Enhancing The Fault-Tolerance of Nonmasking Programs[1]

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

Abstract

In this paper, we focus on automated techniques to enhance the fault-tolerance of a nonmasking fault-tolerant program to masking. A masking program continually satisfies its specification even if faults occur. By contrast, a nonmasking program merely guarantees that after faults stop occurring, the program recovers to states from where it continually satisfies its specification. Until the recovery is complete, however, a nonmasking program can violate its (safety) specification. Thus, the problem of enhancing fault-tolerance from nonmasking to masking requires that safety be added and recovery be preserved. We focus on this enhancement problem for high atomicity programs –where each process can read all variables– and for distributed programs –where restrictions are imposed on what processes can read and write. We present a sound and complete algorithm for high atomicity programs and a sound algorithm for distributed programs. We also argue that our algorithms are simpler than previous algorithms, where masking fault-tolerance is added to a fault-intolerant program. Hence, these algorithms can partially reap the benefits of automation when the cost of adding masking fault-tolerance to a fault-intolerant program is high. To illustrate these algorithms, we show how the masking fault-tolerant programs for triple modular redundancy and Byzantine agreement can be obtained by enhancing the fault-tolerance of the corresponding nonmasking versions. We also discuss how the derivation of these programs is simplified when we begin with a nonmasking fault-tolerant program.

Keywords : Automatic addition of fault-tolerance, Formal methods, Fault-tolerance, Program synthesis, Program transformation, Distributed programs

1 Introduction

In this paper, we concentrate on automatic techniques to enhance the fault-tolerance level of a program from nonmasking to masking. We focus our attention on masking fault-tolerance as it is often desirable –if not ideal– property for system design. A masking fault-tolerant program ensures that its specification (both safety and liveness) is satisfied even if faults occur. By contrast, after the occurrence of faults, a nonmasking fault-tolerant program merely ensures that it recovers to states from where its specification is satisfied. However, until such a state is reached, safety may be violated.

[1]Email: sandeep@cse.msu.edu, ebnenasi@cse.msu.edu
Web: `http://www.cse.msu.edu/~{sandeep,ebnenasi}`
Tel: +1-517-355-2387, Fax: +1-517-432-1061

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

Since a masking fault-tolerant program synthesized thus is correct by construction, there is no need for its proof of correctness. Also, if a masking fault-tolerant program is designed by reusing an existing program then there is a potential that the synthesized program will preserve properties such as efficiency that are difficult to model in an automated synthesis algorithm. Based on these motivations, in [1, 2], Kulkarni, Arora, and Chippada have presented algorithms for adding fault-tolerance to a fault-intolerant program that provides no guarantees about its behavior if faults occur.

One of the problems in automating the addition of masking fault-tolerance to a fault-intolerant program is the complexity of such addition. Specifically, in [1], Kulkarni and Arora have shown that the addition of masking fault-tolerance to distributed fault-intolerant programs is NP-hard in the state space of the fault-intolerant program. Therefore, if we were to implement the brute-force deterministic implementation of the algorithm in [1], its application would be limited. Although the heuristic based polynomial-time implementation in [2] solves some of these difficulties, we find that there are other problems in developing such heuristics; e.g., given a state that the fault-intolerant program reaches in the presence of faults, it is difficult to determine if *safe recovery* can be added from that state.

Given the complexity of adding fault-tolerance to a distributed fault-intolerant program and the difficulty of determining if safe recovery can be added from a state reached in the presence of faults, we consider the following question. *Is it possible to reduce the complexity of adding masking fault-tolerance if we begin with a program that provides additional guarantees about its behavior in the presence of faults?* In the context of tools that automate the addition of masking fault-tolerance, the above question is crucial. This question identifies additional properties that should be (possibly, manually) added to a fault-intolerant program in order to benefit from automation. Moreover, if these additional properties are satisfied by the program at hand, we can simplify the addition of masking fault-tolerance.

Adding masking fault-tolerance to a fault-intolerant program involves two tasks: (1) ensuring that the program recovers to states from where it satisfies both the safety and the liveness specification, and (2) ensuring that the program does not violate safety during recovery. The first task requires the automation algorithm to add (recovery) transitions, whereas the second task requires the automation algorithm to remove (safety violating) transitions. The conflicting nature of these tasks increases the complexity of adding masking fault-tolerance. More specifically, during the addition of masking fault-tolerance to a fault-intolerant program, it is possible that the transitions that were removed to deal with safety violation may be important for recovery. Also, the recovery transitions may themselves conflict with each other and, hence, using one recovery tran-

sition may force the removal of another recovery transition.

Synthesizing a masking fault-tolerant program from a nonmasking fault-tolerant program separates the above two tasks; the nonmasking fault-tolerant program deals with the first task (recovery) whereas the addition of masking fault-tolerance only needs to deal with the second task (safety). Such separation of concerns has been found to be useful in manual addition of fault-tolerance [3]. More specifically, in [3], Arora and Kulkarni have presented a stepwise method for adding masking fault-tolerance. In their method, one begins with a fault-intolerant program, and adds nonmasking fault-tolerance to it in the first step. Then, in the second step, one enhances the fault-tolerance from nonmasking to masking. They have argued that such a stepwise approach is useful in providing new insights in the design of masking fault-tolerance as well as in simplifying the complexity of (manually) adding masking fault-tolerance.

In this paper, we focus on the enhancement problem that deals with automating the second step in the stepwise addition of masking fault-tolerance. We consider the enhancement problem in the context of two models: the high atomicity model and the low atomicity model. In both models, the program consists of a set of variables and a set of processes. In the high atomicity model, the processes can read all variables in the program. By contrast, for each process, the low atomicity model specifies the set of variables that it can read and a (possibly different) set of variables that it can write. Thus, the low atomicity model allows us to capture several computation models, e.g., shared memory model and message passing model, for distributed programs.

To illustrate our algorithms, we choose two examples: the triple modular redundancy (TMR) and Byzantine agreement programs. We have chosen these examples as the (manual) derivation of the masking version of these programs from their nonmasking version is presented in [3], and because, we want to determine if our automated algorithm can use the nonmasking fault-tolerant program from [3] to obtain the corresponding masking version. Also, our Byzantine agreement example enables us to compare the derivation of masking Byzantine agreement from its intolerant version (presented in [2]) to its derivation from a nonmasking Byzantine agreement. We note that we have used our algorithm to derive the masking version of alternating-bit protocol from the corresponding nonmasking version. However, for reasons of space, we do not include that example in this paper.

The main contributions of the paper are as follows: For the high atomicity model, we present a sound and complete algorithm for enhancing fault-tolerance. Thus, if there exists a program that enhances the fault-tolerance of the given nonmasking program to masking, our algorithm will succeed in identifying a masking fault-tolerant program. The complexity of this algorithm is polynomial in the state space of the nonmasking fault-tolerant program. We show that this complexity is asymptotically less than the complexity of adding masking fault-tolerance to a fault-intolerant program. For the low atomicity model, we present a sound algorithm for enhancing fault-tolerance. The complexity of this algorithm is also polynomial in the state space of the nonmasking fault-tolerant program. As an illustration of this algorithm, we show how masking fault-tolerant Byzantine agreement [4] can be designed using a solution for nonmasking Byzantine agreement. We show that this algorithm simplifies the difficulties encountered in the design in [2].

