Noname manuscript No.
(will be inserted by the editor)

Symbolic Synthesis of Masking Fault-Tolerant Distributed Programs

Borzoo Bonakdarpour · Sandeep S. Kulkarni · Fuad Abujarad

Abstract We focus on automated addition of masking fault-tolerance to existing fault-intolerant distributed programs. Intuitively, a program is *masking* fault-tolerant, if it satisfies its safety and liveness specifications in the absence and presence of faults. Masking fault-tolerance is highly desirable in distributed programs, as the structure of such programs are fairly complex and they are often subject to various types of faults. However, the problem of synthesizing masking fault-tolerant distributed programs from their fault-intolerant version is NP-complete in the size of the program's state space, setting the practicality of the synthesis problem in doubt. In this paper, we show that in spite of the high worst-case complexity, synthesizing moderate-sized masking distributed programs is feasible in practice. In particular, we present and implement a BDD-based synthesis heuristic for adding masking fault-tolerance to existing fault-intolerant distributed programs automatically. Our experiments validate the efficiency and effectiveness of our algorithm in the sense that synthesis is possible in reasonable amount of time and memory. We also identify several bottlenecks in synthesis of distributed programs depending upon the structure of the program at hand. We conclude that unlike verification, in program synthesis, the most challenging barrier is not the state explosion problem by itself, but the time complexity of the decision procedures.

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

1 Introduction

Asserting correctness in a program is the most important aspect and application of formal methods. Two approaches to achieve correctness automatically in system design are:

– correct-by-verification, and
– correct-by-construction.

Automated verification (and in particular model checking) is arguably one of the most successful contributions of formal methods in hardware and software development in the past three decades. However, if verification of a program against a mathematical model (e.g., a set of properties) identifies an error in the system, one has to fix the error manually. Such manual revision inevitably requires another step of verification in order to ensure that the error is indeed resolved and that, no new errors are introduced to the program at hand. Thus, accomplishing correctness through verification involves a cycle of design, verification, and subsequently manual revision, if the verification step does not succeed. This iterative procedure of verification and manual revision of programs often requires a vast amount of resources.

A preliminary version of this paper appeared in International Conference on Distributed Computing Systems (ICDCS'07). This work is partially sponsored by Canada NSERC DG 357121-2008, ORF RE03-045, ORE RE-04-036, and ISOP IS09-06-037 grants, and, by USA AFOSR FA9550-10-1-0178 and NSF CNS 0914913 grants.

Borzoo Bonakdarpour
Department of Electrical and Computer Engineering
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada, N2L 3G1
E-mail: borzoo@ece.uwaterloo.ca

Sandeep S. Kulkarni and Fuad Abujarad
3115 Engineering Building
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
E-mail: {sandeep, abujarad}@cse.msu.edu

In other words, achieving correctness by verification is an *after-the-fact* task, which may potentially be costly.

Another common scenario in computing systems is where requirements of a program evolve during the program life cycle. Evolution of requirements is largely due to two factors known as *incomplete specification* and *change of environment*. While the former is usually a consequence of poor requirements engineering, the latter is a maintenance issue. This notion of maintenance turns out to be critical for systems where programs are integrated with large collections of sensors and actuators in hostile physical environments. Deeply embedded systems [1,2] are examples of cases where change of environment occurs fairly frequently. In such systems, it is essential that programs react to physical events such as faults, delays, signals, attacks, etc., so that the system specification is not violated. Since it is impossible to anticipate all possible such physical events at design time, it is highly desirable to have automated techniques that revise programs with respect to newly identified physical events according to the system specification. In fact, it often is impractical to redesign and redeploy systems that are tightly coupled with physical processes from scratch due to changes in specification or determining unanticipated new physical events.

The above scenarios clearly motivate the need for automated methods that revise programs so that the output program preserves existing properties of the input program in addition to satisfying a set of new properties. Using such revision, there is no need to re-verify the correctness of the revised program, as it is correct-by-construction. Taking the paradigm of correct-by-construction to extreme leads us to automated *synthesis from specification*, where a program is constructed from a set of properties from scratch. Alternatively, in *program revision*, an algorithm transforms an input program into an output program that meets additional properties.

In this paper, we focus on automated synthesis of distributed fault-tolerant programs. In particular, we study the problem of revising an existing distributed fault-intolerant program by solely *adding* fault-tolerance to the program. One problem with such an automated synthesis, however, is that both time and space complexity of such algorithms are often high, making them difficult to apply them for large problems.

The time complexity of automated addition of fault-tolerance can be characterized in two parts. The first part has to deal with questions such as *which* recovery transitions/actions should be added to guarantee liveness, 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 [3, 4], 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, suppose a process, say p, in a program executes a transition. In a distributed setting, this transition may correspond to several possible transitions depending upon the state of other processes. This correspondence forms an equivalence class called a group transition predicate. Thus, when a transition is included, it would be necessary to ensure that the corresponding group predicate can be executed (e.g., inclusion of the corresponding equivalence class, given the state of other processes, 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 distributed processes. In fact, the decision problem of manipulating distributed programs to satisfy some safety or liveness properties is NP-complete in both absence and presence of faults [5, 6]. To cope with this complexity, in our previous work, we developed heuristics for identifying recovery/safety-violating transitions (i.e., a witness solution to the decision problem) by considering the interdependence between different transitions of distributed processes [3].

Observe that the solution to the "what" part of the problem is independent of the "how" issue such as representation of programs, faults, specifications, etc. Hence, in the previous work, we utilized explicit-state (enumerative) techniques to design our 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 the state explosion problem and prevent one from synthesizing programs with large state space. In other words, although the polynomial time complexity of the heuristics in [3] allows us to deal with the problem of synthesis of distributed programs, which is known to be NP-complete [5,6], their explicit-state implementation is problematic with scaling up for larger 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 modelled using Boolean formulae and represented by Bryant's Ordered Binary Decision Diagrams [7]. Although symbolic techniques have been shown to be successful in model checking [8], they have not been extensively 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 permits synthesizing programs beyond small toy examples.

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

1. We introduce a symbolic algorithm that adds masking fault-tolerance to distributed programs. The core of the algorithm involves BDD-based state exploration in order to (1) removing unsafe transitions, (2) eliminating deadlock states, (3) computing recovery transitions, and (4) generating a revised invariant predicate.
2. We illustrate that our symbolic technique can significantly improve the performance of synthesis in terms of both time and space complexity with several orders of magnitude. 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 with five non-general processes, the time for explicit-state synthesis was 15 minutes whereas the time with symbolic synthesis is less than one second.
3. We rigorously analyze the cost incurred in different tasks during synthesis through a variety of case studies. These case studies are adapted from problems in the literature of distributed computing (e.g., [9–11]) and real-world examples (e.g., [12]). Our analysis identifies four bottlenecks that need to be overcome, namely (1) deadlock resolution, (2) computation of recovery paths, (3) computation of reachable states in the presence of faults, and (4) cycle resolution. We show that depending upon the structure of distributed programs, a combination of these bottlenecks may affect the performance of automated synthesis. We also demonstrate how a simple user input can significantly assist in improving the performance of program synthesis. We also show that the time and space needed to complete synthesis is competitive with the corresponding verification problem.

We 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 as large as the case studies in this paper. However, the results in this paper do illustrate that a large state space by itself is not an obstacle to permit efficient synthesis.

Organization of the paper. In Section 2, we present the formal definition of distributed programs and specifications. Then, in Section 3, we describe the notion of faults and formalize the concept of fault-tolerance. Section 4 is dedicated to the formal statement of the synthesis problem. We introduce our symbolic synthesis algorithm for adding fault-tolerance to distributed programs in Section 5. Experimental results and analysis of different aspects of our synthesis algorithm using a wide variety of case studies are presented in Sections 6-10. Related work is presented in Section 11. Finally, in Section 12, we outline a roadmap for further research and present concluding remarks. For reader's convenience, Appendix A provides a summary of notions. Appendix B illustrates a sample synthesized program using our tool SYCRAFT [13].

2 Distributed Programs and Specifications

Intuitively, we model a distributed program in terms of a set of processes. Each process is in turn specified by a transition system and is constrained by read/write restrictions over its set of variables. The notion of specification is adapted from the one introduced by Alpern and Schneider [14]. We use a canonical version of the *Byzantine agreement* problem [9] to demonstrate the concepts introduced in this section.

2.1 Distributed Programs

Let $V = \{v_0, v_1 \cdots v_n\}$ be a finite set of Boolean variables. 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 $v_j \in V$, $0 \leq j \leq n$, and $l(v_j)$ denotes a *literal*, which is either v_j itself or its negation $\neg v_j$. Let v be a variable and s be a state. We use $v(s)$ to denote the value of v in state s.

Since non-Boolean variables with finite domain D can be represented by $\log(|D|)$ Boolean variables, our notion of state is not restricted to Boolean variables. The set of all possible states obtained from variables in V and their respective domains is called the *state space*. Intuitively, a state predicate is any subset of the state space.

Definition 1 (state predicate) Let S be the set of states $\{s_0, s_1 \cdots s_m\}$. We specify the state predicate S by the disjunction $\bigvee_{i=0}^{m}(s_i)$, where each s_i is the conjunction of a set of literals (defined above). $\square$

Observe that although the Boolean formula defined in Definition 1 is in disjunctive normal form, one can represent a state predicate by any equivalent Boolean expression. We denote the membership of a state s in a state predicate 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*). Primed variables are meant to show the new value of variables prescribed by a transition. Thus, we define a transition (s, s') by the conjunction $s \wedge s'$ where $s' = \bigwedge_{j=0}^{n} l(v'_j)$ such that $v'_j \in V'$, $0 \leq j \leq n$.

Definition 2 (transition predicate) A *transition predicate* P is a finite set of transitions $\{(s_0, s'_0), (s_1, s'_1) \cdots (s_m, s'_m)\}$ formally defined by $T = \bigvee_{i=0}^{m}(s_i \wedge s'_i)$. We denote the membership of a transition (s, s') in a transition predicate T by $(s, s') \models T$. $\square$

Notation. Let S (respectively, s) be a state predicate (respectively, a state). We use $\langle S \rangle'$ (respectively, $\langle s \rangle'$) to denote the state predicate (repectively, state) obtained by replacing all variables that participate in S (respectively, s) by their corresponding primed variables. Also, let T be a transition predicate. We use $\mathit{Guard}(T)$ to denote the source state predicate of T (i.e., $s \models \mathit{Guard}(T)$ iff $\exists s' : (s, s') \models T$). $\square$

Definition 3 (process) A process p is specified by the tuple $\langle V_p, T_p, R_p, W_p \rangle$ where V_p is a set of variables, T_p is a transition predicate in the state space of p (denoted $\mathcal{S}_p$), R_p is a set of variables that p is allowed to read, and W_p is a set of variables that p is allowed to write such that $W_p \subseteq R_p \subseteq V_p$ (i.e., we assume that p cannot blindly write a variable). $\square$

We now present the issue of distribution. Informally, we model distributed processes by their ability in reading and writing variables defined next.

Write restrictions. Let $p = \langle V_p, T_p, R_p, W_p \rangle$ be a process. T_p must be disjoint from the following transition predicate due to the inability of p to change the value of variables that p is not allowed to write:

$$NW_p \ = \ \bigvee_{(s,s')} \bigvee_{v \notin W_p} (v(s) \neq v(s')).$$

Read restrictions. Let $p = \langle V_p, T_p, R_p, W_p \rangle$ be a process, v be a variable in V_p, and $(s_0, s'_0) \models T_p$ where $s_0 \neq s'_0$. If v is not in R_p, then p must include a corresponding transition from all states s_1 where s_1 and s_0 differ only in the value of v. Let (s_1, s'_1) be one such transition. Now, it must be the case that s'_0 and s'_1 are identical except for the value of v, and, this value must be the same in s_1 and s'_1. For instance, let $V_p = \{a, b\}$ and $R_p = \{a\}$. Thus, since p is not allowed to read b, 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 as far as p is concerned. Thus, each transition (s_0, s'_0) in T_p is associated with the following *group predicate*:

$$\begin{aligned} \mathit{Group}_p(s_0, s'_0) = \bigvee_{(s_1, s'_1) \models T_p} & \\ & (\textstyle\bigwedge_{v \notin R_p}(v(s_0) = v(s'_0) \ \wedge \ \ v(s_1) = v(s'_1)) \ \wedge \\ & \textstyle\bigwedge_{v \in R_p}(v(s_0) = v(s_1) \ \wedge \ \ v(s'_0) = v(s'_1))) \end{aligned}$$

Definition 4 (distributed program) A *distributed program* (or simply *program*) $\mathcal{P}$ is specified by a set $\Pi_{\mathcal{P}}$ of processes. $\square$

For simplicity and without loss of generality, we assume that the state space of all processes that participate in a program are identical. More specifically, given a program $\mathcal{P} = \Pi_{\mathcal{P}}$, we have $\forall p, q \in \Pi_{\mathcal{P}} \ : \ (V_p = V_q)$. In this sense, the state space of $\mathcal{P}$ is identical to the state spaces of its processes as well.
Notation. Let $\mathcal{P} = \Pi_{\mathcal{P}}$ be a program. We use $T_{\mathcal{P}}$ to denote the transition predicate of $\mathcal{P}$ which is formally the collection of transition predicates of all processes in $\Pi_{\mathcal{P}}$, i.e., $T_{\mathcal{P}} = \bigvee_{p \in \Pi_{\mathcal{P}}}(T_p)$.

