Short Papers

The Effect of the Specification Model on the Complexity of Adding Masking Fault Tolerance

Sandeep S. Kulkarni and Ali Ebnenasir

Abstract—In this paper, we investigate the effect of the representation of safety specification on the complexity of adding `masking` fault tolerance to programs—where, in the presence of faults, the program 1) recovers to states from where it satisfies its (safety and liveness) specification and 2) preserves its safety specification during recovery. Specifically, we concentrate on two approaches for modeling the safety specifications: 1) the *bad transition* (BT) model, where safety is modeled as a set of bad transitions that should not be executed by the program, and 2) the *bad pair* (BP) model, where safety is modeled as a set of finite sequences consisting of at most two successive transitions. If the safety specification is specified in the BT model, then it is known that the complexity of automatic addition of masking fault tolerance to high atomicity programs—where processes can read/write all program variables in an atomic step— is polynomial in the state space of the program. However, for the case where one uses the BP model to specify safety specification, we show that the problem of adding masking fault tolerance to high atomicity programs is NP-complete. Therefore, we argue that automated synthesis of fault-tolerant programs is likely to be more successful if one focuses on problems where safety can be represented in the BT model.

Index Terms—Fault-tolerance, automatic addition of fault tolerance, safety specification, formal methods, program synthesis.

1 Introduction

AUTOMATIC addition of fault tolerance is desirable in the design of fault-tolerant programs as it is difficult (if not impossible) to anticipate all classes of faults at design time. Since it is often the case that the designer is aware of a program that is correct in the absence of faults, automatic addition of fault tolerance to an existing program generates a fault-tolerant program that is correct by construction. Moreover, such automated addition of fault tolerance has the potential to reuse the computations of the existing fault-intolerant program during the addition of fault tolerance, thereby synthesizing fault-tolerant programs that preserve the efficiency of their fault-intolerant version.

Kulkarni and Arora [1] present synthesis algorithms for adding fault tolerance to high atomicity programs—where processes can read/write program variables in an atomic step. They show that the complexity of the addition of fault tolerance is polynomial in the state space of the fault-intolerant program if the safety specification is represented as a set of bad transitions. In [2], Gärtner and Jhumka conjecture that representing safety specification as a set of *sequences of transitions* results in exponential complexity for adding fault tolerance. They validate their claim in the context of some examples.

In this paper, we focus on the effect of the representation of safety specification on the complexity of adding masking fault tolerance to high atomicity programs. In the presence of faults, a masking fault-tolerant program 1) recovers to states from which it satisfies its (safety and liveness) specification and 2) preserves its safety specification during recovery. We consider two approaches that are restricted models of the safety specification specified by Alpern and Schneider [3]. Specifically, in [3], the safety specification is specified as a set of computation prefixes, where a computation prefix is a finite sequence of transitions. A computation violates the safety specification if one of its prefixes is ruled out by the safety specification.

The first approach (used in [1]) considers the restrictive model of [3] where the safety specification is specified in terms of a set of bad transitions that must not occur in program computations. In other words, intuitively, a program computation violates safety specification if there exists a bad transition in that computation. We denote this model as the *bad transition* (BT) model. Clearly, this model is more restrictive than that in [3]; given the safety specification specified in terms of "bad transitions" that should not occur in program computations, we can obtain the corresponding set of prefixes that should not occur in program computations.

The second approach is a generalization of the BT model where safety specification is specified in terms of a set of sequences of *at most* two transitions. In this model, a computation violates the safety specification if and only if it contains any sequence ruled out by the safety specification. We denote this model as the *bad pair* (BP) model. It is straightforward to observe that the BP model is a generalization of the BT model and a specialization of the model presented by Alpern and Schneider.

We show that synthesizing a masking fault-tolerant program from its fault-intolerant version in the BP model is significantly more complex than synthesizing a fault-tolerant program in the BT model. Specifically, for high atomicity programs, the synthesis in the BT specification model can be performed in polynomial time. (This result has been previously shown in [1].) However, for the same program model, the synthesis in the BP specification model is NP-complete. (This result is shown in this paper.) It follows that the problem of adding masking fault tolerance for the case where safety is represented as a set of computation prefixes that should not occur in a program computation is NP-hard. With this result, we argue that the synthesis of fault-tolerant programs will be more successful if we focus on more restrictive specifications from the BT model.

Organization of the paper. In Section 2, we present preliminary concepts. Then, in Section 3, we state the problem of adding masking fault tolerance to fault-intolerant programs. Subsequently, in Section 4, we show that adding masking fault tolerance to high atomicity programs is NP-complete for the BP model. Finally, in Section 5, we make concluding remarks.

2 Preliminaries

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 [3] and Kulkarni and Arora [1]. The definition of faults and fault tolerance is adapted from Arora and Gouda [4] and Kulkarni [5].

2.1 Program

A `program` p (denoted $p = \langle S_p, \delta_p \rangle$) is specified by a finite state space, S_p, and a set of transitions, δ_p, where δ_p is a subset of $S_p \times S_p$. A `state predicate` of p is any subset of S_p. A state predicate S is `closed` in the program p iff (if and only if) the condition

• *The authors are with the Software Engineering and Network Systems Laboratory, Department of Computer Science and Engineering, Michigan State University, East Lansing MI 48824.*
E-mail: {sandeep, ebnenasi}@cse.msu.edu.

