Polynomial Time Synthesis of Byzantine Agreement[1]

Sandeep S. Kulkarni* Anish Arora† Arun Chippada*

*Department of Computer Science and Engineering Michigan State University East Lansing MI 48824 USA

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

Abstract

We present a polynomial time algorithm for automatic synthesis of fault-tolerant distributed programs starting from fault-intolerant versions of those programs. Since this synthesis problem is known to be NP-hard, our algorithm relies on heuristics to reduce the complexity. We demonstrate that our algorithm suffices to synthesize an agreement program that tolerates a byzantine fault.

Keywords : Fault-tolerance, Formal methods, Program synthesis, Program transformation, Distributed programs

1 Introduction

We focus our attention on automating the synthesis of fault-tolerant distributed programs. The synthesis begins with a fault-intolerant program that is assumed to be correct in the absence of faults, and designs a fault-tolerant program that is *derived* from that fault-intolerant program. In [1], we showed that the problem of adding ("masking") fault-tolerance is NP-hard, and presented a non-deterministic polynomial time algorithm for synthesizing fault-tolerant programs. It follows that a brute-force deterministic implementation of that algorithm will be exponential in the state space of the fault-intolerant program.

An exponential algorithm for synthesis would limit application only to those programs whose state space is small. One way to redress this limitation is to identify a set of heuristics under which the synthesis algorithm takes polynomial time. To develop these heuristics, we analyze the algorithm in [1] to identify all situations where a non-deterministic choice is made in that algorithm. The heuristics are designed to enable a deterministic choice in those situations. It follows that if the heuristics are applicable for a given synthesis problem, our algorithm will synthesize a fault-tolerant program in polynomial time. Otherwise, our algorithm will fail by declaring that it cannot synthesize a fault-tolerant program for the given problem.

To justify our heuristics, we show how they suffice for synthesizing a program for the byzantine agreement problem [2]. This problem is recognized as an important but difficult one for distributed computing (e.g., [3, 4]). Perhaps due to the difficulties in characterizing byzantine faults or due to the logical complexities in the agreement problem, extant synthesis algorithms do not suffice for the synthesis of a solution to this problem. To deal with these difficulties, we introduce *auxiliary* variables to characterize byzantine faults, and we capture the semantics of the agreement problem by using those auxiliary variables in the specification. For pedagogical reasons, we present our heuristics hand-in-hand with the synthesis of byzantine agreement.

To appreciate the complexity involved in synthesizing byzantine agreement, consider the state space used in the canonical version of the problem where there are four processes; one general process and three non-general processes. For this version, we need the following variables: For the general, there is a variable d (domain 0 and 1) which denotes the decision of the general and a boolean variable b which denotes whether the general is byzantine. And, for each of the three non-general processes, there is a variable d (domain $0, 1, \perp$(= uninitialized decision)) which denotes the decision of that non-general process, a variable f (domain 0 and 1) which denotes whether that non-general process has finalized its decision, and a variable b which denotes whether that non-general process is byzantine. Thus, the state space of the canonical agreement program contains 6912 $(= 4 * 12^3)$ states. Clearly, one cannot execute an algorithm that is exponential in this state space. But, as we show, our polynomial algorithm solves this problem efficiently. As further evidence of the scope of our heuristics, we have also shown [5] how the algorithm can be used to synthesize the solution for the agreement problem where a process suffers from the byzantine fault and a process suffers from the failstop fault.

The algorithm in this paper has been implemented in Java. This implementation will be a part of the synthesis platform that we are building to test the current and future heuristics.

Organization of the paper. This paper is organized as follows: We provide the definitions of programs, specifications, faults and fault-tolerance in Section 2. Using these definitions, in Section 3, we precisely state the transformation problem that characterizes what it means for a fault-tolerant program to be derived from a fault-intolerant program. In Section 4, we define the model of distributed programs, and identify the reasons behind the high complexity while adding fault-tolerance in distributed programs. In Section 5, we show how the byzantine faults are represented, how the requirements of the byzantine agreement are captured, and present the fault-intolerant program for byzantine agreement. In Section 6, we present our algorithm and its application in synthesizing fault-tolerant byzantine agreement. Finally, we make concluding remarks in Section 7.

[1]Email: sandeep@cse.msu.edu, anish@cis.ohio-state.edu, chippada@cse.msu.edu Web: `http://www.cse.msu.edu/~sandeep`, `http://www.cis.ohio-state.edu/~anish`, `http://www.cse.msu.edu/~chippada` Tel: +1-517-355-2387. This work was partially sponsored by NSA Grant MDA904-96-1-0111, NSF Grant NSF-CCR-9972368, NSF CAREER CCR-0092724, DARPA contract F33615-01-C-1901, ONR Grant N00014-01-1-0744, an Ameritech Faculty Fellowship, a grant from Microsoft Research, and a grant from Michigan State University.

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 [6]. And, the definition of faults and fault-tolerance is adapted from [7, 8].

2.1 Program

A program p is a tuple $\langle S_p, \delta_p \rangle$ where S_p is a finite set of states and δ_p is a subset of $\{(s_0, s_1) : s_0, s_1 \in S_p\}$. A state predicate of $p(= \langle S_p, \delta_p \rangle)$ is any subset of S_p. A state predicate S is closed in the program 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 $p(= \langle S_p, \delta_p \rangle)$ 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 $(s_l, s) \in \delta_p$.

The projection of program p on state predicate S, denoted as $p|S$, is the program $\langle S_p, \{(s_0, s_1) : (s_0, s_1) \in \delta_p \ \wedge \ s_0, s_1 \in S\} \rangle$. I.e., $p|S$ 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 $\delta'_p \subseteq \delta_p$.

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

To concisely write the transitions in a program, we use actions. An action is of the form $g \longrightarrow st$, where g is a state predicate, and st is a statement that describes how the program state is updated. Thus, an action $g \longrightarrow st$ denotes the set of transitions $\{(s_0, s_1) : g$ is true in s_0 and s_1 is obtained by changing s_0 as prescribed by $st\}$.

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

Following Alpern and Schneider [6], we let the specification consist of a safety specification and a liveness specification. For our transformation algorithm, the safety specification is specified in terms of a set of bad transitions that should not occur in the program computation. I.e., for program p, its safety specification is a subset of $\{(s_0, s_1) : s_0, s_1 \in S_p\}$. We show that the fault-tolerant program satisfies the liveness specification iff the fault-intolerant program satisfies the liveness specification. Since the initial fault-intolerant program satisfies its specification (including the liveness specification), the liveness specification need not be specified explicitly.

(Since the specification is suffix closed, it is always possible to specify the safety specification as a set of bad transitions. For reasons of space, we refer the reader to [8] for the proof of this claim.)

Given a program p, a state predicate S, and a specification $spec$, we say that p refines $spec$ from S iff (1) S is closed in p, and (2) Every computation of p that starts in a state where S is true is in $spec$. If p refines $spec$ 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 $spec$ iff there exists a sequence of states β such that $\alpha\beta \in spec$. Otherwise, we say that α violates $spec$.

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 $spec$.

2.3 Faults

The faults that a program is subject to are systematically represented by transitions. Specifically, 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.

