Exploiting Symbolic Techniques in Automated Synthesis of Distributed Programs[*]

Borzoo Bonakdarpour Sandeep S. Kulkarni
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
Email: {borzoo,sandeep}@cse.msu.edu
Web: http://www.cse.msu.edu/~{borzoo,sandeep}

Abstract

Automated formal analysis methods such as program verification and synthesis algorithms often suffer from time complexity of their decision procedures and also high space complexity known as the state explosion problem. Symbolic techniques, in which elements of a problem are represented by Boolean formulae, are desirable in the sense that they often remedy the state explosion problem and time complexity of decision procedures. Although symbolic techniques have successfully been used in program verification, their benefits have not yet been exploited in the context of program synthesis and transformation extensively. In this paper, we present a symbolic method for automatic synthesis of fault-tolerant distributed programs. Our experimental results on synthesis of classical fault-tolerant distributed problems such as Byzantine agreement and token ring show a significant performance improvement by several orders of magnitude in both time and space complexity. In particular, we show that synthesis for these problems is feasible with 25 processes, where the size of state space is 2^{102} for Byzantine agreement and 2^{50} for token ring. To the best of our knowledge, this is the first illustration where such large state space is handled during synthesis.

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

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

Report Documentation Page	Form Approved OMB No. 0704-0188

1. REPORT DATE **2007**	2. REPORT TYPE	3. DATES COVERED **00-00-2007 to 00-00-2007**
4. TITLE AND SUBTITLE **Exploiting Symbolic Techniques in Automated Synthesis of Distributed Programs**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Michigan State University,Department of Computer Science and Engineering,East Lansing,MI,48824**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**25**	

1 Introduction

Automated synthesis of programs has the potential to provide high assurance for computing systems, as programs are guaranteed to be correct-by-construction. This property is especially valuable for fault-tolerant systems, due to the difficulty in their verification. In this work, we focus on automated addition of fault-tolerance to existing (fault-intolerant) programs. One of the problems with such automated synthesis, however, is that both time and space complexity of such algorithms are often high, making it difficult to apply them for large problems.

The complexity of automated synthesis can be characterized in two parts. The first part has to deal with questions such as *which* recovery transitions/actions should be added, and *which* transitions/actions should be removed to prevent safety violation in the presence of faults. The second part has to deal with questions such as *how quickly* such recovery and safety violating transitions can be identified.

In our previous work [1], we focused on the first part, where we have identified classes of problems where efficient synthesis is feasible and developed different heuristics, especially for dealing with the constraints imposed by distributed nature of synthesized programs. To illustrate this, when a process, say j, in a distributed program executes an action, it corresponds to several possible program transitions depending upon the state of other processes. Even if some of these transitions violate safety, their inclusion may be necessary due to other useful transitions in the group. (When such a group of transitions is included, it would be necessary to ensure that when j can execute the corresponding action, the state of other processes is such that execution of the action will not result in safety violation.) In other words, while determining recovery/safety-violating transitions, it is necessary to consider the interdependence between different transitions of the program. Our previous work has developed heuristics for identifying recovery/safety-violating transitions by considering the interdependence between different transitions of the program.

Observe that the solution to the first part is independent of issues such as representation of programs, faults, specifications etc. Hence, we have utilized explicit-state (enumerative) techniques to identify the heuristics. Explicit-state techniques are especially valuable in this context, as we can identify how different heuristics affect a given program, and thereby enable us to identify circumstances where they might be useful. Explicit-state techniques, however, are undesirable for the second part, as they suffer from state explosion problem and prevent one from synthesizing programs where the state space is large. In other words, although the polynomial time complexity of the heuristics in [1] allows us to deal with the problem of synthesis of distributed programs, which is known to be NP-hard [2], their explicit-state implementation is problematic with scaling up for large programs.

With this motivation, in this paper, we focus on the second part of the problem to improve the time and space complexity of synthesis. Towards this end, we focus on symbolic synthesis (implicit-state) where programs, faults, specifications etc., are modeled using Boolean formulae (represented by Bryant's Ordered Binary Decision Diagrams [3]). Although symbolic techniques have been shown to be very successful in model checking [4] (e.g., model checkers SMV and SAL), they have not been greatly used in the context of program synthesis and transformation in the literature. Thus, in this paper, our goal is to evaluate how such symbolic synthesis can assist in reducing the time and space complexity, and thereby permit synthesis of large(r) programs.

To compare the efficiency improvement with symbolic synthesis, we compare it with problems that have been considered in the context of explicit-state synthesis. In particular, we focus on the problem of Byzantine agreement [5] and token ring [6], which are equally important in distributed computing. And, increasing the number of processes can create larger instances in both of them. This allows us to evaluate the trend in the time and space complexity for symbolic synthesis.

Related work. While there is an extensive line of research in the area of symbolic model checking (e.g., [4, 7, 8]), little work has been done in symbolic synthesis of programs. In [9], Asarin, Maler, and Pnueli introduce a symbolic method to synthesize discrete and timed controllers. At the semantic level, in their approach, the controller is synthesized by finding a winning strategy for either safety or reachability games (but not both) defined by traditional finite state automata or by timed-automata.

More recently, Wallmeier, Hütten, and Thomas [10] introduce an algorithm for synthesizing finite state controllers by solving infinite games over finite state spaces. In their work, the winning constraint is modeled by safety conditions and a set of request-response properties as liveness conditions. They transform this game into a Büchi game which involves an inevitable exponential blow-up. However, their approach does not address the issue of distribution. Moreover, the reported maximum number of variables in their experiments is 23.

In [11], de Alfaro, Faella, and Legay introduce the tool TICC for facilitating and game-based modeling of software and distributed systems. TICC is developed based on the theory of interface automata [12]. In TICC, components are modeled both via variables (to describe state) and actions (to describe synchronization). The implementation of TICC relies on symbolic methods, yielding algorithms for component and system analysis. In particular, TICC can synthesize input assumptions of *some* environment in which a distributed program can be executed without reaching a deadlock due to unanticipated moves by the environment.

Contributions of the paper. Our contributions in this paper are as follows.

1. We illustrate that our symbolic technique can significantly improve the performance of synthesis in terms of both

time and space complexity. In particular, our analysis shows that the growth of the total synthesis time is *sublinear* in the state space. For example, in case of Byzantine agreement for five processes, the time for explicit-state synthesis was 15 minutes whereas the time with symbolic synthesis was 1.2 seconds.

2. Symbolic synthesis significantly assists in coping with the space complexity. For example, we could synthesize a solution for Byzantine agreement with 25 processes. The size of state space of such a program is 2^{102} and the size of reachable states is 2^{60}, whereas in our implementation the amount of memory used during synthesis was only 131 KB ($< 2^{18}$). To the best of our knowledge, this paper is the first that can deal with such large state space in the context of program synthesis.

3. We analyze the cost incurred in different tasks during synthesis. In particular, our analysis identifies three bottlenecks that need to be overcome, namely, (1) deadlock resolution, (2) computation of reachable states in the presence of faults, and (3) checking whether a group of transitions violates the safety specification. We show that depending upon the structure of distributed programs, a combination of these bottlenecks may affect the performance of automated synthesis.