To concisely write the transitions in a process, we use *guarded commands* (also called *actions*). A guarded command is of the form:

$$L \ :: \ G \ \ \longrightarrow \ \ ST;$$

where L is a label, G is a state predicate (called the *guard*), and ST is a *statement* that describes how the program state is updated. Thus, an action $G \longrightarrow ST$ denotes the following transition predicate:

$$\{(s, s') \mid (s \models G) \text{ and } s' \text{ is obtained by changing } s \text{ as prescribed by } ST\}.$$

Example (Byzantine agreement). The Byzantine agreement program (denoted $\mathcal{BA}$) consists of a *general*, say g, and three (or more) *non-general* processes: j, k, and l. Since the general process only provides a decision, it is modelled implicitly by two variables. Thus, $\Pi_{\mathcal{BA}} = \{j, k, l\}$. Each process of $\mathcal{BA}$ maintains a decision variable 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 the value $\perp$ denotes that the corresponding process has not yet received the decision from the general. Each non-general process also maintains a Boolean variable f that denotes whether or not that process has finalized its decision. For each process, a Boolean variable b shows whether or not the process is Byzantine. Thus, the state space of each process is obtained by the variables in the following set:

$$\begin{aligned} V_{\mathcal{BA}} = \; & \{d.g, d.j, d.k, d.l\} \; \cup && \text{(decision variables)} \\ & \{f.j, f.k, f.l\} \; \cup && \text{(finalized?)} \\ & \{b.g, b.j, b.k, b.l\}. && \text{(Byzantine?)} \end{aligned}$$

The set of variables that a non-general process, say j, is allowed to read and write are respectively:

$R_j = \{b.j, d.j, f.j, d.k, d.l, d.g\}$, and
$W_j = \{d.j, f.j\}$.

The read/write restrictions of processes k and l can be symmetrically instantiated.

The *fault-intolerant* version of $\mathcal{BA}$ works as follows. Each non-general process copies the decision from the general and then finalizes (outputs) that decision, provided it is non-Byzantine. Thus, the transition predicate of a non-general process, say j, is specified by the following two actions:

$$\begin{array}{lcl} \mathcal{BA}1_j :: (d.j = \bot) \wedge (f.j = \mathit{false}) & \longrightarrow & d.j := d.g; \\ \mathcal{BA}2_j :: (d.j \neq \bot) \wedge (f.j = \mathit{false}) & \longrightarrow & f.j := \mathit{true}; \end{array}$$

Definition 5 (computation) Let $\mathcal{P}$ be a program. A *computation* of $\mathcal{P}$ is a finite or infinite sequence of states of the form:

$$\overline{s} = s_0 \to s_1 \to \cdots$$

iff $\forall i \geq 0 : (s_i, \langle s_{i+1} \rangle') \models T_{\mathcal{P}}$.

We distinguish between a terminating computation and a *deadlocked* computation. Precisely, when a computation $\overline{s}$ *terminates* in state s_l, we include the transition (s_l, s_l') in $T_{\mathcal{P}}$. Thus, $\overline{s}$ can be extended to an infinite computation by stuttering at s_l. On the contrary, if there exists a state s_d such that there is no outgoing transition (or a self-loop) that originates from s_d, then s_d is a *deadlock* state.

Definition 6 (deadlock state) We say that a state s_0 in program $\mathcal{P}$ is a *deadlock state* iff for all states s_1 in the state space of $\mathcal{P}$, $(s_0, \langle s_1 \rangle') \not\models T_{\mathcal{P}}$. $\square$

2.2 Specification and Invariant

In this section, we formally present the concept of specifications and define what it means for a program to satisfy a specification.

Definition 7 (specification) A *specification* (or *property*), denoted *SPEC*, is a set of infinite computations of the form $\overline{s} = s_0 \to s_1 \to \cdots$, where s_i is a state for all $i \in \mathbb{Z}_{\geq 0}$. $\square$

In this paper, since we use specifications to reason about the correctness of a program, we assume that the state space of a specification is identical to the state space of the program under consideration. In order to reason about the correctness of programs, we consider invariance properties defined later in this subsection. One key feature of an invariance property is its *closure* in execution of the respective programs.

Definition 8 (closure) Let T be a transition predicate and S be a state predicate. We say that a state predicate S is *closed* in T iff $\bigwedge_{(s,s')\models T}((s \models S) \Rightarrow (s' \models \langle S \rangle'))$ holds. $\square$

Following Definition 8, we say that a state predicate S is closed in program $\mathcal{P}$ iff S is closed in $T_{\mathcal{P}}$. We are now ready to formally define what it means for a program $\mathcal{P}$ to satisfy a specification *SPEC*.

Definition 9 (satisfies) Let $\mathcal{P}$ be a program, S be a state predicate, and *SPEC* be a specification. We say that $\mathcal{P}$ *satisfies SPEC from S* and write $\mathcal{P} \models_S \mathit{SPEC}$ iff

1. S is closed in $\mathcal{P}$, and
2. for all computations $\overline{s} = s_0 \to s_1 \to \cdots$ of $\mathcal{P}$ where $s_0 \models S$, $\overline{s}$ is in *SPEC*. $\square$

Definition 10 (invariant) Let $\mathcal{P}$ be a program, *SPEC* be a specification, and I be a state predicate where $I \neq \mathit{false}$. We say that I is an *invariant predicate of $\mathcal{P}$ for SPEC* iff $\mathcal{P}$ satisfies *SPEC* from I; i.e., $\mathcal{P} \models_I \mathit{SPEC}$. $\square$

In this paper, since an invariant predicate is an essential constituent in establishing correctness about programs, a program is always accompanied by its invariant predicate[1]. In fact, from this point, we denote a program $\mathcal{P}$ by the tuple $\langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$, where $\Pi_{\mathcal{P}}$ is a set of processes and $I_{\mathcal{P}}$ is an invariant predicate.

Assumption. Given a program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$, if the invariant predicate $I_{\mathcal{P}}$ has a state s_0, such that there does not exist state s_1, where $(s_0, s_1) \models T_{\mathcal{P}}$, then we assume that state s_0 is a terminating state. In other words, a computation that reaches s_0 stutters at s_0 using the self-loop (s_0, s_0).

Observe that the notion of *satisfies* characterizes the property of infinite sequences with respect to a program. In order to characterize finite sequences, we introduce the notion of *maintains*.

Definition 11 (maintains) Let $\mathcal{P}$ be a program, *SPEC* be a specification, and S be a state predicate. We say that program $\mathcal{P}$ *maintains SPEC* from S iff

[1] Observe that Definition 9 is similar to the standard definition of satisfaction in temporal logic. However, Definition 9 is concerned with satisfaction from a predicate S to distinguish (1) correct behavior of programs in the absence of faults, (2) in the presence of fault (i.e., when $\neg S$ holds), and (3) reasoning about recovery. This issue will be explained in detail in Section 3.

1. S is closed in $\mathcal{P}$, and
2. for all computation prefixes α of $\mathcal{P}$ that start from S, there exists an infinite sequence of states β such that $\alpha\beta$ is in *SPEC*. $\Box$

Definition 12 (violates) Let $\mathcal{P}$ be a program, *SPEC* be a specification, and S be a state predicate. We say that program $\mathcal{P}$ *violates SPEC* from S iff it is not the case that $\mathcal{P}$ *maintains SPEC* from S. $\Box$

We let the specification consist of a *safety specification* and a *liveness specification*. Following Alpern and Schneider [14], the safety specification can be characterized by a set of *bad prefixes* that should not occur in any computation. Throughout this paper, we let the length of such bad prefixes be two. In other words, we characterize the safety specification by a set of *bad transitions* that should not occur in any program computation. We denote this transition predicate by $SPEC_{bt}$. Thus, the safety specification can be formally defined by the set $SPEC_{\overline{bt}}$ of infinite sequences, such that no infinite sequence contains a transition in $SPEC_{bt}$.

A liveness specification of *SPEC* is a set of infinite sequences of states that meets the following condition: for each finite sequence of states α, there exists a suffix β such that $\alpha\beta \in SPEC$. In our synthesis problem (cf. Section 4), we begin with an initial program that satisfies its specification (including the liveness specification). Moreover, we require our synthesis algorithm to preserve liveness. In other words, if the input program satisfies a liveness specification, then the transformed program satisfies that liveness specification as well. Hence, the liveness specification need not be specified explicitly.

Example (cont'd). The safety specification of $\mathcal{BA}$ requires *validity*, *agreement*, and *persistency*:

- *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.
- *Agreement* means that the final decision of any two non-Byzantine processes must be equal.
- *Persistency* requires that once a non-Byzantine process finalizes (outputs) its decision, it cannot change it.

Thus, the following transition predicate characterizes the above requirements as the safety specification of $\mathcal{BA}$:

$$
\begin{aligned}
SPEC_{bt_{\mathcal{BA}}} \;=\; & \\
& (\exists p \in \{j,k,l\} \;::\; \neg b'.g \;\wedge\; \neg b'.p \;\wedge \\
& \qquad (d'.p \neq \bot) \;\wedge\; f'.p \;\wedge\; (d'.p \neq d'.g)) \;\vee \\
& (\exists p,q \in \{j,k,l\} \;::\; \neg b'.p \;\wedge\; \neg b'.q \;\wedge\; f'.p \;\wedge\; f'.q \;\wedge \\
& \qquad (d'.p \neq \bot) \;\wedge\; (d'.q \neq \bot) \;\wedge\; (d'.p \neq d'.q)) \;\vee \\
& (\exists p \in \{j,k,l\} \;::\; \neg b.p \;\wedge\; \neg b'.p \;\wedge\; f.p \;\wedge \\
& \qquad ((d.p \neq d'.p) \;\vee\; (f.p \neq f'.p))).
\end{aligned}
$$

In this context, an example of a liveness specification can be "all non-general non-Byzantine processes eventually reach the same decision".

One possible invariant predicate of $\mathcal{BA}$ consists of the following sets of states:

1. First, we consider the set of states where the general is non-Byzantine. In this case:
   - one of the non-general processes may be Byzantine,
   - if a non-general process, say j, is non-Byzantine, it is necessary that $d.j$ be initialized to either $\bot$ or $d.g$, and
   - an undecided non-Byzantine process does not finalize its decision.
2. We also consider the set of states where the general is Byzantine. In this case, g can change its decision arbitrarily. It follows that other processes are non-Byzantine and $d.j, d.k$ and $d.l$ are initialized to the same value that is different from $\bot$.

Thus, the invariant predicate is as follows:

$$
\begin{aligned}
& I_{\mathcal{BA}} = \\
& \quad \neg b.g \;\wedge\; (\forall p,q \in \{j,k,l\} \;::\; (\neg b.p \;\vee\; \neg b.q)) \quad \wedge \\
& \quad (\forall p \in \{j,k,l\} \;::\; \neg b.p \Rightarrow (d.p = \bot \vee d.p = d.g)) \;\wedge \\
& \quad (\forall p \in \{j,k,l\} \;::\; (\neg b.p \wedge f.p) \Rightarrow (d.p \neq \bot)) \\
& \quad \vee \\
& \quad (b.g \;\wedge\; \neg b.j \;\wedge\; \neg b.k \;\wedge\; \neg b.l \;\wedge \\
& \qquad\qquad (d.j = d.k = d.l \;\wedge\; d.j \neq \bot))
\end{aligned}
$$

An alert reader can easily verify that $\mathcal{BA}$ satisfies $SPEC_{\overline{bt}_{\mathcal{BA}}}$ from $I_{\mathcal{BA}}$.

3 Fault Model and Fault-Tolerance

Following Arora and Gouda [15], the faults that a program is subject to are systematically represented by a transition predicate. Precisely, a class of *faults* F for program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$ is a transition predicate in the state space of $\mathcal{P}$, i.e., $F \subseteq \mathcal{S}_{\mathcal{P}} \times \mathcal{S}_{\mathcal{P}}$. We use $\mathcal{P}[]F$ to denote the program $\mathcal{P}$ in the presence of faults F. Hence, transitions of program $\mathcal{P}$ in the presence of F is obtained by taking the disjunction of the transitions in $T_{\mathcal{P}}$ and the transitions in F, i.e., $T_{\mathcal{P}} \vee F$.

We emphasize that such representation is possible notwithstanding the type of the faults (be they stuck-at, crash, fail-stop, timing, performance, Byzantine, message loss, etc.), the nature of the faults (be they permanent, transient, or intermittent), or the ability of the program to observe the effects of the faults (be they detectable or undetectable). We also note that since a program does not have control over the occurrence of faults, they are *not* limited to processes' restrictions. That is, fault transitions are independent of read/write restrictions and program specification.

Example (cont'd). The fault transitions that affect a process, say j, of $\mathcal{BA}$ are as follows: (We include similar actions for k, l, and g)

$$F_0 \ :: \ \neg b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l \quad \longrightarrow \quad b.j := \mathit{true};$$
$$F_1 \ :: \ b.j \quad \longrightarrow \quad d.j,\ \ f.j \ := \ 0|1, \ \ \mathit{false}|\mathit{true};$$

where $d.j \ := \ 0|1$ means that $d.j$ could be assigned either 0 or 1. In case of the general process, the second action does not change the value of any f-variable.

Just as we introduced the notion of invariant for reasoning about the correctness of programs in the absence of faults, we introduce the notion of *fault-span* to reason about program correctness in the presence of faults.

Definition 13 **(fault-span)** Let $\mathcal{P} = \langle \Pi_\mathcal{P}, I_\mathcal{P} \rangle$ be a program and F be a set of faults. We say that a state predicate $S_\mathcal{P}$ is an F-span (read as *fault-span*) of $\mathcal{P}$ from $I_\mathcal{P}$ iff the following conditions are satisfied:

1. $I_\mathcal{P} \ \Rightarrow \ S_\mathcal{P}$, and
2. $S_\mathcal{P}$ is closed in $\mathcal{P}[]F$. $\Box$

Observe that for all computations of $\mathcal{P}$ that start from states where $I_\mathcal{P}$ is true, $S_\mathcal{P}$ is a boundary in the state space of $\mathcal{P}$ up to which (but not beyond which) the state of $\mathcal{P}$ may be perturbed by the occurrence of the transitions in F. Subsequently, as we defined the computations of $\mathcal{P}$, one can define computations of program $\mathcal{P}$ in the presence of faults F as follows. We say that a sequence of states, $\overline{s} = s_0 \rightarrow s_1 \rightarrow \cdots$, is a computation of $\mathcal{P}[]F$ iff the following three conditions are satisfied:

1. $\forall j \geq 0 \ :: \ (s_j, s_{j+1}) \ \models \ (T_\mathcal{P} \vee F)$,
2. if $\overline{s}$ is finite and terminates in state s_l, then there does not exist state s such that $(s_l, \langle s \rangle') \models T_\mathcal{P}$, and
3. $\exists n \geq 0 \ :: \ (\forall j \geq n \ :: \ (s_j, s_{j+1}) \models T_\mathcal{P})$.