Just as we defined the computation of p, we say that a sequence of states, $\langle s_0, s_1, ... \rangle$, is a computation of $p(= \langle S_p, \delta_p \rangle)$ in the presence of f iff the following three conditions are satisfied: (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) $\exists n : n \geq 0 : (\forall j : j > n : (s_{j-1}, s_j) \in \delta_p)$. Thus, in each step, either a program or a fault transition is executed. The computation may terminate in a state from where there are no program transitions. In other words, fault transitions may not be used to obtain progress from deadlocked states. And, the number of fault occurrences in a computation are finite.

Using the above definitions, we now define what it means for a program to be fault-tolerant. We say that p is f-tolerant (read as fault-tolerant) to $spec$ from S iff the following two conditions hold:

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

3 The Transformation Problem

In this section, we formally define the transformation problem that characterizes what it means for a fault-tolerant program, say p', to be derived from a fault-intolerant program, say p. Our definition of derivation is based on the premise that p' is obtained by only adding fault-tolerance to p, i.e., p' does not introduce new ways of refining $spec$ when no faults have occurred. More specifically, if S is the invariant of p and S' is the invariant of p', we would like to be able to prove that p' refines its $spec$ from S' by only using the assumption that p refines $spec$ from S. Towards this end, we identify the relation between S and S' and p and p'.

Since p refines $spec$ from S, we have no knowledge about the behavior of p if it starts from a state outside S. Hence, if S' contains a state outside S, we cannot prove that p' refines $spec$ from S' by only using the assumption that p refines $spec$ from S. Hence, we require that $S' \subseteq S$.

Likewise, if $p'|S'$ includes a transition that is not in p, we would not be able to prove that a computation of p' that uses that transition is in $spec$. Hence, we require that $p'|S' \subseteq p$ (or equivalently, $p'|S' \subseteq p|S'$). (Of course p' may contain additional transitions that originate outside S'.) Thus, we define the transformation problem as follows:

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

The above problem is motivated from our previous work [8] that shows that the functionality of a fault-tolerant program in the absence of faults and its fault-tolerance can be separated. Thus, the above transformation problem captures the notion that we should reuse the functionality aspect given by the fault-intolerant program and design only the fault-tolerance aspect.

Remark. Our notion of derivation suggests that we should consider a program which has the weakest invariant and maximal nondeterminism. Also, it requires that the fault-intolerant program should be such that it contains all the variables needed for the fault-tolerant program as new variables cannot be added during synthesis. We have imposed this restriction to permit the addition of tolerance to different types of faults; if new variables could be added while deriving a fault-tolerant program then one needs to determine how the faults may affect them and what are the legitimate values for those variables. And, the answers to these questions are fault-dependent.

4 The Transformation Problem for Distributed Programs

In this section, we present the distributed program model that specifies how read/write restrictions imposed on a distributed program are captured during synthesis. We identify this model and difficulties associated with synthesizing distributed programs in Section 4.1. Then, in Section 4.2, we present the non-deterministic algorithm, Add_ft (from [1].) which is made deterministic in Section 6 using heuristics.

4.1 Modeling Distributed Programs

Recall that a program p consists of a state space S_p and transitions δ_p. For distributed programs, we introduce the notion of variables and their (finite) domains. Given a set of variables $x_1, ..., x_n$ with domains $d_1, ..., d_n$, we obtain the state space as $\{\langle v_1, ..., v_n\rangle : v_1 \in d_1 \wedge ... \wedge v_n \in d_n\}$. Thus, a state is obtained by giving a value to each variable. Also, to capture the notion that all variables cannot be read/written simultaneously, we introduce the notion of processes; a process specifies the set of variables it can read and a set of variables it can write.

In this section, we describe how we determine whether a transition (s_0, s_1) can be used while synthesizing the transitions of a process based on the read/write restrictions imposed on that process.

We first define the following two notations.

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

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

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

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

To represent the write-restrictions, for each transition, we associate the process (or fault) that is responsible to execute it. And, if transition (s_0, s_1) is associated with process j and $(s_0, s_1) \in write(j, w_j)$ then we say that (s_0, s_1) violates safety.

Read-restrictions. Unlike write-restrictions that create no new difficulties, read restrictions are difficult to deal with. In this paper, for simplicity, we consider the case where $w_j \subseteq r_j$, i.e., we assume that j cannot blindly write a variable. (A more general case is discussed in [1]; we omit it here as this simple case suffices for our current problem.) Let (s_0, s_1) be some transition of process j such that $s_0 \neq s_1$. Now, consider a state s'_0 such that the values of all variables in r_j are identical to that in s_0. Since j can only read variables in r_j, j cannot distinguish between s_0 and s'_0. Hence, j must have a transition of the form (s'_0, s'_1) such that s_1 and s'_1 are identical as far as j is concerned, i.e., the values of variables in r_j in s'_1 must be the same as that in s_1. And, the values of variables outside w_j in state s'_1 must be the same as that in s'_0. Moreover, since $w_j \subseteq r_j$, it follows that the values of variables outside r_j must remain unchanged in the transition (s'_0, s'_1). Considering all states where the values of r_j are same, we get a group of transitions; if (s_0, s_1) is a transition of j then all transitions in that group must also be transitions of j. We define these transitions as $group(j, r_j)(s_0, s_1)$ for the case $w_j \subseteq r_j$, where

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

Thus, the inability of a process to read is characterized in terms of grouping of transitions. Note that while describing the read/write restrictions, we only considered transitions $\{(s_0, s_1) : s_0 \neq s_1\}$. If a program p includes a transition of the form (s_0, s_0) and s_0 is reached in the program computation, it is possible that all subsequent states in that computation are same as s_0. Now, in the context of the transformation problem, consider two cases (1) $s_0 \in S'$ and (2) $s_0 \notin S'$. In the first case, from the second requirement of the transformation problem, the transition (s_0, s_0) can be included in p' iff (s_0, s_0) is included in p. In the second case, the transition (s_0, s_0) can be included in p' only if state s_0 is never reached in the execution of $p'[]f$.

The grouping introduced by the read restrictions increase the complexity of synthesizing distributed programs. To support this claim, we consider the case where transitions (s_0, s_1) and (s_2, s_3) are grouped together, (s_0, s_1) is a desirable transition (e.g., because it is used to satisfy the specification in the absence of faults), and (s_2, s_3) should never be executed (e.g., because it causes the safety specification to be violated). In this scenario, the transformation algorithm has two choices (1) include this group and ensure that s_2 is never reached, or (2) exclude this group and lose the useful transition (s_0, s_1). Thus, the transformation algorithm needs to perform a tradeoff between states and transitions. We have used this crucial fact to show in [1] to show that the transformation problem is NP-hard.

4.2 Non-deterministic Algorithm for Distributed Program Synthesis

We now present the non-deterministic algorithm Add_ft (from [1]) that solves the transformation problem. The Add_ft algorithm first computes the set ms; a state s is included in ms iff execution of fault transitions alone from s can violate safety. It follows that if the fault-tolerant program ever reaches a state in ms then execution of fault transitions can violate safety. In other words, the fault-tolerant

program should not reach a state in ms. The Add_ft algorithm then computes mt; a transition is in mt iff either (1) it violates safety or (2) it reaches a state in ms. It follows that a fault-tolerant program should not execute a transition in mt. Then, the program non-deterministically guesses the fault-tolerant program, p', its invariant, S' and its fault-span, T'. Finally, by using the values of ms and mt, it verifies that the three conditions of the transformation problem (cf. Section 3) are satisfied. Thus, the non-deterministic algorithm, Add_ft, is as shown in Figure 1.

