Polynomial Time Synthesis of Byzantine Agreement

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

In this paper, we present a polynomial time algorithm for automating the synthesis of a fault-tolerant distributed program from a fault-intolerant distributed program. Since the problem of synthesizing fault-tolerant distributed program is NP-hard, we present heuristics that allow us to reduce the complexity. We demonstrate our algorithm by automating the synthesis of the byzantine agreement program where there is one byzantine process. We also show how our algorithm can be used for the case where one process suffers from the byzantine fault and another suffers from the failstop fault. Finally, we also describe the tool that implements the heuristics.

Keywords : Fault-tolerance, Formal methods, Program Synthesis Program transformation, Concurrent programs, Low Atomicity

[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, an Ameritech Faculty Fellowship, a grant from Microsoft Research, and a grant from Michigan State University.

1 Introduction

In this paper, 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. Synthesizing a distributed fault-tolerant program is known to be a hard problem. In [1], we showed that this problem is NP-hard, and presented a non-deterministic algorithm for synthesizing fault-tolerant programs. It follows that a bruteforce deterministic implementation of that algorithm will be exponential in the state space of the fault-intolerant program.

An exponential algorithm for synthesis is limited to only those programs where the state space is small. One way to deal with this problem is to identify a set of heuristics to obtain a polynomial implementation of the synthesis algorithm. In this paper, we develop four such heuristics and present a polynomial algorithm for synthesizing distributed fault-tolerant programs using them. If the heuristics are applicable for the given synthesis problem, our algorithm will synthesize the fault-tolerant program in polynomial time. Otherwise, it will fail by declaring that a fault-tolerant program does not exist.

To develop these heuristics, we analyze the algorithm in [1] to identify the situations where non-deterministic choice is made in that algorithm. The heuristics are the designed so that in those situations, we can deterministically select one of those choices. It follows that if the heuristics select the suitable choice then the fault-tolerant program will be synthesized in polynomial time.

We demonstrate the applicability of our heuristics in synthesizing the byzantine agreement program [2]. The byzantine agreement problem has been recognized as an important but difficult problem in the literature (e.g., [3, 4]). However, due to the inherent difficulties in the agreement problem and the difficulties in characterizing byzantine faults during the synthesis algorithm, previous synthesis algorithms have not been able to automate the design of byzantine agreement. 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.

To understand the complexity involved in synthesizing byzantine agreement, consider the state space used in the canonical byzantine agreement, where the program consists of a general process and three non-general processes. For this program, we need the following variables: For the general, we need 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, we need 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 use an algorithm that is exponential in the state space to synthesize the fault-tolerant byzantine agreement program. However, the polynomial algorithm designed using our heuristics solves this problem efficiently.

To analyze the efficiency of our heuristics, we also describe a prototype tool that implements them. The tool is implemented in Java, and is part of the synthesis platform we are building to test different heuristics and allow interaction with users.

Contributions of the paper. The main contributions of the paper are as follows: We first identify the heuristics to obtain a polynomial algorithm for synthesizing fault-tolerant distributed programs. We demonstrate our synthesis algorithm by designing the canonical version of byzantine agreement program. Subsequently, we show how those heuristics can be used to synthesize byzantine agreement programs for the case where the number of non-general processes is increased. Specifically, we point out the synthesis of the algorithm where there are four non-general processes, at most one process is byzantine and at most one process suffers from failstop fault. Finally, we also describe our tool that is used to automate the synthesis of byzantine agreement.

Organization of the paper. This paper is organized as follows: We provide the definitions of programs, specifications, faults and fault-tolerance in Section 2. Using these definitions, we state the transformation

problem in Section 3. In Section 4, we 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 7, we present the heuristics that will be used to add fault-tolerance to the program in Section 6. In Section 8, we present high-level details of the tool that we are building to test these heuristics. Finally, we discuss other problems where these heuristics are useful in Section 9 and make concluding remarks in Section 10.

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

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 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 the 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 [5], 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 $\{(s_0, s_1) : s_0, s_1 \in S_p\}$. The liveness specification is not specified in our transformation algorithm; we show that the fault-tolerant program satisfies the liveness specification iff the fault-intolerant program satisfies the liveness specification. Moreover, in the transformation problem, the initial fault-intolerant program satisfies its specification (including the liveness specification). Thus, 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. We also refer the reader to [8] where we show that it is possible to convert a set of state sequences that is not suffix closed

and/or fusion closed into an equivalent set that is suffix closed and fusion closed. Also, Felix Gartner [9], has recently shown how specifications that are not suffix closed and/or fusion closed can be used directly during synthesis.)

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*. Similarly, we say that α violates *spec* iff it is not the case that α 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*.

2.3 Faults

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

A fault for $p(= \langle S_p, \delta_p \rangle)$ is a subset of $\{(s_0, s_1) : s_0, s_1 \in S_p\}$. We use $p[]f$ 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 $p[]f$. Thus, at each state where an invariant S of p is true, an f-span T of p from S is also true. Also, T, like S, is closed in p. Moreover, if any transition in f is executed in a state where T is true, the resulting state is also one where T is true. It follows that for all computations of p that start at states where S is true, T is a boundary in the state space of p up to which (but not beyond which) the state of p may be perturbed by the occurrence of the 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) $\exists n : n \geq 0 : (\forall j : j > n : (s_{j-1}, s_j) \in \delta_p)$. 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 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.

Note that the above definition captures masking tolerance as defined in [8]. Since we present heuristics for adding masking fault-tolerance, we use the term fault-tolerance to mean masking fault-tolerance. 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 7) can be used for adding failsafe fault-tolerance and the heuristics 1, 2 and 4 (cf. Section 7) can be used for adding nonmasking fault-tolerance.

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

Using the definitions from the previous section, we are now ready to formally define what it means for a fault-tolerant program p' to be derived from a fault-intolerant program, p. Our definition of derivation is based on the premise that p' is obtained by adding fault-tolerance alone to p, i.e., p' does not introduce new ways of refining *spec* when no faults have occurred. More specifically, 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'.