Organization of the paper. This paper is organized as follows: In Section 2, we provide a few basic concepts such as programs, computations, specifications, faults and fault-tolerance. In Section 3, we state the problem of enhancing the fault-tolerance from nonmasking to masking. In Section 4, we present our solution for the high atomicity model. In Section 5, we present our solution for distributed programs. Finally, we make concluding remarks in Section 6.

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 [5]. The definitions of faults and fault-tolerance are adapted from Arora and Gouda [6] and Kulkarni [7]. The issues of modeling distributed programs is adapted from [1]. A similar modeling of distributed programs in read/write atomicity was independently identified by Attie and Emerson [8, 9].

2.1 Program

A program p is a set of finite variables and a set of finite processes. Each variable is associated with a finite domain of values. Let $v_1, v_2, .., v_n$ be variables of p, and let $D_1, D_2, .., D_n$ be their respective domains. A state of p is obtained by assigning each variable a value from its respective domain. Thus, a state s of p is of the form: $\langle l_1, l_2, .., l_n \rangle$, where $\forall i : 1 \leq i \leq n : l_i \in D_i$. The state space of p, S_p, is the set of all possible states of p.

A process, say j, in p is associated with a set of program variables, say r_j, that it can read and a set of variables, say w_j, that it can write. Also, process j consists of a set of transitions δ_j; each transition is of the form (s_0, s_1), where $s_0, s_1 \in S_p$. The transitions of p, δ_p, is the union of the transitions of its processes. We let program p be the tuple $\langle S_p, \delta_p \rangle$, where S_p is a finite set of states and δ_p is a subset of $\{(s_0, s_1) \mid s_0, s_1 \in S_p\}$.

A state predicate of p is any subset of S_p. A state predicate S is closed in p (respectively, δ_p) iff $(\forall s_0, s_1 : (s_0, s_1) \in \delta_p : (s_0 \in S \Rightarrow s_1 \in S))$. A sequence of states, $\langle s_0, s_1, ... \rangle$, is a computation of p iff the following two conditions are satisfied: (1) $\forall j : j > 0 : (s_{j-1}, s_j) \in \delta_p$, and (2) if $\langle s_0, s_1, ... \rangle$ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in \delta_p$. A sequence of states, $\langle s_0, s_1, ... \rangle$, is a computation prefix of p iff $\forall j : j > 0 : (s_{j-1}, s_j) \in \delta_p$, i.e., a computation prefix need not be maximal.

The projection of program p on state predicate S, denoted as $p|S$, consists of transitions of p that start in S and end in S. Given two programs, $p = \langle S_p, \delta_p \rangle$ and $p' = \langle S'_p, \delta'_p \rangle$, we say $p' \subseteq p$ iff $S'_p = S_p$ and $\delta'_p \subseteq \delta_p$.

Remark. To avoid confusion between $S_{p'}$ (S with subscript p') and S_p, (S_p followed by a comma), we use S'_p to denote the invariant of p'. Likewise, we use δ'_p to denote the transitions of p'.

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

2.2 Issues of Distribution

Now, we present the issues that distribution introduces during the addition of fault-tolerance. More specifically, we identify how read/write restrictions on a process affect its transitions.

Write restrictions. Given a transition (s_0, s_1), it is straightforward to determine the variables that need to be changed in order to modify the state from s_0 to s_1. Hence, if process j can only write

the variables in w_j and the value of a variable other than that in w_j is changed in transition (s_0, s_1) then that transition cannot be used in obtaining the transitions of j. In other words, if j can only write variables in w_j then j cannot use the transitions in $nw(j, w_j)$, where

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

Read restrictions. Given a single transition (s_0, s_1), it appears that all the variables must be read in order for that transition to be executed. For this reason, read restrictions require us to group transitions and ensure that the entire group is included or the entire group is excluded. As an example, consider a program consisting of two variables a and b, with domains $\{0, 1\}$. Suppose that we have a process that cannot read b. Now, observe that the transition from the state $\langle a = 0, b = 0\rangle$ to $\langle a = 1, b = 0\rangle$ can be included iff the transition from $\langle a = 0, b = 1\rangle$ to $\langle a = 1, b = 1\rangle$ is also included. If we were to include only one of these transitions, we would need to read both a and b. However, when these two transitions are grouped, the value of b is irrelevant, and hence, we do not need to read it.

More generally, consider the case where r_j is the set of variables that j can read, w_j is the set of variables that j can write, and $w_j \subseteq r_j$. Now, process j can include the transition (s_0, s_1) iff it also includes the transition (s'_0, s'_1) where s_0 (respectively, s_1) and s'_0 (respectively, s'_1) are identical as far as the variables in r_j are considered. We define these transitions as $group(j, r_j)(s_0, s_1)$ for the case $w_j \subseteq r_j$, where

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

2.3 Specification

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 $\langle \alpha, s, \gamma\rangle$ and $\langle \beta, s, \delta\rangle$ are in that set then so are the state sequences $\langle \alpha, s, \delta\rangle$ and $\langle \beta, s, \gamma\rangle$, where α and β are finite prefixes of state sequences, γ and δ are suffixes of state sequences, and s is a program state.

Following Alpern and Schneider [5], we let the specification consist of a safety specification and a liveness specification. For a suffix closed and fusion closed specification, the safety specification can be specified [7] as a set of bad transitions, that is, for program p, its safety specification is a subset of $\{(s_0, s_1) : s_0, s_1 \in S_p\}$. The liveness specification is not required in our algorithm; the liveness specification satisfied by the nonmasking fault-tolerant program is preserved in the synthesized masking fault-tolerant program.

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

For a finite sequence (of states) α, we say that α maintains (does not violate) $spec$ iff there exists a sequence of states β such that $\alpha\beta \in spec$. We say that p maintains (does not violate) $spec$ from S iff (1) S is closed in p, and (2) every computation prefix of p that starts in a state in S maintains $spec$.

Notation. Let $spec$ be a specification. We use the term safety of $spec$ to mean the smallest safety specification that includes $spec$. Also, 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. A fault f for program $p = \langle S_p, \delta_p\rangle$ is a subset of the set $\{(s_0, s_1) : s_0, s_1 \in S_p\}$. 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) $S \Rightarrow T$, 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, $\langle s_0, s_1, ...\rangle$, is a computation of p in the presence of f iff the following three conditions are satisfied: (1) $\forall j : j > 0 : (s_{j-1}, s_j) \in (\delta_p \cup f)$, (2) 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 (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, i.e., if the program reaches a state where only a fault transition can be executed, it is not required that the fault transition be executed. It follows that fault transitions cannot be used to deal with deadlocked states. Finally, the third requirement captures that the number of fault occurrences in a computation is finite.

We say that p is masking f-tolerant (read as fault-tolerant) to $spec$ from S iff 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, $p[]f$ maintains $spec$ from T, and every computation of $p[]f$ that starts from a state in T contains a state of S.