Add_ft(p, f : transitions, S : state predicate, $spec$: specification,
$g_0, g_1, ..., g_{max}$: groups of transitions)
{ $ms := \{s_0 : \exists s_1, s_2, ...s_n : (\forall j : 0 \leq j < n : (s_j, s_{(j+1)}) \in f)$
$\wedge \quad (s_{(n-1)}, s_n)$ violates $spec$ };
$mt := \{(s_0, s_1) : ((s_1 \in ms) \ \vee \ (s_0, s_1)$ violates $spec)$ };

Guess S', T', and
$p' := \bigcup (g_i : g_i$ is chosen to be in the fault-tolerant program);
Verify the following
(F1) $p'|S' \subseteq p|S'$;
(F2) $S' \Rightarrow T'$; T' is closed in $p'[]f$;
(F3) $T' \cap ms = \{\}$; $(p'|T') \cap mt = \{\}$;
(F4) $(\forall s_0 : s_0 \in T' : (\exists s_1 :: (s_0, s_1) \in p'))$;
(F5) $S' \neq \{\}$; $S' \subseteq S$; S' is closed in p';
(F6) $p'|(T' - S')$ is acyclic
}

Figure 1. Program Add_ft

The algorithm Add_ft verifies the six formulae given above: The first formula $F1$ checks that $p'|S' \subseteq p|S'$ is true. The second formula, $F2$, checks that T' is a valid fault-span. The third formula, $F3$, checks that safety is not violated from any state in T'. The fourth formula, $F4$, checks that the program does not deadlock in a state in T'. The fifth formula, $F5$, checks that S' is a valid invariant, i.e., S' is nonempty and S' is closed in p'. That formula also checks if S' is a subset of S. Finally, the last formula, $F6$, checks that the program cannot stay in $T' - S'$ forever. In this paper, we develop heuristics that are used to obtain a deterministic implementation of Add_ft.

5 Byzantine Faults : Model and Fault-Intolerant Program

In this section, we first identify, in Section 5.1, the state space for the canonical version of the byzantine agreement problem. Then, we specify the safety specification for byzantine agreement in Section 5.2. We specify the fault-intolerant program for byzantine agreement in Section 5.3, and show how the faults are modeled in Section 5.4.

5.1 State Space for Byzantine Agreement

The program consists of a "general" (g) and three "non-general" processes (j, k, l). Each process maintains a decision d; for the general, the decision can be either 0 or 1, and for the non-general processes, the decision can be 0, 1 or $\perp$. The value $\perp$ denotes that the corresponding process has not yet received the value from the general. Each non-general process also maintains a boolean variable f that denotes whether that process has finalized its decision. Also, at most one process (from g, j, k and l) may be byzantine.

To represent a byzantine process, we introduce a variable b for each process; if $b.j$ is true then j is byzantine. A byzantine process can change its d and f values arbitrarily in order to confuse other processes. A non-general process can read the d values of other processes and update its d and f values. Thus, the state space for the byzantine agreement problem consists of the following variables.

- $d.g : \{0, 1\}$
- $d.j, d.k, d.l : \{0, 1, \perp\}$
- $b.g, b.j, b.k, b.l : \{true, false\}$
- $f.j, f.k, f.l : \{0, 1\}$

The set of variables j is allowed to read, r_j, is $\{b.j, d.j, f.j, d.k, d.l, d.g\}$, i.e., j can read the d values of other processes and all its variables. The set of variables that j is allowed to write, w_j, is $\{d.j, f.j\}$. However, process j is not allowed to write $d.j$ and $f.j$ if $b.j$ is true. (Of course, as we will see in Section 5.4, if $b.j$ is true then fault transitions can change $d.j$ and $f.j$.)

5.2 Safety Specification of Byzantine Agreement

The safety specification requires that validity and agreement be satisfied. Validity requires that if the general is non-byzantine then the final decision of a non-byzantine process must be the same as that of the general. And, the agreement requires that the final decision of two non-byzantine processes cannot be different. In other words, the safety specification requires that the program should not reach a state where the following predicate is true:

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

A transition violates safety if it reaches a state where S_{sf} is true. Also, to capture the notion that once a process finalizes its decision it cannot change that decision, we say that a transition that changes the decision of a process after it has finalized also violates the safety. Thus, the transitions that violate safety are as follows:

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

Also, our model of byzantine agreement prevents a byzantine process to change its d and f values. Note that this restriction is imposed only on the process, and not on faults. As stated earlier, each transition will be marked with the process (or fault) that is responsible for executing it. Thus, a transition of the form $\{(s_0, s_1) : b.j(s_0) \wedge s_0 \neq s_1\}$ should not be included in the transitions of process j. If we included such transitions in the transitions of process j, then it would imply that even if j is byzantine, it is required to execute these transitions, and requiring a byzantine process to execute some transition is contrary to the traditional understanding of a byzantine process. Of course, as we will see next, if j is byzantine, a fault can execute a transition where $d.j$ and/or $f.j$ is changed.

5.3 Fault-intolerant Byzantine Agreement

If no processes were byzantine, an algorithm that copies the value from the general and then finalizes that value will be sufficient to satisfy the specification of byzantine agreement. Thus, the fault-intolerant program, IB consists of the following two actions for process j (Likewise, actions for k and l are also included.)

$$\begin{aligned} d.j = \perp \ \wedge \ f.j = 0 \quad & \longrightarrow \quad d.j := d.g \\ d.j \neq \perp \ \wedge \ f.j = 0 \quad & \longrightarrow \quad f.j := 1 \end{aligned}$$

Grouping of transitions in IB. Note that each action in the above program consists of several groups: For example, the first action consists of 18 groups (2 values of $d.g * 3$ values of $d.k * 3$ values of $d.l$). Moreover, each group consists of 32 transitions (2 values of $b.g * 2$ values of $b.k * 2$ values of $b.l * 2$ values of $f.k * 2$ values of $f.l$). We also implicitly assume that if s_0 is a deadlocked state in program p, i.e., if there are no transitions of p that originate in state s_0, then (s_0, s_0) is included in p. Section 4.1 identifies restrictions on when we can include transitions of the form (s_0, s_0) while adding fault-tolerance.

Invariant of IB. The invariant of IB, S_{IB}, captures the set of states from where execution of IB satisfies its specification. It follows that IB is likely to have multiple invariants. For example, one possible invariant is the set of states reached from states where all b values are $false$, $d.j, d.k$ and $d.l$ are equal to $\perp$, and $f.j, f.k$ and $f.l$ are equal to 0. As mentioned in Section 3, it is desired that the invariant specified should be the largest possible invariant. In other words, if program IB satisfies its specification from some state where a process is byzantine, we should include that state in S_{IB}. With this insight, we proceed as follows:

First, we consider the set of states where the general is non-byzantine. In this case, one of the non-general processes may be byzantine. However, if a non-general process, say j, is non-byzantine, it is necessary that $d.j$ be initialized to either $\perp$ or $d.g$. Also, a non-byzantine process cannot finalize its decision if its decision equals $\perp$. Thus, the states in S_1 should be included in the invariant, where

$$\begin{aligned} S_1 = \quad & \neg b.g \wedge (\neg b.j \vee \neg b.k) \wedge (\neg b.k \vee \neg b.l) \wedge (\neg b.l \vee \neg b.j) \\ & \wedge (\forall p :: \neg b.p \Rightarrow (d.p = \perp \vee d.p = d.g)) \\ & \wedge (\forall p :: (\neg b.p \wedge f.p) \Rightarrow (d.p \neq \perp)) \end{aligned}$$ [2]

Now, we consider the set of states where the general is byzantine. In this case, g can change $d.g$ value arbitrarily. It follows that if other processes are non-byzantine and $d.j, d.k$ and $d.l$ are initialized to the same value that is different from $\perp$, IB refines its specification. Thus, the states in S_2 should be included in the invariant, where

$$S_2 = b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l \wedge (d.j = d.k = d.l \quad \wedge d.j \neq \perp)$$

From the above discussion, we let the invariant $S_{IB} = S_1 \vee S_2$.