We would like to note that, just as with model checking, this work does not imply that synthesis would be feasible for all programs where the size of state space is 2^{102}. However, this work does illustrate that large state space by itself is not an obstacle to permit efficient synthesis. Finally, while techniques such as symmetry reduction or other abstraction techniques have the potential to reduce the complexity of synthesis, with the use of such techniques, the actual state space of the given problem may not correspond to the state space encountered during synthesis. Since our goal in this work has been to focus on feasibility of dealing with large state space and its impact to the automated synthesis and transformation of distributed programs, we have chosen not to use such techniques.

Organization of the paper. In Section 2, we present the formal definition of distributed programs, specifications, and fault-tolerance. In Section 3, the formal statement of the synthesis problem is presented. Then, in Section 4, we model the heuristics introduced in [1] symbolically. In Section 5, we present our experimental results on different aspects and subtasks of symbolic synthesis of Byzantine agreement and token ring. Finally, in Section 6, we outline a roadmap for further research and present concluding remarks.

2 Preliminaries

In this section, we formally define the notions of distributed programs, specifications, and fault-tolerance. The notion of distributed programs is adapted from [2]. The formal definition of specifications is due to Alpern and Schneider [13]. Definition of faults and fault-tolerance are based on the ones given by Arora and Gouda [14] and Kulkarni [15]. Since in this paper, we deal with symbolic algorithms, all definitions presented in this section are in terms of Boolean formulae. For the sake of completeness, we briefly recap the concept of Ordered Binary Decision Diagrams (OBDDs) [3] as well.

2.1 Program

Let V be a finite set of *Boolean variables* $\{v_0, v_1 \cdots v_n\}$. A *state* is determined by the function $s : V \mapsto \{\mathit{true}, \mathit{false}\}$, which maps each variable in V to either true or false. Thus, we represent a state s by the conjunction $s = \bigwedge_{j=0}^{n} l(v_j)$ where $l(v_j)$ denotes a *literal*, which is either v_j itself or its negation $\neg v_j$. In general, non-Boolean variables (e.g., integers) with finite domain D can be represented by $\log(|D|)$ Boolean variables. Hence, our notion of state is not restricted to Boolean variables.

A *state predicate* is a finite set of states. Formally, we specify a state predicate $S = \{s_0, s_1 \cdots s_m\}$ by the disjunction $S = \bigvee_{i=0}^{m}(s_i)$. Observe that although the resulting formula is in disjunctive normal form, one can represent a state predicate by any equivalent Boolean expression. The *state space* is the set of all possible states obtained from the associated variables. We denote the membership of a state s in a state predicate S (i.e., truthfulness of $s \Rightarrow S$) by $s \models S$.

A *transition* is a pair of states of the form (s, s') specified as a Boolean formula as follows. Let V' be the set $\{v' \mid v \in V\}$ (called *primed variables*). We use these variables to show the new value of variables assigned by a transition. Thus, we define a transition (s, s') by the conjunction: $s \wedge s'$. A *transition predicate* P is a finite set of transitions $\{t_0, t_1 \cdots t_n\}$ defined by $P = \bigvee_{j=0}^{n}(t_j)$. We denote the membership of a transition (s, s') in a transition predicate P (i.e., truthfulness of $(s \wedge s') \Rightarrow P$) by $(s, s') \models P$.

Notation. Let X be a state predicate. We use $\langle X \rangle'$ to denote a state predicate equal to X whose variables are primed. Let P be a transition predicate whose source and target state predicates are X_1 and X_2, respectively. We use $\langle P \rangle''$ to denote the state predicate equal to X_2 whose variables are unprimed. Also, we use $\mathit{Guard}(P)$ to denote the source state predicate of P (i.e., $\mathit{Guard}(P) = X_1$).

A *program* is specified by a set of variables V and a transition predicate P in its state space (denoted S_p). We say that a state predicate S is *closed* in the program P iff $\bigwedge_{(s,s') \models P}((s \models S) \Rightarrow (s' \models S))$ holds, i.e., if a transition of P originates from state predicate S then it ends in state predicate S as well. A sequence of states, $\langle s_0, s_1, ...\rangle$, is a *computation* of P iff the following two conditions are satisfied: (1) $\forall j \mid j > 0 : (s_{j-1}, s_j) \models P$, 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 $(s_l, s) \models P$. The *projection* of program P on state predicate S, denoted as $P|S$, is the program (i.e., transition predicate) $\bigvee_{(s,s')\models P}((s \models S) \wedge (s' \models S))$. I.e., $P|S$ consists of transitions of P that start in S and end in S.

2.2 Specification

A *specification* is a set of infinite sequences of states. Following Alpern and Schneider [13], we let the specification consist of a *safety specification* and a *liveness specification*. For our synthesis algorithm, the safety specification of a program P is specified by a transition predicate $SPEC$ in the state space of P which represents a set of "bad transitions" that should not occur in the program computation. We show that the synthesized program (i.e., the fault-tolerant program) satisfies the liveness specification iff the original program (i.e., 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.

Given a program P, a state predicate S, and a specification $SPEC$, we say that P *satisfies* $SPEC$ *from* S iff (1) S is closed in P, and (2) for all computations $\langle s_0, s_1, \cdots\rangle$ of P, where $s_0 \models S$, $(s_{j-1}, s_j) \not\models SPEC$. If P satisfies $SPEC$ from S and $S \neq false$, 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 no transition in $\alpha\beta$ is in $SPEC$. Otherwise, we say that α *violates* $SPEC$.

Notation. 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 and Fault-Tolerance

The faults that a program P is subject to are systematically represented by a transition predicate F in the state space of 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 in the presence of F* iff the following three conditions are satisfied: (1) $\forall j \mid j > 0 : (s_{j-1}, s_j) \models (P \vee F)$, (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 $(s_l, s) \models P$, and (3) $\exists n \mid n \geq 0 : (\forall j \mid j > n : (s_{j-1}, s_j) \models P)$. 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 is 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 satisfies $SPEC$ 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.

2.4 Modeling Distributed Programs

In this subsection, we present the distribution model that specifies how read/write restrictions imposed on a distributed program are captured during synthesis. To capture the notion that all variables cannot be read/written simultaneously, we introduce the notion of *processes*; a process j is specified by (1) a set V_j of variables, (2) a transition predicate P_j, (3) a set R_j of variables it can read, and (4) a set W_j of variables it can write. We now describe how read/write restrictions on a process affect its transitions.

