cup \text{f\_nonmasking\_masking}))$

scond2_1: THEOREM $(S_2 \subseteq S) \supset (\mathrm{Sp2} \subseteq S)$

scond2_3_2: THEOREM

$(\mathrm{proj}(\mathrm{pp2}, S_2) \subseteq \mathrm{proj}(p, S_2)) \supset (\mathrm{proj}(\mathrm{pp2}, \mathrm{Sp2}) \subseteq \mathrm{proj}(p, \mathrm{Sp2}))$

scond2_3_3: THEOREM $(\mathrm{Sp2} = S_2) \supset \text{closed?}(\mathrm{Sp2}, \mathrm{pp2})$

scond2_3_5: THEOREM $(\mathrm{Sp2} \subseteq \mathrm{Tp2})$

scond2_3_1_1: THEOREM

$\text{disjoint?}(T_2, \mathrm{ms}) \supset$

$(\forall\ (j\text{: nat})\text{:}$

```
      $\forall$ ($c$: prefix((pp2 $\cup$ f_failsafe_masking), $j$)):
        ($c(0) \in$ Sp2) $\supset$ ($\forall$ (($k$: nat | $k < j$)): $\neg$ ($c(k) \in$ ms)))


scond2_3_1_2: THEOREM
  disjoint?($T_2$, ms) $\supset$
   ($\forall$ ($j$: nat):
      $\forall$ ($c$: prefix((pp2 $\cup$ f_failsafe_masking), $j$)):
        ($c(0) \in$ Sp2) $\supset$
         ($\forall$ (($k$: nat | $k < j$)): $\neg$ (($c(k)$, $c(k+1)$) $\in$ sf)))


scond2_3_3_1: THEOREM empty?(deadlock_states(($p_1 \setminus$ mt))(Tp2))


scond2_3_3_2: THEOREM empty?(deadlock_states(pp2)(Tp2))


scond2_8: THEOREM
  ((Sp2 = $S_2$) $\wedge$ (Tp2 = $T_2$)) $\supset$
   ($\forall$ (($s$: state | ($s \in$ Tp2))): reachable?(Sp2, Tp2, pp2, $s$))


scond2_3_6: THEOREM
  $\exists$ ($T$: StatePred): fault_span?($T$, Sp2, (pp2 $\cup$ f_failsafe_masking))


END add_multift
```

Bibliography

[1] K. Bhargavan, D. Obradovic, and C. Gunter. Formal verification of standards for distance vector routing protocols. *Journal of the ACM*, 49(4):538 – 576, 2002.

[2] S. S. Kulkarni. *Component-based design of fault-tolerance*. PhD thesis, Ohio State University, 1999.

[3] 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.

[4] 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.

[5] 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.

[6] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.

[7] 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.

[8] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 2000.

[9] 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.

[10] S. S. Kulkarni and A. Ebnenasir. Automated synthesis of multitolerance. IEEE Conference on Dependable and Network Systems *(DSN'04)*, 2004.

[11] S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. *International Conference on Distributed Computing Systems*, 2003.

[12] Ali Ebnenasir and Sandeep S. Kulkarni. A framework for automatic synthesis of fault-tolerance. `http://www.cse.msu.edu/~sandeep/software/Code/synthesis-framework/`.

[13] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[14] 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.

[15] 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.

[16] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide, Version 2.4*. Computer Science Laboratory, SRI International, Menlo Park, CA, December 2001. URL: http://pvs.csl.sri.com/manuals.html.

[17] 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.

[18] H. P. Barendregt. *Handbook of Theoretical Computer Science: Volume B, Chapter 7, Functional Programming and Lambda Calculus*. Elsevier Science Publishers B. V., 1990.

[19] S. S. Kulkarni and A. Ebnenasir. The complexity of adding failsafe fault-tolerance. *International Conference on Distributed Computing Systems*, 2002.

[20] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

[21] Tomoyuki Yokogawa, Tatsuhiro Tsuchiya, and Tsuchiya Kikuno. Automatic verification of fault tolerance using model checking. *International Symposium on Dependable Computing*, 2001.

[22] Wilfried Steiner and John Rushby. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. IEEE Conference on Dependable and Network Systems *(DSN'04)*, 2004.

[23] John Rushby. Sal tutorial: Analyzing the fault-tolerant algorithm om(1). Technical report, CSL, SRI International, 2004.

[24] F. Schneider, S. M. Easterbrook, J. R. Callahan, and G. J. Holzmann. Validating requirements for fault tolerant systems using model checking. *Third IEEE Conference on Requirements Engineering*, 1998.