5.4 Faults for Byzantine Agreement

A fault-transition can cause a process to become byzantine if no process is initially byzantine. Also, a fault can 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.)

$$\begin{array}{lcl} \neg b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l & \longrightarrow & b.j := true \\ b.j & \longrightarrow & d.j, f.j := 0|1, 0|1 \end{array}$$ [3]

Remark. We have allowed j to read the d values of other processes. A byzantine process could change its d value just before j executes its transition. Thus, we capture the notion that a process could send different values to different processes. Although we could have maintained copies of the $d.k, d.g, d.l$ at j, we have chosen not to do so to keep the state space small.

Now, the problem of transformation requires us to identify a fault-tolerant program, FB, its invariant, S_{FB} such that the three conditions of the transformation problem are satisfied.

6 Synthesis Algorithm and its Application in Byzantine Agreement

In this section, we present our deterministic polynomial time algorithm for synthesis of fault-tolerant distributed programs. Our algorithm removes the non-determinism from Add_ft (cf. Section 4.2) based on the scenario discussed at the end of Section 4.1. Recall that in Section 4.1, we argued that while synthesizing distributed fault-tolerant programs, we are faced with the following choice: either (1) ensure that some state, say s_2, is not reached, or (2) some transition, say (s_0, s_1), is not included in the fault-tolerant program. To make the suitable choice, we develop the heuristics by considering whether the transition being excluded is in the fault-intolerant program, whether the state being excluded is in the invariant of the fault-intolerant program, whether recovery is possible from the state that is to be excluded, and so on. By default, we prefer to exclude a transition (and its corresponding group) over excluding a state. This is due to the fact that if we choose to exclude a state, we need to exclude all transitions that reach that state. We also consider the invariant states to be valuable, i.e., every attempt is made to ensure that a state in the invariant of the fault-intolerant program is not excluded. We also prefer groups that are in the fault-intolerant program as using those groups is likely to result in a fault-tolerant program that is derived from the given fault-intolerant program. Based on these preferences, we develop four heuristics, and use them in our synthesis algorithm.

Our algorithm consists of eight steps. The first two steps, as in Add_ft, compute ms and mt respectively. The remaining steps proceed to identify the fault-tolerant program, p', its invariant, S', and its fault-span, T', so that formulae $F1$-$F6$ are true. Specifically, all steps ensure that $F1$ is always true. The third step computes a new fault-span and ensures that the formula $F2$ is true. The fourth step then removes transitions and/or states from the fault-span computed in step 3 so that formula $F3$ becomes true. Since step 4 may violate $F2$, we redo steps 3 and 4. When no *progress* is made in the last repetition of steps 3-4, we continue to step 5 where we deal with states that are outside S' and where formula $F4$ is false. (Also, to prevent an infinite loop, we keep an upper bound on how often each iteration may continue.) After step 5, we repeat steps 3-5 to re-satisfy $F2$ and $F3$. Throughout steps 3-5, we do not modify the invariant. This requirement is based on the premise that states in the invariant are valuable and should not be removed as far as possible. When there is no progress from steps 3-5, we continue to step 6 where we recompute the invariant while ensuring that $F5$ stays true. In step 7, we deal with states in S' where $F4$ is false. Finally, in step 8, we deal with the situation where formula $F6$ is false. (The structure of our algorithm is as shown in Figure 2.)

For pedagogical reasons, we present our algorithm hand-in-hand with its application to the problem of byzantine agreement. Specifically, for each step of the algorithm, we provide the reasoning behind that step and how it applies to the byzantine agreement program. We also present the heuristic in the step where it is used for the first time. The eight steps of our algorithm are as follows:

Step 1: Identifying a set of states from where execution of faults alone can violate safety. Consider a transition, (s_0, s_1), which is a fault transition and violates safety. We must ensure that the program

[2] In the byzantine agreement problem, unless specified otherwise, all quantifications are on non-general processes

[3] $d.j := 0|1$ means that $d.j$ could be assigned either 0 or 1.

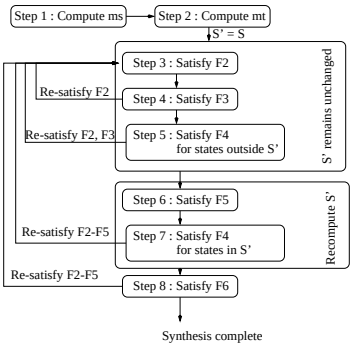

Figure 2. Deterministic Implementation of Add_ft

never reaches state s_0. Also, in this scenario, if (s_{-1}, s_0) is a fault transition then we must ensure that the program never reaches the state s_{-1}. Hence, we identify the set of states, ms, from where execution of one or more fault actions violates safety.

Reasoning behind step 1. Computation of ms is already included in Add_ft. The computation of this set will permit us to identify the set of states that should not be reached in any fault-tolerant program.

Application in the agreement problem. In the context of the current program, a fault transition violates safety iff it is in t_{sf}. Note that t_{sf} consists of two parts: the first part considers transitions that reach a state in S_{sf}. Observe that a fault transition reaches a state in S_{sf} only if it begins in a state in S_{sf}. The second part of t_{sf} considers transitions where a non-byzantine process changes its decision after finalizing it. Since faults affect only the byzantine process, a fault transition cannot fall in the second part. Thus, by going through the transitions once, we identify ms to be equal to S_{sf}.

Step 2: Identifying a set of transitions that should not be executed by the program. Note that a transition that violates safety cannot be executed in the fault-tolerant program. Moreover, if a transition reaches a state in ms (from where faults alone may violate the safety), then that transition should not be included either. Hence, we identify the set of transitions, mt, that should not be executed in the fault-tolerant program.

Reasoning behind step 2. Similar to the computation of ms, mt is already included in the non-deterministic algorithm presented in Section 4.2. The computation of this set will permit us to identify the set of transitions that should not be included in the program computation of a fault-tolerant program.

Application in the agreement problem. In the context of the current program, transitions in t_{sf} violate safety. Also, t_{sf} includes all transitions that reach a state in ms. Hence, by going through the transitions once, we identify mt to be equal to t_{sf}.

Heuristic 1 : A transition that starts in a state in ms may be used by the fault-tolerant program.