Manuscript received 22 July 2004; revised 24 June 2005; accepted 1 Aug. 2005; published online 2 Sept. 2005.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-0111-0704.

$$(\forall s_0, s_1 :: (((s_0, s_1) \in \delta_p) \wedge (s_0 \in S)) \Rightarrow (s_1 \in S))$$

holds. A sequence of states, $\sigma = \langle s_0, s_1, \ldots \rangle$ with $len(\sigma)$ states, is a `computation` of p iff the following two conditions are satisfied: 1) $\forall j : 0 < j < len(\sigma) : (s_{j-1}, s_j) \in \delta_p$ and 2) if σ is finite and terminates in state s_l, then there does not exist state s such that $(s_l, s) \in \delta_p$. A sequence of states, $\langle s_0, s_1, \ldots, s_n \rangle$, is a `computation prefix` of p iff

$$\forall j : 0 < j \leq n : (s_{j-1}, s_j) \in \delta_p,$$

i.e., a computation prefix need not be maximal.

The `projection` of program p on state predicate S, denoted as $p|S$, is the program

$$\langle S_p, \{(s_0, s_1) : (s_0, s_1) \in \delta_p \wedge s_0, s_1 \in S\} \rangle.$$

In other words, $p|S$ consists of transitions of p that start in S and end in S.

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

2.2 Specification

Following Alpern and Schneider [3], we let the specification consist of a `liveness specification` and a `safety specification`. The liveness specification is represented by a set of infinite sequences of states. A computation `satisfies the liveness` specification if it contains a suffix that is in the liveness specification. For specifying the safety specification, we consider two models: the BT model and the BP model.

2.2.1 The BT Model

In this model, the safety specification is specified in terms of a set of bad transitions that should not occur in program computations. Thus, in the BT model, safety is specified by a subset of $S_p \times S_p$. As a result, a computation `violates` safety specification iff it contains a bad transition. To illustrate a specification in BT model, let x represent an integer counter whose domain is $\{0 \cdots k\}$, where $k > 1$. Then, x `never increases by more than one` is an example of safety specification in the BT model.

2.2.2 The BP Model

Now, we consider a special case of the model presented by Alpern and Schneider [3], [6]. In the *bad pair* (BP) model, the safety specification is modeled as a set of finite sequences that consist of at most two transitions. Thus, in this model, safety can be modeled as a subset of $(S_p \times S_p) \cup (S_p \times S_p \times S_p)$, i.e., a computation `violates` the safety specification iff it contains a subsequence that is ruled out by the safety specification. For example, a requirement that an `increase` in x `must not be immediately followed by a decrease in` x is a safety specification in the BP model. In the context of the BT and the BP models, we say a computation `satisfies the safety of specification` iff it does not violate the safety specification.

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

2.3 Faults and Fault Tolerance

The faults that a program is subject to are systematically represented by transitions. A class of `faults` f for program $p = \langle S_p, \delta_p \rangle$ is a subset of the set $S_p \times S_p$. We use $p[]f$ to denote the transitions obtained by taking the union of the transitions in p and the transitions in f. We say that a state predicate T is an f-span (read as `fault-span`) of p from S iff the following two conditions are satisfied: 1) $S \subseteq T$ and 2) T is closed in $p[]f$. 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 occurrence of the transitions in f.

We say that a sequence of states, $\sigma = \langle s_0, s_1, \ldots \rangle$ with $len(\sigma)$ states, is a `computation of` p `in the presence of` f iff the following three conditions are satisfied: 1) $\forall j : 0 < j < len(\sigma) : (s_{j-1}, s_j) \in (\delta_p \cup f)$, 2) if σ is finite and terminates in state s_l, then there does not exist state s such that $(s_l, s) \in \delta_p$, and 3) $\exists n : n \geq 0 : (\forall j : j > n : (s_{j-1}, s_j) \in \delta_p)$. The first requirement captures that, in each step, either a program transition or a fault transition is executed. The second requirement captures that faults do not have to execute. Finally, the third requirement captures that the number of fault occurrences in a computation is finite. This requirement is the same as that made in previous work [7], [8], [4], [9] to ensure that eventually recovery can occur.

2.3.1 Masking Fault Tolerance

We say a program p is `masking` f-`tolerant from` S `for` *spec* iff 1) p satisfies *spec* from S and 2) there exists a state predicate T such that the following three conditions hold: a) T is an f-span of p from S, b) $p[]f$ satisfies the safety of *spec* from T, and c) every computation of $p[]f$ that starts from a state in T contains a state of S. Condition b stipulates that p satisfies its safety specification even in the presence of faults and condition c captures that, in the presence of faults, p will eventually recover to its invariant S if f perturbs p to $T - S$.

Note that, in this paper, we only focus on the problem of adding masking fault tolerance in high atomicity programs. Questions related to other fault tolerance properties (e.g., failsafe and nonmasking) considered in [1] and questions related to adding fault tolerance to distributed programs are outside the scope of this paper. We refer the interested reader to [11] for current results (in the BT model) related to these questions.

3 Problem Statement

In this section, we reiterate the problem of adding masking fault tolerance from [1]. During automated addition of fault tolerance, we begin with the fault-intolerant program, its invariant, faults, and the safety specification that needs to be satisfied in the presence of faults. The goal is to *only* add masking fault tolerance to develop a program that *reuses* the given fault-intolerant program. In other words, we require that no new computations are introduced when faults do not occur.