Informally, the third condition requires that the number of occurrence of faults in a computation has to be finite. This constraint is necessary to ensure *recovery* in the presence of faults.

We are now ready to define what it means for a program to be *masking fault-tolerant*. Intuitively, a program is masking fault-tolerant if it satisfies its safety and liveness specifications in both absence and presence of faults. In other words, the program masks the occurrence of faults.

Definition 14 **(masking fault-tolerance)** Let $\mathcal{P} = \langle \Pi_\mathcal{P}, I_\mathcal{P} \rangle$ be a program with specification SPEC and F be a set of faults. We say that $\mathcal{P}$ is *masking F-tolerant to SPEC from $I_\mathcal{P}$* iff the following conditions hold:

1. $\mathcal{P} \ \models_{I_\mathcal{P}} \ \mathit{SPEC}$, and
2. there exists a state predicate S such that:
   (a) S is an F-span of $\mathcal{P}$ from $I_\mathcal{P}$,
   (b) $\mathcal{P}[]F$ maintains SPEC from S, and
   (c) every computation of $\mathcal{P}[]F$ that starts from a state in S eventually reaches a state s, such that $s \models I_\mathcal{P}$. $\Box$

Notice that condition 2-c explicitly expresses the notion of *recovery* to the invariant predicate as a set of *legitimate states*. It also implicitly implies that the program reaches a state from where it can satisfy its liveness specification.

4 The Synthesis Problem

Given are a fault-intolerant program $\mathcal{P} = \langle \Pi_\mathcal{P}, I_\mathcal{P} \rangle$, a set F of faults, and a specification SPEC, such that $\mathcal{P} \models_{I_\mathcal{P}} \mathit{SPEC}$. Our goal is to synthesize a program $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, I_{\mathcal{P}'} \rangle$ such that $\mathcal{P}'$ is F-tolerant to SPEC from $I_{\mathcal{P}'}$. We require our synthesis method to obtain $\mathcal{P}'$ from $\mathcal{P}$ by *adding fault-tolerance* to $\mathcal{P}$ without introducing new behaviors in the absence of faults. To this end, we first define the notion of *projection*. Informally, projection of a transition predicate T on a state predicate S consists of transitions of T that start in S and end in S.

Definition 15 **(projection)** Let T be a transition predicate and S be a state predicate. The *projection* of T on S (denoted $T|S$) is the following transition predicate:

$$T \mid S \ = \ \bigvee\nolimits_{(s,s') \models T} (s \models S \ \ \wedge \ \ s' \models \langle S \rangle'). \quad \Box$$

Now, observe that in the absence of faults:

1. If $I_{\mathcal{P}'}$ contains states that are not in $I_\mathcal{P}$ then, in the absence of faults, $\mathcal{P}'$ may include computations that start outside $I_\mathcal{P}$. Since we require that $\mathcal{P}'$ satisfies SPEC from $I_{\mathcal{P}'}$, it implies that $\mathcal{P}'$ is using a new way to satisfy SPEC in the absence of faults. Thus, we require that $I_{\mathcal{P}'} \Rightarrow I_\mathcal{P}$.
2. If $T_{\mathcal{P}'} \mid I_{\mathcal{P}'}$ contains a transition that is not in $T_\mathcal{P} \mid I_{\mathcal{P}'}$ then $\mathcal{P}'$ can use this transition in order to satisfy SPEC in the absence of faults. Thus, we require that $(T_{\mathcal{P}'} \mid I_{\mathcal{P}'}) \Rightarrow (T_\mathcal{P} \mid I_{\mathcal{P}'})$.

Following the above observations, the synthesis problem is as follows:

Problem Statement. Given a program $\mathcal{P} = \langle \Pi_\mathcal{P}, I_\mathcal{P} \rangle$, specification SPEC, and a set F of faults such that $\mathcal{P} \models_{I_\mathcal{P}} \mathit{SPEC}$, identify $\mathcal{P}' = \langle \Pi_{\mathcal{P}'}, I_{\mathcal{P}'} \rangle$ such that:

$(C1)$ $I_{\mathcal{P}'} \Rightarrow I_{\mathcal{P}}$,
$(C2)$ $(T_{\mathcal{P}'} \mid I_{\mathcal{P}'}) \Rightarrow (T_{\mathcal{P}} \mid I_{\mathcal{P}'})$, and
$(C3)$ $\mathcal{P}'$ is masking F-tolerant to *SPEC* from $I_{\mathcal{P}'}$. □

We recall that the stated problem, if posed as a decision problem is known to be NP-complete in the size of state space [5,6] and our focus in this paper is to cope with this complexity for synthesizing witness solutions. Notice that constraint $C3$ obviously requires that the synthesized program has to be masking fault-tolerant. Based on Definition 14, this requirement implicitly implies that the synthesized program is not allowed to exhibit *new* finite computations, which in turn means that synthesis algorithms are not allowed to introduce deadlock states to the input program. This condition ensures that if the given intolerant program satisfies a universally quantified liveness property, then the synthesized fault-tolerant program satisfies the property as well.

Our formulation of the synthesis problem is in spirit close to both controller synthesis, where program and fault transitions may be modelled as controllable and uncontrollable actions, and game theory, where program and fault transitions may be modelled in terms of two players. In both problems, the objective is to *restrict* the program actions at each state through synthesizing a controller or a wining strategy such that the behavior of the entire system satisfies some safety or reachability conditions in the presence of an adversary. Notice that the conditions $C1$ and $C2$ precisely express this notion of restriction. Furthermore, the conjunction of all conditions expresses the notion of *language inclusion*, where the synthesized program is supposed to exhibit a subset of behaviors of the input intolerant program in the absence of faults.

Finally, unlike controller synthesis and game theory algorithms where arbitrary specifications are often considered, our algorithms are tailored for properties typically used in specifying distributed and fault-tolerance requirements and, hence, they synthesize programs more efficiently. We elaborate on comparison and contrast with controller synthesis and game theory in Section 11 in detail.

Another point in the problem statement is that it does not allow extending the state space during synthesis. However, there are different ways to modify the problem statement and corresponding algorithms so that redundancy will be added during synthesis. One approach is as follows: If addition of fault-tolerance fails then we add one bit to each process. The original program can update this variable in any way it wishes. Now, we can run the algorithm with this added redundancy. If it is possible to add fault-tolerance then eventually this algorithm will succeed.

Although this approach looks promising, there are several concerns. One problem is that with the above approach, the algorithm may never terminate. In fact, the problem will be undecidable [16]. Even if the algorithm terminates, the complexity could be extremely high. Another, and more serious problem, is that adding such bits may not be consistent with the underlying model. For example, it is not always obvious what read/write restrictions should be imposed on these new bits. For certain models such as read/write or shared memory, one can identify these restrictions. But, in our work, we have allowed the designer to identify their own read/write restrictions to allow for more generality. Hence, adding such bits *during the algorithm* will make the algorithms less general. Yet another serious problem is how one can model the effect of faults on the new bits. For example, if one uses transient faults that perturb the program to an arbitrary state, then the bit could be corrupted arbitrarily. For fail-stop faults, presumably, the bit will not be corrupted. Again, this reduces the generality of the algorithms considered in the paper.

Additionally, if the approach for enhancing state space results in the use of more processes (to obtain additional redundancy) then that has another cost (in terms of hardware, money etc.). We believe that an automation algorithm should never add this cost on its own, as an automation algorithm cannot determine reasonableness of the increased cost.

For these reasons, we advocate the one approach where addition of such bits should be done (manually) by the designer if synthesis fails. With this approach, the state space remains unchanged during the synthesis algorithm. Another approach we advocate is to use components that can increase the state space. In [17], we have developed one such algorithm where inability to synthesize is used to identify a type of component that should be added. The added component has its own variables thereby increasing the state space. We believe that the designer can use one of the two approaches to deal with the case where sufficient redundancy is unavailable.

5 The Symbolic Synthesis Algorithm

In this section, we present our symbolic algorithm for solving the synthesis problem presented in Section 4. As mentioned in Section 4, the corresponding decision problem is known to be NP-complete [5,6] and our goal in this section is to develop an efficient symbolic (i.e., BDD-based) heuristic that can synthesize a rich class of programs. We validate this claim in Sections 6-10 through presenting different case studies. Our heuristic is inspired by the explicit-state algorithms developed in [3, 4].

Algorithm sketch. The algorithm takes an intolerant program, a safety specification, and a set of fault transitions as input and synthesizes a fault-tolerant program that satisfies the constraints of the Problem Statement in Section 4. The algorithm consists of five steps. The first step is initialization, where we identify state and transition predicates from where execution of faults alone may violate the safety specification. In Step 2, we identify the fault-span by computing the state predicate reachable by the program in the presence of faults, starting from the program invariant. In Step 3, the algorithm identifies and rules out program transitions whose execution may lead to violation of the safety specification. Then, in Step 4, we resolve deadlock states. Resolving deadlock states is crucial to ensure that no new finite computations are introduced to the input fault-intolerant program. Hence, in order to ensure that the synthesized fault-tolerant program satisfies liveness specification of the input program, deadlock states must be handled by either adding recovery paths or state elimination. Finally, in Step 5, we re-compute the invariant predicate so that it is closed in the final program. We repeat steps 2-3, 2-4, and 2-5 until a fixpoint is reached. The fixpoint computations are represented by three nested repeat-until loops in the algorithm. Thus, the algorithm terminates when no progress is possible in all the steps described above.

We now describe the algorithm Symbolic_Add_FT (cf. Algorithm 5.1) in detail:

- **Step 1: Initializations (Lines 1-3).** First, we compute the state predicate ms from where execution of faults alone violate the safety specification (Line 1). To this end, we start from state predicate where $Guard(F \wedge SPEC_{bt})$ is true (i.e., states from where faults directly violate the safety specification) and explore backward reachable states by applying fault transitions only. This is achieved by invoking the procedure BWReachStates as a black box. The first parameter of the procedure is the state predicate from where we start computing backward reachable states. The second parameter is the transition predicate applied for exploring reachable states. The procedure FWReachStates works in a similar fashion. The only difference is it explores forward reachable states. We do not present the details of how we compute the set of forward and backward reachable states, as the problem has been thoroughly studied and already implemented in symbolic model checkers [8, 18, 19]. Nonetheless, as we will illustrate in Sections 6-10, small changes in implementation may have tremendous impact on the efficiency of state exploration with respect to different programs. This impact is often more dramatic in program synthesis than verification since reachability analysis procedures may be applied multiple times during the course of synthesizing a program.
Since a program does not have control over the occurrence of faults, one has to ensure that the state predicate ms never becomes true in any program computation. Otherwise, faults alone may lead the program to a state where the safety is violated. Thus, we remove ms from the invariant of the fault-tolerant program (Line 2). We also compute the transition predicate mt whose transitions should not be executed by the fault-tolerant program. Initially, mt is equal to the union of $SPEC_{bt}$ and transitions that start from any arbitrary state and end in ms (Line 3). 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 3). Notice that in Algorithm Add_Symbolic_FT and its sub-procedures, any addition or removal of a transition predicate has to be applied along with its group predicate due to the issue of read restrictions. Thus, issues with regard to distributed processes are all handled in computing group predicates. Since ms is unreachable, we allow the program to execute transitions that originate in ms. This inclusion is along with its group predicate which results in increasing the level of nondeterminism of the synthesized program in the sense that the program is more likely to have multiple choices of execution paths at each reachable state. Such diversity often increases chances for successful synthesis. Observe that although a state predicate, in Line 3, one can interpret ms as a transition predicate that starts in ms and ends in *true*.

- **Step 2: Re-computing the fault-span (Lines 9-11).** After initializations, we re-compute the invariant predicate, program transition predicate, and fault-span of the fault-tolerant program in three nested loops, respectively. Each loop resembles a fixpoint calculation. We start with the most inner loop, where we re-compute the fault-span. The reason for re-computing the fault-span is due to the fact that in other steps of our algorithm, we add and remove transitions that originate in the fault-span. Hence, new states may become reachable and some states

Algorithm 5.1 Symbolic_Add_FT

Input: program transition predicate $T_{\mathcal{P}}$, invariant predicate $I_{\mathcal{P}}$, fault transitions F, and bad transition predicate $SPEC_{bt}$.
Output: program transition predicate $T_{\mathcal{P}'}$ and invariant predicate $I_{\mathcal{P}'}$.