Reasoning behind heuristic 1. If (s_2, s_3) is a transition such that $s_2 \in ms$, then (s_2, s_3) may be included in the transitions of the fault-tolerant program. This heuristic is based on the premise that the synthesis algorithm will ensure that state s_2 will never be reached. This heuristic is useful when (s_2, s_3) is grouped with some other transition that is desirable in the fault-tolerant program. (Thus, in the scenario discussed at the start of the section, we can choose to include the group that contains (s_2, s_3) and ensure that state s_2 is not reached.)

Application in the agreement problem. In the current program, transitions that originate in S_{sf} may be used by the synthesis algorithm. Hence, we remove these transitions from mt.

Step 3: Identifying the fault-span of the fault-intolerant program. In this step, we determine the set of states reached by the computations –that start in a state in the invariant of the fault-intolerant program– of the fault-intolerant program in the presence of faults.

Reasoning behind step 3. As discussed earlier, while synthesizing distributed programs, we are faced with the following choice: either (1) ensure that some state, say s_2, is not reached, or (2) some transition, say (s_0, s_1), is not included in the fault-tolerant program. The set of states reached by the fault-intolerant program in the presence of faults will be used to determine which of the choices is made. Specifically, if state s_2 is not reached by the fault-intolerant program, we follow the first choice. Once again, as in heuristic 1, we define heuristic 2 as follows:

Heuristic 2 : If a transition (s_0, s_1) is in mt and s_0 is not reached in a computation – that starts in a state in the invariant of the fault-intolerant program – of the fault-intolerant program in the presence of faults, then (s_0, s_1) may be included in the fault-tolerant program.

Application in the agreement problem. We compute the set of states, say fs_{IB}, reached in the execution of IB and the faults from states in S_{IB}, where $S_{IB} = S_1 \vee S_2$ (cf. Section 5.3) .

In a state in S_1, if no process is byzantine then a fault transition can cause one process to become byzantine. And, if some process is byzantine then the fault can change the d and f values of the byzantine process. If the faults do not cause g to become byzantine then the set of states reached from S_1 is the same as S_1. And, 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_1 is $(S_1 \cup (b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l))$.

From a state in S_2, the d values of non-general processes will remain unchanged. It follows that the set of states reached from S_2 is S_2. Since $S_2 \subseteq (b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l)$, the set of states reached from S_{IB} is fs_{IB}, where

$$fs_{IB} = S_1 \cup (b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l).$$

Step 4. Identifying transitions in the fault-intolerant program that may be included in the fault-tolerant program. Beginning with the fault-tolerant program that consists of no transitions, we use the following heuristic to include the groups in the fault-intolerant program.

Heuristic 3 : A transition can be included in the fault-tolerant program if it is not in mt or if it is permitted by heuristics 1 and/or 2. A group can be included only if all its transitions can be included.

To ensure that $F1$ also remains true in step 4, if we add a transition that originates from the invariant, we ensure that the resulting state is also in the invariant. During the first iteration of step 4, this is straightforward as the invariant of the fault-intolerant program is closed in the fault-intolerant program. However, this check is done explicitly after the invariant is recomputed in steps 6-7.

Reasoning behind step 4 and heuristic 3. We use this heuristic to determine how long the fault-intolerant program can continue safely even if faults occur. By ensuring that states in ms and transitions in mt are removed, we ensure that $F3$ becomes true.

Application in the agreement problem. In the current program, the fault-intolerant program consists of 54 groups for each process (18 corresponding to the first action and 36 corresponding to the second action). Consider the case where j executes its first action in a state where $d.j = \bot, d.k = 0, d.l = 1, d.g = 0$. In the resulting state, $d.j$ is set to 0. The group corresponding to this action consists of 32 transitions. Since none of these transitions are in mt, we include this group.

Note that in choosing the above group, we needed to use heuristic 1. For example, in the above group, if $f.k$ and $f.l$ were equal to 1, the resulting transition would have violated safety. However, in this case, the initial state was in ms and, hence, we had removed this transition from mt. Likewise, all 18 groups for each process can be included in the fault-tolerant program.

Notation. We use the sequence $\langle x_1, x_2, x_3, x_4\rangle$ to denote the set of states where the value of $d.g$ equals x_1, the value of $d.j$ equals x_2, the value of $d.k$ equals x_3 and the value of $d.l$ equals x_4. We use '$*$' to denote that the decision value of a certain process is irrelevant. For example $\langle *, *, 0, 1\rangle$ denotes the set of states where $d.k$ is 0, $d.l$ is 1, $d.g$ is either 0 or 1, and $d.j$ is either 0, 1 or $\bot$.

While considering groups induced by the second action, consider a state, say s, in $\langle 0, 0, \bot, \bot\rangle$ where $f.j$ is 0. Process j can execute the second action from state s and set $f.j$ to 1. Note that irrespective of the values of other variables (e.g., b and f values of g, k and l), such a transition cannot be in mt. This follows from the fact that for a transition to be in mt, it must reach a state in S_{sf} or it must change the d or f value of a process that has finalized its decision. In the current setting, the final value of $d.k$ and $d.l$ is $\bot$ and hence, the resulting state cannot be in S_{sf}. Moreover, this action only modifies $f.j$, whose value was 0 in the initial state. Likewise, the 8 groups corresponding to the transitions from states in $\langle 0, 0, \bot, \bot\rangle$, $\langle 0, 0, \bot, 0\rangle$, $\langle 0, 0, 0, \bot\rangle$, $\langle 0, 0, 0, 0\rangle$, $\langle 1, 1, \bot, \bot\rangle$, $\langle 1, 1, \bot, 1\rangle$, $\langle 1, 1, 1, \bot\rangle$, $\langle 1, 1, 1, 1\rangle$ are included in the fault-tolerant program.

Now, consider a state, say s, in $\langle 0, 1, \bot, \bot\rangle$ where $f.j$ is 0. If j executes the second action from s and sets $f.j$ to 1, some transitions included in this group violate safety; if $b.g$ is false in s and if j finalizes its decision, its decision will be different from the general. Hence, safety will be violated. However, in this scenario, s will not be in fs_{IB}; in a state in fs_{IB}, $d.j$ must be equal to either $\bot$ or $d.g$ if g is non-byzantine. We leave it to the reader to verify that all transitions in this group that violate safety originate from states outside fs_{IB}. Thus, the 8 groups corresponding to the execution of the second action from states in $\langle 0, 1, \bot, \bot\rangle$, $\langle 0, 1, \bot, 1\rangle$, $\langle 0, 1, 1, \bot\rangle$, $\langle 0, 1, 1, 1\rangle$, $\langle 1, 0, \bot, \bot\rangle$, $\langle 1, 0, \bot, 0\rangle$, $\langle 1, 0, 0\bot\rangle$, and $\langle 1, 0, 0, 0\rangle$ are included in the fault-tolerant program.

