Automating the Addition of Fault-Tolerance

Sandeep S. Kulkarni
Department of Computer
Science and Engineering
Michigan State University
East Lansing MI 48824 USA

Anish Arora
Department of Computer
and Information Science
Ohio State University
Columbus Ohio 43210 USA

Abstract

In this paper, we focus on automating the transformation of a given fault-intolerant program into a fault-tolerant program. We show how such a transformation can be done for three levels of fault-tolerance properties, failsafe, nonmasking and masking. For the high atomicity model where the program can read all the variables and write all the variables in one atomic step, we show that all three transformations can be performed in polynomial time in the state space of the fault-intolerant program. For the low atomicity model where restrictions are imposed on the ability of programs to read and write variables, we show that all three transformations can be performed in exponential time in the state space of the fault-intolerant program. We also show that the the problem of adding masking fault-tolerance is NP-hard and, hence, exponential complexity is inevitable unless $P = NP$.

1 Introduction

In this paper, we focus on automating the transformation of a fault-intolerant program into a fault-tolerant program. The motivations behind this work are multi-fold. The first motivation comes from the fact that the designer of a fault-tolerant program is often aware of a corresponding fault-intolerant program that is known to be correct in the absence of faults. Or, the designer may be able to develop a fault-intolerant program and its manual proof in a simple way. In these cases, it is expected that the designer will benefit from reusing that fault-intolerant program rather than starting from scratch. Moreover, the reuse of the fault-intolerant program will be virtually mandatory if the designer has only an incomplete specification.

The second motivation is that the use of such automated transformation will obviate the need for manually constructing the proof of correctness of the synthesized fault-tolerant program as the synthesized program will be correct by construction. This advantage is especially useful when designing concurrent and fault-tolerant programs as

[1] Email: sandeep@cse.msu.edu, anish@cis.ohio-state.edu. Web: `http://www.cse.msu.edu/~sandeep`, `http://www.cis.ohio-state.edu/~anish`. Tel: +1-517-355-2387. Arora is currently on sabbatical leave at Microsoft Research. This work was partially sponsored by NSA Grant MDA904-96-1-0111, NSF Grant NSF-CCR-9972368, an Ameritech Faculty Fellowship, a grant from Microsoft Research, and a grant from Michigan State University.

it is well-understood that manually constructing proofs of correctness for such programs is especially hard.

The third motivation stems from our previous work [1–3] that shows that a fault-tolerant program can be expressed as a composition of a fault-intolerant program and a set of 'fault-tolerance components'. The fault-intolerant program is responsible for ensuring that the fault-tolerant program works correctly in the absence of faults; it plays no role in dealing with fault-tolerance. The fault-tolerance components are responsible for ensuring that the fault-tolerant program deals with the faults in accordance to the level of tolerance desired; they play no role in ensuring that the program works correctly in the absence of faults. We have also found that the fault-tolerance components help in manually designing fault-tolerant programs [2, 3] as well as in manually constructing their proofs [3, 4]. Moreover, we have found that programs designed using fault-tolerance components are easier to understand and have a better structure [3] than programs designed from scratch.

The third motivation suggests that given a fault-intolerant program p, we should focus on transforming it to obtain a fault-tolerant program p' such that the transformation is done solely for the purpose of dealing with faults according to the level of fault-tolerance desired. More specifically, it suggests that p' should not introduce new ways to satisfy the specification in the absence of faults.

We study the problem of transforming a fault-intolerant program into a fault-tolerant program for three levels of fault-tolerance properties, namely, failsafe, nonmasking and masking. Intuitively, a failsafe fault-tolerant program only satisfies the safety of its specification, a nonmasking fault-tolerant program recovers to a state from where its subsequent computation is in the specification, and a masking fault-tolerant program satisfies the specification even in the presence of faults. (See Section 2 for precise definitions.)

For each of the three levels of fault-tolerance properties, we study the transformation problem in the context of two models; the high atomicity model and the low atomicity model. In the high atomicity model, the program can read and write all its variables in one atomic step. In the low atomicity model, the program consists of a set of processes, and the model specifies restrictions on the ability of processes to atomically read and write program variables. Thus, the transformation problem in the low atomicity model requires us to derive a fault-tolerant program that respects the restrictions imposed by the low atomicity model.