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

Remark. Our notion of derivation suggests that the fault-intolerant program we start with should be maximal, i.e., we should consider a program which has the weakest invariant and maximal non-determinism. Also, the above transformation problem requires that the state space of the fault-tolerant program is the same as that of the fault-intolerant program. In other words, it requires that the fault-intolerant program should be such that it contains all the variables needed for the fault-tolerant program. 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 which are the legitimate values for those variables. And, the answers to these questions are fault-dependent. We refer the reader to [1, 9] for details on how new variables can be introduced while adding fault-tolerance.

4 Low Atomicity Model

In the low atomicity model, a program consists of a set of variables and a set of processes. For each process, the model specifies the variables it can read and write. In this section, we show how we incorporate the constraints imposed by the low atomicity model in adding fault-tolerance. Specifically, given a set of restrictions in the ability of a process to read and write variables, we describe how to determine whether a transition (s_0, s_1) can be used while synthesizing the transitions of that process. In Section 4.1, we present the main difficulties involved in using the low atomicity model while adding fault-tolerance.

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, being able to write only a subset of variables, w_j, is equivalent to not being able to use the set of transitions, $write(j, w_j)$, in the synthesis algorithm, 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, we simply generalize the safety specification. Specifically, with 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., 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 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 where $w_j \subseteq r_j$, 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. Thus, if a transition in some group violates the restrictions imposed by the inability to write, then that entire group must be excluded in the design of fault-tolerant program. It follows that after combining the read restrictions and the write-restrictions, we get another grouping of transitions; we need to choose zero or more such groups to obtain the transitions of that process.

Note that while describing the read/write atomicity, 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 (with fault-intolerant program p, its invariant S, the fault-tolerant program p' and its invariant S'), 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 $p' [] f$.

4.1 Difficulties Associated with Low Atomicity

We now point out why it is difficult to solve the transformation problem in low atomicity. This discussion also points how this problem can be solved using a non-deterministic algorithm, and it leads to the types of heuristics that may be used to reduce the complexity.

Consider a scenario where two transitions (s_0, s_1) and (s_2, s_3) are grouped together due to read restrictions on some process. In this scenario, suppose that (s_0, s_1) is desirable (e.g., because it is a useful recovery transition or it is used to satisfy the specification in the absence of faults). Also, suppose that the transition

(s_2, s_3) should never be executed (e.g., because it causes the safety specification to be violated). In this scenario, the process 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).

The above scenario suggests that the process must tradeoff between the states the fault-tolerant program may reach and the transitions that may be used by the fault-tolerant program. Moreover, the choices made by different processes are interrelated. For example, in the above scenario, if the process chooses the first choice, then other processes also must guarantee that s_2 is never reached. Note that for each such choice that processes must make, if we were able to obtain the *correct* answer, we would be able to design the fault-tolerant program in polynomial time. However, if we need to do a bruteforce search through all choices, the resulting algorithm will be exponential. We used this crucial fact to show in [1] that the transformation problem is NP-hard.

To make the suitable choice in the above scenario, we present heuristics in Section 7. Also, we use the following non-deterministic algorithm (from [1]) that solves the transformation problem to verify that the program obtained by using those heuristics is indeed fault-tolerant.

Add_ft(p, f : transitions, S : state predicate, $spec$: specification,
$g_0, g_1, ..., g_{max}$: groups of transitions)
{
$ms := \{s_0 : \exists s_1, s_2, ...s_n :$
$(\forall j : 0 \leq j < n : (s_j, s_{(j+1)}) \in f) \quad \wedge \quad (s_{(n-1)}, s_n)$ 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 included in the fault-tolerant program);
Verify the following
(1) $S' \neq \{\}$; $S' \subseteq S$; S' is closed in p';
(2) $S' \Rightarrow T'$; T' is closed in $p' [] f$;
(3) $p'|S' \subseteq p|S'$;
(4) $T' \cap ms \ = \ \{\}$; $(p'|T') \cap mt \ = \ \{\}$;
(5) $(\forall s_0 : s_0 \in T' : (\exists s_1 :: (s_0, s_1) \in p'))$; $p'|(T' - S')$ is acyclic
}

The above 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 program 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 p', the fault-tolerant program, S', its invariant, and T', its fault-span. Finally, by using the values of ms and mt, it verifies that the three conditions of the transformation problem are satisfied.

The first verification condition checks that S' is a valid invariant, i.e., S' is nonempty and S' is closed in p'. It also checks that the first condition ($S' \subseteq S$) of the transformation problem is satisfied. The second condition checks that T' is valid fault-span. The third condition checks the second condition of the transformation problem ($p|S' \subseteq p'|S'$). The fourth condition checks that safety is not violated from any state in T'. And, the fifth condition checks that the program does not deadlock in a state in T' and that it cannot stay in T' forever.

5 Representation of Byzantine Faults

We consider the canonical version of the byzantine agreement problem: 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 value 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 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 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, if $b.j$ is true then fault transitions can change $d.j$ and $f.j$.)

As discussed in Section 4, the inability to write prevents j from using a transition that affects variables other than $d.j$ and $f.j$. Also, if $b.j$ is true then j cannot include transitions where $d.j$ and/or $f.j$ are changed. The inability to read forces j to choose a group of transitions instead of choosing individual transitions.

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.)

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

Remark. We have allowed j to read the d values of other processes. Of course, if one of them is a byzantine process, it 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.

6 Fault-intolerant Byzantine Agreement

In this section, we identify the fault-intolerant program for byzantine agreement, identify its invariant, and discuss how the transitions in it are grouped together.