Now, consider a state, say s, in $\langle 0, 0, 1, \bot\rangle$ where $f.j$ is 0. In state s, if $b.g$ is true, $b.k$ is false, $f.k$ is 1, $b.l$ is false and $f.l$ is 0 then the transition that sets $f.j$ to 1 is in mt. Moreover, in that case, s is in fs_{IB}, as fs_{IB} permits all possible values of d and f if $b.g$ is true. For this reason, we do not permit j to execute from a state in $\langle 0, 0, 1, \bot\rangle$. Likewise, the remaining 20 groups included in the second action are not included in the fault-tolerant program. Thus, after completing step 4, in the resulting program, say FB, the actions of a non-byzantine process j are as follows:

$$\begin{array}{lll} d.j = \bot \ \wedge \ f.j = 0 & \longrightarrow & d.j := d.g \\ d.j \neq \bot \ \wedge \ f.j = 0 \ \wedge & & \\ (d.k = \bot \ \vee d.k = d.j) \ \wedge & & \\ (d.l = \bot \ \vee d.l = d.j) & \longrightarrow & f.j := 1 \end{array}$$

Repeat steps 3 and 4. After completing step 4, we recompute the fault-span with the revised program to determine if any additional transitions of the fault-intolerant program may be included. This repetition can proceed until there are no more changes. If there are no changes in steps 3-4, F2 and F3 have become true. (Also, to prevent an infinite loop, we put an upper bound on the number of repetitions. After the upper bound is reached, we continue to step 5. In this case, we will attempt to satisfy $F2$, $F3$ when we repeat steps 3-5.)

Application in the agreement problem. Once again, we recompute the fault-span using the transitions of FB. Note that fs_{FB} is a subset of fs_{IB} as we have only removed transitions from IB. Specifically, we remove the following set of states where the decisions of two processes are different and both have finalized their decision. Formally, the new fault-span is

$fs_{FB} = fs_{IB} - (\forall p, q :: \neg b.p \wedge \neg b.q \wedge d.p \neq \bot \wedge d.q \neq \bot \wedge d.p \neq d.q \wedge f.p \wedge f.q)$

Even with this revised fault-span no new transitions can be added. Thus, the program after repetition is the same as that obtained in the first iteration of step 4.

Step 5 : Resolving deadlocks. After repeating steps 3 and 4, we identify the deadlock states in the resulting program. State s is deadlocked if there is no program transition that originates in state s. Note that such states cause violation of formula $F4$. If s is in the invariant of the fault-intolerant program and s is a deadlocked state in the fault-intolerant program, as mentioned in Section 4.1, we ignore the deadlock at s. We deal with remaining deadlocks using the following heuristic.

Heuristic 4. Given a deadlocked state s,

(**Step 5.1**) If it is possible to add a transition from s to a state in the invariant we attempt to add such a transition. Note that in distributed programs, we must add the group corresponding to that transition. We require that the added group satisfies the following two conditions: (1) no transition in that group is in mt (except as permitted by heuristics 1-3), and (2) if any transition in that group originates in the invariant of the fault-intolerant program, then it satisfies the second condition of the transformation problem, i.e., the transitions in that group that originate in the invariant of the fault-intolerant program are included in the fault-intolerant program. If such a group can be found, we add that group.

(**Step 5.2**) If such a group cannot be added, we consider whether s can be reached from the invariant with the execution of faults alone.

If yes (**Step 5.2.1**), we leave s as is.

If no (**Step 5.2.2**), we ensure that the fault-tolerant program does not reach s.

Reasoning behind step 5 and heuristic 4. The above heuristic is based on the principle that we would not like to eliminate any states and/or transitions unless absolutely required to do so. Hence, if we can recover from a state, we keep that state in the fault-span of the fault-tolerant program. If it is not possible to recover from s, and s can be reached by execution of faults alone from a state in the invariant, we allow s to be included temporarily. This is due to the fact that if we were to require that s is not reached, we would have to eliminate the corresponding state(s) from the invariant. We, however, consider states in the invariant of the fault-intolerant program to be valuable as the invariant of the fault-tolerant program is a subset of the invariant of the fault-intolerant program. And, if we prematurely eliminate the states in the invariant of the fault-intolerant program, it may prevent us from obtaining a fault-tolerant program.

However, if a deadlocked state, s, cannot be reached due to fault transitions alone, it implies that some program transition, say t, must be executed before s is reached. Hence, we could prevent the fault-tolerant program from reaching the deadlocked state by removing t. Hence, we attempt to eliminate s, i.e, we ensure that state s is never reached. Towards this end, we consider transitions of the form (s', s). If (s', s) is a fault transition, we ensure that state s' is never reached. This is due to the fact that if state s' is reached then state s can be reached by the execution of the fault. If (s', s) is the transition of the program obtained in step 4, we may choose to ensure that (1) (s', s) is not included in the fault-tolerant program or (2) state s' is never reached. Following the principle that states are more valuable than transitions, we remove the transition (s', s) from consideration in the fault-tolerant program. However, if removal of such transitions causes some state, say s_0, to be a deadlocked state, we follow the second approach. I.e., we include the transitions originating from state s_0 and attempt to ensure that state s_0 is never reached. Also, during this algorithm, if we encounter a state that can be reached from a state in the invariant by execution of faults alone, we do not pursue further elimination. Such states will be considered later in step 6. Thus, the algorithm to eliminate a state s from program transitions p and invariant S is as follows: (We let S to be the invariant of the fault-intolerant program and p to be the set of transitions obtained in step 4.)

```
eliminate(s : state, S : state predicate, p : transitions)
{   If s was considered earlier for elimination    return
    Remove transitions of the form (s', s) from p
    If there exists a fault transition (s', s)    eliminate(s', S, p)
    If all the transitions from some state, say s0, are removed
       Add the transitions of p that start from s0
       If s0 is not reachable from S by execution of faults alone
            eliminate(s0, S, p)   }
```

Note that the above program does not eliminate all deadlocked states as it does not remove states from the invariant.

Application in the agreement problem. In the current program, we consider the set of states where program FB deadlocks. Once again, we limit ourselves only to the fault-span of FB. Note that if the d value of a non-general process, say j, is $\perp$, then FB is not deadlocked as j can execute the first action. Hence, in a deadlocked state, the d values of non-general processes will be either 0 or 1.

Also, if the d values of the non-byzantine non-general processes are identical and if these processes have finalized, then in such states, the fault-intolerant program is also deadlocked. Moreover, such states are in the invariant of the fault-intolerant program. (We leave it to the reader to verify this claim.) Hence, such deadlocks (fixpoints) can be included in the fault-tolerant program. It follows that we only need to deal with states where d value of some process is 0 and the d value of some process is 1. Let's consider one such canonical state in $\langle *, 0, 0, 1\rangle$. Now, depending upon the values of $f.l$ and $b.l$, we consider the following scenarios:

$f.l$ is 0 and $b.l$ is false. For any such state in fs_{FB}, we find that l can change $d.l$ to 0 and reach a state in the invariant.

This fact is verified by considering all possibilities for the remaining variables, $d.g, b.g, b.j, b.l, f.j$ and $f.k$. We find that given any values for these variables, if the resulting state is in fs_{FB} then $b.g$ must be true in that state. (This follows from the fact that if $b.g$ is false then, from fs_{FB}, it is required that $d.g$ be equal to $d.l(=1)$. However, since $d.j$ and $d.k$ are 0, it requires that $b.j$ and $b.k$ be true. However, fs_{FB} does not permit such a state.) Moreover, the resulting state is in the invariant as the d values of all processes are identical. Hence, we add the groups corresponding to the following action to FB. (Note that we add similar groups for processes j and k as well.)

$$d.j = 1 \ \wedge \ d.k = 1 \ \wedge d.l = 0 \ \wedge \ f.l = 0 \ \longrightarrow \ d.l, f.l := 1, 0 \,|\, 1$$
$$d.j = 0 \ \wedge \ d.k = 0 \ \wedge d.l = 1 \ \wedge \ f.l = 0 \ \longrightarrow \ d.l, f.l := 0, 0 \,|\, 1$$

$b.l$ is true. Consider any state, say s, in $\langle *, 0, 0, 1\rangle$ where $b.l$ is true. If s is in fs_{FB}, $b.g$ must be false and $d.g$ must be equal to 0. It follows that any such state is in S_1 (and, hence, S_{IB}). Hence, we leave such states as is for now.