Since a nonmasking fault-tolerant program need not satisfy safety in the presence of faults, p is nonmasking f-tolerant to $spec$ from S iff 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 $p[]f$ that starts from a state in T contains a state of S.

Note that a specification is a set of infinite sequences of states. Hence, if p satisfies $spec$ from S then all computations of p that start in S must be infinite. In the context of nonmasking and masking fault-tolerance, every computation from the fault-span reaches a state in its invariant. Hence, if fault-span T is used to show that p is nonmasking (respectively, masking) f-tolerant to $spec$ from S then all computations of p that start in a state in T must also be infinite. Also, note that p is allowed to contain a self-loop of the form (s_0, s_0); we use such a self-loop whenever s_0 is an *acceptable fixpoint* of p.

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

3 Problem Statement

In this section, we formally define the problem of enhancing fault-tolerance from nonmasking to masking. The input to the enhancement problem includes the (transitions of) nonmasking program, p, its invariant, S, faults, f, and specification, $spec$. Given p, S, and f, we can calculate an f-span, say T, of p by starting at a state in S and identifying states reached in the computations of $p[]f$. Hence, we include fault-span T in the inputs of the enhancement problem.

The output of the enhancement problem is a masking fault-tolerant program, p', its invariant, S', and its f-span, T'.

Since p is nonmasking fault-tolerant, in the presence of faults, p may temporarily violate safety. More specifically, faults may perturb p to a state in $T-S$. After faults stop occurring, p will eventually reach a state in S. However, p may violate $spec$ while it is in $T-S$. By contrast, a masking fault-tolerant program p' needs to satisfy both its safety and liveness specification in the absence and in the presence of faults.

As mentioned in the Introduction, the goal of the enhancement problem is to separate the tasks involved in adding recovery transitions and the tasks involved in ensuring safety. The enhancement problem deals only with adding safety to a nonmasking fault-tolerant program. With this intuition, we define the enhancement problem in such a way that only safety may be added while adding masking fault-tolerance. In other words, we require that during the enhancement, no new transitions are added to deal with functionality or to deal with recovery. Towards this end, we identify the relation between state predicates T and T', and the relation between the transitions of p and p'.

If $p'[]f$ reaches a state that is outside T then new recovery transitions must be added while obtaining the masking fault-tolerant program. Hence, we require that the fault-span of the masking fault-tolerant program, T', be a subset of T. Likewise, if p' does not introduce new recovery transitions then all the transitions included in $p'|T'$ must be a subset of $p|T'$. Hence, this is the second requirement of the enhancement problem. Thus, the enhancement problem is as follows:

The Enhancement Problem

Given p, S, $spec$, f, and T such that p satisfies $spec$ from S
and T is an f-span used to show that p is nonmasking fault-tolerant for $spec$ from S

Identify p' and T' such that
$T' \subseteq T$,
$p'|T' \subseteq p|T'$, and
p' is masking f-tolerant to $spec$ from T'. □

Comments on the Problem Statement

(1) While the invariant, S, of the nonmasking fault-tolerant program is an input to the enhancement problem, it is not used explicitly in the requirements of the enhancement problem. The knowledge of S permits us to identify the transitions of p that provide functionality and the transitions of p that provide recovery. We find that such classification of transitions is useful in solving the enhancement problem. Hence, we include S in the problem statement.

(2) If S' is an invariant of p', $S' \Rightarrow T'$, every computation of p' that starts from a state in T' maintains safety, and every computation of p' that starts from a state in T' eventually reaches a state in S' then every computation of p' that starts in a state in T' also satisfies its specification. In other words, in this situation, T' is also an invariant of p' (see [7] for proof). Hence, we do not explicitly identify an invariant of p'. Predicates T' and $T' \cap S$ can be used as the invariants of p'.

4 Enhancement in High Atomicity Model

In this section, we present our algorithm for the enhancement problem in high atomicity model. Thus, given a high atomicity nonmasking fault-tolerant program p, our algorithm derives masking fault-tolerant program p' that ensures that safety is added while the recovery provided by p is preserved. Hence, we obtain a solution for the enhancement problem by tailoring the algorithm $Add\_failsafe$ (from [1]); $Add\_failsafe$ deals with the addition of safety to a fault-intolerant program in the presence of faults.

In our algorithm, first, we compute the set of states, ms, from where fault actions alone violate safety. Clearly, we must ensure that the program never reaches a state in ms. Hence, in addition to the transitions that violate safety, we cannot use the transitions that reach a state in ms. We use mt to denote the transitions that cannot be used while adding safety.

Using ms and mt, we compute the fault-span of p', T', by calling function HighAtomicityConstructInvariant ($HACI$). The first guess for T' is $T-ms$. However, due to the removal of transitions in mt, it may not be possible to provide recovery from some states in $T-ms$. Hence, we remove such states while obtaining T'. If the removal of such states causes other states to become deadlocked, we remove those states as well. Moreover, if (s_0, s_1) is a fault transition such that s_1 was removed from T' then we remove s_0 to ensure that T' is closed in f. We continue the removal of states from T' until a fixpoint is established. After computing T', we compute the transitions of p' by removing all the transitions of $p-mt$ that start in a state in T' but reach a state outside T'. Thus, our high atomicity enhancement algorithm is as follows:

```
High_Atomicity_Enhancement(p, f : transitions,
                           T : 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) };
   T' := HACI(T − ms, p−mt, f);
   if (T' = {}) declare no masking f-tolerant program p' exists;
   else p' :=  (p − mt)  −  {(s0, s1) : s0∈T'  ∧  s1 ∉ T'}
}

HACI(T : state predicate, p, f : transitions)
    { while (∃s0 : s0 ∈ T : (∀s1 : s1 ∈ T : (s0, s1) ∉p) ∨
                                (∃s1 : s1 ∉ T : (s0, s1) ∈ f))
          T := T − {s0}  }
```

Theorem 4.1 The algorithm High_Atomicity_Enhancement is sound and complete and its complexity is polynomial in the state space of the nonmasking fault-tolerant program.
Please refer to [10] for proof. □

4.1 Example: Triple Modular Redundancy

As an illustration of our high atomicity algorithm, we show how the masking triple modular redundancy (TMR) program can be designed by enhancing the fault-tolerance level of the corresponding nonmasking program.

First, we present the nonmasking version of TMR program, the specification of TMR, and the fault actions for TMR. Then, we show how our high atomicity algorithm is used to enhance the level of fault-tolerance to masking. For brevity, we present the (program and fault) transitions in terms of guarded commands; a guarded command $g \longrightarrow st$ captures the transitions $\{(s_0, s_1)$: the state predicate g is true in s_0, and s_1 is obtained by *executing* statement st in state s_0 $\}$.