// *initializations*
1: ms := BWReachStates($Guard(F \wedge SPEC_{bt}),\ F$);
2: $I_1,\ fte := I_{\mathcal{P}} - ms,\ \ false$;
3: $mt := (\langle ms \rangle' \vee SPEC_{bt}) \wedge Group(\neg ms)$;

// *recomputing the invariant predicate*
4: **repeat**
5: $\quad I_2 := I_1$;
$\quad$// *recomputing the transition predicate*
6: $\quad$**repeat**
7: $\quad\quad S_1,\ T_2 := I_1,\ T_1$;
$\quad\quad$// *recomputing the fault-span*
8: $\quad\quad$**repeat**
9: $\quad\quad\quad S_2 := S_1$;
10: $\quad\quad\quad S_1$:= FWReachStates($I_1,\ T_1 \vee F$);
11: $\quad\quad\quad S_1 := S_1 - fte$;
12: $\quad\quad\quad mt := mt \wedge S_1$;
$\quad\quad\quad$// *removing unsafe transitions*
13: $\quad\quad\quad T_1 := T_1 - Group(T_1 \wedge mt)$;
14: $\quad\quad$**until** $(S_1 = S_2)$;
$\quad\quad$// *Resolving deadlock states through adding recovery or state elimination*
15: $\quad\quad ds := S_1 \wedge \neg Guard(T_1)$;
16: $\quad\quad T_1 := T_1 \vee$ AddRecovery($ds,\ I_1,\ S_1,\ mt$);
17: $\quad\quad ds := S_1 \wedge \neg Guard(T_1)$;
18: $\quad\quad T_1,\ fte$:= Eliminate($ds,\ T_1,\ I_1,\ S_1,\ F,\ false, false$);
19: $\quad$**until** $(T_1 = T_2)$;
20: $\quad T_1,\ I_1$:= ConstructInvariant($T_1,\ I_1,\ fte$);
21: **until** $(I_1 = I_2)$;
22: $I_{\mathcal{P}'},\ T_{\mathcal{P}'} := I_1,\ T_1$;
23: **return** $I_{\mathcal{P}'},\ T_{\mathcal{P}'}$;

may become unreachable. Thus, re-computation is needed to update the fault-span.
Let the initial fault-span S_1 be equal to the invariant I_1 (Line 7). We re-compute the fault-span by starting exploration from states where the invariant I_1 is true and applying the program transition predicate in the presence of faults (i.e., $T_1 \vee F$). To this end, we invoke the procedure FWReachStates (Line 10). After recomputing the fault-span, we remove the state predicate *fte* from the new fault-span S_1 (Line 11). This state predicate is identified later in Step 4 where we resolve deadlock states. For now, *fte* contains states failed to eliminate during deadlock resolution.

– **Step 3: Identifying and removing unsafe transitions (Lines 12-13).** We first identify unsafe transitions. Suppose there exists a state predicate which is a subset of $Guard(mt)$, but it is unreachable by transitions in $T_1 \vee F$ starting from the invariant. In other words, it does not intersect with the fault-span. Since $Guard(mt)$ is unreachable by computations of the program even in the presence of faults, we let the transitions that originate in this state predicate be in the fault-tolerant program (Line 12). In other words, the necessary condition for a transition to be unsafe is its source state has to be in the fault-span. Otherwise, the transition can exist in the fault-tolerant program transition predicate, even if it is in $SPEC_{bt}$. The reason for not including such transitions in mt is due to the fact that their corresponding group predicates are also included in the synthesized program transition predicate which in turn adds to non-determinism and diversity of the program.
The fault-tolerant program transition predicate is computed based on the following principle: a transition can be included if it is not unsafe. Thus, once we identify unsafe transitions (i.e., transition predicate mt), we rule them out from the program transition predicate (Line 13). Note that a transition can be included in the fault-tolerant program if its entire corresponding group predicate can be included.

– **Step 4: Resolving deadlock states (Lines 15-18).** Since the algorithm may remove some transitions from the input program, some states may have no outgoing transitions due to this removal. Thus, when the execution of the algorithm reaches a fixpoint in the inner loop, we identify *deadlock states* inside the fault-span. We deal with deadlock states in two ways. We either add a safe *recovery* path from a deadlock state to the program invariant, or (if recovery is not possible) we *eliminate* the deadlock state. That is, we make the deadlock state unreachable. The deadlock resolution mechanisms are implemented in two procedures AddRecovery and Eliminate (cf. Procedures 5.2 and 5.3), respectively. We now describe these mechanisms in detail.
First, the algorithm Symbolic_Add_FT identifies the deadlock state predicate ds (Line 15). Intuitively, a state in the fault-span is deadlocked if the guard of T_1 is false in that state. We now proceed as follows to resolve deadlock states:

  – *(Adding safe recovery paths)* The Procedure AddRecovery (cf. Procedure 5.2) takes a deadlock state predicate ds, invariant predicate I, fault-span S, and unsafe transition predicate mt as input. It returns a transition predicate which contains new recovery transitions as output. We add recovery paths in an iteratively layered fashion. Let the first layer that recovery can reach be the program invariant, i.e., $lyr = I_1$ (Line 1). Also, let rt be the transition predicate that

Procedure 5.2 AddRecovery

Input: deadlock states ds, invariant I, fault-span S, and transition predicate mt.
Output: recovery transition predicate rec.

```
1: lyr, rec := I, false;
2: repeat
3:     rt := Group(ds ∧ ⟨lyr⟩');
4:     rt := rt − Group(rt ∧ mt);
5:     if DetectCycles(T ∨ rec ∨ rt, S) then
6:         rt := false;
7:     end if
8:     rec := rec ∨ (rt − Group(rt ∧ I ∧ ¬⟨I⟩'));
9:     lyr := Guard(ds ∧ rt);
10: until (lyr = false);
11: return rec;
```

originates from the deadlock state predicate ds and ends in lyr (Line 3). Since we require the fault-tolerant program to satisfy safety during recovery, rt must be disjoint from mt. Thus, we check whether the group predicate of rt maintains the safety specification (Line 4). If so, we check whether addition of rt to the program transition predicate creates a cycle that is entirely in the fault-span. Existence of such a cycle may prevent the program to recover to the invariant predicate within a finite number of steps. To this end, we invoke the procedure DetectCycles with parameters $T \vee rec \vee rt$ and S. If there is indeed a cycle in the fault-span created due to addition of rt to the program transition predicate, we do not add the new recovery transitions in rt to rec which is the final set of recovery transitions. Otherwise, it is safe to add rt to rec (Line 8). There exist several symbolic approaches in the literature for detecting cycles [20]. We, in particular, incorporate the approach introduced by Emerson and Lei [21].

For the next iteration, we let lyr be the state predicate from where one-step safe recovery is possible (Line 9). We continue adding recovery steps until no new recovery transition is added.

– *(Eliminating deadlock states)* If safe recovery is not possible from a deadlock state predicate, we choose to eliminate it. By state elimination we mean making that state unreachable. The recursive Procedure Eliminate takes a deadlock state predicate ds, program transition predicate T, invariant predicate I, fault-span S, fault transitions F, a predicate vds of deadlock states visited while eliminating, and a predicate fte of deadlock states failed to eliminate as input. It returns a revised program transition predicate T', visited deadlock states vds', and states fte' failed to eliminate as output.
The Procedure Eliminate works as follows. First, if all deadlock states in ds have already been considered for elimination, the procedure returns immediately (Lines 1-5). Otherwise, we add ds to the predicate vds of states already visited (Line 6). There are potentially two ways to reach a state in ds: (1) by a program transition, or (2) by a fault transition. If ds is reachable by a fault transition, we need to backtrack to the source state predicate and make that predicate unreachable. This is due to the fact that the program does not have control over the occurrence of faults and if there exist states in ds that are reachable by fault transitions from another state predicate, say fs (Line 9), then one has to backtrack and eliminate fs. Thus, we mark the states reachable by fault transitions from fs as *failed to eliminate* (Line 10)[2]. We note that elimination

[2] Let T be a transition predicate. T'' denotes the state predicate obtained from unpriming all variables in the target state predicate of T, i.e., the state predicate reachable by T.

Procedure 5.3 Eliminate

Input: deadlock states ds, transition predicate T, invariant I, fault-span S, fault transitions F, visited deadlock states vds, and predicate fte failed to eliminate.
Output: revised program transition predicate T, visited deadlock states vds, and predicate fte failed to eliminate, where states in ds become unreachable.

```
1: ds := ds − vds;
2: if (ds = false) then
3:     T', vds', fte' := T, vds, fte;
4:     return T', vds', fte';
5: end if
6: vds := vds ∨ ds;

    // eliminating states in ds
7: tmp := (S − I) ∧ T ∧ ⟨ds⟩';
8: T := T − Group(tmp);

    // eliminating source states of incoming fault transitions
9: fs := Guard(S ∧ F ∧ ⟨ds⟩');
10: fte := fte ∨ ⟨fs ∧ F⟩'';
11: OffendingStates := OffendingStates ∨ (fs ∧ I);
12: if (fs ≠ false) then
13:     T, vds, fte := Eliminate(fs − I, T, I, S, F, vds, fte);
14: end if

    // testing whether removal of incoming program transitions
    creates new deadlock states
15: nds := Guard(S ∧ Group(tmp) ∧ ¬T);
16: T := T ∨ (Group(tmp) ∧ nds);
17: fte := fte ∨ ((nds ∧ ⟨Group(tmp)⟩'') − I);
18: T', vds', fte' := Eliminate(nds, T, I, S, F, vds, fte);

19: return T', vds', fte';
```

of states in *fs* is based on the following principle: we do not eliminate states in the invariant predicate. Thus, if *fs* intersects with the invariant, we mark the intersection as *OffendingStates* and deal with them when we re-compute the invariant predicate in Step 5. At this point, we invoke Eliminate recursively with parameter $fs - I$ (Line 13).

If *ds* is reachable by program transitions, we remove those transitions and their corresponding group predicates from T, provided no new deadlock states are introduced. Thus, we temporarily remove such transitions, say *tmp*, with the hope that this removal makes *ds* unreachable (Lines 7-8). Once we deal with incoming fault transitions, we ensure that removal of $Group(tmp)$ does not introduce new deadlock states to the program. If there exist new deadlock states, the predicate *nds* (computed in Line 15) is not equal to false. Thus, we take the following steps: We

1. reinstate the transitions originating from *nds* back to the program (Line 16),
2. mark the states in *nds* as failed to eliminate (Line 17), and
3. attempt to ensure that *nds* is never reached by invoking Eliminate with parameter *nds* recursively (Line 18).

As mentioned earlier the Algorithm Symbolic_Add_FT exploits the above mechanisms to deal with deadlock states. As can be seen, first it invokes the Procedure AddRecovery on the existing deadlock states (Line 16). Then, if AddRecovery fails to resolve some deadlock states, we make them unreachable by invoking the Procedure Eliminate (Line 18). Finally, notice that in Line 11 of the algorithm, we exclude state predicate *fte* from the re-computed fault-span. This is because these states have to eventually become unreachable and if we include them in the fault-span, in the next iteration of resolving deadlock states, they will be unnecessarily re-considered for elimination.

– **Step 5: Re-computing the invariant (Line 20).** Once we reach a fixpoint after re-computing fault-span and program transition predicate, we re-compute the invariant by invoking the Procedure ConstructInvariant (see Procedure 5.4). Re-computation of invariant has to be done due to identifying offending states during state elimination. Recall that a computation, say $\overline{s} = s_0 \rightarrow s_1 \rightarrow \cdots$, that starts from an offending state may reach a state that was considered for elimination. In particular, since the first transition of $\overline{s}$ is a fault transition, s_0 which is an offending state has to become unreachable. To this end, we first remove offending states from the invariant (Line 2). Then, due to this removal, we need to ensure that the invariant predicate is closed in T. Thus, we remove the transition predicate that violates the closure of T (Lines 3 and 4). We continue these steps until a fixpoint is reached in the sense that no offending states exist and I is closed in T.

The algorithm keeps repeating steps 1-5 until the three fixpoints are reached. At the end of each fixpoint computation, we verify the correctness of conditions of the Problem Statement. Hence, when the algorithm terminates, we are guaranteed the solution is sound.

Theorem 1 *The algorithm* Symbolic_Add_FT *is sound.*

Proof In order to prove soundness, we show that the output of the algorithm meets the constraints of our Problem Statement in Section 4:

– (Constraint $C1$) The first constraint requires that the invariant of the synthesized program must be a subset of the input program's invariant. Notice that the invariant predicate can only be modified by Procedure ConstructInvariant. Moreover, ConstructInvariant may only remove states from the given invariant predicate. Hence, $I_{\mathcal{P}'} \Rightarrow I_{\mathcal{P}}$.

– (Constraint $C2$) The second constraint ensures that transitions that start and end in the invariant predicate are not added to the program. Observe that the algorithm only adds transitions to the program in Procedure AddRecovery. In this procedure, transitions that start from the invariant predicate and end outside this predicate are ruled out in Line 8 (i.e., only transitions that start outside the invariant are added). Hence, $(T_{\mathcal{P}'}|I_{\mathcal{P}'}) \Rightarrow (T_{\mathcal{P}}|I_{\mathcal{P}'})$.

– (Constraint $C3$) Following constraints $C1$ and $C2$, in the absence of faults, the synthesized program does not exhibit new behaviors. Hence, if $\mathcal{P} \models_{I_{\mathcal{P}}}$

Procedure 5.4 ConstrucInvariant

Input: invariant predicate I, program transition predicate T, and offending states *OffendingStates*.

Output: revised invariant predicate I' and program transition predicate T'.

1: **while** $(\mathit{OffendingStates} \neq \mathit{false})$ **do**
2: $\quad I := I - \mathit{OffendingStates}$;
3: $\quad \mathit{tmp} := T \wedge I \wedge \langle \neg I \rangle'$;
4: $\quad T := T - \mathit{Group}(\mathit{tmp})$;
5: $\quad \mathit{OffendingStates} := \langle \mathit{tmp} \rangle''$;
6: **end while**

7: $I',\ T' := I,\ T$;
8: **return** $I',\ T'$;

SPEC, then $\mathcal{P}' \models_{I_{\mathcal{P}'}} SPEC$. In the presence of faults:

- The algorithm ensures that the synthesized program executes no transition in $SPEC_{bt}$ by removing all transitions (i.e., mt) that lead to a state in ms (i.e., the set of states from where faults alone can violate the safety specification). Hence, $\mathcal{P}'$ maintains $SPEC_{\overline{bt}}$ from the fault-span in the presence of faults.
- Moreover, if the synthesized program reaches a state outside the invariant predicate, the algorithm ensures that $\mathcal{P}'$ does not deadlock through adding safe recovery paths by invoking Procedure AddRecovery. If adding safe recovery is not possible, then the algorithm makes deadlock states unreachable via invoking Procedure Eliminate. Thus, the synthesized program never deadlocks and eventually reaches a state in the invariant predicate from where satisfaction of liveness is guaranteed.