The main contributions are as follows: (1) For the high atomicity model, we present a sound and complete algorithm that solves the transformation problem. The complexity of our algorithm is polynomial in the state space of the fault-intolerant program (cf. Section 4). (2) For the low atomicity model, we present a sound and complete algorithm that solves the transformation problem. The complexity of our algorithm is exponential in the state space of the fault-intolerant program (cf. Section 5.1). (3) We also show that for the low atomicity model, the problem of transforming a fault-intolerant program into a masking fault-tolerant program is NP-hard. It follows that there is no sound and complete polynomial algorithm to solve the problem of adding masking fault-tolerance unless $P = NP$ (cf. [**?**]).

Organization of the paper. This paper is organized as follows: We provide the definitions of programs, specifications, faults and fault-tolerance in Section 2. Using these definitions, we state the transformation problem in Section 3. In Section 4, we show how the transformation problem is solved in the high atomicity model. In Section 5, we show how to characterize the low atomicity model and how sketch our algorithm for the low atomicity model. Finally, we discuss related work in Section 6 and make

concluding remarks in Section 7. (For reasons of space, in [?], we show that the problem of transforming a fault-intolerant program into a masking fault-tolerant program is NP-hard. Also, we refer the reader to [?] for examples of programs constructed using our algorithms)

2 Programs, Specifications Faults, and Fault-Tolerance

In this section, we give formal definitions of programs, problem specifications, faults, and fault-tolerance. The programs are specified in terms of their state space and their transitions. The definition of specifications is adapted from Alpern and Schneider [5]. And, the definition of faults and fault-tolerances is adapted from Arora and Gouda [6] and Arora and Kulkarni [1].

2.1 Program

Definition. A program p is a tuple $\langle S_p, \delta_p \rangle$ where S_p is a finite set of states, and δ_p is a subset of $\{(s_0, s_1) : s_0, s_1 \in S_p\}$. □

Definition (*State predicate*). A state predicate of $p(= \langle S_p, \delta_p \rangle)$ is any subset of S_p. □

Notation. A state predicate S is true in state s iff $s \in S$.

Definition (*Projection*). Let $p(= \langle S_p, \delta_p \rangle)$ be a program, and let S be a state predicate of p. We define the projection of p on S, denoted as $p|S$, as the program $\langle S_p, \{(s_0, s_1) : (s_0, s_1) \in \delta_p \ \wedge \ s_0, s_1 \in S\} \rangle$. □

Note that $p|S$ consists of transitions of p that start in S and end in S.

Definition (*Subset*). Let $p(= \langle S_p, \delta_p \rangle)$ and $p'(= \langle S'_p, \delta'_p \rangle)$ be programs. We say $p' \subseteq p$ iff $S'_p = S_p$ and $\delta'_p \subseteq \delta_p$. □

Definition (*Closure*). A state predicate S is closed in a set of transitions δ_p iff $(\forall (s_0, s_1) : (s_0, s_1) \in \delta_p : (s_0 \in S \Rightarrow s_1 \in S))$. □

Definition. (*Computation*). A sequence of states, $\langle s_0, s_1, ... \rangle$, is a computation of $p(= \langle S_p, \delta_p \rangle)$ iff the following two conditions are satisfied:

- $\forall j : j > 0 : (s_{j-1}, s_j) \in \delta_p$
- if $\langle s_0, s_1, ... \rangle$ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in \delta_p$. □

Notation. We call δ_p as the transitions of p. When it is clear from context, we use p and δ_p interchangeably, e.g., we say that a state predicate S is closed in $p(= \langle S_p, \delta_p \rangle)$ to mean that S is closed in δ_p.