Nonmasking TMR program. Nonmasking version of TMR program consists of three processes j, k, and l that share an output variable out. Each process j has an input variable $in.j$. The values

of these input variables are obtained from a common sensor. The domain of each input variable is $\{0, 1\}$ and the domain of out is $\{0, 1, \bot\}$ ($\bot$ means no value has been assigned to out). For each process j, if the value of out is not yet assigned, j copies (using guarded command $N1$) its input $in.j$ to out. And, if out is assigned a wrong value, i.e., the value other than the majority value, and the value of $in.j$ is not corrupted then process j *corrects* (by guarded command $N2$) out by copying $in.j$ to out. Both nonmasking and masking programs for TMR include a self-loop for states in which out has been assigned a *correct* value. However, for brevity, in this section, we keep such self-loops implicit. Thus, the actions of each process j in the nonmasking version of TMR are as follows ($\oplus$ denotes modulo 3 addition):

$$N1 : (out = \bot) \longrightarrow out := in.j$$
$$N2 : (out \neq \bot) \wedge (out \neq in.j) \wedge ((in.j = in.(j \oplus 1)) \vee (in.j = in.(j \oplus 2))) \longrightarrow out := in.j$$

Faults. Faults may perturb one of the inputs when all of them are equal. Thus, the fault action that affects j is represented by the following action:

$$F : (\forall p :: in.j = in.p) \longrightarrow in.j := 0 \mid 1$$

Invariant. The following state predicate is an invariant of TMR.

$$S_{TMR} = (out = \bot \wedge (\forall p, q :: in.p = in.q)) \vee (\exists p, q : p \neq q : out = in.p = in.q)$$

Safety specification. The safety specification of TMR requires the program not to reach states in which there exist two processes whose input values are equal but these inputs are not equal to out (where $out \neq \bot$). The safety specification also stipulates that variable out cannot change if it is different from $\bot$. Thus, safety specification requires that following transitions are not included in a program computation.

$sf_{TMR} = sf_1 \cup sf_2$, where
$sf_1 = \{(s_0, s_1) \mid (\exists p, q : (p \neq q) : (in.p(s_1) = in.q(s_1)) \wedge (in.q(s_1) \neq out) \wedge (out(s_1) \neq \bot))\}$, and
$sf_2 = \{(s_0, s_1) \mid (out(s_0) \neq \bot) \wedge (out(s_0) \neq out(s_1))\}$

Fault-span. If all the inputs are equal, the value of out is either $\bot$ or equal to those inputs. Thus, fault-span of the nonmasking version of TMR is T, where

$$T_{TMR} = ((\forall p, q :: in.p = in.q) \Rightarrow ((out = \bot) \vee (\forall p :: out = in.p)))$$

Remark. The TMR program consists of three variables whose domain is $\{0, 1\}$ and one variable whose domain is $\{0, 1, \bot\}$. Enumerating the states associated with these variables, the state space of TMR program includes 24 states. Of these, 10 states are in the invariant, 12 additional states are in the fault-span, and two states are outside the fault-span.

The program consisting of actions $N1$ and $N2$ is nonmasking fault-tolerant in that if the faults perturb it to a state that is outside S_{TMR} then it eventually recovers to a state where S_{TMR} is true. However, until such a state is reached, safety specification may be violated.

Enhancing the tolerance of TMR. We trace the execution of our high atomicity algorithm for nonmasking TMR program.

(1) **Compute** ms**.** ms includes all the states from where one or more fault transitions violate safety. In case of TMR, fault transitions do not violate safety if they execute in a state in T_{TMR} . Faults only change the value of one of the inputs and then safety may be violated if the corresponding process executes guarded command $N1$. Thus, $T_{TMR} \cap ms = \{\}$.

(2) **Compute** mt**.** From the definition of ms, $mt = sf_{TMR}$.

(3) **Construct** T'_{TMR} **and** p'**.** After removing transitions in mt, states where out differs from $\bot$ and out differs from the majority of the inputs are deadlocked. Hence, we need to remove those states while obtaining T'_{TMR}. After removal of these states, there are no other deadlock states. Hence, our algorithm will let T'_{TMR} to be the state predicate:

$$T'_{TMR} = T_{TMR} - \{s : (\exists p, q : (p \neq q) : (in.p(s) = in.q(s)) \wedge (out(s) \neq \bot) \wedge (out(s) \neq in.p(s)))\}$$

Moreover, to obtain the transitions of masking version of TMR, we consider the transitions of p that preserve the closure property of T'_{TMR}. Thus, the masking version of TMR consists of the following guarded command:

$$M1 : (out = \bot) \wedge ((in.j = in.(j \oplus 1)) \vee (in.j = in.(j \oplus 2))) \longrightarrow out := in.j$$

The predicate T'_{TMR} computed by our algorithm is both an invariant and a fault-span for the above program; every computation of the above program satisfies the specification if it begins in a state in T'_{TMR}. Moreover, T'_{TMR} is closed in both the program and fault transitions.

Remark. Note that transitions included in $N2$ are removed from the above masking fault-tolerant program as those transitions violate sf_2. However, if safety consisted of only sf_1 then the fault-tolerant program would include the transitions included in $N2$. While a masking fault-tolerant program can be obtained without using the transitions in $N2$, their inclusion follows from the heuristic in [1] that the output program should be maximal. In [1], Kulkarni and Arora have argued that if the output of a synthesis algorithm is to be used as an input, say to add fault-tolerance for a new fault, it is desirable that the intermediate program be maximal.

4.2 Enhancement versus Addition

In this section, we compare our enhancement algorithm with $Add\_masking$ algorithm in [1]. We compare these two algorithms with respect to their complexity.

Complexity issues. We compare the High_Atomicity_Enhancement algorithm to high atomicity algorithm $Add\_masking$ from [1] for adding masking fault-tolerance to a fault-intolerant program. Since $Add\_masking$ tries to add both safety and liveness simultaneously, it is more complex than High_Atomicity_Enhancement presented in this paper. More specifically, the asymptotic complexity of High_Atomicity_Enhancement is less than that of $Add\_masking$. Thus, if the state space of the problem at hand prevents the addition of masking fault-tolerance to a fault-intolerant program, it may be possible to partially automate the design of a masking fault-tolerant program by manually designing a nonmasking fault-tolerant program and enhancing its fault-tolerance to masking using automated techniques.

We note that the asymptotic complexity of High_Atomicity_Enhancement is the same as the complexity of adding failsafe fault-tolerance to a fault-intolerant program. Thus, in High_Atomicity_Enhancement, the recovery is preserved for free!

5 Enhancement for Distributed Programs

In this section, we present an algorithm to enhance the fault-tolerance level of a *distributed* nonmasking fault-tolerant program

to masking. First, we discuss the issues involved in the enhancement problem for distributed programs. Then, we present our algorithm. As a case study, we apply our algorithm to the Byzantine agreement problem.

In high atomicity model, the main issue in enhancing the fault-tolerance level of a nonmasking fault-tolerant program p was to ensure that p does not execute a safety violating transition (s_0, s_1). In order to achieve this goal, we could either (i) ensure that p will never reach s_0, or (ii) remove (s_0, s_1). For the high atomicity model, we chose the latter option as it was strictly a better choice. However, for distributed programs, we cannot simply remove a safety violating transition (s_0, s_1) as (s_0, s_1) could be grouped with some other transitions (due to read restrictions). Thus, removal of (s_0, s_1) will also remove other transitions that are potentially useful recovery transitions. In other words, for distributed programs, the second choice is not necessarily the best option. Since an appropriate choice from the above two options cannot be identified easily for distributed programs, the synthesis of distributed programs becomes more difficult.