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

$$\begin{array}{lll} d.j = \perp \ \wedge \ f.j = false & \longrightarrow & d.j := d.g \\ d.j \neq \perp \ \wedge \ f.j = false & \longrightarrow & f.j := true \end{array}$$

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 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 some initial state, 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. For this case, if a non-general process, say j, is non-byzantine, it is necessary that $d.j$ be initialized to either $\perp$ or $d.g$. It is also necessary that at most one non-general process is byzantine. Thus, the states in S_1 should be included in the invariant, where

$$S_1 = \quad \neg b.g \wedge (\neg b.j \vee \neg b.k) \wedge (\neg b.k \vee \neg b.l) \wedge (\neg b.l \vee \neg b.j) \wedge (\forall j :: \neg b.j \Rightarrow (d.j = \perp \vee d.j = d.g))$$

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 satisfies its specification. Thus, the states in S_2 should be included in the invariant, where

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

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

Safety specification. 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:

$$\begin{array}{lll} S_{sf} = & & \exists j, k :: \neg b.j \wedge \neg b.k \wedge d.j \neq \perp \wedge d.k \neq \perp \wedge d.j \neq d.k \wedge f.j \wedge f.k \\ & \vee & \exists j :: \neg b.g \wedge \neg b.j \wedge d.j \neq \perp \wedge d.j \neq d.g \wedge f.j \end{array}$$

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:

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

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, if j is byzantine, a fault can execute a transition where $d.j$ and/or $f.j$ is changed.

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.

7 Heuristics for Deriving Masking Fault-Tolerant Program

In Section 4.1, we argued that in the low atomicity model, 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-tolerant 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) than 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 to present the derivation of the fault-tolerant byzantine agreement program from program IB in Section 6. We have divided the derivation in 8 steps. For each step, we provide the reasoning behind that step and how it applies for the byzantine agreement program. Also, we present each of the four heuristics in a step where it is used for the first time.

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 (s_0, s_1) violates safety, we must ensure that the program 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 violate safety.

Reasoning behind step 1. Computation of ms is already included in the non-deterministic algorithm presented in Section 4.1. 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 current program. In the context of the current program, a fault transition violates the 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 it has finalized 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 4.1. 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 current program. In the context of the current program, transitions in t_{sf} violates 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 procedure 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 current program. In the current program, transitions that originate in S_{sf} may be used by the synthesis algorithm. Hence, we remove these transitions from mt. Hence, the new value of mt is equal to $\{(s_0, s_1) : \neg b.j(s_0) \wedge \neg b.j(s_1) \wedge f.j(s_0) = 1 \wedge (d.j(s_0) \neq d.j(s_1) \vee f.j(s_0) \neq f_j(s_1))\}$. (Note that this set is the same as the second part in t_{sf}.)

Step 3: Identifying the fault-span of the fault-intolerant program. Recall that the fault-span refers to the set of states that may be reached in the execution of the program actions and the fault actions. In this step, we compute 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, in the low atomicity model, 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 followed. Specifically, if state s_2 is not reached by the fault-intolerant program, we follow the first choice. Once again, using heuristic 1, we now define heuristic 2, where

> **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 current program. We compute the fault-span fs_{IB} by computing the states reached in the execution of IB and the fault actions from states in S_{IB}. Note that $S_{IB} = S_1 \vee S_2$.

Observe that starting from 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. Now, consider the set of states that may be reached in the presence of these faults: 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. (This is due to the fact that when g becomes byzantine, from fault action 1, all non-general processes are non-byzantine. And, they cannot become byzantine later as the fault action is disabled.) Thus, the set of states reached from S_1 is $(S_1 \cup (b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l))$.

Starting from S_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_2 is S_2. Finally, since S_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_{IB} is fs_{IB}, where

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

Step 4. Identifying transitions in the fault-intolerant program that may be included in the fault-tolerant program. We begin with the fault-tolerant program that consists of no transitions. We then consider the groups included in the fault-intolerant program and decide which of those groups can be included in the fault-tolerant program. Clearly, we include a group only if all transitions in that group can be included. We use the following heuristic for this purpose.

> **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.

Reasoning behind step 4 and heuristic 3. Since the goal is to reuse the transitions of the fault-intolerant program, we use this heuristic to determine how long the fault-intolerant program can continue safely even if faults occur. By ensuring that transitions in mt are not included, we ensure that safety is never violated.

Application in the current program. 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). Again, each of these groups consists of several transitions, and we are required to include all the transitions in a group or none at all.

Now 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 transition, 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, we consider all 18 groups for each process, and find that each group 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, $d.g$ could be 0 or 1, and $d.k$ could be 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 false. Process j can execute the second action from state s and set $f.j$ to true. 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 false in the initial state. Likewise, j can execute the second action from $\langle 0, 0, \bot, 0\rangle$, $\langle 0, 0, 0, \bot\rangle$, $\langle 0, 0, 0, 0\rangle$, $\langle 1, 1, \bot, \bot\rangle$, $\langle 1, 1, \bot, 1\rangle$, $\langle 1, 1, 1, \bot\rangle$, $\langle 1, 1, 1, 1\rangle$. Thus, the eight groups corresponding to these states of j are included in the fault-tolerant program. (Likewise, we include the groups corresponding to k and l.)

Now, consider a state, say s, in $\langle 0, 1, \bot, \bot\rangle$ where $f.j$ is false. If j executes the second action from s and sets $f.j$ to true, some transitions included in this group violate safety. Specifically, if $b.g$ is false in state s and if j finalizes its decision, its decision will be different from the general and, hence, safety will be violated. However, in this scenario, s is not 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}. For this reason, j can execute the second action from states in $\langle 0, 1, \bot, \bot\rangle$. Likewise, j can execute the second action from $\langle 0, 1, \bot, 1\rangle$, $\langle 0, 1, 1, \bot\rangle$, $\langle 0, 1, 1, 1\rangle$, $\langle 1, 0, \bot, \bot\rangle$, $\langle 1, 0, \bot, 0\rangle$, $\langle 1, 0, 0\bot\rangle$, and $\langle 1, 0, 0, 0\rangle$. Thus, the eight groups corresponding to these actions of j are included in the fault-tolerant program.