2.2 Specification

Definition. A specification is a set of infinite sequences of states that is suffix closed and fusion closed. Suffix closure of the set means that if a state sequence σ is in that set then so are all the suffixes of σ. Fusion closure of the set means that if state sequences $\langle \alpha, x, \gamma \rangle$ and $\langle \beta, x, \delta \rangle$ are in that set then so are the state sequences $\langle \alpha, x, \delta \rangle$ and $\langle \beta, x, \gamma \rangle$, where 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 [5], it can be shown that any specification is the intersection of some "safety" specification that is suffix closed and fusion closed and some "liveness" specification. Intuitively, the safety specification identifies a set of bad prefixes. A sequence is in the safety specification iff none of its prefixes are identified

as bad prefixes. Intuitively, a liveness specification requires that any finite sequence be extensible in order to satisfy that liveness specification. Formally,

Definition (*Safety*). A safety specification is a set of state sequences that meets the following condition: for each state sequence σ not in that set, there exists a prefix α of σ, such that for all state sequences β, $\alpha\beta$ is not in that set (where $\alpha\beta$ denotes the concatenation of α and β). □

Definition (*Liveness*). A liveness specification is a set of state sequences that meets the following condition: for each finite state sequence α there exists a state sequence β such that $\alpha\beta$ is in that set. □

Notation. Let *spec* be a specification. We use the term 'safety of *spec*' to mean the smallest safety specification that includes *spec*.

Note that the synthesis algorithm must be provided with a specification that is described in finite space. To simplify further presentation, however, we have defined specifications to contain infinite sequences of states. A concise representation of these infinite sequences is given in Section 2.6.

2.3 Program Correctness with respect to a Specification

Let *spec* be a specification.

Definition (*Refines*). p refines *spec* from S iff (1) S is closed in p, and (2) Every computation of p that starts in a state where S is true is in *spec*. □

Definition (*Maintains*). Let α be a finite sequence of states. The prefix α maintains *spec* iff there exists a sequence of states β such that $\alpha\beta \in$ *spec*. □

Definition (*Violates*). Let α be a finite sequence of states. The prefix α violates *spec* iff it is not the case that α maintains *spec*. □

Notation. We say that p maintains *spec* from S iff S is closed in p and every computation prefix of p that starts in a state in S maintains *spec*. We say that p violates *spec* from S iff it is not the case that p refines *spec* from S.

Definition (*Invariant*). S is an invariant of p for *spec* iff $S \neq \{\}$ and p refines *spec* from S. □

Notation. Henceforth, whenever the specification is clear from the context, we will omit it; thus, "S is an invariant of p" abbreviates "S is an invariant of p for *spec* ".

2.4 Faults

The faults that a program is subject to are systematically represented by transitions. We emphasize that such representation is possible notwithstanding the type of the faults (be they stuck-at, crash, fail-stop, omission, timing, performance, or Byzantine), the nature of the faults (be they permanent, transient, or intermittent), or the ability of the program to observe the effects of the faults (be they detectable or undetectable).

Definition (*Fault*). A fault for $p(= \langle S_p, \delta_p \rangle)$ is a subset of $\{(s_0, s_1) : s_0, s_1 \in S_p\}$. □

For the rest of the section, let *spec* be a specification, T be a state predicate, S an invariant of p, and f a fault for p.

Definition (*Computation in the presence of faults*). A sequence of states, $\langle s_0, s_1, ... \rangle$, is a computation of $p(= \langle S_p, \delta_p \rangle)$ in the presence of f iff the following three conditions are satisfied:

- $\forall j : j \geq 0 : (s_{j-1}, s_j) \in (\delta_p \cup f)$.
- If $\langle s_0, s_1, ... \rangle$ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in \delta_p$, and
- $\exists n : n \geq 0 : (\forall j : j > n : (s_{j-1}, s_j) \in \delta_p)$. □

Notation. For brevity, we use '$p[]f$' to mean 'p in the presence of f'. More specifically, a sequence is a computation of '$p[]f$' iff it is a computation of 'p in the presence of f'. And, the transitions of $p[]f$ are obtained by taking the union of the transitions of p and the transitions of f.

Definition (*Fault-span*). T is an f-span of p from S iff $S \Rightarrow T$ and T is closed in $p[]f$. □

Thus, at each state where an invariant S of p is true, and an f-span T of p from S is also true. Also, T, like S, is also closed in p. Moreover, if any action in f is executed in a state where T is true, the resulting state is also one where T is true. It follows that for all computations of p that start at states where S is true, T is a boundary in the state space of p up to which (but not beyond which) the state of p may be perturbed by the occurrence of the actions in f.

Notation. Henceforth, whenever the program p is clear from the context, we will omit it; thus, "S is an invariant" abbreviates "S is an invariant of p" and "f is a fault" abbreviates "f is a fault for p".