Write restrictions. Let $v(s)$ denote the value of a variable v in state s. Given a transition (s, s'), it is straightforward to determine the variables that need to be changed in order to modify the state from s to s'. Hence, if process j can only write the variables in the set W_j and the value of a variable other than those in W_j is changed by the transition (s, s') then that transition cannot be used in obtaining the transitions of j. In other words, if j can only write variables in W_j then j cannot use the transitions of the following transition predicate:

$$NW_j = \bigvee_{(s,s') \models S_p \times S_p} (\bigvee_{v \notin W_j} (v(s) \neq v(s'))).$$

Likewise, we define the transition predicate in which process j changes the value of one of the variables in W_j as follows:

$$WW_j = \bigvee_{(s,s') \models S_p \times S_p} ((v(s) \neq v(s')) \Rightarrow v \in W_j).$$

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. Observe that executing an individual transition (s_0, s_0') determines the variables that will be written by (s_0, s_0'). Also, to execute (s_0, s_0') individually, a process j must read the value of all program variables to ensure that the current state is s_0 and then take the state of the program to s_0'. However, in a distributed program, a process may have restrictions in reading the local variables of other processes. As an example, consider a program consisting of two variables a and b. Suppose that we have a process j that cannot read b. Now, observe that the transition $\neg a \wedge \neg b \wedge a' \wedge \neg b'$ and the transition $\neg a \wedge b \wedge a' \wedge b'$ have the same effect on the processes as far as j is concerned, as j cannot read b. Thus, the uncertainty of a process regarding the local variables of other processes creates an equivalence class of transitions (called *group predicate*) corresponding to $(s_0, s_0^{'})$ such that for every transition (s_1, s_1') in this equivalence class (i) the value of all readable variables for process j are equal in s_0 and s_1 (respectively, s_0' and s_1'), and (ii) the values of unreadable variables for j remain unchanged in s_0 and s_0' (respectively, s_1 and s_1'). Thus, if (s_0, s_0') is a transition of process j whose set of readable variables is R_j, the corresponding group predicate is defined as follows:

$$\begin{aligned}
&\mathit{Group}(j, R_j)(s_0, s_0') = \\
&\quad \bigvee_{(s_1,s_1') \models S_p \times S_p} (\bigwedge_{v \in R_j} (v(s_0) = v(s_1) \ \wedge \ v(s_0') = v(s_1')) \ \wedge \\
&\qquad\qquad \bigwedge_{v \notin R_j} (v(s_0) = v(s_0') \ \wedge \ v(s_1) = v(s_1')))
\end{aligned}$$

Likewise, one can define the group predicate corresponding to a transition predicate P as the union of group predicates of each transition in P.

2.5 Ordered Binary Decision Diagrams

Ordered Binary Decision Diagrams (OBDDs) [3] represent Boolean formulae as directed acyclic graphs. They form a canonical representation, making testing of functional properties such as satisfiability and equivalence straightforward and extremely efficient.

The main idea in creating the OBDD of a Boolean formula is removing redundant nodes in the complete binary tree that corresponds to the truth table of the Boolean formula. As an example, let us consider the Boolean formula $A = (a_1 \wedge b_1) \ \vee \ (a_2 \wedge b_2) \ \vee \ (a_3 \wedge b_3)$. Figure 1 shows two OBDD representations of the formula A. In this figure, a dashed (respectively, solid) branch denotes the case where the decision variable is false (respectively, true). Moreover, for the case where the variables are ordered $a_1 < b_1 < a_2 < b_2 < a_3 < b_3$, the corresponding OBDD is the one in Figure

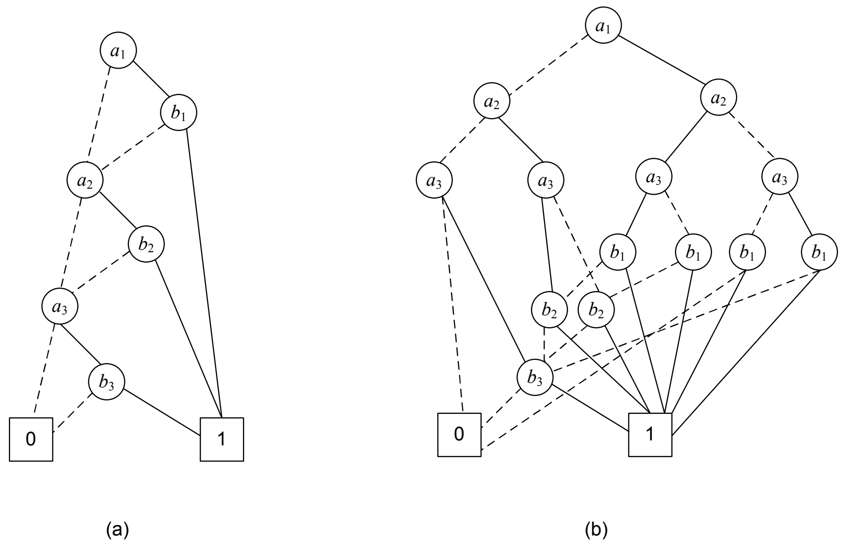

Figure 1: OBDD representations of a single function for two different variable orderings.

1-a, while for the case where the variables are ordered $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$, the corresponding OBDD is in Figure 1-b. Although both OBDDs represent the same Boolean formula, and both are more efficient than the complete binary tree that represents the truth table of A, the one in Figure 1-a obviously requires less memory. Thus, variable ordering can significantly affect the efficiency of OBDDs.

In this paper, we represent state predicates and transitions predicates (e.g., programs, faults, invariant, etc.) by OBDDs. However, we do not focus on how basic Boolean operators over OBDDs are implemented (see [3] for details). In fact, in our implementation, we use the CUDD package [16] which provides us with a library for creating and manipulating OBDDs.

3 Problem Statement

We now formally present the statement of our synthesis problem based on [1]. Given are a program P with invariant S, a class of faults F, and specification $SPEC$ such that P satisfies $SPEC$ from S. Our goal is to find a program P' with invariant S' such that P' is F-tolerant to $SPEC$ from S'. Observe that:

1. If S' contains states that are not in S then, in the absence of faults, P' may include computations that start outside S. Since we require that P' satisfies $SPEC$ from S', it implies that P' is using a new way to satisfy $SPEC$ in the absence of faults. Thus, we require that $S' \Rightarrow S$.

2. If $P'|S'$ contains a transition that is not in $P|S'$ then P' can use this transition in order to satisfy $SPEC$ in the

absence of faults. Thus, we require that $(P'|S') \Rightarrow (P|S')$.

Following the above observation, the synthesis problem is as follows.

Synthesis problem. Given P, S, F, and $SPEC$ such that P satisfies $SPEC$ from S. Identify P' and S' such that:

$(C1)$ $S^{'} \Rightarrow S$,

$(C2)$ $(P^{'}|S^{'}) \Rightarrow (P|S^{'})$, and

$(C3)$ P' is F-tolerant to $SPEC$ from S'. ∎

4 The Symbolic Synthesis Algorithm

In this section, we present a symbolic algorithm based on the heuristics developed in [1]. The symbolic representation of the heuristics in terms of Boolean formulae will later enable us to implement them using OBDDs. Using the definitions given in Section 2, we are now ready to develop our symbolic algorithm.

Algorithm sketch. The algorithm Symbolic_Add_FT (cf. Figure 2) consists of five steps. The first step is initialization, where we identify state and transition predicates from where execution of faults alone violate the safety specification. In the later steps, we ensure that such state and transition predicates will remain unreachable. In Step 2, we identify the fault-span by computing the state predicate reachable by program and fault transitions starting from the program invariant. In Step 3, we identify and rule out transitions whose execution violates the safety specification. Then, in Step 4, we resolve deadlock states. Finally, in Step 5, we recompute the invariant predicate to ensure that it is closed in the program. We repeat steps 2-3, 2-4, and 2-5 until a fixpoint is reached. The fixpoint computations are represented by three nested loops in the algorithm. Thus, the algorithm terminates when no progress is possible in all the steps described above.

