Exploiting Symbolic Techniques in Automated Synthesis of Distributed Programs with Large State Space*

Borzoo Bonakdarpour Sandeep S. Kulkarni
Department of Computer Science and Engineering
Michigan State University
3115 Engineering Building, 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. To the best of our knowledge, this is the first illustration where programs with large state space (beyond* 2^{100}*) is handled during synthesis.*

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

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. In the context of synthesis of fault-tolerant distributed programs, the complexity of automated synthesis can be characterized in two parts. The first part deals 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 deals with questions such as *how quickly* such recovery and safety violating transitions can be identified.

In our previous work [11], 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. Observe that the solution to the first part is independent of issues such as representation of programs, faults, specifications etc. Hence, we 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 [11] allows us to deal with the problem of synthesis of distributed programs, which is known to be NP-complete [10], 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 [6]. Although symbolic techniques have been shown to be very successful in model checking [7] (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

*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.

space complexity, and thereby permit synthesis of large(r) programs. We would like to note that, just as with model checking, this work does not imply that synthesis would be feasible for all programs with large state space. However, this work does illustrate that large state space by itself is not an obstacle to permit efficient synthesis.

Related work. While there is an extensive line of research in the area of symbolic model checking (e.g., [7,8,13]), little work has been done in symbolic synthesis of programs. In [5], 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 [14] 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.

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 on the same hardware setting.

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.

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 [11] 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 make concluding remarks and present future work.

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 [10]. The formal definition of specifications is due to Alpern and Schneider [2]. Definition of faults and fault-tolerance are based on the ones given by Arora and Gouda [3] and Kulkarni [9].

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 $s : V \mapsto \{true, false\}$, 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 $\{t_0, t_1 \cdots t_m\}$ defined by $P = \bigvee_{i=0}^{m}(t_i)$. 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 $Guard(P)$ to denote the source state predicate of P (i.e., $Guard(P) = X_1$).

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. A sequence of states, $c = \langle s_0, s_1 \cdots \rangle$, 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 c is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \models P$. We distinguish between a terminating computation and a deadlocked computation. Precisely, when a computation c *terminates* in state s_l, we include the transition (s_l, s_l) in P, i.e., c can be extended to an infinite computation by stuttering at s_l. On the other hand, if there exists a state s_d such that there is no outgoing transition (or a self-loop) from s_d then s_d is a *deadlock* state. The *projection* of program P on state predicate S, denoted as $P|S$, is the program (i.e., transition predicate) $\bigvee_{(s,s') \models P}((s \models S) \wedge (s' \models S))$.

2.2. Specification

A *specification* is a set of infinite sequences of states. Following Alpern and Schneider [2], 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. Regarding liveness specification, 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 $P[]F$.

Just as we defined the computation of P, we say that a sequence of states, $\langle s_0, s_1 \cdots \rangle$, is a *computation of* P *in the presence of* F iff the following three conditions are satisfied: (1) $\forall j > 0 : (s_{j-1}, s_j) \models (P \vee F)$, (2) if $\langle s_0, s_1 \cdots \rangle$ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \models P$, and (3) $\exists n \geq 0 : (\forall j > n : (s_{j-1}, s_j) \models P)$.

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: (1) P satisfies $SPEC$ from S, and (2) there exists T such that T is an F-span of P from S, $P[]F$ maintains $SPEC$ from T, and every computation of $P[]F$ that starts from a state in T has a state in S. Notice that our definition of fault-tolerance includes *self-stabilizing* systems as well where the system is not allowed to violate the safety specification during stabilization.

2.4. Modeling Distributed Programs

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. If process j can only write the variables in the set 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. Consider the case where (s_0, s_0'), $s_0 \neq s_0'$, is included in the transitions of j. If j cannot read variable v, then j must include a corresponding transition from s_1 where s_1 and s_0 differ only in the value of v. Let this transition be (s_1, s_1'). Now, it must be the case that s_0' and s_1' are identical except for the value of v. And, value of v must be the same in 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:

$Group(j, R_j)(s_0, s_0') =$
$\bigvee_{(s_1,s_1')|=S_p\times S_p}(\bigwedge_{v\in R_j}(v(s_0)=v(s_1)\wedge v(s_0')=v(s_1'))\wedge$
$\bigwedge_{v\notin R_j}(v(s_0)=v(s_0')\ \wedge\ v(s_1)=v(s_1')))$
Likewise, one can define the group predicate corresponding to a transition predicate P as the union of group predicates of each transition in P.