Now, consider the case where we begin with the fault-intolerant program p, its invariant S, specification *spec*, and faults f. Let p' be the fault-tolerant program derived from p, and let S' be an invariant of p'. If S' contains a state s_0 that does not belong to S, then the computations of p' may reach s_0 and create new computations in the absence of faults that do not belong to p. Hence, we require S' to be a subset of S. Likewise, if $p'|S'$ includes transitions that do not belong to $p|S'$, then p' may introduce new computations in the absence of faults. Thus, the set of transitions of $p'|S'$ must be a subset of $p|S'$. Therefore, the decision problem of adding masking fault tolerance to fault-intolerant programs (from [1]) is as follows:

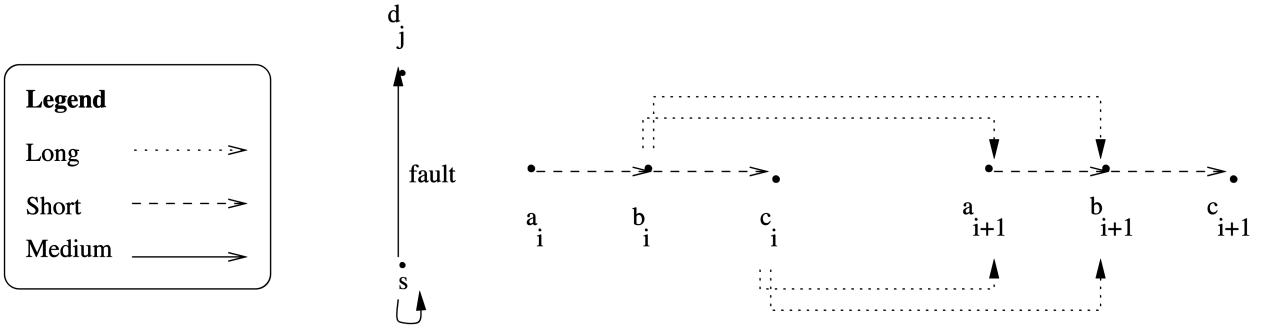

Fig. 1. The states and the transitions corresponding to the propositional variables in the 3-SAT formula. (Except for transitions marked as `fault`, all are program transitions. Also, note that the program has no long transitions that originate from a_i and no short transitions that originate from c_i.)

For a given fault-intolerant program p, its invariant S, the specification *spec*, and faults f, does there exist a masking fault-tolerant program p' and the invariant S' such that $S' \subseteq S$, $p'|S' \subseteq p|S'$ and p' is masking f-tolerant from S' for *spec*?

Remark. Observe that, in the above problem statement, every computation of $p'[]f$ that starts in the fault-span has a suffix that is a computation of p' that starts in a state in S' (which is a subset of S). As $S' \subseteq S$ and $p'|S' \subseteq p|S'$, this suffix is also a computation of p. Since p satisfies its specification (including liveness specification) from S, it follows that p' also satisfies the liveness specification. For this reason, liveness specification is not needed in the above problem statement.

4 NP-Completeness Proof

In this section, we show that, in general, the problem of synthesizing masking fault-tolerant programs from their fault-intolerant version becomes NP-complete if the safety specification is specified in the BP model.[1] Toward this end, in Section 4.1, we present a mapping between a given instance of the 3-SAT problem and an instance of the (decision) problem of adding masking fault tolerance. Then, in Section 4.2, we show that the given 3-SAT instance is satisfiable iff the answer to the decision problem is affirmative.

4.1 Mapping 3-SAT to the Addition of Masking Fault Tolerance

The problem statement for the 3-SAT problem [12] is as follows:

Given is a set of propositional variables, $x_1, x_2, \ldots, x_n$, and a Boolean formula $y = y_1 \wedge y_2 \wedge \ldots \wedge y_M$, where each y_j $(1 \leq j \leq M)$ is a disjunction of exactly three literals.

Does there exist an assignment of truth values to $x_1, x_2, \ldots, x_n$ such that y is satisfiable?

Next, we identify each entity of the instance of the problem of adding masking fault tolerance, based on the given 3-SAT formula. Recall that the instance of the decision problem of synthesizing masking fault tolerance consists of the fault-intolerant program, p, its invariant, S, its (safety) specification, and a class of faults f.

The state space and the invariant of the fault-intolerant program, p. The invariant, S, of the fault-intolerant program, p, includes only one state, say s. Corresponding to the propositional variables and disjunctions of the given 3-SAT instance, we include additional *states outside the invariant* (cf. Fig. 1). Specifically, for each propositional variable x_i, we introduce three states a_i, b_i, and c_i $(1 \leq i \leq n)$. Also, for simplicity, we introduce a propositional variable x_{n+1} which is always true and, corresponding to x_{n+1}, we introduce two states a_{n+1} and b_{n+1}. For each disjunction y_j, we introduce a state d_j outside the invariant $(1 \leq j \leq M)$.

The transitions of the fault-intolerant program. For the convenience of representing safety specification, we classify transitions as *short*, *long*, and *medium* transitions. The only transition inside the invariant of the fault-intolerant program is the medium transition (s, s). Also, we introduce *short* transitions (a_i, b_i) and (b_i, c_i) for each propositional variable x_i $(1 \leq i \leq n)$. We also introduce a short transition (a_{n+1}, b_{n+1}) for x_{n+1}.

Moreover, corresponding to each propositional variable x_i, we introduce *long* transitions (b_i, a_{i+1}), (b_i, b_{i+1}), (c_i, a_{i+1}), and (c_i, b_{i+1}) $(1 \leq i \leq n)$. From b_{n+1}, we introduce a long transition (b_{n+1}, s) to the invariant. Corresponding to each disjunction y_j, we have the following long transitions:

- If x_i is a literal in y_j, then we include the long transition (d_j, a_i).
- If $\neg x_i$ is a literal in y_j, then we include the long transition (d_j, b_i).

Fault transitions. The class of faults f is equal to the set of *medium* transitions $\{(s, d_j) : 1 \leq j \leq M\}$.