Step 1: Initialization (Lines A.1-A.3) First, we compute the state predicate ms from where execution of faults alone violate the safety specification (Line A.1). This state predicate should never become true in any program computation (Line A.2), as faults (which are uncontrollable events) may lead the program to a state where there exists a transition in $SPEC \wedge F$. We compute the state predicate ms by exploring backward reachable states using fault transitions starting from where faults directly violate safety (i.e., $Guard(SPEC \wedge F)$). We let the Procedure BackwardReachableStates compute the set of backward reachable states. Since this procedure is a standard state exploration algorithm, we do not include the details in Figure 2. (see [4, 7] for more details)

Likewise, we compute the transition predicate mt whose transitions should not be executed by the fault-tolerant program. Initially, mt is equal to the union of $SPEC$ and transitions that start from any arbitrary state and end ms (denote $\langle ms \rangle'$).

Since the fault-tolerant program is not supposed to reach a state in ms, we allow the transitions that originate in ms to be in the fault-tolerant program (Line A.3). Observe that although a state predicate, in Line A.3, we interpret ms as a transition predicate that starts in ms and ends in $true$.

Step 2: Recomputing the fault-span (Lines A.6-A.8). After initializations, we start recomputing invariant, program transition predicate, and fault-span of the fault-tolerant program in 3 nested loops, respectively. We first describe the inner loop, where we recompute the fault-span. Initially, we let the fault-span T_1 be equal to the invariant S_1 (Line A.5). We recompute the fault-span by exploring the state predicate reachable by both program and fault transitions (i.e., $P_1 \vee F$) starting from the invariant S_1 (Line A.7). While exploring the reachable states, we ignore the states in state predicate ne. This state predicate is identified later in Step 4 in this section where we explain how we resolve deadlock states.

Now, suppose there exists a state predicate which is a subset of $Guard(mt)$, but it is unreachable by transitions in $P_1 \vee F$ starting from the invariant (i.e., it does not intersect with the fault-span). Since this state predicate is unreachable by computations of the program even in the presence of faults, we let the transitions that originate in that state predicate be in the fault-tolerant program (Line A.8).

Step 3: Identifying unsafe transitions (Lines B.1-B.4). Once we recompute the fault-span, we invoke the procedure CheckGroupSafety to rule out the transitions that violate the safety specification (Line A.9). The fault-tolerant program transition predicate is computed based on the following principle: a transition can be included if it is not in mt. Also, note that a transition can be included in the fault-tolerant program if its corresponding group predicate can be included. Thus, for each process j, first, we compute the intersection of P_j and mt inside the fault-span T (Line B.1) to identify the program transitions that violate safety inside the fault-span. The guard of the resulting transition predicate gives us the state predicate (denoted $Witness$) in the fault-span from where safety is violated due to executing program transitions. Thus, we ensure that no process can execute its unsafe actions where $Witness$ is true by conjoining actions of each process with $\neg Witness$ (Line B.2).

Step 4: Resolving deadlock states (Lines A.11-A.15 and D.1-D.14). Since we might have removed some transitions from the original program when we reach a fixpoint in the inner loop, we identify *deadlock states* inside the fault-span. A state s is deadlocked if there is no outgoing program transition from s. We treat such deadlock states in two ways: (1) we add a safe recovery path from a deadlock state to the program invariant, or (2) (if recovery is not possible) we eliminate the deadlock state. We now describe both mechanisms. Since we are dealing with a symbolic algorithm, first, we identify the deadlock state predicate ds (Line A.11).

- (*Adding safe recovery paths : Lines A.11-A.15*) We add recovery paths in an iteratively layered fashion. Let the

```
Algorithm Symbolic_Add_FT($P, F, SPEC$: transition predicate, $S$: state predicate,
                                  $R_1..R_n, W_1..W_n$: set of variables)
{
    $ms$ :=BackwardReachableStates($Guard(SPEC \wedge F), F$);                    (A.1)
    $S_1 := S - ms$;                                                              (A.2)
    $mt := (\langle ms \rangle' \vee SPEC) \wedge \neg ms$;                       (A.3)
    repeat
        $S_2, ne := S_1, false$;                                                  (A.4)
        repeat
            $T_1, P_2 := S_1, P_1$;                                               (A.5)
            repeat
                $T_2 := T_1$;                                                     (A.6)
                $T_1 :=$ ForwardReachableStates($S_1$, $P_1 \vee F$, $ne$);       (A.7)
                $mt := mt \wedge T_1$;                                            (A.8)
                $P_1 :=$ CheckGroupSafety($P_1, R_1..R_n$);                       (A.9)
            until $(T1 = T2)$;
            $lyr := S_1$;                                                         (A.10)
            repeat
                $ds := T_1 \wedge \neg Guard(P_1)$;                               (A.11)
                $rt := ds \wedge \langle lyr \rangle' \wedge \neg mt$;            (A.12)
                $P_1 := P_1 \vee$ CheckGroupSafety($rt, R_1..R_n$);               (A.13)
                $lyr := ds \wedge \neg(T_1 \wedge \neg Guard(P_1))$;              (A.14)
            until $(lyr = false)$;
            $P :=$ Eliminate($ds, T_1, S_1, false, P_1, F$);                      (A.15)
        until $(P_1 = P_2)$;
        $P_1, S_1 :=$ ConstructInvariant($P_1$, $S_1$, $ne$);                     (A.16)
    until $S1 = S2$;
    return $S_1$, $P_1$;
}

Procedure CheckGroupSafety($P$: transition predicate, $R_1..R_n$: set of variables)
{
    for each process $j \in 1..n$:
        $Witness = Guard(Group(P \wedge T \wedge mt, R_j))$;                      (B.1)
        $P_j := (\neg Witness) \wedge P_j$;                                       (B.2)
    $P := \vee_{j=1}^{n} P_j$;                                                    (B.3)
    return $P$;                                                                   (B.4)
}

Procedure ConstructInvariant($P$: transition predicate, $S, ne$: state predicate)
{
    $OffendingStates := S \ \wedge$ BackwardReachableStates($ne, F$);             (C.1)
    repeat
        $S := S \wedge \neg OffendingStates$;                                     (C.2)
        $tmp := (S \vee OffendingStates) \wedge P \wedge \neg \langle S \rangle'$;  (C.3)
        $P := P \wedge \neg Group(tmp)$;                                          (C.4)
        $OffendingStates := \langle tmp \rangle''$;                               (C.5)
    until $(OffendingStates = false)$;
    return $S, P$;                                                                (C.6)
}
```

Figure 2: Symbolic algorithm for synthesizing fault-tolerant distributed programs.

first layer be the program invariant S_1 (Line A.10). Also, let rt be the transition predicate that originates from deadlock state predicate ds, ends in lyr, and is disjoint from mt (Line A.12). We check whether the group predicate of rt satisfies the safety specification. If so, we add this group predicate to P_1 (Line A.13). In the next iteration, we let lyr be the state predicate from where one-step safe recovery is possible (Line A.14). We continue adding recovery transition predicates until no new transition is added.