[25] P. Ramanathan and K. G. Shin. Use of common time base for checkpointing and rollback recovery in distributed systems. *IEEE Transactions on Software Engineering*, 19:571–583, 1993.

[26] Jean-Charles Fabre, Thomas Losert, Eric Marsden, Nick Moffat, Michael Paulitsch, David Powell, and William Simmonds. Validation of fault tolerance and timing properties. Project Deliverable DSC2 for DSoS, Research Report 20/2003, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2003.

[27] Ali Ebnenasir. Automatic synthesis of distributed programs: A survey. 2002. http://www.cse.msu.edu/~ebnenasi/survey.pdf.

[28] N. S. Bjorner. A servey of reactive synthesis. 1996. http://theory.stanford.edu/people/nikolaj/dimacs96.ps.

[29] E. A Emerson. *Handbook of Theoretical Computer Science: Chapter 16, Temporal and Modal Logics.* Elsevier Science Publishers B. V., 1990.

[30] 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.

[31] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[32] 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.

[33] 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.

[34] Paul C. Attie and E. Allen Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(2):187 – 242, 2001.

[35] 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.

[36] W. Thomas. *Handbook of Theoretical Computer Science: Chapter 4, Automata on Infinite Objects.* Elsevier Science Publishers B. V., 1990.

[37] A. Pnueli and R. Rosner. On the synthesis of a reactive module. *In Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 179–190, 1989.

[38] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. *In Proceeding of 16th International Colloqium on Automata, Languages, and Programming*, Lec. Notes in Computer Science 372, Springer-Verlag:652–671, 1989.

[39] O. Kupferman and M.Y. Vardi. Synthesizing distributed systems. In *Proc. 16th IEEE Symp. on Logic in Computer Science*, July 2001.

[40] M. Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. *Computer Aided Verfification, volume 939 of LNCS,*, pages 267–278, 1995.

[41] A. Arora. *A foundation of fault-tolerant computing.* PhD thesis, The University of Texas at Austin, 1992.

[42] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of Byzantine agreement. *Symposium on Reliable Distributed Systems*, 2001.

[43] 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.

[44] P.S. Miner and S.D. Johnson. Verification of an optimized fault-tolerant clock synchronization circuit. *Third Workshop on Designing Correct Circuits (DCC96)*, 1996.

[45] *NASA Langley Research Center Formal Methods Group. SPIDER homepage.* `http://shemesh.larc.nasa.gov/fm/spider/`.

[46] Alfons Geser and Paul S. Miner. A formal correctness proof of the SPIDER diagnosis protocol. In *Theorem Proving in Higher-Order Logics (TPHOLs) – Track B Proceedings*, number NASA/CP-2002-211736, pages 71–86, NASA Langley Research Center, Hampton, VA, August 2002.

[47] T.F. Melham. Abstraction mechanisms for hardware verification. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 129–157, Boston, 1988. Kluwer Academic Publishers.

[48] Ben L. Di Vito and Ricky W. Butler. Formal techniques for synchronized fault-tolerant systems. *3rd IFIP Working Conference on Dependable Computing for Critical Applications*, 1992.

[49] John Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–661, 1999.

[50] Natarajan Shankar. Mechanical verification of a generalized protocol for byzantine fault tolerant clock synchronization. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 217–236. Springer-Verlag, 1992.

[51] Patrick Lincoln and John Rushby. The formal verification of an algorithm for interactive consistency under a hybrid fault model. In Costas Courcoubetis, editor, *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, pages 292–304, Elounda, Greece, June/July 1993. Springer-Verlag.

[52] John Rushby. Formal verification of an oral messages algorithm for interactive consistency. Technical Report CSL-91-01, SRI International, 1992.

[53] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, (1):5–6, 1986.

[54] Lee Pike, Jeffery Maddalon, Paul Miner, and Alfons geser. Abstraction for fault-tolerant distributed system verification. *Theorem Proving in Higher-Order Logics (TPHOLs)*, 2004.

[55] S. S. Kulkarni, Borzoo Bonakdarpour, and Ali Ebnenasir. Mechanical verification of automatic synthesis of fault-tolerant programs. *International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)*, 2004.

[56] S. S. Kulkarni, Borzoo Bonakdarpour, and Ali Ebnenasir. Mechanical verification of automatic synthesis of failsafe fault-tolerance. *Theorem Proving in Higher-Oreder Logics (TPHOLs) Emerging Trends*, 2004.