The safety specification of the fault-intolerant program, p. Safety will be violated if a short (respectively, long) transition is followed by another short (respectively, long) transition. Note that (s, s) and fault transitions are medium transitions (cf. Fig. 1). Hence, they can be followed by (respectively, preceded by) any transition. Also, all transitions except those identified above violate the safety specification. This is to ensure that transitions such as (d_j, s), (a_i, s), (b_i, s), (c_i, s) $((1 \leq j \leq M) \wedge (1 \leq i \leq n))$, and (a_{n+1}, s) cannot be used for recovery.

4.2 Reduction from 3-SAT

In this section, we show (with Lemmas 1 and 2) that the given instance of 3-SAT is satisfiable iff masking fault tolerance can be added to the problem instance identified in Section 4.1.

Lemma 1. *If the given 3-SAT formula is satisfiable then there exists a masking fault-tolerant program for the instance of the decision problem identified in Section 4.1.*

Proof. Since the 3-SAT formula is satisfiable, there exists an assignment of truth values to the propositional variables x_i, $1 \leq i \leq n$, such that each y_j, $1 \leq j \leq M$, is *true*. Now, we identify a masking fault-tolerant program, p', that is obtained by adding fault tolerance to the fault-intolerant program p identified in Section 4.1.

The invariant of p' is the same as the invariant of p (i.e., $\{s\}$). We derive the transitions of the fault-tolerant program p' as follows (as an illustration, we have shown a partial structure of p' where $x_1 = true$, $x_2 = false$, and $x_3 = true$ in Fig. 2):

1. Additional insights about our NP-completeness proof can be found in the Appendix.

Fig. 2. The partial structure of the fault-tolerant program.

- For each propositional variable x_i, $1 \leq i \leq n$, if x_i is *true*, then we include the short transition (a_i, b_i). In this case, we also include the long transition (b_i, a_{i+1}) if x_{i+1} is *true* or (b_i, b_{i+1}) if x_{i+1} is *false*.
- For each propositional variable x_i, $1 \leq i \leq n$, if x_i is *false*, then we include the short transition (b_i, c_i). In this case, we also include the long transition (c_i, a_{i+1}) if x_{i+1} is *true*, or (c_i, b_{i+1}) if x_{i+1} is *false*.
- We include the transitions (a_{n+1}, b_{n+1}) and (b_{n+1}, s) corresponding to x_{n+1}.
- For each disjunction y_j that includes x_i, we include the transition (d_j, a_i) iff x_i is *true*.
- For each disjunction y_j that includes $\neg x_i$, we include the transition (d_j, b_i) iff x_i is *false*.

Now, we show that p' is masking fault-tolerant in the presence of faults f.

- p' **in the absence of faults**. $p'|S = p|S$. Thus, p' satisfies *spec* in the absence of faults.
- p' **is masking** f-tolerant **for** *spec* **from** S. To show this result, we let T' be the set of states reached in the computations of $p'[]f$ starting from s.
  - p' **satisfies its safety specification from** T'. Since the instance of the 3-SAT formula is satisfiable, each propositional variable x_i is assigned a unique truth value. Thus, for each pair of transitions (a_i, b_i) and (b_i, c_i), one of them is excluded in the set of transitions of p'. Hence, a computation of p' cannot include two consecutive short transitions. Also, the only way to execute two consecutive long transitions in the original fault-intolerant program is to execute a long transition that terminates in state b_i, $1 \leq i \leq n$, and then execute a long transition that originates in b_i. If the former transition is included, then x_i is assigned the truth value false. However, in this case, no outgoing long transition from b_i is included. Thus, p' cannot execute two consecutive long transitions.
  - **Starting from every state in** T'**, a computation of** p' **reaches** s. By construction, p' contains no cycles outside the invariant. Hence, it suffices to show that p' does not deadlock in $T' - S'$. Now, let $y_j = x_i \vee \neg x_k \vee x_r$ be a disjunction in the 3-SAT formula. Since y_j evaluates to true, p' includes a transition from $\{(d_j, a_i), (d_j, b_k), (d_j, a_r)\}$. Also, by considering the truth values of x_i and x_{i+1}, $1 \leq i \leq n$, we observe that, for every state in $\{a_i, b_i, c_i\}$ in T', there is a path that reaches a state in $\{a_{i+1}, b_{i+1}, c_{i+1}\}$. Finally, from a_{n+1} (respectively, b_{n+1}) there is an outgoing transition to b_{n+1} (respectively, s). It follows that p' does not deadlock in $T' - S$. □

Lemma 2. *If there exists a masking fault-tolerant program for the instance of the decision problem identified earlier, then the given 3-SAT formula is satisfiable.*

Proof. Before we use the masking fault-tolerant program p' to identify the truth value assignment to the propositional variables in the 3-SAT formula, we make some observations about p'. Let S' be the invariant of p' and let T' be the fault-span used to show the masking fault tolerance property of p'. Since $S' \neq \{\}$ and $S' \subseteq S = \{s\}$, the conditions $S' = S$ and $p|S' = p'|S'$ must hold.

Since faults may directly perturb p' to d_j ($1 \leq j \leq M$), the condition $d_j \in T'$ holds. Thus, p' must provide safe recovery from each d_j, i.e., each computation of p' starting at d_j must satisfy safety and reach a state in S'. As a result, for each d_j, there exists i, $1 \leq i \leq n$, such that either (d_j, a_i) or $((d_j, b_i)$ and $(b_i, c_i))$ is included in $p'|T'$, i.e., either a_i or c_i must be reachable. Hence, we have

Observation 1. There exists i, $1 \leq i \leq n$, such that either $a_i \in T'$ or $c_i \in T'$.