Hence, $\mathcal{P}'$ maintains *SPEC* from the invariant predicate in the presence of faults. □

The structure of the output program is typically formed by three types of actions as compared to the input intolerant program. These action types are the following:

1. *Unchanged actions.* These actions identically exist in both the tolerant and intolerant programs.
2. *Strengthened actions.* Transitions corresponding to these actions exist in the input program. However, the guard of the corresponding actions are stronger than their counterpart actions in the input program. This is due to the fact that the transformed program makes unsafe states and some deadlock states unreachable.
3. *Recovery actions.* These action do not exist in the input program at all. Recovery actions are added to the input program in order to resolve some deadlock states that are reachable by the program in the presence of faults.

In the next five sections, we present a set of experimental results of implementation of the Algorithm Symbolic_Add_FT and its procedures in our tool SYCRAFT [13]. Our case studies include three classic examples in the literature of distributed fault-tolerant computing, namely, the Byzantine agreement problem [9], Byzantine agreement with fail-stop faults, and the token ring [10] problem. We also present experimental results on addition of fault-tolerance to a bulk data dissemination protocol in wireless sensor networks, known as Infuse [12]. As mentioned in the introduction, our case studies address a wide variety of structural issues and obstacles that can potentially affect the efficiency of our algorithm. In all case studies, we find a considerable improvement in both time and space complexity as compared to the existing approaches.

6 Case Study 1: Byzantine Agreement

Throughout this section and Sections 7, 8, 9, and 10, all experiments are run on a dedicated PC with a 2.2GHz AMD Opteron processor and 1.2GB RAM. The BDD representation of the Boolean formulae is implemented using the Glu/CUDD package [22]. We analyze all the experiments in terms of *time* and *space* to complete synthesizing a fault-tolerant program. From the time perspective, we consider total synthesis time, time spent for resolving deadlock states (including addition of safe recovery and state elimination), cycle detection, and computing the fault-span. From the space perspective, we consider the number of states reachable by each distributed program in the presence of faults (i.e., the size of fault-span) and the actual memory usage. Our first case study is the continuation of our running example, the Byzantine agreement program.

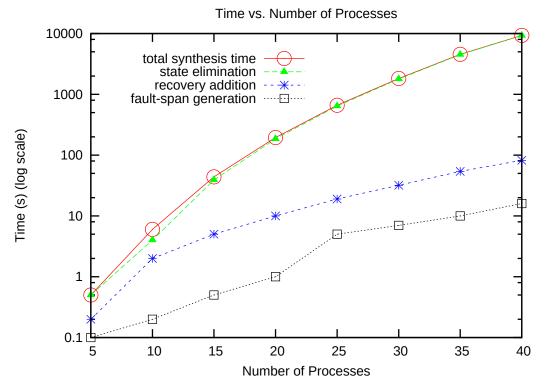
Example (cont'd). The output of our algorithm with respect to program $\mathcal{BA}$ is program $\mathcal{BA}'$ which tolerates the Byzantine faults identified in Section 3 in the sense that $\mathcal{BA}'$ never violates its specification and it does not deadlock when faults occur. We note that our synthesized program is identical to the canonical version of Byzantine agreement program manually designed in [9]. The actions of the synthesized program for a non-general process j are as follows (see Appendix B for the actual output of the tool SYCRAFT with respect to $\mathcal{BA}$):

$$\begin{array}{lll}
\mathcal{BA}'1_j & :: & d.j = \bot \ \wedge \ f.j = false \\
& & \quad \longrightarrow \quad d.j \ := \ d.g; \\
\mathcal{BA}'2_j & :: & d.j \neq \bot \ \wedge \ f.j = false \ \wedge \ (d.k = \bot \ \vee \ d.k = d.j) \ \wedge \\
& & (d.l = \bot \ \vee d.l = d.j) \ \wedge (d.k \neq \bot \vee d.l \neq \bot) \\
& & \quad \longrightarrow \quad f.j \ := \ true; \\
\mathcal{BA}'3_j & :: & d.j = 1 \ \wedge \ d.k = 0 \ \wedge \ d.l = 0 \ \wedge \ f.j = false \\
& & \quad \longrightarrow \quad d.j, \ f.j \ := \ 0, \ false|true; \\
\mathcal{BA}'4_j & :: & d.j = 0 \ \wedge \ d.k = 1 \ \wedge \ d.l = 1 \ \wedge \ f.j = false \\
& & \quad \longrightarrow \quad d.j, \ f.j \ := \ 1, \ false|true; \\
\mathcal{BA}'5_j & :: & d.j \neq \bot \ \wedge \ f.j = false \ \wedge \\
& & ((d.j = d.k \wedge d.j \neq d.l) \ \vee \ (d.j = d.l \wedge d.j \neq d.k)) \\
& & \quad \longrightarrow \quad f.j \ := \ true;
\end{array}$$

Notice that action $\mathcal{BA}'1$ is unchanged, actions $\mathcal{BA}'3$ and $\mathcal{BA}'4$ are recovery actions, and actions $\mathcal{BA}'2$ and $\mathcal{BA}'5$ are strengthened actions.

	Space		*Time(s)*			
	reachable states	**memory (MB)**	**deadlock resolution** Eliminate	**deadlock resolution** Recovery	**fault-span generation**	**total**
$\mathcal{BA}^5$	10^4	3.5	< 1	< 1	< 1	< 1
$\mathcal{BA}^{10}$	10^9	6.2	4	2	< 1	6
$\mathcal{BA}^{15}$	10^{12}	11.5	39	5	< 1	45
$\mathcal{BA}^{20}$	10^{15}	13.68	185	10	1	199
$\mathcal{BA}^{25}$	10^{19}	14.2	642	19	5	669
$\mathcal{BA}^{30}$	10^{22}	15.2	1791	32	7	1836
$\mathcal{BA}^{35}$	10^{26}	15.6	4492	54	10	4565
$\mathcal{BA}^{40}$	10^{30}	16.5	9253	82	16	9366

(a)



(b)

Fig. 1 Experimental results for algorithm Symbolic_Add_FT and the Byzantine agreement problem.

We now present the results of our experiments with respect to the Byzantine agreement program. Figure 1(a) shows actual memory usage in megabytes and time needed to accomplish each subtask of the algorithm in seconds in terms of the number of non-general processes ($\mathcal{BA}^i$ denotes Byzantine agreement program with i non-general processes). Figure 1(b) compares the total synthesis time and the time spent to complete subtasks of the algorithm graphically. Notice that the x-axis is in logarithmic scale. The number of processes synthesized in our experiments ranges over 5 to 40. 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 40 processes is sufficiently clear to make sound judgements. We now analyze the results.

Total synthesis time

Observe that it takes less than one second to synthesize 5 non-general processes. It is noteworthy to mention that an enumerative implementation of similar heuristics [3, 4] requires 900 seconds to synthesize the same number of processes. Moreover, this enumerative implementation can not handle more than 5 processes due to the state explosion problem. To the contrary, our symbolic approach is cable of synthesizing 40 processes (10^{30} reachable states and beyond) in a reasonable amount of time, which is obviously a significant improvement. Note that the size of state space of $\mathcal{BA}^{40}$ is 10^{28} times larger than the size state space of $\mathcal{BA}^5$.

Figure 1(b) shows that the growth rate of total time spent to synthesize Byzantine agreement is *sublinear* to the size of reachable states. In particular, our analysis shows that the fraction $\frac{Time}{ReachableStates^{0.13}}$ remains constant as the number of non-general processes grows. Sublinearity of total synthesis time to the size of reachable states is important in the sense that the exponential blow-up of state space does not affect the time complexity of our synthesis algorithm. More precisely, the size of reachable states is not an obstacle by itself in order to synthesize distributed programs.

Fault-Span generation

As can be seen in Figure 1(b), the generation of fault-span is fairly fast in case of the Byzantine agreement program. This is mainly due to the following contributing factors:

1. The state space of the program is partially reachable. To illustrate the issue of the size of reachable states, let us consider the Byzantine agreement program with i 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 process has 4 Boolean variables. Also, the general has 2 variables. Hence, the program has $4i+2$ Boolean variables in total and the size of state space is 2^{4i+2}. 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., $f.j = \mathit{false}|\mathit{true}$). Hence, each non-general can have 5 different combinations. Furthermore, the general can have either decision value (i.e., $d.g = 0|1$) and be Byzantine or non-Byzantine (i.e., $b.g = \mathit{false}|\mathit{true}$). Hence, the size of reachable states is at most $5^i * 4$, which is considerably less than the size of the entire explicit state space. For instance, in case of $\mathcal{BA}^{40}$, the size of state space is 10^{45}, while only 10^{30} states are reachable.
2. The diameter of the state-transition graph of the program is not long and, hence, shallow reachability

is possible. For instance, in case of $\mathcal{BA}^{30}$, the fault-span can be computed by only 32 rounds of frontier generation in a breadth first manner.

These reasons significantly affect the efficiency of fault-span generation. It is important to notice that the above factors may behave differently in distributed programs with different structures. As a matter of fact, our experiments with respect to the token ring problem (see Section 9) validates this statement.

Deadlock resolution

Figure 1 also shows the time spent to resolve deadlock states for different number of processes. As mentioned earlier, deadlock resolution is crucial in order for the output program to meet liveness properties. We note that deadlock resolution (as defined in Section 5) 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. In fact, there is very little experimental analysis with regard to synthesis of liveness properties in the literature. Since deadlock resolution is achieved through adding recovery paths and state elimination, we present the time spent in each subtask for a more thorough analysis.

Addition of safe recovery is the first attempt that the algorithm makes in order to resolve deadlock states. The results of our experiments in case of Byzantine agreement shows that this mechanism is not costly as compared to the total synthesis time (see Figure 1(b)). This is solely due to the structure of $\mathcal{BA}$ where the size of fault-span and safety specification are not barriers in efficient addition of safe recovery paths.

As can be seen in Figure 1(b), the graph of total synthesis time is almost identical to the graph of state elimination time. In fact, in the range of 5-40 processes, on average, 96% of the total synthesis time is spent to resolve deadlock states through state elimination. In other words, only 4% of the total synthesis time is spent to re-compute the fault-span, checking the safety of group predicates, computing recovery paths, and re-computing the program invariant. This is basically due to the fact that state elimination can potentially involve many backtracking steps. As a matter of fact, this is exactly what is happening in Byzantine agreement. For instance, in case of $\mathcal{BA}^5$, the Procedure Eliminate needs to be called 26 times recursively. Thus, state elimination can potentially be a serious stumbling block in efficiency of synthesis algorithms. We note that the existence and diversity of deadlock states directly depends on the structure of the given program. In Section 9, we show that in case of token ring, for instance, deadlock resolution is not a time-consuming issue.

Finally, Figure 1(a) shows that if we were not required to resolve deadlocks states, identifying a solution to the problem can be accomplished considerably faster. Although such a solution is not a *masking* fault-tolerant program, it is a correct *failsafe* fault-tolerant program. A failsafe program is one that is required to merely satisfy its safety specification (and not necessarily its liveness specification) in the presence of faults. Consequently, there is no need to resolve deadlock states in order to synthesize a failsafe solution. Thus, one can synthesize a failsafe solution to $\mathcal{BA}^{40}$ in 22 seconds. This performance is significantly better than the DiConic approach in [23] where synthesis of a failsafe version of $\mathcal{BA}^{40}$ requires 353 seconds using a cluster of workstations.

Memory usage

Figure 1(a) also shows the amount of virtual memory that the Algorithm Symbolic_Add_FT requires (in MB) for different number of non-general processes. As can be seen, the amount of memory that the algorithm requires to synthesize 40 processes (16.5 MB) is not considerably greater than the amount of memory required to synthesize 5 processes (3.5 KB). The insignificant growth trend of memory usage is more appreciable when it is compared to the growth of number of reachable states in case of 5 and 40 processes. Low memory usage of our algorithm with respect to this case study is clearly due to efficient representation of Boolean formulae by BDDs.

The issue of variable ordering

The key reason to efficient encoding of a Boolean formula in a BDD is to identify an appropriate order of variables when constructing the BDD [7]. In our implementation, we order the variables based on the following two principles, regardless of the structure of the given fault-intolerant program:

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

For instance, in Byzantine agreement program, the order of variables is as follows:

$$d.j < d'.j < f.j < f'.j < b.j < b'.j < d.k < d'.k < f.k < f'.k < b.k < b'.k < \cdots .$$

The first principle is a rule of thumb in existing symbolic model checkers as well, as a transition often updates a subset of program variables, say U, and leave the rest unchanged. Hence, for each variable v where $v \notin U$, $v = v'$ must hold in the BDD that encodes the transition. Therefore, in order to reduce the number of nodes in the BDD, it is more efficient to order each primed variable immediately right after its corresponding unprimed variable.

The second principle reduces the number of nodes in BDDs that encode group predicates. Recall that the value of all readable variables in source state of all transitions in a group predicate are equal. The same premise holds for target states. Thus, it is beneficial to order readable variables of a process subsequently. As a concrete example, an implementation of Symbolic_Add_FT that does not apply the second principle requires one minute to synthesize $\mathcal{BA}^{10}$, which is 10 times slower than the case where the second principle is applied.

Comparison with model checking

In order to demonstrate the effectiveness and efficiency of our synthesis algorithm, we present a comparison with the corresponding verification problem, which is a significantly easier problem[3]. To this end, we use the model checker NuSMV [24] to verify the following properties for Byzantine agreement:

- agreement,
- validity,
- deadlock-freedom, and
- recovery (i.e., from any state outside the invariant predicate, the program eventually reaches the invariant.