- (*Eliminating deadlock states: Lines D.1-D.14*) Now, if safe recovery is not possible from a deadlock state predicate, say ds, we eliminate this predicate by invoking the procedure Eliminate (Line A.15) described next (cf. Figure 3 for pseudo-code). First, if deadlock states ds has been considered for elimination before, the procedure returns (Lines D.1 and D.2). If ds has not been considered for elimination and it is reachable by some program transitions, we temporarily remove such transitions, say tmp, with the hope that this removal makes ds unreachable (Line D.5). However, unlike program transitions, since we do not have control over the occurrence of faults, if ds is reachable by fault transitions from another state predicate, say fs (Line D.7), then we invoke Eliminate recursively with parameter fs (Line D.8). Notice that in Line D.7, we rule out the case where ds is reachable by faults alone from the program invariant. We will consider this case when we recompute the program invariant in Step 5. Now, if the removal of the transition predicate tmp causes some state predicate, say New, to be deadlocked (Line D.9), we add the transitions originating from New back to the program (Line D.11) and attempt to ensure that New is never reached by invoking Eliminate with parameter $(New \wedge \neg S)$ recursively (Line D.13). Again, we consider the case where New is in the program invariant in Step 5. Notice that the variable ne keeps record of the states that are considered but not eliminated in this step (Line D.10).

Step 5: Recomputing the invariant (Lines C.1-C.6). Once we reach a fixpoint for resolving deadlock states, we recompute the invariant by invoking the Procedure ConstructInvariant (Line A.16) due to two reasons. First, let $OffendingStates$ be a state predicate (called *offending state predicate*) inside the invariant from where execution of faults alone can cause the program to reach the state predicate ne identified in Step 4 (Line C.1). We remove such offending states from the invariant, as if a program reaches ne, there is no way to recover to the program invariant (Line C.2). Then, since we remove the offending states from the program invariant, we need to ensure that the invariant predicate is closed in P. To this end, we remove the transition predicate that violates the closure of P (Lines C.3 and C.4). We continue this procedure until a fixpoint is reached in the sense that no offending states exist and S is closed in P. Once we reach a fixpoint, since we have removed some states from the program invariant, we repeat steps 2-5 (the most outer loop in

```
Procedure Eliminate($ds, T, S, Visited$: state predicate, $P, F$: transition predicate)
{
    $ds := ds \wedge \neg Visited$;                                                      (D.1)
    if ($ds = false$) then return $P$;                                                   (D.2)
    $Visited := Visited \vee ds$;                                                        (D.3)
    $Old := P$;                                                                          (D.4)
    $tmp := T \wedge \neg S \wedge P \wedge \langle ds \rangle'$;                         (D.5)
    $P := P \wedge \neg Group(tmp)$;                                                     (D.6)
    $fs = Guard(T \wedge f \wedge \langle ds \rangle' \wedge \neg S) \ \wedge \ \neg$ ForwardReachableStates($S, F, false$);   (D.7)
    $P :=$ Eliminate($fs, T, S, Visited, P, F$);                                         (D.8)
    $New := Guard(T \wedge Group(tmp) \wedge \neg Guard(P))$;                            (D.9)
    $ne := ne \vee \neg \langle Old \wedge \neg P \wedge T \wedge \langle ds \rangle' \rangle''$;   (D.10)
    $P := P \vee (Group(tmp) \wedge New)$;                                               (D.11)
    $New := New \wedge Guard(tmp)$;                                                      (D.12)
    $P :=$ Eliminate($New \wedge \neg S, T, S, Visited, P, F$);                          (D.13)
    return $P$;                                                                          (D.14)
}
```

Figure 3: State elimination algorithm for resolving deadlock states.

Figure 2) to ensure that the faults-span is closed in $P \vee F$, and no new deadlock states are introduced to the fault-span.

5 Experimental Results

In this section, we present the experimental results of implementation of the Algorithm Symbolic_Add_FT presented in Section 4. In particular, we describe the results in the context of two classical examples in the literature of distributed computing, namely, Byzantine agreement [5] and token ring [6]. In both case studies, we find a considerable improvement in both time and space complexity as compared to the explicit-state model.

Throughout this section, all experiments are run on a Sun Fire V40z with a dual-core Opteron processor and 16 GB RAM. The OBDD representation of the Boolean formulae has been done using the C++ interface to the CUDD package developed at University of Colorado [16].

In this section, 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 (called *guard*), and st is a *statement* that describes how the program state is updated. Thus, an action $g \longrightarrow st$ denotes the transition predicate $\{(s, s') \mid s \Rightarrow g$ and s' is obtained by changing s as prescribed by $st\}$.

5.1 Case Study 1: Byzantine Agreement

In this subsection, we present our experimental results on automated synthesis of the Byzantine generals problem due to Lamport, Shostak and Pease [5]. More specifically, we use a canonical version of the problem modeled by Kulkarni, Arora, and Chippada [1].

The program consists of a "general" (g) and three (or more) "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 $\perp$, where $\perp$ 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 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.

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

The set of variables j is allowed to read, R_j, is $\{b.j, d.j, f.j, d.k, d.l, d.g\}$. The set of variables that j is allowed to write, W_j, is $\{d.j, f.j\}$. If $b.j$ is true then fault transitions can change $d.j$ and $f.j$.

Fault-intolerant program. 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.

```
$(d.j = \perp) \wedge \neg f.j$    $\longrightarrow$    $d.j := d.g$
$(d.j \neq \perp) \wedge \neg f.j$    $\longrightarrow$    $f.j := true$
```

Fault actions. 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:

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

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.

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

Moreover, a transition violates safety if it reaches a state where S_{sf} is true. Also, once a process finalizes its decision it cannot change that decision. Thus, the transitions that violate safety are as follows:

$$\begin{aligned} SPEC = \quad & S_{sf} \vee \bigvee_{(s,s') \models S_p \times S_p} \neg b.j(s) \wedge \neg b.j(s') \wedge f.j(s) \\ & \qquad\qquad \wedge \ (d.j(s) \neq d.j(s') \vee f.j(s) \neq f.j(s'))\} \end{aligned}$$

Fault-tolerant program. The output of our implementation is a program that tolerates the Byzantine faults identified above, i.e., it never violates its safety specification and it does not deadlock when faults take the program outside the invariant. Intuitively, the fault-tolerant program consists of (1) strengthened actions of the intolerant program, making deadlock states and states from where safety may be violated unreachable, and (2) new safe recovery actions. Notice that, our synthesized program is the same as the canonical Byzantine agreement program manually designed in [5]. The actions of the synthesized program for a non-general process j are as follows.

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

Analysis of implementation results. We now present the results of our experiments using the implementation of the Algorithm Symbolic_Add_FT. For our analysis, we present three graphs based on (1) total synthesis time (cf. Figure 4), (2) deadlock resolution time (cf. Figure 5), and (3) the amount of required memory (cf. Figure 6) all versus the size of explicit state space. We choose to analyze our data versus the size of explicit state space rather the number of processes since the size of explicit state space shows the exponential blow up of both time and space more clearly if an enumerative approach is applied. Based on the results presented in this section, we argue that automated synthesis of fault-tolerant distributed programs certainly has the potential to be used in practice with comparable scaling factor to that of model checking of such programs.