Now, consider the case where $a_i \in T'$ and $c_i \in T'$. In this case, (a_i, b_i) must be included as all transitions terminating in a_i are long transitions. Further, if $c_i \in T'$, then (b_i, c_i) must be included since it is the only transition that reaches c_i. In this case, $p'[]f$ can violate safety by executing (a_i, b_i) and (b_i, c_i). Hence, we have:

Observation 2. If $a_i \in T'$, then $c_i \notin T'$.

Moreover, if $a_i \in T'$, then $(a_i, b_i) \in p'|T'$ since all transitions terminating in a_i are long transitions. Hence, $b_i \in T'$. Now, to guarantee safe recovery from b_i, p' must include either (b_i, a_{i+1}) or $((b_i, b_{i+1})$ and $(b_{i+1}, c_{i+1}))$. Thus, either $a_{i+1} \in T'$ or $c_{i+1} \in T'$. Also, if $c_i \in T'$, then either (c_i, a_{i+1}) or $((c_i, b_{i+1})$ and $(b_{i+1}, c_{i+1}))$ must be included. Thus, we have:

Observation 3. If $(a_i \in T') \vee (c_i \in T')$ holds, then we have $(\forall l : i < l \leq n : ((a_l \in T') \vee (c_l \in T')))$.

Now, let sm be the smallest value for which $((a_{sm} \in T') \vee (c_{sm} \in T'))$ holds. Based on Observation 3, we have $(\forall l : sm < l \leq n : (a_l \in T') \vee (c_l \in T'))$. Hence, we make value assignment to the propositional variables of the 3-SAT formula as follows:

- For $t < sm$, we assign *true* to x_t.

- For $sm \leq t$, if $a_t \in T'$, then $x_t = true$. And, if $c_t \in T'$, then $x_t = false$.

Based on Observations 1-3, it is straightforward to observe that a unique value is assigned to each x_i $(1 \leq i \leq n)$. To complete the proof, we need to show that, with this truth-value assignment, the 3-SAT formula is satisfiable. We show this for a disjunction y_j $(1 \leq j \leq M)$. Without loss of generality, let $y_j = x_i \vee \neg x_k \vee x_r$. Since state d_j can be reached by the occurrence of a fault from s, p' must provide safe recovery from d_j. Since the only safe transitions from d_j are those corresponding to states a_i, b_k, and a_r, p' must include at least one of the transitions (d_j, a_i), (d_j, b_k), or (d_j, a_r). Now, if $(d_j, a_i) \in p'$, then $a_i \in T'$ and, hence, x_i is assigned $true$. Further, if $(d_j, b_k) \in p'$, then no long transition from b_k can be included as it would allow p' to execute two long transitions successively. Hence, p' must include (b_k, c_k). Thus, $c_k \in T'$ and, hence, x_k is assigned $false$. It follows that, irrespective of which transition is included from d_j, y_j evaluates to $true$. Therefore, the 3-SAT formula is satisfiable. □

Theorem 1. *If the safety specification is specified in the BP model, then the problem of adding masking fault tolerance to high atomicity programs is NP-complete.*

Proof. The NP-hardness of adding masking fault tolerance in the BP model follows from Lemmas 1 and 2. To show that this problem is in NP, we proceed as follows: Given an input for the problem of adding masking fault tolerance, we guess fault-tolerant program p', its invariant S', and its fault-span T'. Now, we need to verify that

1. $S' \subseteq S$,
2. S' is closed in p',
3. $p'|S' \subseteq p|S'$,
4. T' is closed in $p'[]f$,
5. p' does not deadlock in $T' - S'$,
6. safety is not violated in $p'|T'$, and
7. $p'|(T' - S')$ is acyclic.

Since each of these conditions can be verified in polynomial time in the state space, the theorem follows. □

Corollary 1. *If the safety specification is specified by a set of computational prefixes that should not occur in program computations (as in [3]), then the problem of adding masking fault tolerance is NP-hard in the size of S_p.*

Proof. Note that the instance of the problem obtained in Section 4.1 by reducing the 3-SAT formula is also an instance of the synthesis problem where safety specification is specified as defined in [3]. Hence, the corollary follows. □

5 Conclusion

In this paper, we investigated the effect of the representation of the safety specification on the complexity of adding masking fault tolerance. It is shown in the literature [1] that if one represents the safety specification by a set of *bad transitions* (denoted the BT model) that should not be executed by a program, then adding masking fault tolerance to that program in the high atomicity model— where processes can read/write all program variables in an atomic step—can be done in polynomial time in the state space of the input fault-intolerant program. However, in this paper, we showed that if safety is represented by a set of sequences of transitions, where each sequence contains at most two transitions (denoted the *bad pair* (BP) model), then adding masking fault tolerance to programs is NP-complete. With this result, we argue that adding fault tolerance to existing programs can be done more efficiently if we focus on the BT model.

Although the BT model is a restricted version of the BP model, it is general enough to capture other representations for modeling safety considered in the literature. For example, in the *bad state* (BS) model (e.g., [12], [13]), a computation violates safety if it reaches a state that is ruled out by the safety specification. The BS model is a restrictive version of the BT model. Hence, the algorithms in [1] can be extended to the BS model. Thus, the complexity of adding fault tolerance in the BS model is (approximately) in the same class as that of the BT model.

Also, we observe that the expressiveness of the BT model has the potential to capture the safety specification of practical problems. As an illustration, we have modeled the safety specification of several examples including a simplified version of an aircraft altitude controller [10]. After modeling safety in the BT model, we have automatically added masking fault tolerance to the controller program for faults that perturb the altitude sensors. As a result, we argue that, although the results of this paper limit the applicability of *efficient* addition of fault tolerance to problems that can be specified in the BT model, this model can capture a broad range of interesting problems in the synthesis of fault-tolerant programs.