2.5 Fault-Tolerance

In the absence of faults, a program should refine its specification. In the presence of faults, however, it may refine a weaker version of the specification as determined by the level of tolerance provided. With this notion, we define three levels of fault-tolerance below.

Definition (*failsafe f-tolerant for spec from S*). p is failsafe f-tolerant to *spec* from S iff (1) p refines *spec* from S, and (2) there exists T such that T is an f-span of p from S and $p[]f$ maintains *spec* from T. □

Definition (*nonmasking f-tolerant for spec from S*). p is nonmasking f-tolerant to *spec* from S iff (1) p refines *spec* from S, and (2) there exists T such that T is an f-span of p from S and every computation of $p[]f$ that starts from a state in T has a state in S. □

Definition (*masking f-tolerant for spec from S*). p is masking f-tolerant to *spec* from S iff the following two conditions hold:

- p refines *spec* from S, and
- there exists T such that T is an f-span of p from S, $p[]f$ maintains *spec* from T, and every computation of $p[]f$ that starts from a state in T has a state in S. □

Notation. In the sequel, whenever the specification *spec* and the invariant S are clear from the context, we omit them; thus, "masking f-tolerant" abbreviates "masking f-tolerant for *spec* from S", and so on.

2.6 Observations on Programs and Specifications

In this section, we summarize observations about our programs and specifications. Subsequently, we present the form in which specifications are given to the synthesis algorithm.

Note that a specification, say *spec*, is a set of infinite sequences of states. If p refines *spec* from S then all computations of p that start from a state in S are in *spec* and, hence, all computations of p that start from a state in S must be infinite. Using the same argument, we make the following two observations.

Observation 2.1 If p' is (failsafe, nonmasking or masking) f-tolerant for *spec* from S' then all computations of p' that start from a state in S' must be infinite. □

Observation 2.2 If p' is (nonmasking or masking) f-tolerant for *spec* from S' then all computations of $p'[]f$ that start from a state in S' must be infinite. □

Observe that we do not disallow fixed-point computations; we simply require that if s_0 is a fixed-point of p then the transition (s_0, s_0) should be included in the transitions of p.

Concise Representation for Specifications. Recall that a safety specification identifies a set of bad prefixes that should not occur in program computations. For fusion closed and suffix closed specifications, we can focus on only prefixes of length 2. In other words, if we have a prefix $\langle \alpha, s_0 \rangle$ that maintains *spec* then we can determine whether an extended prefix $\langle \alpha, s_0, s_1 \rangle$ maintains *spec* by focusing on the transition (s_0, s_1), and ignoring α. Formally we state this in Lemma 2.3 as follows (cf. [3] for proof.):

Lemma 2.3. Let α be finite sequence of states, and let *spec* be a specification.
If $\langle \alpha, s_0 \rangle$ maintains *spec*
Then $\langle \alpha, s_0, s_1 \rangle$ maintains *spec* iff $\langle s_0, s_1 \rangle$ maintains *spec*. □

From Lemma 2.3, it follows that the safety specification can be concisely represented by the set of 'bad transitions'. For simplicity, we assume that for a given *spec* and a state space S_p, the set of bad transitions corresponding to the minimal safety specification that includes *spec* are given. If this is not the case and *spec* is given in terms of a temporal logic formula, the set of bad transitions can be computed in polynomial time by considering all transitions (s_0, s_1), where $s_0, s_1 \in S_p$.

Our proof that a fault-tolerant program refines the liveness specification solely depends on the fact that the fault-intolerant program refines the liveness specification. Therefore, our algorithm can transform a fault-intolerant program into a fault-tolerant program even if the liveness specification is unavailable.

3 Problem Statement

In this section, we formally specify the problem of deriving a fault-tolerant program from a fault-intolerant program. We first intuitively characterize what it means for a fault-tolerant program p' to be derived from a fault-intolerant program p. We use this characterization to precisely state the transformation problem. Finally, we also discuss the soundness and completeness issues in the context of the transformation problem.