- (*Total synthesis time*) Figure 4 illustrates the time spent to synthesize fault-tolerant non-general processes versus the size of explicit state space. The number of processes synthesized in our experiments ranges over 3 to 25. Although it is feasible to synthesize programs with more number of processes in a reasonable amount of time, the trend of the graph with maximum 25 processes is clear enough to make sound judgments. Notice that both axes are in logarithmic scale. First, observe that it takes 1.2 seconds to synthesize 5 non-general processes. Surprisingly, a previous enumerative implementation of the heuristics of [1] takes 15 minutes to synthesize the same number of processes. Moreover, the previous enumerative implementation could not handle more than 5 processes due to the large size of state space. By contrast, using symbolic techniques, we were able to synthesize up to 35 processes in a reasonable amount of time, which is indeed a significant improvement. Note that the size of state space of the Byzantine agreement with 35 processes is 10^{31} times bigger than the size state space of Byzantine agreement with 5 processes.

  Moreover, the graph in Figure 4 shows that the growth rate of total time spent to synthesize Byzantine agreement is *sublinear* to the size of explicit state space. In particular, our analysis shows that fraction $\frac{Time}{StateSpace^{0.15}}$ remains constant as the number of non-general processes grows. Sublinearity of total synthesis time to the size of state space is important in the sense that the exponential blow-up of state space does not affect the time complexity of our synthesis algorithm.

- (*Deadlock resolution*) Figure 5 shows the time spent to resolve deadlock states versus the size of explicit state space. Surprisingly, in case of Byzantine agreement the graph is almost identical to the graph of total synthesis time. In fact, in the range of 3-25 processes, in average, 94% of the total synthesis time is spent to resolve deadlock states, namely, in adding recovery actions (cf. Lines A.11-A.14 in the Algorithm Symbolic_Add_FT in Figure 2) and in the Procedure Eliminate (cf. Figure 3). In other words, only 6% of the total synthesis time is spent to compute the fault-span of the program, checking safety of groups of transitions, and recomputing the program invariant. Note that deadlock resolution (as defined in Section 4) is a problem that uniquely exists in the context of program synthesis and transformation and, hence, has not been addressed by the model checking community. Note that the existence and diversity of deadlock states directly depends on the structure of the given program. In fact, later in

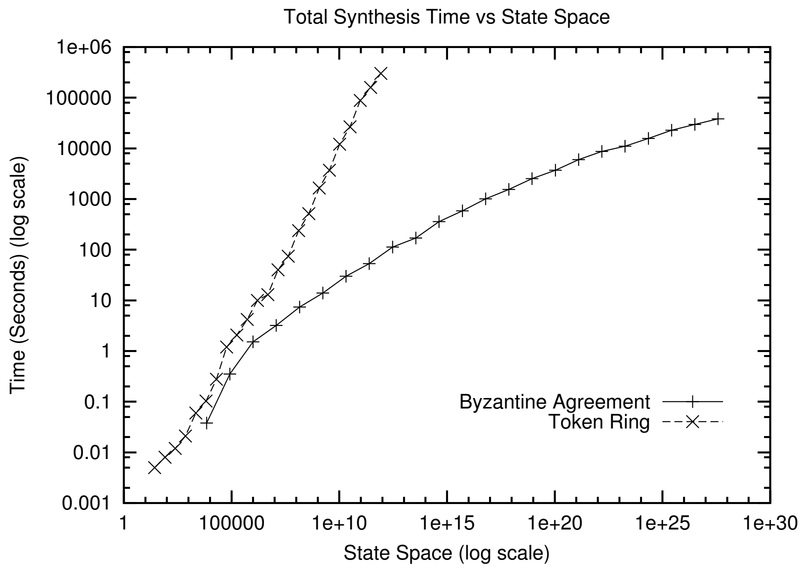

Figure 4: Total synthesis time versus the size of explicit state space in synthesis of Byzantine agreement and token ring for 3-25 processes.

this subsection, we show that in case of token ring, deadlock resolution is not a crucial issue.

Figure 5 also shows that in case we are not required to resolve deadlocks states, synthesis of programs such as Byzantine agreement can be done considerably faster. In other words, synthesis of distributed *failsafe* programs, where a program only guarantees to satisfy its safety specification in the presence of faults and is not required to recover to its invariant after occurrence of faults, can be done more efficiently.

- (*Memory usage*) Figure 6 shows the amount of virtual memory that the Algorithm Symbolic_Add_FT requires (in KB) versus the size of explicit state space. As can be seen, the amount of memory that the algorithm requires to synthesize 25 processes (131 KB) is not considerably greater than the amount of memory required to synthesize 3 processes (16 KB) as compared to the size of explicit state space in case of 3 and 25 processes. This is certainly due to efficient representation of Boolean formulae by OBDDs and partially due to the size of "reachable" states in the fault-span of Byzantine agreement. To illustrate the issue of size of reachable states let us consider the Byzantine agreement program with 25 processes. Since we represent the decision value of each processes by two Boolean variables, as the size of their respective domain is 3, each non-general processes has 4 variables. Also, the general has 2 variables. Hence, the program has 102 variables in total and the size of explicit state space is 2^{102}. In order to compute the size of reachable states approximately, observe that non-general processes are either undecided (i.e., $d.j = \perp$), or they are decided (i.e., $d.j = 0|1$) and their decision is either finalized or not yet finalized (i.e., $f.j = 0|1$). Hence, each non-general can have 5 different combinations. Furthermore, the general

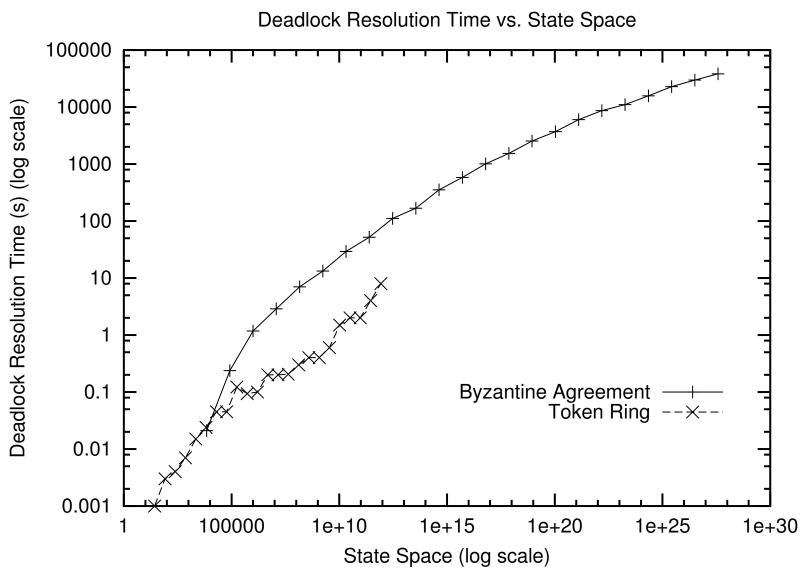

Figure 5: Deadlock resolution time versus the size of explicit state space in synthesis of Byzantine agreement and token ring for 3-25 processes.

can have either decision value (i.e., $d.g = 0|1$) and be Byzantine or non-Byzantine (i.e., $b.g = 0|1$). Hence, the size of reachable states is at least $5^{25} * 4 \simeq 2^{60}$. Thus, the size of reachable states is considerably less than the size of entire explicit state space, but still considerably greater than the amount of memory that the Algorithm Symbolic_Add_FT requires.

The issue of order of variables in OBDDs. As mentioned in Subsection 1, variable ordering can significantly affect the efficiency of OBDDs in terms of both space and time. In our implementation, we order the variables based on the following two principles, regardless of the structure of the given fault-intolerant program:

- Each primed variable is always ordered immediately after its unprimed variable, and
- Variables of each process are ordered one after another.