Appendix A

An Explanation into the NP-Completeness Result in Section 4

Because of the similarity between the BP model and the BT model, the reader may wonder if it is possible to reduce (in polynomial time) an instance of the BP model, say $BP_{instance}$, to an instance of the BT model, say $BT_{instance}$ such that $BP_{instance}$ has a solution iff $BT_{instance}$ has a solution. If such a reduction were possible, it would contradict the result (from [1]) that the synthesis problem for the BT instance can be solved in polynomial time. An anonymous referee of the paper suggested one such reduction. In this supplemental material, we consider this reduction. The counterexample for the correctness of this reduction provides an insight into the NP-completeness result in Section 4.[2]

We proceed as follows: First, we identify the possible reduction from the synthesis in the BP model to the synthesis in the BT model. Then, we show that this reduction is incorrect with the help of a counterexample. Subsequently, we identify the reason behind the failure of such a reduction. This reason identifies the main difference between the BT model and the BP model.

A.1 Polynomial-Time Reduction

The goal is to transform an instance $BP_{instance}$ of the problem of adding masking fault tolerance in the BP model into an instance $BT_{instance}$ of adding masking fault tolerance in the BT model. Given a program p, its state space S_p, its invariant S, its safety specification $spec$ (specified in the BP model), and a class of faults

2. Note that this discussion does not conclusively prove that any such reduction does not exist. The direct proof that (assuming $P \neq NP$) shows the nonexistence of such reduction follows from the proof that synthesis in the BP model is NP-complete (cf. Section 4 in the paper) and the proof that the algorithm in [1] is in P. However, this example does identify one of the main difficulties in adding fault tolerance in the BP model. This difficulty is the main reason behind the NP-completeness of adding masking fault tolerance to an instance of the BP model.

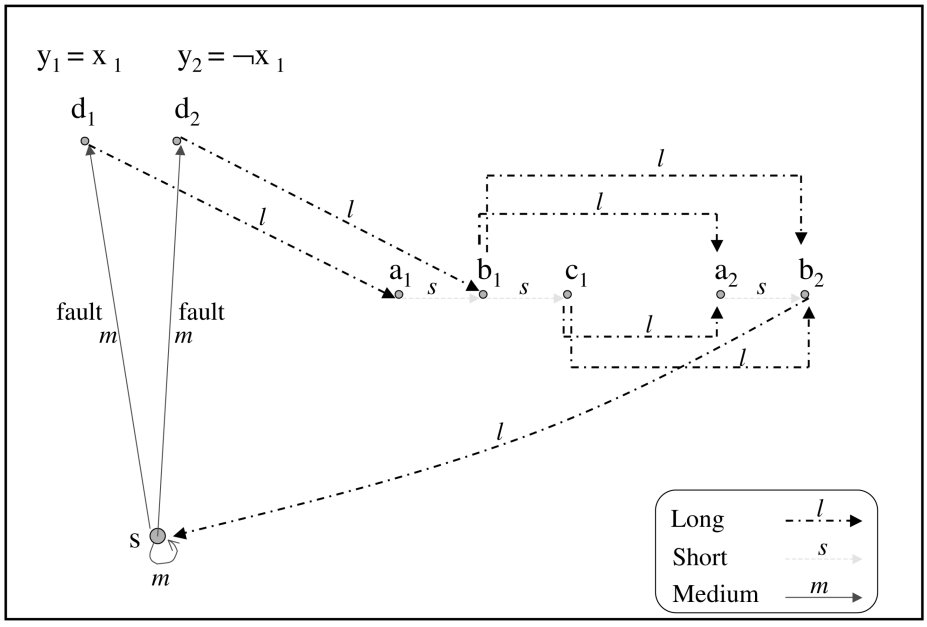

Fig. 3. Program corresponding to the SAT instance $\mathrm{y_1} \wedge \mathrm{y_2}$ in the BP model.

f, we construct another program p', fault-class f', invariant S', and safety specification $spec'$ in the BT model.

- **Constructing the state space**. The new state space, $S_{p'}$, is equal to $(\{\epsilon\} \cup S_p) \times S_p$, where ϵ is a distinguished state not in S_p. The distinguished state ϵ is for representing the case where the program is just starting and there is no historical information available. Each state in $S_{p'}$ tracks both the current state and the historical information of the previous state of the program p.
- **Constructing program transitions**. The transition function $\delta_{p'}$ for program p' will be a pair of state pairs, i.e., $\delta_{p'} \subseteq ((S_p \cup \{\epsilon\}) \times S_p) \times (S_p \times S_p)$. A transition $((s_0, s_1), (s_2, s_3))$ is `well-formed` if and only if $s_1 = s_2$.
- **Constructing the invariant**. The state predicate $S' = \{(\{\epsilon\} \cup S_p) \times S\}$ constructs the invariant of the $BT_{instance}$, which is closed in p'. By contradiction, consider a transition $((s_0, s_1), (s_1, s_2)) \in \delta_{p'}$ such that $(s_0, s_1) \in S'$ and $(s_1, s_2) \notin S'$. Since $(s_1, s_2) \in \delta_p$ and $s_1 \in S$, it follows that $s_2 \in S$. Thus, $(s_1, s_2) \in S'$ must hold. Therefore, S' is closed in p'.
- **Constructing the safety specification**. The specification $spec'$ is equal to $BT_1 \cup BT_2 \cup BT_3$, where