Figure 2 shows the results of our experiments with respect to the property that requires maximum verification time for 10, 20, 30, and 40 processes and compares them with their respective synthesis problem. As can be seen, verification of fault-tolerant Byzantine agreement is faster than its synthesis. The main reason for this clear difference is the issue of deadlock resolution. As mentioned earlier, deadlock resolution is the main bottleneck of synthesizing Byzantine agreement. If we exclude deadlock resolution from synthesizing Byzantine agreement, then the total synthesis time of our method will be equal to the sum of fault-span generation and adding recovery paths. In this case, the total synthesis time would be even less than its corresponding model checking. This latter observation is due to the following reasons:

1. NuSMV employs a general labelling algorithm for model checking in full CTL. The steps of our synthesis algorithm for Byzantine agreement does not suffer from BDD-based cycle detection.
2. Our synthesis algorithm is customized to the assumption that occurrence of faults is finite and the program eventually executes non-fault transitions only. This means that our algorithm can completely overlook searching for cycles involving faults only.
3. Following the previous point, since cycles consisting only fault transitions are of no concern, our NuSMV model employs an integer variable to count the number of occurrence of faults and enforce an upperbound on it. Integer counters often make symbolic model checking inefficient.

[3] In our framework, model checking distributed programs can be done in $O(n^2)$ time, where n is the size of the state space, but their synthesis is NP-complete [5].

	Total verification time(s)	*Total synthesis time(s)*
$\mathcal{BA}^{10}$	0.8	6
$\mathcal{BA}^{20}$	11	199
$\mathcal{BA}^{30}$	50	1836
$\mathcal{BA}^{40}$	332	9366

Fig. 2 Experimental results for model checking Byzantine agreement using NuSMV.

7 Case Study 2: Exploiting Human Knowledge to Assist Synthesis Algorithms

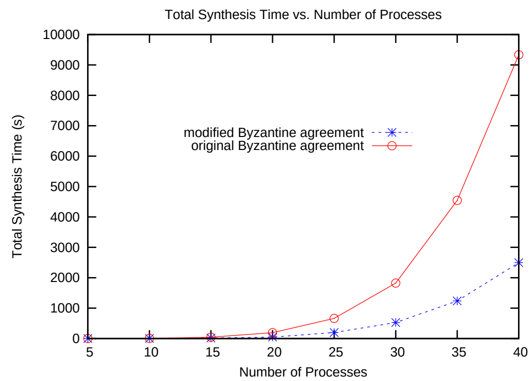
Our experiments on the Byzantine agreement problem clearly exhibited state elimination as a severe bottleneck. One way to remedy this problem is by making states that have to be considered for elimination unreachable by adding constraints to the safety specification (i.e., $SPEC_{bt}$). This can be achieved through labelling transitions that can potentially lead a computation to reach deadlock states as bad transitions in the safety specification. To demonstrate our idea, below we analyze the state elimination step of the Byzantine agreement program by the state-transition graph of the $\mathcal{BA}$.

Let the sequence $\langle x_1, x_2, x_3, x_4 \rangle$ denote the set of states with respect to decision value of processes, i.e., $x_1 = d.g$, $x_2 = d.j$, $x_3 = d.k$, and $x_4 = d.l$. In this notation, an overlined (respectively, underlined) d-value shows that the corresponding process has finalized its decision (respectively, is Byzantine). Now consider the following scenario: Starting from a state s_0 in $\langle 1, \bot, \bot, 1 \rangle$, where the general and process l agree on decision 1 and processes j and k are undecided, the program may reach the following sequence of states due to the occurrence of faults (denoted $\dashrightarrow$) and execution of program actions (denoted $\rightarrow$):

$$\langle 1, \bot, \bot, 1 \rangle \longrightarrow \langle 1, \bot, \bot, \overline{1} \rangle \dashrightarrow \langle \underline{1}, \bot, \bot, \overline{1} \rangle \dashrightarrow$$

	Space		*Time(s)*			
	reachable states	**memory (MB)**	**deadlock resolution** Eliminate	**deadlock resolution** Recovery	**fault-span generation**	**total**
$\mathcal{BA}^5$	10^4	3	0	< 1	< 1	< 1
$\mathcal{BA}^{10}$	10^9	6	0	2	< 1	3
$\mathcal{BA}^{15}$	10^{12}	14.5	0	14	1	17
$\mathcal{BA}^{20}$	10^{15}	18	0	63	1	67
$\mathcal{BA}^{25}$	10^{19}	24	0	188	1	199
$\mathcal{BA}^{30}$	10^{22}	31	0	506	4	526
$\mathcal{BA}^{35}$	10^{26}	44	0	1203	7	1237
$\mathcal{BA}^{40}$	10^{30}	64	0	2428	25	2496

(a)



(b)

Fig. 3 Experimental results for modified Byzantine agreement problem.

$$\langle \underline{0}, \perp, \perp, \overline{1} \rangle \longrightarrow \langle \underline{0}, 0, \perp, \overline{1} \rangle \longrightarrow \langle \underline{0}, 0, 0, \overline{1} \rangle.$$

Let s_1 be a state in $\langle \underline{0}, 0, 0, \overline{1} \rangle$, where non-general processes j and k agree with the Byzantine general on decision 0, but process l has finalized its decision on 1. Observe that s_1 is a deadlock state and since process l has finalized its decision, we cannot resolve s_1 by adding safe recovery. Thus, s_1 has to be eliminated.

One way to make s_1 (and symmetrically equal states) unreachable through adding constraints to the safety specification is by allowing non-general processes to finalize their decision only when there does not exist two or more undecided non-generals. In this case, a non-general process cannot reach a deadlock state such as s_1. Thus, we modify the safety specification of $\mathcal{BA}$ (i.e., the bad transition predicate $SPEC_{bt_{\mathcal{BA}}}$ introduced in Subsection 2.2) as follows:

$$\begin{aligned} SPEC_{bt_{\mathcal{BA}}} \;&=\; SPEC_{bt_{\mathcal{BA}}} \;\vee \\ &\exists p :: \neg b'.p \;\wedge\; f'.p \;\wedge\; (\exists q, r :: d.q \;=\; d.r \;=\; \perp), \end{aligned}$$

where p, q, and r range over non-general processes.

Addition of such constrains is generally straight forward and a user can identify them by a simple analysis of the program behavior. Figure 3 shows the results of our experiments for Byzantine agreement with the above modification in the safety specification. One can make the following observations from this figure:

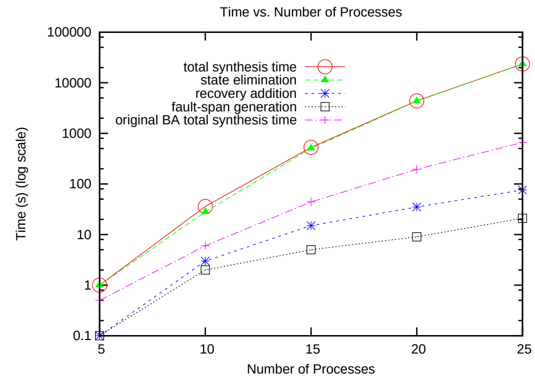
- Figure 3(a) shows that no time is spent for state elimination. This is obviously due to non-existence of deadlock states from where safe recovery is not possible.
- Figure 3(b) compares total synthesis time of the original and modified versions of Byzantine agreement and one can obviously see a considerable improvement. Precisely, in average, the modified version can be synthesized 4 times faster than the original version for the range of 5 to 40 processes. It is expected that this factor becomes larger as the number of processes increases.
- Although constraining a non-general process casts away states that have to be eliminated, it affects the performance of addition of safe recovery. Our analysis shows this is mainly due to enlarging the transition predicate $SPEC_{bt_{\mathcal{BA}}}$ and as a result its corresponding BDD. Recall that recovery paths are not allowed to violate the safety specification and, hence, the large size of the BDD that encodes $SPEC_{bt_{\mathcal{BA}}}$ affects the performance of addition of recovery.
  Nonetheless, this cost does not diminish the improvement of total synthesis time.
- Due to the same reason, memory usage of our modified version of Byzantine agreement is increased. Although this increase suffers by a factor of 2 in average, we argue that the trade-off is worthwhile. A closer look at Figures 1 and 3 uncovers that unlike verification where the time complexity of algorithms is generally not high, our crucial issue in synthesis is time and not space. In other words, in synthesis, *we run out of time before we run out of space*. Thus, given the low memory usage of our case study, it is beneficial to increase memory usage by a factor of 2 to gain a speed-up by a factor of 4.

8 Case Study 3: Byzantine Agreement with Fail-Stop Faults

A *fail-stop* fault is one that halts a process in response to any internal failure and does so before the effects of that failure become visible [11]. In this Section, we introduce fail-stop faults to the Byzantine agreement

	Space		Time(s)			
	reachable states	**memory (MB)**	**deadlock resolution** Eliminate	Recovery	**fault-span generation**	**total**
$\mathcal{BA}^5$	10^5	5.2	1	< 1	< 1	1
$\mathcal{BA}^{10}$	10^8	13	28	3	2	36
$\mathcal{BA}^{15}$	10^{12}	14.5	505	15	5	528
$\mathcal{BA}^{20}$	10^{16}	15.9	4322	35	9	4378
$\mathcal{BA}^{25}$	10^{20}	17.8	23387	76	21	23502

(a)



(b)

Fig. 4 Experimental results for Byzantine agreement subject to fail-stop faults.

problem to make the program more complicated (denoted $\mathcal{BAFS}$). To this end, we first add a Boolean variable u to each process; if u is true then the process is alive and working, otherwise, the process has been stopped and is not working. Thus, actions of a process, say j, are as follows:

$$\begin{array}{lll} \mathcal{BAFS}1_j & :: & (d.j = \bot) \wedge (f.j = \mathit{false}) \wedge (u.j = \mathit{true}) \\ & & \qquad \longrightarrow \quad d.j \ := \ d.g; \\ \mathcal{BAFS}2_j & :: & (d.j \neq \bot) \wedge (f.j = \mathit{false}) \wedge (u.j = \mathit{true}) \\ & & \qquad \longrightarrow \quad f.j \ := \ \mathit{true}; \end{array}$$

In addition to the faults introduced in the example in Section 3, we introduce the following fault action:

$$\forall p :: (u.p) \quad \longrightarrow \quad u.j \ := \ \mathit{false};$$

In other words, at most one process may fail. A Byzantine process can change its decision only if it is alive. Thus, we revise fault action F_1 as follows:

$$\begin{array}{l} F_1 \ :: \ b.j \ \wedge \ u.j \\ \qquad \longrightarrow \qquad d.j, \ \ f.j \ := \ 0|1, \ \ \mathit{false}|\mathit{true}; \end{array}$$

Likewise, the safety specification and invariant of $\mathcal{BAFS}$ must express the fact that a process may decide or finalize its decision only if it has not stopped due to the occurrence of faults. Thus, $SPEC_{bt_{\mathcal{BAFS}}}$ and $I_{\mathcal{BAFS}}$ are as follows:

$$\begin{array}{l} SPEC_{bt_{\mathcal{BAFS}}} \ = \\ \quad (\exists p \in \{j,k,l\} \ :: \ \neg b'.g \ \wedge \ \neg b'.p \ \wedge \\ \qquad (d'.p \neq \bot) \ \wedge \ f'.p \ \wedge \ (d'.p \neq d'.g)) \ \vee \\ \quad (\exists p, q \in \{j,k,l\} \ :: \ \neg b'.p \ \wedge \ \neg b'.q \ \wedge \\ \qquad f'.p \ \wedge \ f'.q \ \wedge \ u'.p \ \wedge \ u'.q \ \wedge \\ \qquad (d'.p \neq \bot) \ \wedge \ (d'.q \neq \bot) \ \wedge \ (d'.p \neq d'.q)) \ \vee \\ \quad (\exists p \in \{j,k,l\} \ :: \ \neg b.p \ \wedge \ \neg b'.p \ \wedge \ f.p \ \wedge \ \neg u'.p \ \wedge \\ \qquad ((d.p \neq d'.p) \ \vee \ (f.p \neq f'.p))), \end{array}$$

and

$$\begin{array}{l} I_{\mathcal{BAFS}} = \\ \quad (\forall p, q \in \{j,k,l\} \ :: \ u.p \ \vee \ u.q) \wedge \\ \quad [\neg b.g \ \wedge \ (\forall p, q \in \{j,k,l\} \ :: \ (\neg b.p \ \vee \ \neg b.q)) \ \ \wedge \\ \quad (\forall p \in \{j,k,l\} \ :: \ \neg b.p \Rightarrow (d.p = \bot \vee d.p = d.g)) \ \wedge \\ \quad (\forall p \in \{j,k,l\} \ :: \ (\neg b.p \wedge f.p) \ \Rightarrow \ (d.p \neq \bot)) \\ \quad \vee \\ \quad (b.g \ \wedge \ \neg b.j \ \wedge \ \neg\, b.k \ \wedge \ \neg\, b.l \ \wedge \\ \quad \forall p, q \in \{j,k,l\} \ :: \ ((u.p \ \wedge \ u.q) \ \ \Rightarrow \\ \qquad\qquad ((d.p \ = \ d.q) \ \wedge \ (d.p \ \neq \ \bot))))]. \end{array}$$

Figure 4 shows the results of our experiments on the Byzantine agreement program subject to fail-stop faults with 5-25 non-general processes. The growth trend of both time and space as the number of processes increases is similar to the normal Byzantine agreement. However, $\mathcal{BAFS}$ has obviously a larger state space and occurrence of fail-stop faults makes the synthesis problem more complex. In particular, state elimination is significantly harder to solve, as the Procedure Eliminate involves more backtracking steps due to existence of a failed process on reachability paths. As a concrete example, observe that it takes about 6.5 hours to synthesize a solution to $\mathcal{BAFS}^{25}$, while it only takes 11 minutes to synthesize a solution to $\mathcal{BA}^{25}$.