We develop our Low_Atomicity_Enhancement algorithm by tailoring the high atomicity algorithm to deal with this grouping of transitions. More specifically, given a nonmasking fault-tolerant program p, we first start by calculating a high atomicity fault-span, T'_{high} , which is closed in $p[]f$. Since the low atomicity model is more restrictive than the high atomicity model and T'_{high} is the largest fault-span for a high atomicity program (cf. [10]), we use T'_{high} as the domain of the states that may be included in the fault-span of our low atomicity program. In other words, if a transition, say (s_0, s_1) violates the safety specification and $s_0 \notin T'_{high}$ then we include the group associated with (s_0, s_1) and ensure that state s_0 is never reached.

Then, we call function LowAtomicityConstructInvariant ($LACI$) to calculate a low atomicity invariant S'_{low} for p'. To calculate S'_{low} , we first call function $HACI$ with $T'_{high} \cap S$ as its first argument. We ignore the fault transitions during this call to $HACI$; the effect of fault transitions is considered subsequently. In this call to $HACI$, we also ignore the grouping of transitions. These requirements are checked on the value, S'_{high} , returned by $HACI$. Specifically, if there exists a group containing transitions (s_0, s_1) and (s'_0, s'_1) such that $s_0, s'_0, s'_1 \in S'_{high}$ and $s_1 \notin S'_{high}$, we remove s_0 from S'_{high} and recalculate the invariant. If no such group exists, $LACI$ returns S'_{high}. Thus, our algorithm for constructing a low atomicity invariant is as follows:

LACI(S : state predicate, p: transitions,
$g_0, \cdots, g_m$: groups of transitions)
{ S'_{high} = HACI($S, p, \emptyset$);
if ($\exists g_i, s_0, s_1, s'_0, s'_1 : (s_0, s_1), (s'_0, s'_1) \in g_i :$
$(s_0, s'_0, s'_1 \in S'_{high} \wedge s_1 \notin S'_{high})$)
then return LACI($S'_{high} - \{s_0\}, p, g_0, \cdots, g_m$);
else return S'_{high};
}

The value returned by $LACI$, S'_{init}, is used as an estimate of the invariant of the masking fault-tolerant program. To compute T', we identify the effect of the fault and program transitions from states in S'_{init}. We use the variable S'_{low} to keep track of states reached in the execution of the program and fault transitions from S'_{init}. Our first estimate for S'_{low} is the same as S'_{init}.

Now, we compute S_2 as the set of states reached in one step (of program or fault). Regarding fault transitions, if (s_0, s_1) is a fault transition, $s_0 \in S'_{low}$ and $s_1 \in (T'_{high} - S'_{low})$ then we add state s_1 to the set S_2. Regarding program transitions, we only consider a group if the following three conditions are satisfied: (1) at least one of the transitions in it begins and ends in S'_{low}, (2) if a transition in that group begins in a state in T'_{high} then it terminates in a state in T'_{high} and it does not violate safety, and (3) if a transition in that group begins in a state in S'_{init} then it terminates in a state in S'_{init}. If such a group has another transition (s'_0, s'_1) such that $s'_0 \in S'_{low}$ and $s'_1 \notin S'_{low}$ then we include state s'_1 in the set S_2. (Note that in the first iteration, S'_{init} equals S'_{low}. Hence, expansion by program transitions need not be considered. However, this expansion may be necessary in subsequent iterations.) Thus, S_2 identifies states from where recovery must be preserved.

We then calculate the set of states from where recovery can be added, in one step. Specifically, if there is a transition (s_0, s_1) such that $s_0 \notin S'_{low}$ and $s_1 \in S'_{low}$ then we include s_0 in set S_3. We require that T'_{high} and S'_{init} are closed in the group being considered for recovery and that safety is not violated by any transition (that starts in a state in T'_{high}) in that group. Subsequently, we add S_3 to S'_{low}. The goal of this step is to ensure that infinite computations are possible from all states in S'_{low}. This result is true about the initial value (S'_{init}) of S'_{low}. Moreover, this property continues to be true since there is an outgoing transition from every state in S_3.

We continue this calculation until no new states can be added to S'_{low}. At this point, if S_2 is nonempty, i.e., there are states from where recovery needs to be added but no new recovery transitions can be added, we declare failure. Otherwise, we identify the transitions of fault-tolerant program p' by considering transitions of p–mt that start in a state in S'_{low}. Thus, our algorithm for enhancing the fault-tolerance of nonmasking distributed programs is as follows:

Low_Atomicity_Enhancement(p : transitions,
$g_0, \cdots, g_m$: groups of transitions, f: faults,
T, S : state predicate, $spec$: specification)
// $p = g_0 \cup g_1 \cup ... \cup g_m$
{ Calculate ms and mt as in High_Atomicity_Enhancement
T'_{high} = HACI($T - ms, p - mt, f$);
$S'_{init} = S'_{low}$ = LACI($S \cap T'_{high}, p - mt, g_0, \cdots, g_m$);
repeat {
$S_2 = \{s_1 : s_1 \in (T'_{high} - S'_{low}) :$
$(\exists s_0 : s_0 \in S'_{low} : (s_0, s_1) \in f \vee$
$(\exists g_i : (s_0, s_1) \in g_i : (((g_i | S'_{low}) \cap (p - mt)) \neq \phi) \wedge$
$(\forall s_2, s_3 : (s_2, s_3) \in g_i \wedge s_2 \in T'_{high} :$
$s_3 \in T'_{high} \wedge (s_2, s_3) \notin mt) \wedge$
$(\forall s_2, s_3 : (s_2, s_3) \in g_i \wedge s_2 \in S'_{init} : s_3 \in S'_{init}))$)}
$S_3 = \{s_0 : s_0 \in (T'_{high} - S'_{low}) :$
$(\exists s_1, g_i : (s_0, s_1) \in g_i \wedge s_1 \in S'_{low} :$
$(\forall s_2, s_3 : (s_2, s_3) \in g_i \wedge s_2 \in T'_{high} :$
$s_3 \in T'_{high} \wedge (s_2, s_3) \notin mt) \wedge$
$(\forall s_2, s_3 : (s_2, s_3) \in g_i \wedge s_2 \in S'_{init} : s_3 \in S'_{init}))\}$
$S'_{low} = S'_{low} \cup S_3$;
} until ($S_3 = \emptyset$);
if ($S_2 \neq \emptyset$) then *declare fault-tolerance cannot be enhanced;* exit().
$T' = S'_{low}$;
$p' = \{g_i : (\forall s_0, s_1 : (s_0, s_1) \in g_i :$
$(s_0 \in T' \Rightarrow (s_1 \in T' \wedge (s_0, s_1) \in (p - mt))) \wedge$
$(s_0 \in S'_{init} \Rightarrow s_1 \in S'_{init}))\}$;
return p', T';
}

Theorem 5.1 The algorithm Low_Atomicity_Enhancement is sound and its complexity is polynomial in the state space of the nonmasking fault-tolerant program (For proof, please see [10].). □

Modifications/Improvements for Low Atomicity Enhancement. There are several improvements that can be made for the above algorithm. We discuss these improvements and issues related to completeness below.