Now, we consider what it means for a fault-tolerant program p' to be derived from p. As mentioned in the introduction, our derivation is based on the premise that p' is obtained by adding fault-tolerance alone to p, i.e., p' does not introduce new ways of refining *spec* when no faults have occurred. We precisely state this concept based on the following two observations: (1) If S' contains states that are not in S then, in the absence of faults, p' will include computations that start outside S. Since p' refines *spec* from S', it would imply that p' is using a new way to refine *spec* in the absence of faults (since p refines *spec* only from S). Therefore, we require that $S' \subseteq S$ (equivalently $S' \Rightarrow S$). (2) If $p'|S'$ contains a transition that is not in $p|S'$, p' can use this transition in order to refine *spec* in the absence of faults. Since this was not permitted in p, we require that $p'|S' \subseteq p|S'$. Thus, we define the transformation problem as follows (This definition will be instantiated for failsafe, nonmasking and masking f-tolerance):

The Transformation Problem
Given p, S, *spec* and f such that p refines *spec* from S
Identify p' and S' such that
$S' \Rightarrow S$,
$p'|S' \subseteq p|S'$, and
p' is f-tolerant to *spec* from S'. □

We also define the corresponding decision problem as follows:(This definition will also be instantiated for failsafe f-tolerance, nonmasking f-tolerance and masking f-tolerance):

The Decision Problem

Given p, S, *spec* and f such that p refines *spec* from S

Does there exist p' and S' such that

$S' \Rightarrow S$,

$p'|S' \subseteq p|S'$, and

p' is f-tolerant to *spec* from S'? □

Notations. Given a fault-intolerant program p, specification *spec*, invariant S and faults f, we say that program p' and predicate S' solve the transformation problem for a given input iff p' and S' satisfy the three conditions of the transformation problem. We say p' (respectively S') solves the transformation problem iff there exists S' (respectively p') such that p', S' solve the transformation problem.

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

4 Adding Fault-Tolerance in High Atomicity Model

In this section, we consider the transformation problem for programs in the high atomicity model, where a program transition can read any number of variables as well as update any number of variables in one atomic step. In other words, if the enumerated states of the program are $s_0, s_1, ...s_{max}$ then the program transitions can be any subset of $\{(s_j, s_k) : 0 \leq j \leq max\}$. We present our algorithm for adding failsafe, nonmasking and masking fault-tolerance in Sections 4.1, 4.2, and 4.3 respectively.

4.1 Problem of Designing Failsafe Tolerance

As shown in Section 2, the safety specification identifies a set of bad transitions that should not occur in program computations. Given a bad transition (s_0, s_1), we consider two cases: (1) (s_0, s_1) is not a transition of f, (2) (s_0, s_1) is a transition of f.

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

For case (2), we cannot remove the transition (s_0, s_1) as it would mean removing a fault transition. Therefore, we must ensure that $p'[]f$ never reaches the state s_0. In other words, for all states s, the transition (s, s_0) must be removed in obtaining p'. Also, if any of these removed transitions, say (s'_0, s_0), is a fault transition then we must recursively remove all transitions of the form (s, s'_0) for each state s.

Using the above two cases, our algorithm to obtain the failsafe fault-tolerant program is as follows: it first identifies states from where execution of one or more fault transitions violates safety. Then, it removes transitions of p that reach these states as well as transitions of p that violate the safety of *spec*. If there exist states in the invariant such that execution of one or more fault actions from those states violates the safety of *spec*, then we recalculate the invariant by removing those states. The detailed algorithm is in the Appendix (As mentioned in Section 2, we use a program and its transitions interchangeably.):

4.2 Problem of Designing Nonmasking Tolerance

To design a nonmasking f-tolerant program p', we ensure that if p is perturbed by faults in f then it eventually recovers to a state in S. To obtain the nonmasking f-tolerant program, for each state s_0, $s_0 \notin S$, we add a transition (s_0, s_1) such that $s_1 \in S$. Our algorithm for synthesizing nonmasking f-tolerant program is in Appendix.

4.3 Problem of Designing Masking Tolerance

To design a masking f-tolerant program p', we proceed to identify the weakest invariant S' (which is stronger than S) and the weakest fault-span T'. As argued in Theorem 4.4, our first estimate of S' is S_1 where S_1=ConstructInvariant$(S - ms, p - mt)$. And, we estimate T' to be T_1 where $T_1 = true - ms$, i.e., T_1 includes all states except those in ms.