  1. $BT_1 = \{((s_0, s_1), (s_1, s_2)) \in (S_{p'} \times S_{p'}) | (s_1, s_2) \in spec\}$.
  2. $BT_2 = \{((s_0, s_1), (s_1, s_2)) \in (S_{p'} \times S_{p'}) \mid (s_0, s_1, s_2) \in spec\}$.
  3. $BT_3 = \{((s_0, s_1), (s_2, s_3)) \in (S_{p'} \times S_{p'}) \mid ((s_0, s_1), (s_2, s_3))$ is not well-formed$\}$.
- **Constructing fault transitions**. The class of faults f' in $BT_{instance}$ is equal to

$$\{((s_0, s_1), (s_1, s_2)) \in (S_{p'} \times S_{p'}) \mid (s_1, s_2) \in f\}.$$

Obviously, the complexity of such construction of an instance in the BT model is polynomial in S_p. Now, to reduce the problem of adding masking fault tolerance in the BP model to the problem of adding masking fault tolerance in the BT model, we need to show that:

Conjecture: The original instance $BP_{instance}$ in the BP model has a masking f-tolerant program iff the newly constructed instance $BT_{instance}$ in the BT model has a masking f'-tolerant program.

A.2 Counterexample

Unfortunately, the above conjecture is invalid. Hence, we present a counterexample to refute this conjecture. In this counterexample, we begin with the SAT formula $(x_1 \wedge \neg x_1)$. We derive the corresponding BP model. (Note that, although, in Section 3 of the paper, the reduction is from 3-SAT, it can be easily extended to be from SAT.) Then, we use the above reduction to obtain the BT model. We show that fault tolerance can be added to the instance in the BT model although it cannot be added to the instance of the BP model.

Now, we generate the BP instance for this formula. The BP instance is as shown in Fig. 3. Next, we reduce this BP instance to the BT instance using the above reduction approach. Since there are eight states in the BP instance, according to the above reduction, there would be 9*8 states in the BT model. However, many of these states are not *interesting*; for example, if the current state is d_1 (cf. Fig. 3), then the previous state must be s. Likewise, if the current state is b_1, then, previous state can either be d_2 or a_1. Hence, in reducing BP instance to BT instance, for simplicity of presentation, we only consider the 14 states shown in Fig. 4. (Note that fault tolerance can be added to this instance of the BT solution by only considering these 14 states. Therefore, the addition of remaining states does not affect the existence of the fault-tolerant program obtained by the addition of masking fault tolerance to the BT instance.)

Masking fault tolerance *cannot* be added to the instance in the BP model in Fig. 3.

Proof. Both d_1 and d_2 are reachable from s by fault transitions. Hence, recovery must be added from these states. Therefore, (d_1, a_1) and (d_2, b_1) must be in the fault-tolerant program. Since there is only one transition from a_1, (a_1, b_1) must be included in the fault-tolerant program. From b_1, the fault-tolerant program cannot include the transition (b_1, c_1) as execution of (a_1, b_1)

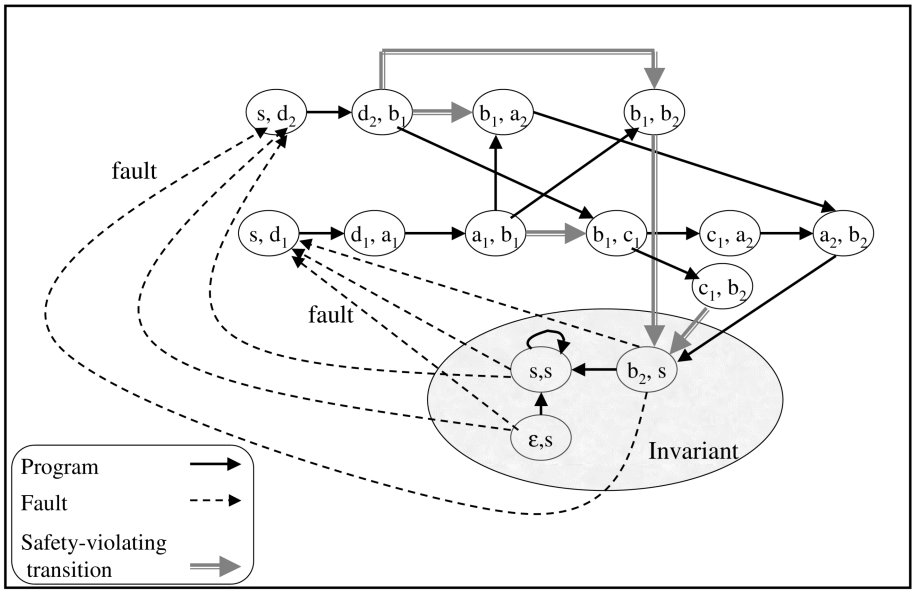

Fig. 4. Fault-intolerant program in the BT model derived using the above reduction.

followed by (b_1, c_1) violates safety. Likewise, (b_1, a_2) cannot be included; inclusion of this transition will cause the execution of two consecutive long transitions (d_2, b_1) and (b_1, a_2). This causes b_1 to be a deadlock state. Since this is not permitted in a masking fault-tolerant program, it follows that masking fault tolerance *cannot* be added to the instance in the BP model in Fig. 3. □

Masking fault tolerance *can* be added to the instance in BT model in Fig. 4.

Proof. Fig. 5 shows the program obtained by adding masking fault tolerance to the BT instance. □

Based on the above discussion, it follows that the above conjecture is incorrect.