$f.l$ is 1 and $b.l$ is false. We find that any such state in fs_{IB} cannot be reached by starting from a state in S_{IB} and executing faults alone. Moreover, we also find that it is not possible to add a transition (of any process) from such state to a state in S_{IB}.

Based on the above heuristic, we ensure that any state in $\langle *, 0, 0, 1\rangle$, where $f.l$ is 1 and $b.l$ is false, is not reached. Towards this end, consider a state, say s, in $\langle 0, 0, 0, 1\rangle$ where $f.l$ is 1 and $b.l$ is false. Again, as in the first case, $b.g$ is true if s is in fs_{FB}. The state s can be reached from $\langle 1, 0, 0, 1\rangle$ if the fault changes $d.g$ to 0. Hence, we need to eliminate states in $\langle 1, 0, 0, 1\rangle$ where $b.l$ is false, $f.l$ is 1 and $b.g$ is true. Since this elimination is identical to that of s, we simply discuss the elimination of s here.

The only transitions in FB that reach s are the transitions where either j or k executes the first action. I.e., state s can be reached in program FB by states in $\langle 0, \perp, 0, 1\rangle$, $\langle 0, 0, \perp, 1\rangle$. Let s' be one such state. Clearly, if we prevent j (respectively k) to execute the first action in a state in $\langle 0, \perp, 0, 1\rangle$ (respectively $\langle 0, 0, \perp, 1\rangle$), no transition due to any process can be executed in that state. Hence, we must remove states in $\langle 0, \perp, 0, 1\rangle$, $\langle 0, 0, \perp, 1\rangle$ where $f.l$ is 1, $b.l$ is false, and $b.g$ is true. Likewise, we need to remove states in $\langle 0, \perp, \perp, 1\rangle$ where $f.l$ is 1 and $b.l$ is false, and $b.g$ is true. Also, as prescribed by the above algorithm, we add actions that let j (respectively k) execute the first action from states in $\langle 0, \perp, 0, 1\rangle$, $\langle 0, 0, \perp, 1\rangle$, and $\langle 0, \perp, \perp, 1\rangle$.

Eliminating a state, say s_1, in $\langle 0, \perp, \perp, 1\rangle$ where $b.l$ is false, $f.l$ is 1 and $b.g$ is true, however, requires the elimination of the state, say s_1', in the invariant where all processes are non-byzantine, $d.g$ and $d.l$ are equal to 1, $f.l$ is equal to 1, and $d.j$ and $d.k$ are equal to $\perp$. If the fault causes g to be byzantine in s_1', and it changes the value of $d.g$, state s_1 will be reached. Hence, we stop at s_1 and do not attempt further elimination.

Repeat Step 3-5. After completion of step 5, we repeat steps 3-5. Specifically, we have a revised program, say p_r (obtained from steps 4 and 5). We use p_r while repeating steps 3-5. In step 3, we use the transitions of p_r to identify the fault-span. However, while computing the fault-span, we do not explore states that were not eliminated in step 5.2.2. (If we explore these states, we will get the same deadlocked states which we were trying to eliminate in step 5.2.2) Then, in step 4, we consider transitions of p_r and identify if all its transitions can still be used. We also determine if transitions from the original fault-intolerant program can also be added; this may occur if the fault-span (re)computed in step 3 is different. Subsequently, we resolve the deadlocks as mentioned in step 5. While repeating step 5, additional recovery transitions could be added due to the revised fault-span. Also, in repeating step 5, the recovery transitions could also be added to states from where recovery was added in earlier steps. We continue this until a fixpoint is reached. (Alternatively, we could stop after certain iterations and continue to step 6.)

Application in the agreement problem. In the next repetition, recovery is possible from states in $\langle *, 0, 0, 1\rangle$ where $b.l$ is true. This is due to the fact that if a state in $\langle *, 0, 0, 1\rangle$ is in the revised fault-span, then either $b.l$ is true or $f.l$ is 0. In either case, we can add the groups of transitions where process j (respectively k) finalizes its decision. Thus, groups corresponding to the following action are added to the transitions of j when j is non-byzantine. (Likewise, groups are added to k and l).

$d.j \neq \bot \wedge f.j = 0 \wedge$
$((d.j = d.k \wedge d.j \neq d.l) \vee (d.j = d.l \wedge d.j \neq d.k)) \longrightarrow f.j := 1$

Step 6 : Removing states from the invariant. Steps 3-5 ensure that no state in the invariant is removed. More specifically, if s_0 is a state in the invariant, execution of faults alone from state s_0 can cause the program to reach state s_1, state s_1 is a deadlock state, and no recovery is possible from state s_1, step 5.2.1 simply quits. Likewise, step 5.2.2 also quits if it encounters a state whose elimination would require the elimination of a state in the invariant. For both these situations, we remove the offending states from the invariant in this step. Note that by removing states thus, the revised invariant of the fault-tolerant program will be a subset of the invariant of the fault-intolerant program.

Reasoning behind this step. Since repetition of steps 3-5 have reached a fixpoint, all deadlocked states fall in category 5.2.1 or 5.2.2. This suggests that there are some offending states in the invariant which should not be in the invariant of the fault-tolerant program.

Application in the agreement problem. Recall that in step 5, while eliminating states in $\langle 0, 0, 0, 1\rangle$ where $b.l$ is false, $f.l$ is 1 and $b.g$ is byzantine, we reached a state, say s, where $\langle 0, \bot, \bot, 1\rangle$ where $b.l$ is false, $f.l$ is 1 and $b.g$ is true. As shown in step 5, s could be reached by the execution of faults alone from a state in S_{IB}. For this reason, we did not eliminate state s in step 5. However, after step 5 has failed to make progress, in step 6, we eliminate state s, and the elimination of state s causes the elimination of states in $\langle *, \bot, \bot, 1\rangle$, where $f.l$ is 1 and no process is byzantine.

Step 7 : Recomputing the invariant. After step 6, we recompute the new invariant for the fault-tolerant program. In step 6, we may have eliminated some state(s) in the invariant. We use the following program to recompute the invariant.

ConstructInvariant(S : state predicate, p : transitions)
// Returns a subset of S such that computations of p
// within that subset are infinite
{ while $(\exists s_0 : s_0 \in S : (\forall s_1 : s_1 \in S : (s_0, s_1) \notin p))$ $S := S - \{s_0\}$ }

We instantiate the above function as follows: To identify S, we remove the set of states eliminated in step 6 from the invariant of the fault-intolerant program. And, we let p to be the set of transitions of the program obtained after step 6. ConstructInvariant removes the state s_0 from S if there is no transition of the form (s_0, s_1) such that (s_0, s_1) is a transition of p and s_1 is in S. Step 6 can produce such state s_0 if it eliminates the state s_1. (Once again, we ignore the case where the deadlocked state is included in the fault-intolerant program.)