Now, consider a state, say s, in $\langle 0, 0, 1, \bot\rangle$ where $f.j$ is false. Consider the group of transitions obtained by the execution of the second action from s. Some of the transitions in this group violate safety. Specifically, 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 true is in mt. Moreover, 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, j cannot execute the second action from states in $\langle 0, 0, 1, \bot\rangle$, $\langle 0, 0, 1, 0\rangle$, $\langle 0, 0, 1, 1\rangle$, $\langle 0, 0, \bot, 1\rangle$, $\langle 0, 0, 0, 1\rangle$, $\langle 1, 0, 1, \bot\rangle$, $\langle 1, 0, 1, 0\rangle$, $\langle 1, 0, 1, 1\rangle$, $\langle 1, 0, \bot, 1\rangle$, $\langle 1, 0, 0, 1\rangle$, $\langle 0, 1, 0, \bot\rangle$, $\langle 0, 1, 0, 1\rangle$, $\langle 0, 1, 0, 0\rangle$, $\langle 0, 1, \bot, 0\rangle$, $\langle 0, 1, 1, 0\rangle$, $\langle 1, 1, 0, \bot\rangle$, $\langle 1, 1, 0, 1\rangle$, $\langle 1, 1, 0, 0\rangle$, $\langle 1, 1, \bot, 0\rangle$, $\langle 1, 1, 1, 0\rangle$. Thus, the twenty groups corresponding to these transitions cannot be included in the fault-tolerant program.

After completing step 4, all 18 groups of the first action, and 16 (out of 36) of the second action are included in the fault-tolerant program. The actions for process j in the resulting program, say FB, are as follows:

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

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. Or, we can put an upper bound on the number of repetitions.

Application in the current program. 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 decision of two processes is different and both have finalized their decision. Formally, the new fault-span is

$$fs_{FB} = fs_{IB} - (\forall j, k :: \neg b.j \wedge \neg b.k \wedge d.j \neq \bot \wedge d.k \neq \bot \wedge d.j \neq d.k \;\; \Rightarrow \;\; (\neg(f.j \wedge f.k)))$$

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. 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 4, 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 the low atomicity model, 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.
- If such a group cannot be added, we consider whether that s can be reached from the invariant with the execution of faults alone.
  - If yes (**Step 5.2**), we leave s as is.
  - If no (**Step 5.3**), 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 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 must 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 s_0, are removed
        Add the transitions of p that start from s_0
        If s_0 is not reachable from S by execution of faults alone
            eliminate(s_0, 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 current program. 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-general processes are identical, 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.)

  $$\begin{array}{lll} d.j = 1 \ \wedge \ d.k = 1 \ \wedge d.l = 0 \ \wedge \ f.l = false & \longrightarrow & d.l = 1 \\ d.j = 0 \ \wedge \ d.k = 0 \ \wedge d.l = 1 \ \wedge \ f.l = false & \longrightarrow & d.l = 0 \end{array}$$

- *$b.l$ is true.* We find that any such state in fs_{FB} can be reached from a state in S_{FB} by execution of faults alone.

  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. Now consider a state s' where the value of $d.l$ is 0 and the values of the remaining variables are the same as s. Clearly, s' is in S_1, and s can be reached from s' if the byzantine fault changes $d.l$ to 1. 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_{FB} cannot be reached by starting from a state in S_{FB} and executing faults alone. Moreover, we also find that it is not possible for any transition (of any process) from such state cannot reach a state in S_{FB}.

  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 $\langle 0, \bot, 0, 1\rangle$, $\langle 0, 0, \bot, 1\rangle$. Let s' be one such state. Clearly, if we prevent j (respectively k) to execute the first action in a state in $\langle 0, \bot, 0, 1\rangle$ (respectively $\langle 0, 0, \bot, 1\rangle$), no transition of any process can be executed in that state. Hence, we must remove states in $\langle 0, \bot, 0, 1\rangle$, $\langle 0, 0, \bot, 1\rangle$ where $f.l$ is 1, $b.l$ false, and $b.g$ is true. Likewise, we need to remove states in $\langle 0, \bot, \bot, 1\rangle$ where $f.l$ is 1 and $b.l$ is false, and $b.g$ is true (cf. Figure ??). Also, as prescribed by the above algorithm, we add actions that let j (respectively k) execute the first action from states in $\langle 0, \bot, 0, 1\rangle$, $\langle 0, 0, \bot, 1\rangle$, and $\langle 0, \bot, \bot, 1\rangle$.

  Eliminating a state, say s_1, in $\langle 0, \bot, \bot, 1\rangle$ 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 $\bot$. 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 5.3. 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. We continue this until a fixpoint is reached. (Alternatively, we could stop after certain iterations and continue to step 6.)

Application in the current program. While we omit the details, we would like to point out that 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.

Step 6 : Removing states from the invariant. Steps 3-5 attempt to 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 simply quits. Likewise, step 5.3 also quits if it encounters a state whose elimination would require the elimination of a state in the invariant. In both these situations, in this step, we remove the offending states from the invariant.

Reasoning behind this step. Since repetition of steps 3-5 have reached a fixpoint, all deadlocked states fall in category 5.2 or 5.3. 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 current program. Recall that in step 5, while eliminating states in $\langle 0, 0, 0, 1\rangle$ where $b.l$ was false, $f.l$ was 1 and $b.g$ was byzantine, we reached a state, say s, where $\langle 0, \bot, \bot, 1\rangle$ where $b.l$ was false, $f.l$ was 1 and $b.g$ was 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 s'. Specifically, we eliminate all states in $\langle *, \bot, \bot, 1\rangle$ where $f.l$ is equal to 1 and $b.l$ is false.

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 the 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 5 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.)