We continue to strengthen our S_1 and T_1 while ensuring that if some S' solves the transformation problem then $S' \Rightarrow S_1$. We first identify and remove states in T_1 from where it is not possible to reach a state in S_1 without violating the safety of *spec*. We then find the largest subset of the remaining states that is closed in f. This represents the new estimate for fault-span. Since S_1 must be a subset of T_1, we recalculate S_1 to be the largest subset of $S_1 \wedge T_1$ such that all the computations from that subset are infinite. We continue this process until we reach a fixpoint. Our algorithm for adding masking f-tolerance is in the Appendix.

5 Adding Fault-Tolerance in Low Atomicity Model

The synthesis algorithm in Section 4 assumes that the fault-tolerant program can contain a transition (s_0, s_1) for any two states s_0, s_1. If we think of the program state to consist of variables and their corresponding values, the synthesis algorithm assumes that the program can read the values of all variables and write the values of all variables in an atomic step. In this section, we first describe how a low atomicity model that imposes restrictions on how processes can read and write variables. Then, we will outline our algorithm in Section 5.1

We assume that the program consists of processes; each process can atomically read a subset of the program variables and write (a possibly different) set of variables. To systematically use these read/write restrictions imposed by the model, we now define what it means for a process to read and write a variable. First, we define the following two notations.

Notation. Let x be a variable. $x(s_0)$ denotes the value of variable x in state s_0.

Notation. Let r_j denote the set of variables j is allowed to read and w_j denote the set of variables that j is allowed to write.

For simplicity, we assume that j can *atomically* read all variables in r_j and write all variables in w_j. If this is not the case, we split process j into multiple processes that satisfy this assumption. We leave it to the reader to verify that this can always be done.

Remark. Note that the above restrictions are for the program actions only. Faults are not restricted in any way, i.e., a fault transition can read and write all the variables in one atomic step.

Write-restrictions. If j can only write the subset of variables w_j and the value of a variable other than that in w_j is changed in the transition (s_0, s_1) then that transition cannot be used in synthesizing the transitions of j. In other words, being able to write the subset w_j is equivalent to providing a set of transitions $write(j, w_j)$ that j cannot use synthesis algorithm, where

$$write(j, w_j) = \{(s_0, s_1) : (\exists x : x \notin w_j : x(s_0) \neq x(s_1))\}$$

Read-restrictions. Initially, we consider the case where $w_j \subseteq r_j$, i.e., j can write a variable only if it can read it. Let (s_0, s_1) be some transition of process j such that $s_0 \neq s_1$. Now, consider a state s'_0 such that the values of all variables in r_j are identical to that in s_0. Since j can only read variables in r_j, j must have a transition of the form (s'_0, s'_1). Moreover, the values of variables in r_j in s'_1 must be the same as that in s_1. And, since $w_j \subseteq r_j$, the values of variables that are not in r_j must be the same as that in s'_0. Considering all states where the values of r_j are same, we get a group of transitions; if (s_0, s_1) is a transition of j then all transitions in that group must also be transitions of j. We define these transitions as $group(j, r_j)(s_0, s_1)$, for the case where $w_j \subseteq r_j$, where

$$\begin{aligned} group(j, r_j)(s_0, s_1) = \{(s'_0, s'_1) : & (\forall x : x \in r_j : x(s_0) = x(s'_0) \ \wedge x(s_1) = x(s'_1)) \ \wedge \\ & (\forall x : x \notin r_j : x(s'_0) = x(s'_1) \ \wedge \ x(s_0) = x(s_1)) \ \} \end{aligned}$$

Now, we consider the case where $w_j \not\subseteq r_j$, i.e., j writes variables without reading them. To motivate such cases, consider the following scenario: Let $chan_j$ denote the sequence of messages on channel $chan$ which is an outgoing channel from process j. When j sends a message, it writes $chan_j$. However, j cannot read what messages are still pending on channel $chan$, i.e., j cannot read $chan_j$. When j updates $chan_j$, the new value of $chan_j$ depends upon the initial state of the program (including the initial value of $chan_j$). In other words, there exists a function f_{chan_j} such that when j executes in state s_0, j assigns the value $f_{chan_j}(s_0)$ to $chan_j$.