After computing the invariant using ConstructInvariant, we recompute the program transitions to ensure that the revised invariant is closed in the program. Towards this end, if the program obtained in step 6 contains a transition of the form (s_0, s_1) where s_0 is in the revised invariant but s_1 is not in the revised invariant then we remove the transition (s_0, s_1) (and the group associated with it). With the revised program, there may be new deadlock states created within the invariant. Hence, we apply ConstructInvariant again with the revised program. We continue this until a revised invariant is closed in the program transitions or the revised invariant is the empty set. In the latter case, we declare that synthesis is not possible.

Reason behind this step. If the invariant contains a state s_0 such that there is no transition from p that starts in s_0 and p begins in state s_0, p will be deadlocked. We cannot add other transitions from state s_0 due to the second requirement of the transformation problem. Hence, we must ensure that p never reaches state s_0 in the absence of faults. Therefore, we remove s_0 from the invariant.

Application in the agreement program. After steps 6 and 7, the new invariant is S_{FB}, where

$$S_{FB} = S_{IB} - \{\exists p : f.p \wedge (\forall q : p \neq q : d.q = \bot)\}$$

Also, due to the revised invariant S_{FB} we need to remove the transition where process j finalizes its decision even if $d.k$ and $d.l$ are equal to $\bot$. Thus, the actions of FB for a non-byzantine process j are as follows: (This program is same as the canonical byzantine agreement program [2].)

$d.j = \bot \wedge f.j = 0$
$\longrightarrow$ $d.j := d.g$
$d.j \neq \bot \wedge f.j = 0 \wedge (d.k = \bot \vee d.k = d.j) \wedge$
$(d.l = \bot \vee d.l = d.j) \wedge (d.k \neq \bot \vee d.l \neq \bot)$
$\longrightarrow$ $f.j := 1$
$d.j = 1 \wedge d.k = 0 \wedge d.l = 0 \wedge f.j = 0$
$\longrightarrow$ $d.j, f.j := 0, 0 | 1$
$d.j = 0 \wedge d.k = 1 \wedge d.l = 1 \wedge f.j = 0$
$\longrightarrow$ $d.j, f.j := 1, 0 | 1$
$d.j \neq \bot \wedge f.j = 0 \wedge$
$((d.j = d.k \wedge d.j \neq d.l) \vee (d.j = d.l \wedge d.j \neq d.k))$
$\longrightarrow$ $f.j := 1$

Repeat steps 3-7. After completing step 7, we redo steps 3-7, i.e., with this reduced invariant, we compute the new fault-span. Then,

we decide which transitions of the fault-intolerant program may be used in step 4. While redoing step 4, we use a program transition of that originates in S_{FB} only if it also reaches a state in S_{FB} (This requirement was not there previously as if the transition of program IB originated in a state in S_{IB}, it was guaranteed that it will reach a state in S_{IB}.) The computation for steps 5-7 remains as is. In the next repetition of steps 3-7, the invariant and program transitions will remain the same and, hence, we will continue to step 8.

Step 8: Removing cycles. Let p' be the program obtained after repetitions of steps 3-7, let S' be its invariant, and let T' be its fault-span. In step 8, we consider cycles of the form $\langle s_0, s_1, \ldots s_0 \rangle$ where $s_0 \notin S'$ and $s_0 \in T'$. Clearly, we need to remove such cycles; otherwise the computation of p' can remain in these states forever. For this reason, we arbitrarily drop one transition (and the corresponding group) from this cycle.

Application in the agreement problem. In the current program, after step 7, there is no cycle in states outside S_{FB}. And, we can verify all six formulae in Add_ft. Hence, the program FB from step 7 is the required fault-tolerant program.

Repeat steps 3-8. If the program obtained after step 8 does not satisfy the formulae in Add_ft, we need to repeat steps 3-8 with the program obtained in step 8. If after some predetermined number of iterations, a fault-tolerant program is not found, our algorithm declares that no fault-tolerant program could be found. In the problem of byzantine agreement, this iteration is not needed as the fault-tolerant program is already found at the end of step 7.

7 Conclusion and Future Work

We presented heuristics for transforming a fault-intolerant distributed program into a fault-tolerant distributed program. We also presented a polynomial time algorithm that uses those heuristics. We showed how the heuristics suffice for the synthesis of a fault-tolerant byzantine agreement program. As mentioned in the Introduction, the canonical byzantine agreement program consists of 6912 states. Clearly, it would be impossible to use an exponential algorithm in this state space. However, our heuristics allowed polynomial time implementation by deterministically deciding which transitions and/or states should be included in the fault-tolerant program. The heuristics also permitted us to inspect only a small number of states as only those states reached in the presence of faults were inspected. The resulting program is the same as that in [2]. For reasons of space, we have discussed the application of our synthesis algorithm to other programs in [5].

Our algorithm differs from previous work on synthesis (e.g., [9–11]) in that the algorithms in [9–11] start with a specification (typically in some temporal logic). By contrast, we start with a fault-intolerant program that is known to be correct. For this reason, our algorithm only needs the safety specification that the program is supposed to satisfy in the presence of faults; the algorithm does not need the liveness specification. Further, ours is the first algorithm that has synthesized a solution to the byzantine agreement.

Our algorithm will be especially valuable in the case where the designer of a fault-tolerant program is aware of a corresponding fault-intolerant program that is known to be correct in the absence of faults. In this case, we expect that the designer will benefit from reusing that fault-intolerant program rather than starting from scratch. Moreover, such a situation occurs frequently, e.g., when the designer needs to incrementally adapt a program to deal with new types of faults. Further, the reuse of the fault-intolerant program will be virtually mandatory if the designer has only an incomplete specification.

The definition of fault-tolerance in this paper captures masking tolerance as defined in [8]. In [8], we have also defined two other types of fault-tolerance failsafe, where only the safety specification is satisfied in the presence of faults, and nonmasking, where the program eventually reaches a state from where the specification is satisfied. While the heuristics to add these types of fault-tolerance is outside the scope of this paper, we would like to point out that the heuristics 1, 2 and 3 (cf. Section 6) can be used for adding failsafe fault-tolerance and the heuristics 1, 2 and 4 (cf. Section 6) can be used for adding nonmasking fault-tolerance. One interesting future work in this direction is the derivation of the self-adjusting byzantine agreement algorithm by Zhao and Bastani [12].

The algorithm presented in Section 6 has been implemented in Java. It takes approximately 25 seconds to run on a SUN Ultra-5 workstation. This implementation will be used to develop a synthesis platform so that the user can specify a set of precomputed fault-tolerance components [8] which often occur in fault-tolerant programs, and use these components directly in the synthesis algorithm. We expect that these precomputed fault-tolerance components will not only help in reducing the complexity of synthesizing fault-tolerant programs but also permit the synthesized fault-tolerant program to be more efficient.

References

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

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

[3] M. Barborak, A. Dahbura, and M. Malek. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171–220, 1993.

[4] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. In *European Dependable Computing Conference*, pages 71–87, 1999.

[5] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of byantine agreement. Technical Report MSU-CSE-01-21, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, July 2001.

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

[7] 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.

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

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

[10] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6:68–93, 1984.

[11] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synchronize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[12] Y. Zhao and F. B. Bastani. A self-adjusting algorithm for Byzantine agreement. *Distributed Computing*, 5:219–226, 1992.