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 to the current program. In the current program, all the states removed in step 5 were outside S_{IB}. Hence, no further states need to be removed in this step.

Step 8: Removing cycles.

8 Implementation

In this section, we describe the high-level details of the prototype synthesis tool that solves the transformation problem. The prototype is implemented in Java, and its main goals are:

- Demonstrate the feasibility of the transformation
- Test the different heuristics for solving the transformation problem.
- To determine the data structures that would reduce the time complexity of solving the transformation problem by identifying the various tasks involved and their frequencies.

8.1 Input to the tool

The inputs that the user needs to provide to the tool are:

- Variables and their possible values : It can be observed that all possible combinations of the assignment of values to the variables determine the state space in which the solution for the problem is being attempted.
- The fault-intolerant program : The fault-intolerant program is specified in a text file that contains the various actions for the different processes.
- Faults : Faults are specified as actions (similar to program actions) in another text file.
- Invariant for the fault-intolerant program : The invariant is specified as a predicate involving the various variables.

- Read-write restrictions : The read-write restrictions for each process describe what variables that process can read or write.
- Safety specification : Specifying the set of transitions that violate safety indicates the safety specification.

8.2 Object Design of the tool

We use a graph-based approach to implement the algorithm. The design of the tool is based on object-oriented methodology. The primary idea in our graph-based approach can be summed up using the following description. We construct the state-transition graph for the fault-intolerant program by starting from its invariant and using the program transitions of the fault-intolerant program and the fault-transitions. Then we modify the graph by adding or removing transitions, in terms of the low atomicity model, as outlined by the algorithm. The modification of the state-transition graph results in the transformation of the fault-intolerant program into a fault-tolerant program.

The design of the tool involves the following major objects : Graph, Program, Fault and Safety Specification. We describe these objects next.

Design of the Graph object. The Graph object represents the state-transition graph. A state refers to a particular assignment of values to the different variables and is represented by the State object. A transition can be a program-transition obtained by applying a program-action to a state or a fault-transition obtained by applying a fault-action to a state. The State object internally uses an array to store the values for the different variables. Initially, we construct the state-transition graph by performing a reachability analysis on all states that are reachable from the invariant states by applying the different program and fault actions. We observe that all states, reachable from the invariant states by applying the different program and fault actions, correspond to a fault-span of the fault-intolerant program. Every application of a program or a fault action to a state produces a transition that represents an edge in the state-transition graph. The different states that are reachable from the invariant represent the different nodes in this graph. Each State object also stores the following information

- A list of all outgoing program transitions from this state
- A list of all incoming program transitions to this state
- A list of all outgoing fault transitions from this state
- A list of all incoming fault transitions to this state
- A Boolean variable (flag) identifying whether this is an ms state or not.
- A Boolean variable (flag) identifying whether subsequent program transitions should be explored from this state.

A fault-transition just stores the information regarding what states it is connecting. A program-transition, on the other hand, also stores the additional information regarding which program statement produced this transition. This information is necessary, as we shall see shortly, in identifying other program transitions that are grouped with this transition.

Identifying ms-states. An ms-state is a state from where application of one or more fault actions violates safety. In other words, there exists a sequence of fault transitions from this state that includes a safety violating transition. We mark a state as an ms-state if it has a safety violating outgoing fault-transition. Then we also mark all states, which are reachable from this ms-state using incoming fault transitions, as ms-states.

Removing safety violating program transitions. For every non ms-state in the state-transition graph, we examine all the outgoing program transitions from this state. We remove an outgoing program-transition from the state-transition graph if it satisfies any one of the following two conditions

- The program-transition is an incoming program-transition for an ms-state.
- The program-transition directly violates safety.

When we remove a program-transition, we also remove all other program transitions that are grouped with this program-transition.

Design of the Program object. The Program is a set of processes. Each process has certain read-write restrictions and actions of the form
Guards $\longrightarrow$ Statement
The actions in each process are represented using an organization of Guard objects and Statement objects within the Program object. Each guard within an action is represented by a Guard object and the statement within an action is represented by a Statement object. A Guard object determines what should be the values of the various read-variables in a state, s_1, in order for the action to become applicable to s_1. When the Guard objects evaluate that the action can be applied to s_1, the Statement object produces the transition, (s_1, s_2), where s_2 is the resultant state.

In order to efficiently determine what actions within a process become applicable to a state we use a tree data structure to organize the actions. For example, the figure shows a part of the data structure used for the organization of actions, as specified in Section 6, for the subordinate (process) j in Byzantine agreement.