More generally, if j can write multiple variables, say $x_1, x_2, ...$, without being able to read any of them, the model provides a function f (or polynomial number of different functions) such that when j executes in state s_0, j assigns the value $x_i(f(s_0))$ to variable x_i. Using f (or for each possible function f), we now define a group of transitions, $group(j, f, r_j)(s_0, s_1)$, where

$$\begin{aligned} group(j, f, r_j)(s_0, s_1) = \{(s'_0, s'_1) : & (\forall x : x \in r_j : x(s_0) = x(s'_0) \ \wedge x(s_1) = x(s'_1)) \ \wedge \\ & (\forall x : x \notin r_j : x(s'_1) = x(f(s'_0)) \ \wedge \ x(s_1) = x(f(s_0))) \ \} \end{aligned}$$

Remark. The above grouping is done for the case where the transition is not a self-loop. Regarding the self-loop, there are no restrictions. We model this by introducing a group (s_0, s_0) for each state s_0. Note, however, given a program p with invariant S, the masking (respectively, nonmasking) fault-tolerant program p' can contain a self-loop only if it is in $p|S$.

Combining read-restrictions and write-restrictions. The inability of a process to read is characterized in terms of grouping of transitions. Thus, if a transition in some group violates the restrictions imposed by the inability to write, then that entire group must be excluded in the design of fault-tolerant program. It follows that after combining the read restrictions and the write-restrictions, we get another grouping of transitions; we need to choose zero or more such groups to obtain the transitions of that process. Moreover, the time to compute these groups is polynomial in the size of the input. Thus, we have

Observation 5.1 The groups of transitions corresponding to the given fault-intolerant program and the low atomicity model describing the processes (with the restriction on their ability to read and write) can be computed in polynomial time. □

Typically, the state space of the fault-intolerant program is much larger than the description of the low atomicity model. Therefore, the time required to compute the groups is polynomial in the state space of the fault-intolerant program. Now, using these groups, we describe our algorithm, next.

5.1 Algorithm Sketch

Now, we sketch our algorithm for adding fault-tolerance in the low atomicity model. Our algorithm is in NP and, hence, the complexity of the corresponding (brute-force) deterministic algorithm is at most exponential. Being in NP, we simply guess the solution, namely, the invariant S', the fault-span T', and the groups of transitions which would be included in the fault-tolerant program p'. Subsequently, we verify that the three conditions of the transformation problem are satisfied. In this verification, the first two conditions can be easily verified in polynomial time. The third condition about f-tolerance is verified by using T' as the fault-span. We leave it to the reader to verify that time required to compute the guess as well as the time for verification is polynomial. (For reasons of space, we relegate the detailed algorithm to [?].)

6 Related Work

Previous work [7–9] on program synthesis has addressed the problem of synthesizing a program that satisfies a given specification. These synthesis algorithms are based on the decision procedure for testing temporal satisfiability proposed by Emerson and Clarke [7] and Manna and Wolper [8]. Previous work on synthesizing fault-tolerant programs includes [9–12]. In [9, 12], the authors assume that the fault cannot affect the internal state of a process; the fault can interact only through input and output signals. By way of contrast [11] and this paper permits a more general model of faults, where faults can affect the all variables of a process. Also, our low atomicity model is more general than the Read/Write model used in [11, 13].

In terms of complexity, our low atomicity algorithm as well as that of the algorithms in [9, 13] is exponential in the state space of the fault-tolerant program. Although the algorithm in [11] is polynomial in the state space of the fault-intolerant program, it is incomplete.

7 Conclusion and Future Work

In this paper, we focused on transforming a fault-intolerant program into a fault-tolerant program. We considered three levels of fault-tolerance, namely failsafe, nonmasking and masking. We showed that in the high atomicity model, where the program can read and write all the variables in one atomic step, all these transformations can be performed in polynomial time in the size of the fault-intolerant program. We also showed that in the low atomicity model, where the program consists of processes each of which can only read and write only a limited set of variables, all these transformations can be performed in exponential time in the size of the fault-intolerant program. Moreover, we demonstrated that in the low atomicity model the problem of adding masking fault-tolerance to a given fault-intolerant program is NP-hard. For reasons of space, discussion about examples of programs that can be designed using these algorithms, namely, triple modular redundancy, byzantine agreemnt and token ring circulation is relegated to [?].