A.3 Explanation

The following describes why the conjecture is incorrect. To prove the conjecture, one needs to show that 1) if fault tolerance can be added to the BP instance, then it can be added to the BT instance and 2) if fault tolerance can be added to the BT instance, then it can be added to the BP instance. Of these, the first result can be shown. But, the second result is incorrect. This is because, as shown in the above example, the BT model chooses the transition (b_1, c_1) (respectively, (b_1, a_2)) if the program reaches b_1 from d_2 (respectively, a_1). However, the BT model drops the transition (b_1, c_1) (respectively, (b_1, a_2)) if the program reaches b_1 from a_1 (respectively, d_2). Now, taking the solution from the BT model to obtain a solution in the BP model, we cannot do this "fine-tune" modification to the program. Specifically, we have to either include the transition (b_1, c_1) (respectively, (b_1, a_2)) or drop it. We

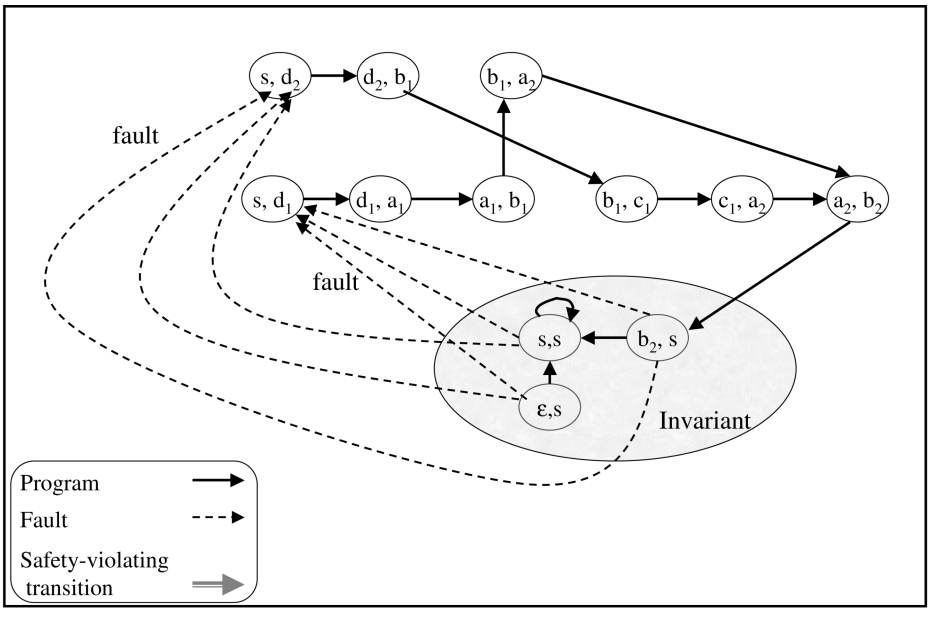

Fig. 5. Masking fault-tolerant program in the BT model.

cannot include it conditionally, as done in the BT model. Now, if we include the transition (b_1, c_1) (respectively, (b_1, a_2)), then it leads to safety violation and if we exclude it, then it leads to deadlock. Therefore, even if the BT instance has a solution, the corresponding BP instance may not.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive feedback. They especially thank the reviewer who provided a possible polynomial-time reduction (considered in the Appendix) from the BP model to the BT model. The counter-example for this reduction provides additional insights into the complexity of adding masking fault tolerance in BP model. This work was partially sponsored by US National Science Foundation (NSF) grant NSF CAREER CCR-0092724, US Defense Advanced Research Projects Agency Grant OSURS01-C-1901, the Office of Naval Research Grant N00014-01-1-0744, NSF grant EIA-0130724, and a grant from Michigan State University.

REFERENCES

[1] S.S. Kulkarni and A. Arora, "Automating the Addition of Fault-Tolerance," *Proc. Sixth Int'l Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems,* pp. 82-93, 2000.

[2] F.C. Gärtner and A. Jhumka, "Automating the Addition of Failsafe Fault-Tolerance: Beyond Fusion-Closed Specifications," *Proc. Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT),* pp. 183-198, Sept. 2004.

[3] B. Alpern and F.B. Schneider, "Defining Liveness," *Information Processing Letters,* vol. 21, no. 4, pp. 181-185, Oct. 1985.

[4] A. Arora and M.G. Gouda, "Closure and Convergence: A Foundation of Fault-Tolerant Computing," *IEEE Trans. Software Eng.,* vol. 19, no. 11, pp. 1015-1027, Nov. 1993.

[5] S.S. Kulkarni, "Component-Based Design of Fault-Tolerance," PhD thesis, Ohio State Univ., 1999.

[6] F.B. Schneider, "Enforcing Security Policies," *ACM Trans. Information and System Security,* vol. 3, no. 1, pp. 30-50, 2000.

[7] E.W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Comm. ACM,* vol. 17, no. 11, 1974.

[8] A. Arora and S.S. Kulkarni, "Designing Masking Fault-Tolerance via Nonmasking Fault-Tolerance," *IEEE Trans. Software Eng.,* vol. 24, no. 6, pp. 435-450, June 1998.

[9] G. Varghese, "Self-Stabilization by Local Checking and Correction," PhD thesis, Massachusetts Inst. Technology, 1993.

[10] S.S. Kulkarni and A. Ebnenasir, "A Framework for Automatic Synthesis of Fault-Tolerance," Technical Report MSU-CSE-03-27, Michigan State Univ., http://www.cse.msu.edu/~sandeep/software/Code/synthesis-framework/, 2003.

[11] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[12] E.A. Emerson and E.M. Clarke, "Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons," *Science of Computer Programming,* vol. 2, no. 3, pp. 241-266, 1982.

[13] P. Attie and A. Emerson, "Synthesis of Concurrent Programs for an Atomic Read/Write Model of Computation," *ACM Trans. Programming Languages and Systems,* vol. 23, no. 2, Mar. 2001.