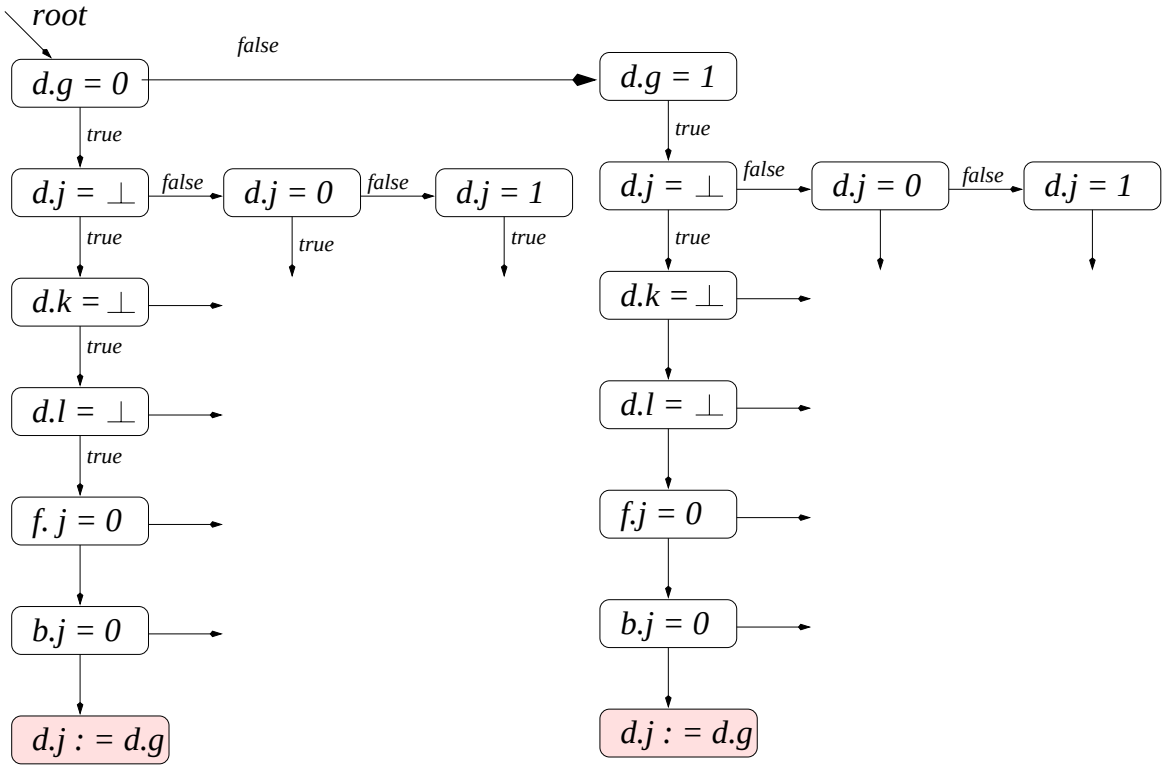

Figure 1: Process j in Program object for Byzantine agreement

The shaded boxes represent the Statement objects and the un-shaded boxes represent the Guard objects. Every possible action in the process corresponds to a particular traversal from the root node (which is generally a Guard object) to a Statement object. There can be more than one Statement object for the same set of Guards though this situation does not occur in the current example. When the process is required to apply actions to a state, say s, the state is simply routed along a particular path using the Guard objects to the Statement objects to produce the transitions for s. For example, let us consider a state, s, as

$d.g = 0 \wedge d.j = 0 \wedge d.k = \perp \wedge d.l = 1 \wedge f.j = 0 \wedge f.k = 0 \wedge f.l = 0 \wedge b.g = 1 \wedge b.j = 0 \wedge b.k = 0 \wedge b.l = 0.$

The state s first encounters the root, which in the above figure is the guard, $d.g = 0$. The condition, $d.g = 0$, is true in s. Hence, the guard, $d.g = 0$, routes s along its true edge. s now encounters the guard, $d.j = \perp$. However, the condition, $d.j = \perp$, is not true in s. Hence, the guard, $d.j = \perp$, routes s along its false edge. We continue in this way until s encounters a Statement object. Then the Statement object executes the action, producing a program-transition, (s, r), where r is the resultant state. The reason for choosing this data structure for the organization of actions within a Program object is as follows. We need to determine what actions within a process can be applied to a particular state. However, we require each action to be at the most fundamental level and correspond to the smallest possible group of transitions. Consequently, the number of actions within a process is very large. A possible organization for the actions is to store them in a list. In this case, we need to check sequentially, the applicability of each action against an input state. This is very less time efficient. Instead the above organization uses the values in an input state to directly determine what actions become applicable to that state. Thus it avoids examining all the actions within a process and is hence time-efficient.

Capturing Grouping of program transitions. The read-restrictions on a process introduce the concept of groups of program transitions. All the transitions produced by a particular Statement object form a particular group of transitions. This group of transitions is to be treated as a single unit. To capture the grouping concept, a Statement object stores all program transitions produced by it as a List within itself.

Design of the Fault object. Given the fault actions in the form,
Guards $\longrightarrow$ Statement
, the Fault object can be contructed corresponding to these actions. However, there is no concept of read restrictions for a fault. Thus, we need not consider grouping of fault-transitions. This observation allows us to use a very simple and efficient data structure for a Fault object. We use a linked-list of fault actions as the data structure for a Fault object. For example, the figure shows a part of the data structure for the Fault object in Byzantine agreement.