For instance, in Byzantine agreement program, for all i, where i is the process number, the order of variables is as follows:

$$d.i < d'.i < f.i < f'.i < b.i < b'.i < d.(i+1) < d'.(i+1) < f.(i+1) < f'.(i+1) < b.(i+1) < b'.(i+1) < \cdots.$$

The first principle is also being applied in existing model checkers (e.g., SAL), which is due to the fact that transitions often update a subset of variables, say U, and leave the rest unchanged. Hence, for each variable that is not in V, XNOR of the variable and its primed variable must hold. Therefore, to keep the OBDD of XNORs small, it is more efficient to order each primed variable immediately after its unprimed variable.

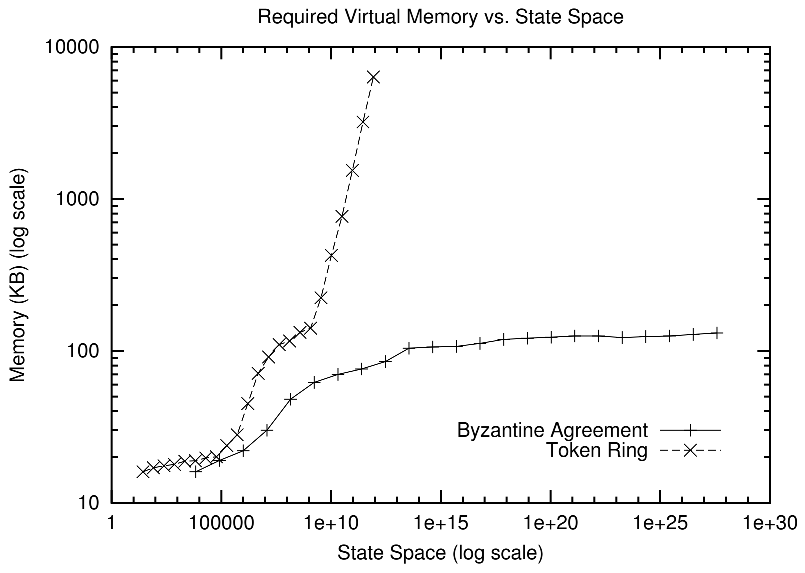

Figure 6: Required memory versus the size of explicit state space in synthesis of Byzantine agreement and token ring for 3-25 processes.

On the other hand, the second principle is based on our experiments. In particular, if we change the order of variables of processes externally, it decreases the performance. For instance, a counterintuitive result is if we order the variables such that the ones that can be read by all processes are next to each other, the performance drops down. (In that case, it takes one 1 minute to synthesize 5 processes.) However, changing the order of variables of a process internally (e.g., $d.i$ and $f.i$) does not change the performance of synthesis. Note that most OBDD packages, including CUDD, offer very efficient heuristics for variable ordering and one does not need to order variables manually. We did so to analyze the behavior of our synthesis algorithm.

5.2 Case Study 2: Token Ring

In a token ring program, the processes $0..N$ are organized in a ring and the token is circulated along the ring in a fixed direction. Each process, say j, maintains a variable with the domain $\{0, 1, \perp\}$, where $\perp$ denotes a corrupted value. Process j, $j \neq 0$, has the token iff $x.j$ differs from its successor $x.(j+1)$ and process N has the token iff $x.N$ is the same as its successor $x.0$. Each process can only write its local variable (i.e., $x.j$). Moreover, a process can only read its own local variable and the variable of its predecessor.

Fault-intolerant program. The program, consists of two actions for each process j. Formally, these actions are as follows (where $+_2$ denotes modulo 2 addition):

$$(j \neq 0) \;\wedge\; (x.j \neq x.(j-1)) \quad \longrightarrow \quad x.j := x.(j-1)$$

$$(j = 0) \;\wedge\; (x.j \neq (x.N +_2 1)) \quad \longrightarrow \quad x.j := x.N +_2 1$$

Fault action. Faults can restart at most $N-1$ processes. Thus, the fault action is as follows:

$$\exists i, j \mid (i \neq j) \;:\; (x_i \neq \bot) \wedge (x_j \neq \bot) \quad \longrightarrow \quad x.k := \bot$$

Safety specification. The safety specification requires that a process whose state is uncorrupted should not copy the value of a corrupted process. Formally, the safety specification is the following set of bad transitions:

$$SPEC = \quad \bigvee_{j=0}^{N} (x.j \neq \bot \;\wedge\; x'.j = \bot)$$

Note that in token ring (unlike Byzantine agreement), we require that the safety specification can only be violated by execution of program actions. In other words, when a fault action restarts a process, safety is not violated.

Fault-tolerant program. The output of our implementation is a program that tolerates the above fault actions. Intuitively, a process in the synthesized program is allowed to copy the value of its predecessor, if this value in not corrupted. Note that the actions of the synthesized program stipulate recovery actions that start from outside program invariant as well. The actions of the synthesized program are as follows:

$$(j \neq 0) \;\wedge\; (x.j \neq x.(j-1)) \;\wedge\; (x.(j-1) \neq \bot) \quad \longrightarrow \quad x.j := x.(j-1)$$

$$(j = 0) \;\wedge\; (x.j \neq (x.N +_2 1)) \;\wedge\; (x.N \neq \bot) \quad \longrightarrow \quad x.j := x.N +_2 1$$

Analysis of implementation results. Similar to Byzantine agreement, our analysis is based on three criteria, namely, (1) total synthesis time, (2) deadlock resolution time, and (3) memory usage, presented in Figures 4-6, respectively. Although token ring has a less complex structure than Byzantine agreement, the experimental results surprisingly show that token ring exhibits features that Byzantine agreement does not. One of these features is the structure of its fault-span in the sense that unlike Byzantine agreement, the fault-span of token ring is almost equal to its entire state space.

- (*Total synthesis time*) Similar to Byzantine agreement, in our experiments with token ring, the number of processes ranges over 3 to 25. As can be seen in Figure 4, in case of token ring the graph has sharper slope as compared to Byzantine agreement. In particular, the total synthesis time for 3..20 processes in token ring is less than the total synthesis time with the same number of processes in Byzantine agreement. However, in token ring with 21..25 processes, the total synthesis time increases dramatically. We explain the reason as we proceed. Notice that the total synthesis time to the size of state space is still sublinear.

- (*Deadlock resolution*) Unlike Byzantine agreement, in synthesis of token ring, our algorithm does not encounter a diverse set of deadlock states. In fact, in case of token ring, all deadlock states can be easily resolved by adding safe recovery transitions and, thus, our synthesis algorithm does not need to eliminate any states. As can be seen in Figures 4 and 5, the amount time spent for resolving deadlock states is considerably less than the total synthesis time. In fact, in average, 92% of the total time is spent in computing the fault-span.