3. Problem Statement

We now formally present the problem of synthesizing a fault-tolerant program by starting from its fault-intolerant version. This problem [11] requires that the synthesized fault-tolerant program is not allowed to exhibit new computations in the absence of faults. Thus, 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'. $\square$

4. The Symbolic Synthesis Algorithm

In this section, we present a symbolic pseudo-code for the set of heuristics introduced in [11] with some minor modifications. The symbolic representation of the heuristics in terms of Boolean formulae will later enable us to implement them using Ordered Binary Decision Diaragms (OBDD) [6].

Intuitively, the algorithm Symbolic_Add_FT (cf. Figure 1) 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. For details, we refer the reader to [11].

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 [12] and token ring [4]. In both case studies, we find a considerable

Algorithm Symbolic_Add_FT(P, F, $SPEC$: transition predicate,
S: state predicate, $R_1 \ldots R_n, W_1 \ldots W_n$: set of variables){
Step 1: ms :=BackwardReachableStates($Guard(SPEC \wedge F), F$);
$S_1 := S - ms$; $mt := (\langle ms\rangle' \vee SPEC) \wedge \neg ms$;
repeat
$S_2, ne := S_1, false$;
repeat
$T_1, P_2 := S_1, P_1$;
repeat
$T_2 := T_1$;
Step 2: T_1 := ForwardReachableStates(S_1, $P_1 \vee F$, ne);
$mt := mt \wedge T_1$;
Step 3: P_1 := CheckGroupSafety(P_1, $R_1 \ldots R_n$);
until $(T1 = T2)$;
$lyr := S_1$;
Step 4: **repeat**
$ds := T_1 \wedge \neg Guard(P_1)$;
$rt := ds \wedge \langle lyr\rangle' \wedge \neg mt$;
$P_1 := P_1 \vee$ CheckGroupSafety($rt, R_1 \ldots R_n$);
$lyr := ds \wedge \neg(T_1 \wedge \neg Guard(P_1))$;
until $(lyr = false)$;
P_1 := Eliminate($ds, T_1, S_1, false, P_1, F$);
until $(P_1 = P_2)$;
Step 5: P_1, S_1 := ConstructInvariant(P_1, S_1, ne);
until $(S1 = S2)$;
return S_1, P_1;}
Procedure CheckGroupSafety(P: transition predicate,
$R_1 \ldots R_n$: set of variables){
for each process $j \in 1 \ldots n$:
$Witness = Guard(Group(P \wedge mt, R_j))$;
$P_j := (\neg Witness) \wedge P_j$;
$P := \vee_{j=1}^{n} P_j$;
return P;}
Procedure ConstructInvariant(P: transition predicate,
S, ne: state predicate){
$OffendingStates := S \wedge$ BackwardReachableStates(ne, F);
repeat
$S := S \wedge \neg OffendingStates$;
$tmp := (S \vee OffendingStates) \wedge P \wedge \neg\langle S\rangle'$;
$P := P \wedge \neg Group(tmp)$;
$OffendingStates := \langle tmp\rangle''$;
until $(OffendingStates = false)$;
return S, P;}
Procedure Eliminate($ds, T, S, Visited$: state predicate,
P, F: transition predicate){
$ds := ds \wedge \neg Visited$;
if $(ds = false)$ **then return** P;
$Visited := Visited \vee ds$;
$Old := P$;
$tmp := T \wedge \neg S \wedge P \wedge \langle ds\rangle'$;
$P := P \wedge \neg Group(tmp)$;
$fs = Guard(T \wedge f \wedge \langle ds\rangle' \wedge \neg S) \wedge$
$\neg$ ForwardReachableStates($S, F, false$);
$P :=$ Eliminate($fs, T, S, Visited, P, F$);
$New := Guard(T \wedge Group(tmp) \wedge \neg Guard(P))$;
$ne := ne \vee \neg\langle Old \wedge \neg P \wedge T \wedge \langle ds\rangle'\rangle''$;
$P := P \vee (Group(tmp) \wedge New)$;
$New := New \wedge Guard(tmp)$;
$P :=$ Eliminate($New \wedge \neg S, T, S, Visited, P, F$);
return P;}

Figure 1. Symbolic algorithm for synthesizing fault-tolerant distributed programs.

improvement in both time and space complexity as compared to the corresponding implementation in 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 [1].

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 $\{(s, s') \mid s \models g$ 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 [12]. We use a canonical version of the problem modeled by Kulkarni, Arora, and Chippada [11] as follows. 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.

To represent a Byzantine process, we introduce a variable b for each process; if $b.j$ is true then j is Byzantine. Also, at most one process (from g, j, k and l) may be 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 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 that 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 process 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 consists of the following two actions for process j.

$$\begin{array}{lcl} (d.j = \perp) \ \wedge \ \neg f.j & \longrightarrow & d.j := d.g \\ (d.j \neq \perp) \ \wedge \ \neg f.j & \longrightarrow & f.j := true \end{array}$$

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:

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

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.