(1) In the above algorithm, if the value of S_2 is the empty set then we can break out of the loop before computing S_3. Subsequently, we can use value of S'_{low} at that time to compute p' and T'. However, we continue in the loop to determine whether recovery can be added from new states. This allows the possibility that a larger fault-span is computed and additional transitions are included in the masking fault-tolerant program (As mentioned in [1], this is a desirable feature for an automatic algorithm).

(2) In the above algorithm, in the calculation of S_3, we calculate states from where recovery is possible. One heuristic is to focus on states in S_2 first as recovery must be added from states in S_2. If recovery from states in S_2 is not possible then other states in $T'_{high} - S'_{low}$ should be considered. However, considering states in S_2 alone may be insufficient as it may not be possible to add recovery from those states in one step; adding recovery from other states can help in recovering from states in S_2.

(3) Our algorithm is incomplete in that it may be possible to enhance the fault-tolerance of a given nonmasking program although our algorithm fails to find it. One of the causes for incompleteness is in our calculation of $LACI$; when $LACI$ needs to remove states/transitions to deal with grouping of transitions, the choice is non-deterministic. Since this choice may be inappropriate, the algorithm is incomplete. Based on [11], where we showed that adding failsafe fault-tolerance to distributed programs is NP-hard, it is expected that the complexity of a deterministic sound and complete algorithm for enhancing the fault-tolerance of a distributed nonmasking program will be exponential unless $P = NP$.

5.1 Example: Byzantine Agreement

We show how our algorithm for the low atomicity model is used to enhance the fault-tolerance level of a nonmasking Byzantine agreement program to masking. First, we present the nonmasking program, its invariant, its safety specification, faults, the fault-span for the given faults, and read/write restrictions. Finally, we show how our algorithm is used to obtain the masking program (in [4]) for Byzantine agreement.

Variables for Byzantine agreement. The nonmasking program consists of three non-general processes j, k, l and a general g. Each non-general process has three variables d, f, and b. Variable $d.j$ represents the decision of a non-general process j, $f.j$ denotes whether j has finalized its decision, and $b.j$ denotes whether j is Byzantine or not. Process g also has a variable $d.g$ and $b.g$. Thus, the variables and their corresponding domains in the Byzantine agreement program are as follows: $d.g : \{0,1\}$; $d.j, d.k, d.l : \{0,1,\bot\}$; $b.g, b.j, b.k, b.l : \{true, false\}$, and $f.j, f.k, f.l : \{0,1\}$.

Transitions of the nonmasking program. If process j has not copied a value from the general, action $NB1$ copies the decision of the general. If j has copied a decision and as a result $d.j$ is different from $\bot$ then j can finalize its decision by action $NB2$. If process j reaches a state, where its decision is not equal to the majority of decisions and all the non-general processes have decided then j corrects its decision by actions $NB3$ or $NB4$. Thus, the actions of each process j in the nonmasking program are as follows:

$$NB1: \ d.j = \bot \ \wedge \ f.j = 0 \quad \longrightarrow \quad d.j := d.g$$
$$NB2: \ d.j \neq \bot \ \wedge \ f.j = 0 \quad \longrightarrow \quad f.j := 1$$
$$NB3: \ (d.j = 1) \ \wedge \ (d.k = 0) \ \wedge \ (d.l = 0) \longrightarrow d.j := 0$$
$$NB4: \ (d.j = 0) \ \wedge \ (d.k = 1) \ \wedge \ (d.l = 1) \longrightarrow d.j := 1$$

Safety specification. The safety specification requires that if g is Byzantine, all the non-general non-Byzantine processes should finalize with the same decision (*agreement*). If g is not Byzantine, then the decision of every non-general non-Byzantine process should be the same as $d.g$ (*validity*). Thus, safety is violated if the program reaches a state in S_{sf}, where (in this section, unless otherwise specified, quantifications are on non-general processes)

$$\begin{aligned} S_{sf} = \ & (\exists p, q :: \neg b.p \wedge \neg b.q \wedge d.p \neq \bot \wedge d.q \neq \bot \wedge \\ & \qquad d.p \neq d.q \wedge f.p \wedge f.q) \vee \\ & (\exists p :: \neg b.g \wedge \neg b.p \wedge d.p \neq \bot \wedge d.p \neq d.g \wedge f.p) \end{aligned}$$

Also, a transition violates safety if it changes the decision of a process after it has finalized. Thus, the set of transitions that violate safety is equal to t_{sf}, where

$$\begin{aligned} t_{sf} = \ & \{(s_0, s_1) : s_1 \in S_{sf}\} \ \cup \ \{(s_0, s_1) : \\ & \quad \exists p :: \neg b.p(s_0) \wedge \neg b.p(s_1) \wedge f.p(s_0) = 1 \wedge \\ & \qquad (d.p(s_0) \neq d.p(s_1) \vee f.p(s_0) \neq f.p(s_1))\} \end{aligned}$$

Invariant. The invariant of nonmasking Byzantine agreement is the state predicate $S_{NB} = S_{NB_1} \ \vee \ S_{NB_2}$, where

$$\begin{aligned} S_{NB_1} = \ & \neg b.g \wedge (\neg b.j \vee \neg b.k) \wedge (\neg b.k \vee \neg b.l) \wedge (\neg b.l \vee \neg b.j) \wedge \\ & \quad (\forall p :: \neg b.p \Rightarrow (d.p = \bot \vee d.p = d.g)) \ \wedge \\ & \qquad (\forall p :: (\neg b.p \wedge f.p) \Rightarrow (d.p \neq \bot)), \quad \text{and} \\ S_{NB_2} = \ & b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l \ \wedge (d.j = d.k = d.l \ \wedge \ d.j \neq \bot) \end{aligned}$$

Read/Write restrictions. Each non-general process j is allowed to read $\{b.j, d.j, f.j, d.k, d.l, d.g\}$. Thus, j can read the d values of other processes and all its variables. The set of variables that j can write is $\{d.j, f.j\}$.

Faults for Byzantine agreement. A fault transition can cause a process to become Byzantine if no process is initially Byzantine. A fault can also change the d and f values of a Byzantine process. Thus, the fault transitions that affect j are as follows (We include similar fault-transitions for k, l, and g):

$$F1: \neg b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l \quad \longrightarrow \quad b.j := true$$
$$F2: b.j \quad \longrightarrow \quad d.j, f.j := 0|1, 0|1$$

Fault-Span. Starting from a state in S_{NB_1}, if no process is Byzantine then a fault transition can cause one process to become Byzantine. Then, faults can change the d and f values of the Byzantine process. Now, if the faults do not cause g to become Byzantine then the set of states reached from S_{NB_1} is the same as S_{NB_1}. However, if the faults cause g to become Byzantine then the d and f values of non-general processes may be arbitrary. However, the b values of non-general processes will remain false. Thus, the set of states reached from S_{NB_1} is $(S_{NB_1} \cup (b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l))$.