We would like to note that our low atomicity model is more general than the Read/Write model considered elsewhere [11, 13] in the literature. More specifically, the Read/Write model is an instance of our low atomicity model (cf. Section 5).

Unlike previous work that starts with a specification (typically in some temporal logic), we start with a fault-intolerant program that is known to be correct. For this reason, we believe that this approach will be especially useful if a fault-intolerant program is already known or if other constraints (such as unavailability of a complete specification of the given fault-intolerant program) require that we reuse the fault-intolerant program. Also, due to the same reason, our algorithms only needed the

safety specification that the program is supposed to satisfy in the presence of faults; the algorithms did not need the liveness specification.

Regarding extensions to our work, we will be focusing on reducing the complexity of our algorithms by identifying heuristics. These heuristics will be specially tailored according to the type of fault being considered. By focusing on only a limited types of faults, it may be possible to solve the transformation problem in polynomial time.

Another extension of this work is to precompute a set of fault-tolerance components which often occur in fault-tolerant programs and use these components directly in the synthesis algorithm. We expect that these precomputed fault-tolerance components will not only help in reducing the complexity of designing fault-tolerant programs but also permit the derived fault-tolerant program to be more efficient.

References

1. A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *International Conference on Distributed Computing Systems*, pages 436–443, May 1998. An extended version of this paper has been invited to *IEEE Transactions on Computers*.
2. A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.
3. S. S. Kulkarni. *Component-based design of fault-tolerance*. PhD thesis, Ohio State University, 1999.
4. 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.
5. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
6. 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.
7. E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synchronize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.
8. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6:68–93, 1984.
9. A. Pnueli and R. Rosner. On the synthesis of a reactive module. *ACM Symposium on Principles of Programming Languages*, pages 179–190, 1989.
10. A. Anuchitanukul and Z. Manna. Reliability and synthesis of reactive modules. *International Conference on Computer-Aided Verification*, pages 156–169, 1994.
11. 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.
12. O. Kupferman and M. Vardi. Synthesis with incomplete information. *ICTL*, 1997.
13. D. Dill and H. Wong-Toi. Synthesizing processes and schedulers from temporal specifications. *International Conference on Computer-Aided Verification*, 1990.

```
Add_failsafe(p, f : transitions, S : state predicate, spec : specification)
{
    ms := {s0 : ∃s1, s2, ...sn : (∀j : 0≤j<n : (sj, s(j+1)) ∈ f)   ∧   (s(n−1), sn) violates spec };
    mt := {(s0, s1) : ((s1 ∈ ms) ∨ (s0, s1) 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
    { while (∃s0 : s0 ∈ S : (∀s1 : s1 ∈ S : (s0, s1) ∉ p)) S := S − {s0}   }

ConstructTransitions(p : transitions, S : set of states)
    { return p−{(s0, s1) : s0 ∈ S  ∧  s1 ∉ S} }
```

Theorem 4.3 The algorithm *Add_failsafe* is sound, complete and in P. □

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

Theorem 4.6 The algorithm *Add_nonmasking* is sound, complete, and in P. □

```
Add_masking(p, f : transitions, S : state predicate, spec : specification)
{
    Define ms and mt as in Add_failsafe.
    S1, T1 := ConstructInvariant(S − ms, p−mt), true−ms;
    repeat
        T2, S2 := T1, S1;
        p1 := p|S1 ∪ {(s0, s1) : s0 ∉ S1 ∧ s0 ∈ T1 ∧ s1 ∈ T1}−mt;
        T1 := ConstructFaultSpan(T1 − {s : S1 is not reachable from s in p1 }, f);
        S1 := ConstructInvariant(S1 ∧ T1, p1);
        if (S1 = {} ∨ T1 = {}) declare no masking f-tolerant program p' exists;
    until (T1 = T2 ∧ S1 = S2);
    For each state s : s ∈ T1 :
        Rank(s) = length of the shortest computation prefix of p1
                  that starts from s and ends in a state in S1;
        p', S', T' := {(s0, s1) : ((s0, s1) ∈ p1) ∧ (s0 ∈ S1 ∨ Rank(s0) > Rank(s1))}, S1, T1;
}

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

Theorem 4.12 The algorithm *Add_masking* is sound, complete, and in P. □