- (*Memory usage*) Figure 6 completes the chain of premises to conclude our explanation on the counterintuitive behavior of synthesis of token ring. As mentioned earlier, in case of token ring, the total synthesis time increases dramatically beyond 20 processes. The same pattern occurs in Figure 6 more clearly; the slope of the graph increases rapidly where we synthesize more than 20 processes. This is due to the fact that the program can reach almost the entire state space in the presence of faults. Since the algorithm recomputes the fault-span in all iterations by starting from the program invariant and using a (possibly) modified set of program transitions, the size intermediate fault-spans (during recomputation) becomes crucial in memory usage. Moreover, in case of of token ring the number of iterations to recompute the fault-span is proportional to the number of processes, whereas in Byzantine agreement, this number is independent of the number of processes. In this situation, the size of state space of token ring with more than 20 processes is large enough to increase the size of intermediate faults-spans and iterations, which in turn affects the overall performance of synthesis.

6 Conclusion and Future Work

In this paper, we demonstrated that using techniques from symbolic analysis, the state of art in synthesis could be significantly improved. In particular, we showed that symbolic synthesis approaches could assist in overcoming state space explosion encountered during synthesis. Using the symbolic analysis and heuristics for addition of fault-tolerance [1], we demonstrated that synthesis of distributed programs with a large state space (2^{102} in case of Byzantine agreement with 25 processes, 2^{50} in case of the token ring with 25 processes) can be achieved in reasonable amount of time. Moreover, this analysis also shows that the growth of the time complexity is sublinear in the state space.

Furthermore, we showed that the symbolic approach also has the potential to significantly reduce the space complexity. In particular, the state space used during synthesis of Byzantine agreement program with 25 processes was 131 KB whereas the actual state space (with explicit state space approach) of that program is 2^{102}. The reduction in space complexity is especially valuable, as it will permit synthesis in scenarios where previous synthesis algorithms have failed due to lack of memory.

As mentioned in the introduction, while this result does not demonstrate feasibility of synthesis for all large programs (the problem is not solved even for model checking which is arguably a simpler problem than synthesis). However, this work does demonstrate that large state space by itself is not an obstacle for permitting synthesis of distributed fault-tolerant programs.

Based on the analysis of our algorithm and experimental results, we identified three different bottlenecks (depending upon the structure of the program being synthesized), namely, (1) deadlock resolution, (2) computation of fault-span, and (3) checking safety of groups of transitions. In particular, we observed that in case of Byzantine agreement, in average, 94% of the total synthesis time is spent to resolve deadlocks. Also, in case of token ring, in average, 92% of the total time is pent to compute the fault-span of the program. This analysis suggests that more efficient deadlock resolution and reachability analysis algorithms are strongly needed to make synthesis of fault-tolerant distributed programs more efficient. Thus, we categorize open problems and suggest a comprehensive roadmap for further research as follows:

- In our implementation, the Procedure ForwardReachableStates is implemented simply by next-state relation. This approach is efficient for cases where the size of OBDDs are small (e.g., in Byzantine agreement). However, as soon as the size of OBDDs become larger (e.g., in token ring), next-state reachability analysis can be as bad as enumerative methods. Hence, we are planning to incorporate more recent symbolic techniques such as clustering [17] and saturation-based reachability analysis [18, 19] in our current implementation. These techniques will certainly improve computation of state predicates such as program invariant and fault-span. However, due to the dynamic nature of synthesis, since we add and remove transitions and states, in each iteration of the algorithm, we need to recompute a *new* fault-span starting from the program invariant using the modified set of program transitions. Thus, an open problem is to develop algorithms that reuse the fault-span from previous iterations and remove unreachable states.

- We are currently working on extending the current implementation by developing new symbolic methods for deadlock resolution as well as by including existing heuristics from [20]. Note that deadlock resolution (in the sense presented in Section 4) is a problem that uniquely exists in the context of program synthesis and transformation and, hence, has not been addressed by the model checking community. Moreover, if deadlock resolution and finding recovery paths for a distributed program are costly (e.g., in Byzantine agreement), we expect that synthesizing a failsafe fault-tolerant version of the program, where only satisfaction of safety specification in the presence of faults is required, can be done more efficiently.

- Observe that in case of both Byzantine agreement and token ring, all processes have similar structures. Since this symmetry often occurs in distributed programs, we expect that a significant improvement can be achieved using symmetry reduction techniques [21–23].

- Observe that in case of Byzantine agreement, the first action of the program never violates safety. This fact suggests that it is beneficial if we can somehow identify such actions and rule them out in early stages of synthesis. Also, observe that if processes of a distributed program are allowed to read and write only few number of variables (e.g., in token ring), the size of associated group predicates become relatively large. Since violation of safety can be modeled as a satisfiability problem [24], we expect that integrating our implementation with SAT and SMT (satisfiability modulo theories) solvers is beneficial. In SMT solvers (e.g., Yices [25]), in addition to Boolean variables, one can use other types such as abstract data types, integers, reals etc., in formulae that involve arithmetic and quantifiers as well. We expect that such integration improves the performance of synthesis.

- OBDD of a Boolean formula is often more space-efficient than the enumerative representation provided a good ordering of variables is chosen. In model checking, since the goal is to "verify" the correctness of a model against a property, once the OBDD of the model is constructed, there is no need to reconstruct it during verification. Hence, an appropriate initial order of variables, is sufficient during the course of verification. However, synthesis is a more dynamic procedure, as we often add and remove states and transitions to manipulate a given program such that it satisfies a desired property (e.g., fault-tolerance). In other words, since the structure of a program changes during synthesis, reordering the variables of OBDDs dynamically is expected to be beneficial. However, there is a trade-off between time spent to reorder variables to reduce the space complexity of the problem on one side, and the time spent to synthesize the program on the other side. Thus, another open problem is to determine the circumstances under which dynamic reordering is beneficial.

We expect that the mentioned further improvements will enable us to synthesize a large class of fault-tolerant distributed programs from their fault-intolerant version.

References

[1] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of Byzantine agreement. In *20th Symposium on Reliable Distributed Systems (SRDS)*, pages 130–140, 2001.

[2] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 82–93, 2000.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[5] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

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

[7] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[8] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Logics in Computer Science (LICS)*, pages 394–406, 199.

[9] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid System*, 1995.

[10] N. Wallmeier, P. Hütten, and W. Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In *Implementation and Application of Automata (CIAA)*, pages 11–22, 2003.

[11] L. de Alfaro, M. Faella, and A. Legay. An introduction to the tool Ticc. Technical Report UCSC-CRL-06-14, School of Engineering, University of California, Santa Cruz, 2006.

[12] L. de Alfaro and T. A. Henzinger. Interface automata. In *European Software Engineering Conference*, pages 109–120, 2001.

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

[14] 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.

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

[16] CUDD: Colorado University Decision Diagram Package. `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`.

[17] R. Ranjan, A. Aziz, R. Brayton, B. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. In *IEEE/ACM International Workshop on Logic Synthesis*, 1995.

[18] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state-space generation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 328–342, 2001.

[19] G. Ciardo and A. J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 146–161, 2005.

[20] FTSyn: A framework for automatic synthesis of fault-tolerance. `http://www.cse.msu.edu/~sandeep/software/`.

[21] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Computer Aided Verification (CAV)*, pages 450–462, 1993.

[22] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):105–131, August 1996.

[23] 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.

[24] A. Ebnenasir and S. S. Kulkarni. SAT-based synthesis of fault-tolerance. Fast Abstracts of the International Conference on Dependable Systems and Networks (DSN), 2004.

[25] Yices: An SMT Solver. `http://yices.csl.sri.com`.