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

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.

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

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 occur. 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 [12].

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

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 2), (2) deadlock resolution time (cf. Figure 3), and (3) the amount of required memory (cf. Figure 4) 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 2 illustrates the time spent to synthesize fault-tolerant non-general processes versus the size of explicit state space. The

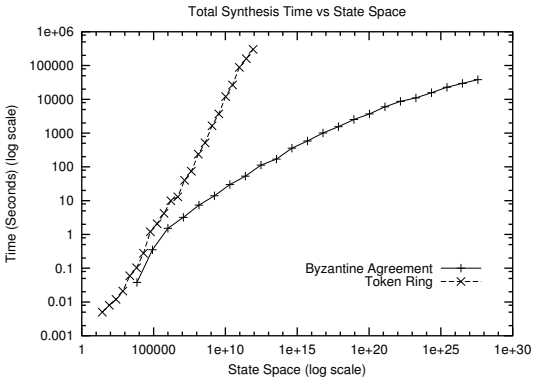

Figure 2. Total synthesis time versus the size of explicit state space in synthesis of Byzantine agreement and token ring for 3-25 processes.

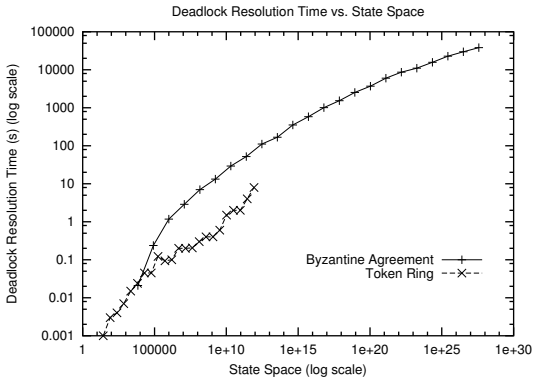

Figure 3. Deadlock resolution time versus the size of explicit state space in synthesis of Byzantine agreement and token ring for 3-25 processes.

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. First, observe that it takes 1.2 seconds to synthesize 5 non-general processes. Surprisingly, a previous enumerative implementation of the heuristics of [11] takes 15 minutes to synthesize the same number of processes on the same hardware setting. 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 25 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 25 processes is 10^{27} times larger than the size state space of Byzantine agreement with 5 processes.

Moreover, the graph in Figure 2 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 the 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 3 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 and in the Procedure Eliminate. 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 is a problem that 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 fault-intolerant program. In fact, later in this subsection, we show that unlike Byzantine agreement, in case of the token ring problem, deadlock resolution is not a crucial issue.

  Figure 3 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 4 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 process by two Boolean

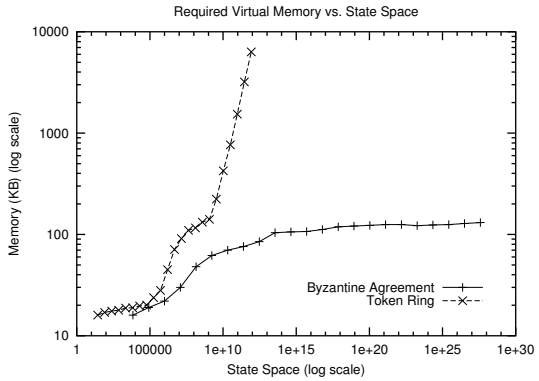

Figure 4. Required memory versus the size of explicit state space in synthesis of Byzantine agreement and token ring for 3-25 processes.

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 Boolean 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., $d.j = \bot$), or they are decided (i.e., $d.j = 0|1$) and their decision is either finalized or not yet finalized (i.e., $f.j = false|true$). Hence, each non-general can have 5 different combinations. Furthermore, the general can have either decision value (i.e., $d.g = false|true$) 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.

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 $\{0, 1, \bot\}$, where $\bot$ 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 for process j is as follows:

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

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 2-4, 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 2, 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 2 and 3, 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 4 completes the chain of premises to conclude our explanation on the counter-intuitive 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 4 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 of 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 techniques could assist in overcoming state space explosion encountered during synthesis. Using the symbolic analysis and heuristics for addition of fault-tolerance [11], 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 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 size of state space (with explicit state space approach) of that program is 2^{102}. Hence, we expect this work to demonstrate that similar to model checking, the explosion of state space by itself is not an impediment in the context of automated synthesis.

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. Backed by this analysis, we are planning to focus on resolving the above bottlenecks to improve our approach so that we can synthesize distributed programs with larger state space.

References

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

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

[3] 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.

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

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

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

[7] 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.

[8] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

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

[10] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, volume 1926 of *Lecture Notes in Computer Science*, pages 82–93, Pune, India, 2000. Springer.

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

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

[13] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[14] 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.