Starting from S_{NB_2}, no process can become Byzantine. Hence, the d values of non-general processes will remain unchanged. It follows that the set of states reached from S_{NB_2} is S_{NB_2}. Finally, since S_{NB_2} is a subset of $(b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l)$, the set of states reached from S_{NB} is T_{NB}, where

$$T_{NB} = S_{NB_1} \cup (b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l)$$

Application of our algorithm. First, we compute ms and mt that are needed by our algorithm. Every fault transition originating at S_{sf} reaches S_{sf} because it only affects the Byzantine process and the destination state will remain in S_{sf}. Since the destination of these fault transitions is S_{sf}, they violate the safety. Thus, the set of states from where faults alone violate safety is equal to S_{sf}, and as

a result $ms = S_{sf}$. Since t_{sf} includes all the transitions that reach S_{sf} (which is equal to ms) or violate safety, $mt = t_{sf}$.

To calculate T'_{high}, we use the $HACI$ function of our high atomicity algorithm. This function removes deadlock states and states from where the closure of T'_{high} is violated by fault transitions. Since we have removed ms states and no fault transition can reach a state in ms from a state outside ms, there exists no state from where the closure of T'_{high} can be violated by fault transitions. Now, consider a state, say s_0, where $d.j = 0, d.k = 0, d.l = 1, b.l = false$, and $f.l = 1$. Clearly, s_0 is a deadlock state as no process can execute a safe transition from s_0. Hence, such states must be removed while obtaining T'_{high}.

Now, consider a state, say s_1, where $d.j = \bot, d.k = 0, d.l = 1, b.l = false$, and $f.l = 1$. In state s_1, only process j can execute a transition (by copying $d.g$) without violating safety. However, if j copies the value of the general and $d.g = 0$, the program reaches a state that was removed earlier. Hence, such states must also be removed while obtaining T'_{high}. Continuing thus, we remove all states where a process in the *minority* has finalized its decision. In other words, T'_{high} is equal to $T_{NB} - X$, where

$$X = \{s : \ (\exists p :: f.p(s) = 1 \wedge (\forall q : p \neq q : d.p(s) \neq d.q(s)))\}$$

After this step, function $LACI$ returns $S'_{init} = T'_{high} \cap S_{NB}$. Now, we trace two iterations of the main loop in our algorithm in order to illustrate the way that our algorithm works.

First iteration. To calculate S_2, we search for states in S'_{init} from where we can directly reach a state in $T'_{high} - S'_{init}$ by fault transitions or by program transitions. From S'_{init}, no program transition can reach a state that is outside S'_{init}. However, from a state s, where $(\neg(d.j(s) = d.k(s) = d.l(s)) \vee (\exists p :: d.p(s) = \bot))$, a fault transition can cause the general to become Byzantine and then the program is outside S'_{init}. Hence, in the first iteration, $S_2 = \{s : s \in (T'_{high} - S'_{init}) : b.g(s) \wedge (\neg(d.j(s) = d.k(s) = d.l(s)) \vee (\exists p :: d.p(s) = \bot)) \wedge (\forall p : (d.p(s) \neq \bot) \Rightarrow (d.p(s) = d.g(s)))\}$.

Now, we compute S_3. Consider a state, say s_0, where $d.j = 0, d.k = 0, d.l = 1, b.l = false$, and $f.l = 0$. In s_0, l can change $d.l$ to 0 and reach a state in S'_{init}. Hence, such states are included in S_3. Also, consider a state, say s_1, where $d.j = \bot, d.k = 1, d.l = 1$, and $d.g = 1$. In s_1, process j can copy the value of $d.g$ and take the program to S'_{init}. Therefore, in this iteration $S_3 = P_1 \cup P_2$, where

$$\begin{aligned} P_1 = \{&s : s \in (T'_{high} - S'_{init}) : (\exists p : (d.p(s) \neq \bot) \wedge (f.p(s) = 0) : \\ &(\forall q : (q \neq p) : (d.q(s) \neq \bot) \wedge d.p(s) \neq d.q(s)))\}, \text{ and} \\ P_2 = \{&s : s \in (T'_{high} - S'_{init}) : \ (\exists p : d.p(s) = \bot : \\ &(\forall q : q \neq p : (d.q(s) \neq \bot) \ \wedge \ (d.q(s) = d.g(s))))\}. \end{aligned}$$

Then, we add S_3 states to S'_{low}.

Remark. In the case of Byzantine agreement, the only states from where recovery to S'_{init} can be achieved in a single step are the states of S_3 in the first iteration. Every other recovery path includes these states as its final step to S'_{init}.

Second iteration. In the second iteration $S'_{low} = S'_{init} \cup S_3$ (S_3 in the first iteration.). To calculate S_2 in the second iteration, we search for states in S'_{low} from where we can directly reach a state in $T'_{high} - S'_{low}$ by fault transitions or by program transitions.

To calculate S_2 in the second iteration, we need to calculate the set of states in $T'_{high} - S'_{low}$ that are reachable by a fault transition from S'_{low}. From the first iteration, we already know the set of states reachable from S'_{init}. Thus, we only need to calculate the states of $T'_{high} - S'_{low}$ that are reachable by a fault transition from recently joined states (i.e., $S_3 = P_1 \cup P_2$ of the first iteration) to S'_{low}. Since in P_1 the general process is Byzantine and all non-generals have decided, P_1 is closed in fault transitions. However, since g is Byzantine, faults may change the value of $d.g$ in a state in P_2 and take the program outside S'_{low}. In these states, the condition $(\exists p : d.p = \bot : (\forall q : q \neq p : (d.q \neq \bot) \wedge (d.q \neq d.g)))$ holds. Therefore, in this iteration, the program can reach states of S_2 by a fault transition, where

$$\begin{aligned} S_2 = \{&s : s \in (T'_{high} - S'_{low}) : b.g(s) \wedge \\ &(\exists p : d.p = \bot : (\forall q : q \neq p : (d.q \neq \bot) \wedge (d.q \neq d.g)))\} \end{aligned}$$

To calculate S_3, we find states from where recovery is possible to S'_{low}. Thus, we search for states from where we can reach the states of S_3 calculated in the first iteration. Hence, in this iteration, single-step recovery to S'_{low} is possible from S_3, where

$$\begin{aligned} S_3 = \{&s : s \in (T'_{high} - S'_{low}) : \\ &(\exists p : (d.p(s) = \bot) : (\forall q : q \neq p : d.q(s) \neq \bot)) \vee \\ &(\exists p : (d.p(s) \neq \bot) \wedge (d.p(s) = d.g(s)) : \\ &(\forall q : q \neq p : d.q(s) = \bot))\} \end{aligned}$$

Continuing thus, we get the masking fault-tolerant Byzantine agreement; this program is the same as that in [4]. The actions of this program are as follows:

$$\begin{aligned} MB1: \ & d.j = \bot \wedge \ f.j = 0 && \longrightarrow d.j := d.g \\ MB2: \ & d.j \neq \bot \ \wedge \ f.j = 0 \wedge \\ & \quad ((d.j = d.k) \vee (d.j = d.l)) && \longrightarrow f.j := 1 \\ MB3: \ & (d.j = 1) \ \wedge \ (d.k = 0) \wedge \\ & \quad (d.l = 0) \ \wedge \ (f.j = 0) && \longrightarrow d.j := 0 \\ MB4: \ & (d.j = 0) \ \wedge \ (d.k = 1) \wedge \\ & \quad (d.l = 1) \ \wedge \ (f.j = 0) && \longrightarrow d.j := 1 \end{aligned}$$

5.2 Enhancement versus Addition

In this section, we compare the cost of adding masking fault-tolerance to a fault-intolerant distributed program and the cost of enhancing the fault-tolerance of a nonmasking fault-tolerant distributed program to masking.