9 Case Study 4: Token Ring

In a token ring program (denoted $\mathcal{TR}$) for solving distributed mutual exclusion, processes $0..N$ are organized in a ring and the token is circulated along the ring in a fixed direction. Each process, say p where $p \in \{0..N\}$, maintains a variable $x.p$ with domain $\{0, 1, \bot\}$, where $\bot$ denotes a corrupted value. Process p, $0 \leq p \leq N-1$, has the token and can enter the critical section iff $x.p$ differs from its successor $x.(p+1)$ and process N has the token iff $x.N$ is the same as its successor $x.0$. Each process p can only write its local variable (i.e., $x.p$). Moreover, a process can only read its own local variable and the variable of its predecessor. Thus, the read/write restrictions are as follows:

$V_p = \{x.i \mid 0 \le i \le N\}$, where $p \in \{0..N\}$,
$W_p = \{x.p\}$, where $p \in \{0..N\}$,
$R_p = \{x.p, x.(p-1)\}$, where $p \in \{1..N\}$, and
$R_0 = \{x.0, x.N\}$.

Fault-intolerant program. The program consists of two actions. Formally, these actions are as follows:

$$\begin{array}{lllll} \mathcal{TR}_p & :: & x.p \neq x.(p-1) & \longrightarrow & x.p \ := \ x.(p-1); \\ \mathcal{TR}_0 & :: & x.0 = x.N & \longrightarrow & x.0 \ := \ x.N +_2 1; \end{array}$$

where $p \in \{1..N\}$ and where $+_2$ denotes modulo 2 addition.

Fault actions. Faults can restart at most $N-1$ processes. Thus, the fault action for process p, where $p \in \{0..N\}$ is as follows:

$$\begin{array}{ll} F \ :: \ \exists i,j \in \{0..N\} \mid (i \neq j) \ :: & \\ \qquad (x_i \neq \bot) \ \wedge \ (x_j \neq \bot) \quad \longrightarrow \quad x.p \ := \ \bot; \end{array}$$

Safety specification. The safety specification of $\mathcal{TR}$ 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:

$$SPEC_{bt_{\mathcal{TR}}} = \bigvee_{p=0}^{N} (x.p \neq \bot \ \wedge \ x'.p = \bot).$$

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.

Invariant predicate. The invariant predicate of the token ring program is determined as follows. Consider a state where a process, say p, has the token. In this state, since no other process has the token, the x-value of all processes $0..p$ is identical and the x-value of all processes $(p{+}1)..N$ is identical. Letting X denote the string of binary values $x.0, x.1 \cdots x.N$, we have that X satisfies the regular expression $(0^l 1^{(N+1-l)} \cup 1^l 0^{(N+1-l)})$, which denotes a sequence of length $N+1$ consisting of zeros followed by ones or ones followed by zeros.

Fault-tolerant program. The output of our algorithm is a program $\mathcal{TR}'$ that tolerates the above fault action. Intuitively, a process in the synthesized program is allowed to copy the value of its predecessor, if this value in not corrupted. The actions of the synthesized fault-tolerant program are as follows:

$$\begin{array}{lll} \mathcal{TR}'_p & :: & (x.p \neq x.(p-1)) \ \wedge \ (x.(p-1) \neq \bot) \\ & & \qquad \longrightarrow \qquad x.p \ := \ x.(p-1); \\ \mathcal{TR}'_0 & :: & (x.0 \neq (x.N +_2 1)) \ \wedge \ (x.N \neq \bot) \\ & & \qquad \longrightarrow \qquad x.0 \ := \ x.N +_2 1; \end{array}$$

where $p \in \{1..N\}$. Observe that action $\mathcal{TR}'_0$ stipulates recovery transitions that start from outside program invariant as well.

Figure 5 shows the results of our experiments with respect to the token ring program. Although token ring has a less complex structure than Byzantine agreement, it exhibits features that Byzantine agreement does not. One of these features is the existence of two cycles in both input and output programs which affect the addition of multi-step recovery. Another feature is concerned with the size of fault-span. Unlike Byzantine agreement, the fault-span (i.e., the set of all reachable states) of token ring is almost identical to the state space of the program.

Total synthesis time

Similar to the previous case studies, in case of token ring, the total synthesis time is sublinear to the number of reachable states. We emphasize that the result of our experiments with respect to token ring is considerably different from the results reported in [25]; we can synthesize up to 100 processes in less than two hours while it takes 8 hours to synthesize 25 processes using the method in [25]. This is mainly due to the choice of reachability analysis algorithm.

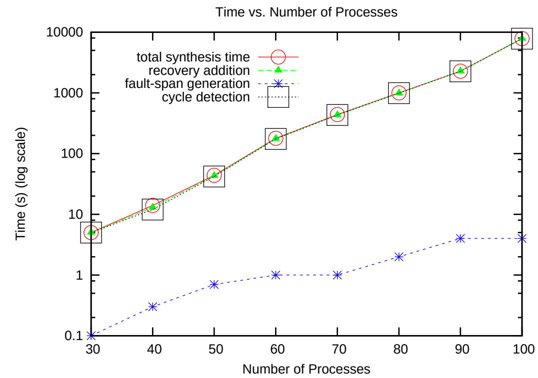
BDD-based computation of reachable states is normally achieved using a breadth-first search algorithm on state-transition graph of the input program. Such a BFS algorithm involves a frontier generation step which can be implemented in two ways:

1. Applying the transition predicate only on unexplored states which at iteration d, consists of all states at distance exactly d from the invariant predicate.
2. Applying the transition predicate to all known states, that is all states at distance at most d from the invariant predicate.

While the second approach may sound wasteful, the cost of applying the transition predicate in a symbolic setting depends on the number of nodes in the corresponding BDD, not on the number of states encoded by the BDD. Thus, even in the verification research community, it is unknown which approach is better. In [25], we implemented the first approach, but through analyzing more case studies, we choose to incorporate the second approach for its generality. An obvious reason for

	Space		*Time(s)*			
	reachable states	**memory (MB)**	**recovery addition**	**cycle detection**	**fault-span generation**	**total**
$\mathcal{TR}^{30}$	10^{14}	8	5	5	< 1	5
$\mathcal{TR}^{40}$	10^{19}	13.3	13	13	< 1	35
$\mathcal{TR}^{50}$	10^{23}	14.5	43	43	< 1	44
$\mathcal{TR}^{60}$	10^{28}	15.2	176	174	1	179
$\mathcal{TR}^{70}$	10^{33}	18.8	433	432	1	438
$\mathcal{TR}^{80}$	10^{38}	25.3	992	990	2	999
$\mathcal{TR}^{90}$	10^{42}	33.7	2272	2270	4	2283
$\mathcal{TR}^{100}$	10^{47}	44.2	7824	7819	4	7837

(a)



(b)

Fig. 5 Experimental results for token ring mutual exclusion program.

better performance of the second approach in the context of our synthesis problem which is highly probable is that the entire state space of programs is reachable in the presence of faults. Thus, many variables may become don't care in the corresponding BDD which in turn reduces the number of nodes in the BDD. Moreover, in programs such as token ring, there exists many transitions that visits states that are already explored. Consequently, the issue of fault-span generation is not a serious bottleneck as it was in [25].

Deadlock resolution and cycle detection

In this case study, we are not concerned with state elimination time, as safe recovery from all deadlock states is possible. Thus, in order to analyze deadlock resolution time, we only focus on addition of recovery. As can be seen in Figure 5, the total synthesis time is almost equal to the time spent for adding recovery paths to the program. Moreover, the time spent for adding recovery is almost equal to the time spent for detecting cycles. First, we note that in previous case studies, cycle detection time were negligible and, hence, was not discussed. However, in this case study, a considerable amount of time is spent for detecting cycles. Recall that in Procedure AddRecovery, after adding a new layer to recovery paths, we check whether or not a cycle has been introduced to the fault-span of the intermediate program (see Line 5 of Procedure 5.2). Since the input program in previous case studies does not contain a cycle, the cycle detection algorithm tends to return a negative answer fairly fast. However, one can easily observe that the token ring intolerant program has two cycles that cover all states in the invariant. Thus, in the steps of adding multi-step recovery paths, new cycles are introduced to the fault-span symmetrically which is reflected in the time spent to detect cycles and subsequently removing transitions involved in the cycles. In fact, one can observe in Figure 5(b) the total synthesis times, and time spent detecting cycles and adding recovery are almost equal. Thus, in programs such as token ring cycle detection becomes a stumbling block of our synthesis algorithm.

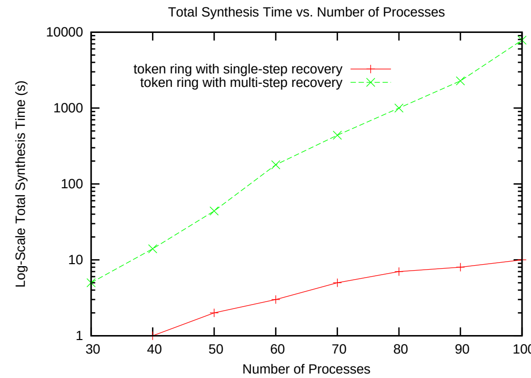
The Effect of Multi-Step Recovery

As mentioned earlier, the issue of cycle detection exists in addition of recovery, as our algorithm constructs *multi-step* recovery paths. Notice that the first recovery step includes transitions that originate from a set of reachable deadlock states and end in the invariant predicate. Recall that each transition has to be added along with its corresponding group predicate. Thus, including additional recovery steps can potentially introduce cycles to the fault-span which in turn prohibits the program to recover to the invariant predicate in a finite number of steps. Hence, an algorithm that synthesizes single-step recovery to an input program need not detect cycles.

Obviously, depending upon the structure of input program, a different type of recovery path may be required. For instance, in case of the token ring program, single-step recovery suffices to resolve all deadlock states. Thus, a respective algorithm need not include the while loop and DetectCycle function in the Procedure AddRecovery. Figure 6 shows the result of experiments using such an algorithm for adding single-step recovery to $\mathcal{TR}$. As can be seen, an enormous speed-up is gained. On average, the total synthesis time drops by a factor of 330. To illustrate the effect of such a small change in the algorithm, we note that one can synthesize token ring with 200 processes (reachable states

	Space		*Time(s)*			
	reachable states	**memory (MB)**	**recovery addition**	**cycle detection**	**fault-span generation**	**total**
$\mathcal{TR}^{30}$	10^{14}	4	< 1	0	< 1	0
$\mathcal{TR}^{40}$	10^{19}	5	< 1	0	< 1	1
$\mathcal{TR}^{50}$	10^{23}	7.6	1	0	< 1	1
$\mathcal{TR}^{60}$	10^{28}	9.8	1	0	< 1	2
$\mathcal{TR}^{70}$	10^{33}	11.4	1	0	1	3
$\mathcal{TR}^{80}$	10^{38}	12.5	1	0	3	5
$\mathcal{TR}^{90}$	10^{42}	12.9	1	0	4	6
$\mathcal{TR}^{100}$	10^{47}	13	2	0	4	8

(a)



(b)

Fig. 6 Experimental results for token ring mutual exclusion with single-step recovery.

of size 10^{95}) in less than 2 minutes.

Comparison with model checking

We now present a comparison with the corresponding verification problem for token ring. To this end, we verify the following recovery property:

> *The invariant predicate is eventually reached by the program even in the presence of faults.*

Figure 7 shows the results of our experiments for 20, 40, 60, 80, and 100 processes and compares them with their respective synthesis problem. As can be seen, our synthesis algorithm for adding single-step recovery exhibits a significantly better performance than the corresponding model checking experiment. Similar to Byzantine agreement, this is due to the following reasons:

1. NuSMV employs a general labelling algorithm for model checking in full CTL. Our synthesis algorithm for adding single-step recovery is only concerned with a subset of CTL which does not handle cycles consisting program transitions only.
2. Our synthesis algorithm is customized to the assumption that occurrence of faults is finite and the program eventually executes non-fault transitions only. This means that our algorithm can completely overlook searching for cycles involving faults only.
3. Since in token ring all but one process are subject to faults, our NuSMV model employs an integer variable to count the number faults to ensure that faults do not occur in all processes. Integer counters often make symbolic model checking inefficient.

On the contrary, our multi-step algorithm does not outperform model checking of token ring. This is simply because cycle detection and resolution makes synthesis significantly more difficult.

10 Case Study 5: Infuse

We now focus on Infuse, a time division multiple access (TDMA) based reliable data dissemination protocol in sensor networks [12]. Our intention to present this case study is twofold. First, we intend to demonstrate an application of our algorithm outside the literature of fault-tolerant distributed computing. In other words, we demonstrate the applicability of our algorithm in real-world problems by adding fault-tolerance to a sensor network protocol. Secondly, we use Infuse as yet another case study to analyze the performance of our algorithm.

In Infuse, a base station is responsible for communicating with the outside world. The data is split into fixed size packets. Note that Infuse is not concerned with the contents of the data. In our version of case study, all sensors are located in a simple line topology. The base station sends new data to its sole neighbour. Then, this neighbour forwards the packet to its neighbour and so on.