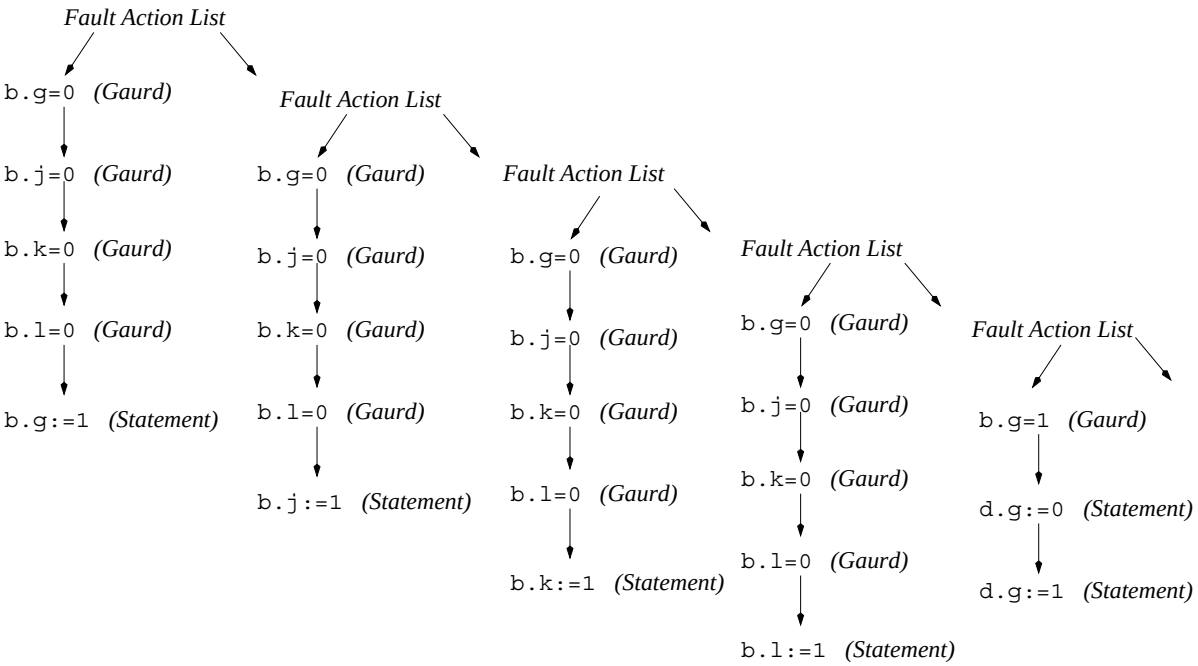

Figure 2: Fault object for Byzantine agreement

In the figure, a single Statement object and a sequence of Guard objects preceding the Statement object correspond to a single fault-action. When we wish to generate the fault transitions for a state, say s, we sequentially test every fault action whether it applies to s. If a fault action becomes applicable to s, it is executed to produce a fault transition, (s, r), where r is the resultant state.

Design of the Safety Specification object. Given the safety specification as a predicate in disjunctive normal form,
$d_1 \vee d_2 \vee d_3 \vee ... \vee d_i \vee .... \vee d_n$
where each d_i is in the form
$d_i = (c_1 \wedge c_2 \wedge c_3 \wedge ... \wedge c_j \wedge ..... \wedge c_m)$
, the Safety Specification object can be constructed corresponding to this predicate. The Safety Specification object is used to determine whether a transition violates safety.

For example, one of the disjuncts in the safety specification predicate for Byzantine agreement is:

$b.i(s_1) = 0 \wedge b.i(s_2) = 0 \wedge f.i(s_1) = 1 \wedge d.i(s_1) = 0 \wedge d.i(s_2) = 1$

where, s_1 is the source state and s_2 is the destination state in a transition, (s_1, s_2). A transition that satisfies the above disjunct violates safety.

Each conjunct, c_j, within a disjunct, d_i, in the above expression for safety specification predicate, can be represented by a Guard object. However, unlike the Guard object used in the Program or the Fault objects described above, the Guard object in the Safety Specification has to evaluate whether a condition holds true for a transition instead of a state. For this reason, we use two types of Guard objects- a Source Guard and a Destination Guard. Given a transition, (s_1, s_2), the Source Guard evaluates whether the source state of the transition, s_1, satisfies a particular condition while the Destination Guard evaluates whether the destination state of the transition, s_2, satisfies a particular condition. We use a linked-list to represent the safety specification predicate in the Safety Specification object. For example, the figure shows a part of the data structure for the Safety Specification object in Byzantine agreement.

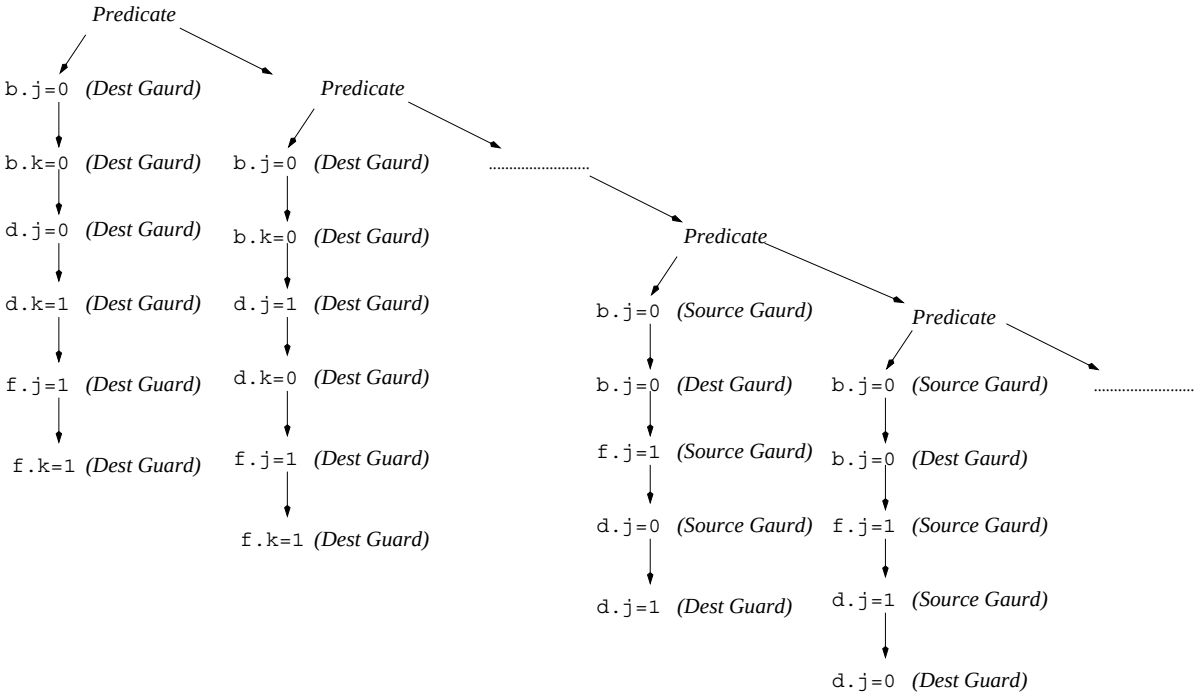

Figure 3: Safety Specification object for Byzantine agreement

In the figure, each vertical sequence of Guard objects corresponds to a disjunct in the safety specification predicate. When we wish to determine whether a transition violates safety, we sequentially test every disjunct in the predicate until we find a disjunct that evaluates true for the transition. If there is no disjunct in the predicate that evaluates true for the transition, the transition does not violate safety.