Asymptotically speaking, adding masking (respectively, failsafe) fault-tolerance to a fault-intolerant distributed program is NP-hard [1, 11]. Therefore, it is expected that the enhancement problem — that adds safety while preserving recovery— for distributed programs will also be NP-hard. Although the enhancement problem may not provide relief in terms of the worst-case complexity, we find that it helps in developing heuristics that determine if safe recovery is possible from states that are reached in the presence of faults. More specifically, consider a state, say s, that is reached in a computation of the fault-intolerant program in the presence of faults. While adding masking fault-tolerance to a fault-intolerant program, we need to exhaustively search all possible transition sequences from s to determine if recovery from s is possible. By contrast, while enhancing the fault-tolerance of a nonmasking fault-tolerant program, we reuse the recovery provided by the nonmasking fault-tolerant program. Hence, we need to check only the transition sequences that the nonmasking fault-tolerant program can produce. It follows that deriving heuristics that determine if safe recovery is possible from a given state is simpler in the enhancement problem.

The enhancement problem also allows us to deduce additional information about states by reasoning in the high atomicity model. We illustrate this by one example that occurs in Byzantine agreement. Consider a state s_0 where all processes are non-Byzantine,

$d.j = d.k = \bot$, $d.g = 1$, $d.l = 1$ and $f.l = 0$. Let s_1 be a state that is identical to s_0 except that the value of $f.l$ in s_1 is 1. Now, consider the transition (s_0, s_1). Note that both s_0 and s_1 are in the invariant, S_{NB}. Hence, for a synthesis algorithm, this appears as a good transition that should be retained in the fault-tolerant program. However, from s_1, if g becomes Byzantine and changes $d.g$, we can reach a state where $d.g, d.j$, and $d.k$ become 0. The resulting state is a deadlock state.

While adding masking fault-tolerance to a fault-intolerant program, it is difficult to check that all computations that (1) start from s_1, (2) in which g becomes Byzantine, and (3) in which g changes $d.g$ to 0 are deadlock states. Moreover, if we ignore the grouping restrictions imposed by the low atomicity model, i.e., if we could read and write all variables in one atomic step, recovery is possible from s_1. However, in the context of the enhancement problem, we concluded that even in the high atomicity model, we could not recover from state s_1 by reusing the transitions of the nonmasking fault-tolerant program.

We expect that such high atomicity reasoning will play an important role in reducing complexity in the enhancement problem. To reduce the complexity of adding fault-tolerance in the low atomicity model, it is desirable to reason about the input program in the high atomicity model, obtain a high atomicity masking fault-tolerant program, and modify that high atomicity masking fault-tolerant program so that the restrictions of the low atomicity model are satisfied while preserving the masking fault-tolerance. As the Byzantine agreement example illustrates, this approach can be followed while enhancing the fault-tolerance of a nonmasking fault-tolerant program. However, this approach could not be used while adding masking fault-tolerance to a fault-intolerant program.

6 Conclusion and Future Work

In this paper, we presented the problem that dealt with enhancing the fault-tolerance level of a nonmasking program to masking. This problem separates (1) the task of recovery, and (2) the task of maintaining the safety specification during recovery. For the high atomicity model, we presented a sound and complete algorithm for the enhancement problem. We showed that the complexity of our high atomicity algorithm is asymptotically less than $Add\_masking$ algorithm from [1]. For distributed programs, we presented a sound algorithm for the enhancement problem. We also showed that our fault-tolerance enhancement algorithm for distributed programs resolves some of the difficulties encountered in adding safe recovery transitions in [2].

As an illustration of our algorithms, we showed how masking fault-tolerant programs for TMR (in high atomicity model) and Byzantine agreement (for distributed programs) can be designed by enhancing the fault-tolerance of the corresponding nonmasking programs. We chose these examples as masking fault-tolerant versions of these programs have been *manually* designed from the corresponding nonmasking fault-tolerant versions [3]. The results in this paper show that those enhancements can in fact be automated as well. Also, we argued that enhancing the fault-tolerance of a distributed program is simpler than adding fault-tolerance to its fault-intolerant version. We validated this result by comparing the derivation of a masking fault-tolerant Byzantine agreement program from the corresponding fault-intolerant version and from the corresponding nonmasking version.

The synthesis approach in this paper focuses on addition of fault-tolerance to an existing program. Thus, it differs from solutions (e.g., [9, 12–14]) where one begins with a specification and obtains a fault-tolerant program. Related work on addition of fault-tolerance to an existing program includes [1,2,11]. In [1], authors have shown that the problem of adding masking fault-tolerance to distributed programs is NP-hard. A heuristic based solution for adding masking fault-tolerance is presented in [2]. The algorithm in [2] adds safety and recovery simultaneously to a fault-intolerant program. As discussed in Sections 5.2, we simplify the addition of recovery transitions by reasoning about the feasibility of adding recovery in the high atomicity model and using the acquired knowledge in the low atomicity model to reduce the complexity. Hence, our enhancement algorithm is simpler than the polynomial time synthesis algorithm of [2]. Thus, if a nonmasking fault-tolerant program is available (or can be manually designed) then our approach simplifies the design of masking fault-tolerance.

References

[1] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 2000. Also available as a Technical Report MSU-CSE-00-13 at Computer Science and Engineering Department, Michigan State University, East Lansing, Michigan.

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

[3] A. Arora and S. S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Transactions on Software Engineering*, pages 435–450, June 1998. A preliminary version appears in the Proceedings of the Fourteenth Symposium on Reliable Distributed Systems, Bad Neuenahr, 1995, pages 174–185.

[4] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 1982.

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

[8] P. Attie and E. Emerson. Synthesis of concurrent systems for an atomic read/write model of computation. *ACM Symposium on the Principles of Distributed Computing (PODC)*, 1996.

[9] P. Attie and A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM TOPLAS*, 23(2), March 2001.

[10] Sandeep S. Kulkarni and Ali Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. Technical Report MSU-CSE-03-2, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, February 2003.

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

[12] 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.

[13] 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.

[14] O. Kupferman and M. Vardi. Synthesis with incomplete information. *International Conference on Temporal Logic*, 1997.