9 Discussion

The heuristics presented in Section 7 are applicable to variations of the byzantine agreement problem as well as other programs. For example, consider the problem where there are four non-general processes, $j, k.l$ and m. The heuristics in Section 7 can be used for this problem as well. The outline of the derivation is as follows: First, in step 4, we identify deadlocked states which are of the form $\langle 0,0,0,0,1\rangle$ or $\langle 0,0,0,1,1\rangle$. (Here, we have generalized the notation on page 11 where $\langle x_1, x_2, x_3, x_4, x_5\rangle$ denotes the set of states where $d.g$ equals x_1, $d.j$ equals x_2, $d.k$ equals x_3, $d.l$ equals x_4 and $d.m$ equals x_5.) Step 5 is first used to recover from states in $\langle 0,0,0,0,1\rangle$. Specifically, as shown in Section 7, we allow m to change $d.m$ to 0 and allow other processes to finalize their decision. Subsequently, recovery is possible from $\langle 0,0,0,1,1\rangle$ by changing $d.l$ (or $d.m$) to 0.

The heuristics in Section 7 are also applicable for the problem where there are four non-general processes, at most one process suffers from the byzantine fault and at most one suffers from the failstop fault. In this problem, the state space needs to expanded to deal with the case that a process may suffer from the failstop fault. However, the heuristics presented earlier allows us to derive the fault-tolerant program that tolerates both the byzantine fault and the failstop fault. In this derivation, in addition to the steps taken in the previous paragraph, we need to add transitions so that process j and k can finalize their decisions from states such as $\langle 0,0,0,1,\bot\rangle$.

The heuristics are also applicable in other problems as well. We leave it to the reader to verify that the same heuristics can be used to design the triple modular redundancy problem where there are three *input*s at most one of which is corrupted, and it is required that the *output* is eventually assigned the correct value. Also, the heuristics can be used to derive the token ring program in [10] where a token is being circulated along a ring, and the fault can *restart* all but one processes. The fault-tolerant program, derived thus, ensures that at any time there is at most one token in the ring, and eventually a state is reached where there is exactly one token. For reasons of space, we omit the derivation of this program.

The heuristics are also useful while designing failsafe or nonmasking fault-tolerance. In the design of failsafe fault-tolerance, where only safety specification needs to maintained in the presence of faults, we simply need to drop step 5 where we add recovery transitions (cf. Step 5.1). In the design of nonmasking fault-tolerance, where the program eventually reaches a state from where the specification is satisfied, we simply need to drop step 4 where transitions that violate safety are removed. Of course, if recovery were not possible from such states, step 5 may drop those transitions.

10 Conclusion and Future Work

Algorithm may fail.

In this paper, we presented heuristics for transforming a fault-intolerant distributed program into a fault-tolerant distributed program. We used those heuristics to present a polynomial time algorithm for synthesizing fault-tolerant programs. To show the effectiveness of heuristics, we presented the synthesis of the 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 to synthesize the solution to byzantine agreement. However, our heuristics allowed polynomial implenentation by allowing us to make a suitable deterministic choice about 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.

We also discussed extensions of the byzantine agreement program. We showed how the derivation will proceed if we had four non-general processes. We also pointed out how our heuristics are used in a program where one process suffers from byzantine fault and one process suffers from the failstop fault.

As the fault-tolerant program obtained by the synthesis algorithm is correct by construction, there is no need to manually develop its proof of correctness. Also, our algorithm will be 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. And, such a situatoin occurs frequently, e.g., when the designer needs to incrementally adapt a program to deal with new types of faults. And, the reuse of the fault-intolerant program will be virtually mandatory if the designer has only an incomplete specification.

Unlike previous work [11–18] that starts with a specification (typically in some temporal logic), we start with a fault-intolerant program that is known to be correct. For this reason, our algorithms only needed the safety specification that the program is supposed to satisfy in the presence of faults; the algorithms did not need the liveness specification. Also, previous synthesis algorithms have not considered the derivation of programs that tolerte byzantine faults.

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 7) can be used for adding failsafe fault-tolerance and the heuristics 1, 2 and 4 (cf. Section 7) 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 [19].

The synthesis tool, described in Section 8 will be used to develop a synthesis platform so that so that the user can specify precomputed a set of 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 designing fault-tolerant programs but also permit the derived fault-tolerant program to be more efficient. The synthesis platform will also provide an interactive interface to users: This would be achieved by allowing the user to see the intermediate program after applying some step, and asking the user to choose an heuristic that the user thinks is useful. Also, the user may specify preferences that determine how the recovery transitions are added and how states are eliminated. For example, the user could specify certain states to be *important* and should not be eliminated.

Regarding other extensions to our work, we will be focusing on further reducing the complexity of our algorithms by identifying heuristics that are fault-dependent and model-dependent. For example, we will identify how heuristics in Section 7 can be tailored based on the fault being considered. Also, we will identify whether it is possible to reduce the complexity further in a specialized model such as Read/Write model –where the process can read its neighbor's variable or write its own state but not both, or a shared memory model, where the process can read the variables of its neighbors but can only write its own variables.

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] 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] A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *International Conference on Distributed Computing Systems*, pages 436–443, May 1998. An extended version of this paper has been invited to *IEEE Transactions on Computers*.

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

[9] F. Gartner. Automating the addition of fault-tolerance: Beyond fusion-closed specifications. Personal Communication.

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

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

[12] 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.

[13] P. Attie and E. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Transactions on Programming Languages and Systems*, 20(1):51–115, 1998.

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

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

[16] A. Anuchitanukul and Z. Manna. Reliability and synthesis of reactive modules. *International Conference on Computer-Aided Verification*, pages 156–169, 1994.

[17] O. Kupferman and M. Vardi. Synthesis with incomplete information. *ICTL*, 1997.

[18] D. Dill and H. Wong-Toi. Synthesizing processes and schedulers from temporal specifications. *International Conference on Computer-Aided Verification*, 1990.

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