Each sensor maintains two variables r and s where r denotes the sequence number of the last packet the sensor has received and s denotes the sequence number of the packet to be sent to its neighbour. Each variable ranges over $0..M$, where M is the number of bytes in each packet. Thus, if sensors are numbered 0 to N, where sensor 0 is the base station, the read/write restrictions for corresponding processes are as follows:

$R_0 = \{s.0, r.0, s.1, r.1\}$,
$W_0 = \{s.0, r.0\}$,

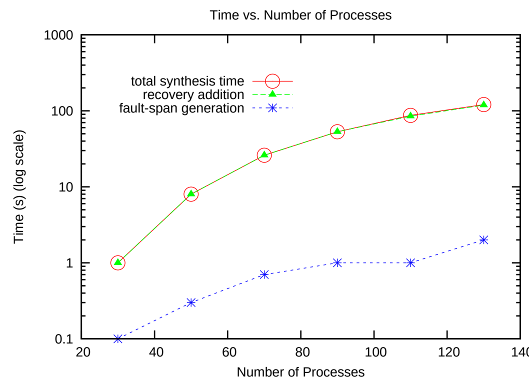
$R_j = \{s.(j-1), r_(j-1), s.j, r.j, s.(j+1), r.(j+1)\}$,
$W_j = \{s.j, r.j\}$, where $1 \leq j \leq N-1$,

	Total verification time(s)	*Total muti-step synthesis time(s)*	*Total single-step synthesis time(s)*
$\mathcal{TR}^{20}$	0.2	2	< 1
$\mathcal{TR}^{40}$	5	35	< 1
$\mathcal{TR}^{60}$	23	179	2
$\mathcal{TR}^{80}$	110	999	5
$\mathcal{TR}^{100}$	632	7837	8

Fig. 7 Experimental results for model checking token ring using NuSMV.

	Space		*Time(s)*		
	reachable states	**memory (MB)**	**recovery addition**	**fault-span generation**	**total**
$\mathcal{IF}^{30}$	10^{15}	4.7	1	< 1	1
$\mathcal{IF}^{50}$	10^{25}	10	8	< 1	8
$\mathcal{IF}^{70}$	10^{34}	11	26	< 1	26
$\mathcal{IF}^{90}$	10^{43}	14.5	53	1	53
$\mathcal{IF}^{110}$	10^{51}	15	85	1	87
$\mathcal{IF}^{130}$	10^{60}	16	118	2	121

(a)



(b)

Fig. 8 Experimental results for Infuse bulk data dissemination protocol in sensor networks.

$R_N = \{s.N, r.N, s.(N-1), r.(N-1)\}$,
$W_N = \{s.N, r.N\}$.

Fault-intolerant program. Initially, all packets are disseminated from the base station. A new packet is sent to sensor 1 when the base station knows that sensor 1 has received the last packet it had sent. Thus, the action for the base station is as follows:

$$\mathcal{IF}_0 \;::\; s.0 = r.1 \quad\longrightarrow\quad s.0 \;:=\; s.0+1;$$

Intuitively, a sensor, say j, where $j \in \{1..N-1\}$, may receive a new packet, if

1. the successor (i.e., $j+1$) has all the packets that j has received, and
2. $j-1$ is transmitting the next packet.

If this is the case, then j receives the packet and transmits it in the next slot. Thus, the action that models packet transmission for sensors $1..N-1$ is as follows:

$$\begin{array}{l} \mathcal{IF}_j \;::\; (r.j = r.(j+1)) \;\wedge\; (s.(j-1) = r.j+1) \\ \qquad\qquad\longrightarrow\quad r.j,\; s.j \;:=\; r.j+1,\; s.j+1; \end{array}$$

Finally, sensor N obtains a packet, if its predecessor has the next packet that sensor N expects. Formally,

$$\mathcal{IF}_N \;::\; s.(N-1) = r.N+1 \quad\longrightarrow\quad r.N \;:=\; r.N+1;$$

Fault actions. A fault causes the base station to transmit a packet that sensor 1 does not expect. In other words, some packet transmitted by the base station is lost. For the rest of sensors, a fault allows a sensor, say j, to receive a packet from its predecessor $j-1$ even though its successor $j+1$ did not obtain the packet that j transmitted last. Thus, the fault actions of Infuse are as follows:

$$\begin{array}{l} F_0 \;:\; true \quad\longrightarrow\quad s.0 \;:=\; s.0 \;+\; 1; \\ F_j \;::\; (r.j \leq r.(j-1)) \;\wedge\; (s.(j-1) = r.j+1) \\ \qquad\qquad\longrightarrow\quad r.j,\; s.j \;:=\; r.j+1,\; s.j+1; \end{array}$$

Safety specification. The safety specification of Infuse informally consists of the following constrains:

– Reception of a packet cannot be undone and packets can be received only in order,
– A sensor is not allowed to receive a packet unless its predecessor neighbour has received it,
– A sensor may not send a packet that it has not obtained yet.
– Finally, the current s-value of a sensor should reflect the packet it expects from the neighbouring sensor.

Thus, the safety specification of Infuse is formally as follows:

$$
\begin{aligned}
&SPEC_{bt_{\mathcal{IF}}} \ = \\
&\quad \exists p \in \{1..N\} \ :: \ ((r'.p \ < \ r.p) \ \vee \ (r'.p \ > \ r.p \ + \ 1)) \ \vee \\
&\quad \exists p \in \{1..N-1\} \ :: \ (r'.p \ = \ r.p \ + \ 1) \ \wedge \\
&\qquad\quad ((r'.p \neq s.(p-1)) \ \wedge \ (r'.p \neq s.(p+1))) \ \vee \\
&\quad (r'.N = r.N + 1 \ \ \wedge \ \ r'.N \neq s.(N-1)) \ \vee \\
&\quad \exists p \in \{0..N\} \ :: \ (r'.p \ < \ s'.p) \ \vee \\
&\quad \exists p \in \{0..N-1\} \ :: \ ((s.p \ > \ r.(p+1)+1) \ \wedge \\
&\qquad\quad (s'.p \ < \ r.(p+1)+1)).
\end{aligned}
$$

Invariant. Informally, the invariant of Infuse specifies the following set of legitimate states:

- It is illegitimate for a sensor to send a packet that it has not received.
- The packet to be sent by a sensor must be expected by its successor sensor.
- The base station initially owns all the packets.
- Finally, a sensor should not have a packet that its predecessor sensor dose not.

Thus, the invariant of Infuse is formally defined as follows:

$$
\begin{aligned}
I_{\mathcal{IF}} \ = \ & (\forall p \in \{0..N\} \ :: \ s.p \ \leq \ r.p) \ \wedge \\
& (s.0 \ \leq \ r.1 \ + \ 1) \ \wedge \\
& (\forall p \in \{1..N-1\} \ :: \ (s.p \ \leq \ r.(p-1)+1) \ \wedge \\
& \qquad (s.p \ \leq \ r.(p+1) \ + \ 1)) \ \wedge \\
& (s.N \ \leq \ r.(N-1) \ + \ 1) \ \wedge \\
& (r.0 \ = \ M) \ \wedge \\
& (\forall p \in \{1..N\} \ :: \ r.p \ \leq \ r.(p-1))
\end{aligned}
$$

Fault-tolerant program. Given $\mathcal{IF}$, F, $SPEC_{bt_{\mathcal{IF}}}$, and $I_{\mathcal{IF}}$, the output of our algorithm is a fault-tolerant version of Infuse, denoted $\mathcal{IF}'$. Adding fault-tolerance to $\mathcal{IF}$ basically results in synthesizing recovery paths. This is because the occurrence of faults does not lead the program to a state from where safety may be violated. Hence, the only task Algorithm Add_Symbolic_FT needs to accomplish is to guarantee deadlock freedom. And, such deadlock freedom can be achieved by adding safe recovery and no state elimination is required. Formally, the fault-tolerant of Infuse is the following program:

$$
\begin{aligned}
&\mathcal{IF}'0_j \ :: (r.j = r.(j+1)) \ \wedge \ (s.(j-1) = r.j + 1) \\
&\qquad\qquad \longrightarrow \qquad r.j, \ s.j \ := \ r.j+1, \ s.j+1; \\
&\mathcal{IF}'1_j \ :: (s.j > r.(j+1)+1) \\
&\qquad\qquad \longrightarrow \qquad s.j \ := \ r.(j+1)+1; \\
&\mathcal{IF}'2_j \ :: (s.j > r.(j+1)+1) \ \wedge \\
&\qquad\qquad (s.(j-1) = r.j \ + \ 1) \\
&\qquad\qquad \longrightarrow \qquad s.j, \ r.j \ := \ r.(j+1)+1, \ r.j+1;
\end{aligned}
$$

where $j \in \{1..N-1\}$. Actions of the base station and sensor N can be derived similarly. Observe that $\mathcal{IF}'0_j$ is an unchanged action. Actions $\mathcal{IF}'1_j$ and $\mathcal{IF}'2_j$ are recovery actions and resolve reachable deadlock states. Essentially, these actions enable the program to retransmit packets that are lost while maintaining the safety specification to keep the correct sequence of packet transmission.

Figure 8 shows the result of our experiments with respect to Infuse. We emphasize that $\mathcal{IF}$ does not reach states that need to be eliminated. In addition, the state-transition graph of Infuse does not include cycles. Thus, cycle detection and state elimination do not play any role in adding fault-tolerance to $\mathcal{IF}$. Given these facts, Figure 8 is self-evident in describing the behavior of Algorithm Symbolic_Add_FT with respect to Infuse. The majority of total synthesis time is spent to add safe recovery which is expected due to the structure of Infuse. One can also observe that the fault-span generation time is negligible. This is due to the fact that the diameter of the state-transition graph of Infuse is short.

11 Related Work

The concept of program synthesis has been studied from different perspectives ranging from synthesis from temporal logic specifications to controller synthesis and synthesizing winning strategies in game theory. In this section, we present the work in the literature that is relevant to our work in this paper.

11.1 Controller Synthesis

Synthesis of discrete-event systems has mostly been studied in the context of controller synthesis and game theory. The seminal work in the area of controller synthesis is due to Ramadge and Wonham [26]. The discrete controller synthesis (DCS) problem is as follows: starting from two languages $\mathcal{U}$ and $\mathcal{D}$, identify a third language $\mathcal{C}$ such that $\mathcal{U} \cap \mathcal{C} \subseteq \mathcal{D}$. In the DCS terminology, the three languages $\mathcal{U}$, $\mathcal{D}$, and $\mathcal{C}$ are called the *plant*, the *desired system*, and the *controller*, respectively. $\mathcal{U} \cap \mathcal{C}$ is called the *controlled system*. Finally, the set $\mathcal{A}$ of alphabets represents events that can occur. Obviously, the languages $\mathcal{U}$ and $\mathcal{D}$ may represent the set of computations of a given program and a safety and/or reachability specification. Moreover, $\mathcal{C}$ identifies the computations that do not violate $\mathcal{D}$ in the presence of uncontrollable transitions.

The idea of transforming a fault-intolerant system into a fault-tolerant system using controller synthesis was first developed by Chao and Lim [27]. Similar to the model in this paper, Chao and Lim consider faults as a system malfunction and failures as something that should not occur in any execution. Their control objective is a set of states that should be reachable by controllable actions or what they define as *recurrent events*. Also, Girault and Rutten [28], demonstrate the application of discrete controller synthesis in automated addition of fault-tolerance in the context of untimed systems. They model different types of faults (e.g., processor crash, Byzantine faults, value corruption) by uncontrollable actions in a labeled transition system (LTS). They show that given a fault-intolerant program as a plant, discrete controller synthesis can automatically add fault-tolerance to the synchronous product of the plant and the fault model LTS with respect to invariance and reachability constraints.

One can notice that our work in this paper is in spirit close to DCS. Specifically, an input program and fault transitions may be modelled as controllable and uncontrollable actions. In fact, in both problems, the objective is to *restrict* the program actions at each state through synthesizing a controller such that the behavior of the entire system is always *desirable* according to safety and reachability conditions, in the presence of an adversary. As mentioned in Section 4, conditions $C1$ and $C2$ of the problem statement precisely express this notion of restriction. Furthermore, the conjunction of all conditions expresses the notion of *language inclusion*, where the synthesized program in the absence of faults is supposed to exhibit a subset of behaviors of the input intolerant program. Although the complexity of controller synthesis for centralized programs is comparable to ours if we eliminate distribution from our formulation [29], our work differs from synthesizing discrete-event controllers in that:

1. The computation model for synthesizing controllers is based on prioritized synchronization[4], whereas ours is based on interleaving.
2. Our synthesis algorithm is concerned with properties typically used in specifying fault-tolerance requirements rather than any arbitrary specification. Hence, our algorithm tends to synthesize programs more efficiently.
3. The notion of dependability and in particular fault-tolerance involves features beyond just invariance and reachability. One such feature is *recovery*, where a program returns to its normal behavior when its state is perturbed by the occurrence of faults. Recovery is an essential building block in fault-tolerant systems and it is the focus of this paper. In controller synthesis [26–28, 30–39], and in particular in [28], which is in spirit close to our work, the recovery mechanism must be given as input to the DCS algorithm. Thus, one key difference between our work in this paper and the methods in is the fact that we automatically synthesize recovery paths. We also analyze the complexity and cost of dealing with livelocks (i.e., cycles outside the invariant predicate) in this paper. Finally, note that adding recovery transitions to a given fault-intolerant program is not trivial due to the distributed nature of programs and read/write restrictions.
4. Finally, we model *distribution* by specifying read/write restrictions, whereas in controller synthesis, decentralized plants are modelled through *partial observability* [36, 37]. As mentioned earlier, the issue of distribution drastically increases the complexity of synthesis [5, 6]. This results is also known in the context of controller synthesis, but to the best of our knowledge, this paper introduces the first instance where synthesis of distributed fault-tolerant programs scales up and moderate-sized programs beyond toy examples are successfully synthesized. We are also not aware of any work in controller synthesis that analyzes efficiency and effectiveness of respective algorithms using different types of programs in terms of structure and size. Unlike existing controller synthesis tools such as SIGALI [40], our synthesis method and corresponding tool SYCRAFT [13] is capable of synthesizing distributed programs.

[4] In prioritized synchronization, different processes have synchronized transitions and some have higher priority. Hence, not all interleavings are poosible.

11.2 Game Theory

Game-theoretic approaches for synthesizing controllers and reactive programs [41] are generally based on the model of two-player games [42]. In such games a program makes moves in response to the moves of its environment. The program and its environment interact through a set of interface variables and, hence, the environment can only update the interface variables. In our model, however, faults can perturb all program variables. Moreover, in a two-player game model, players take turns and the set of states from where the first player can make a move is disjoint from the set of states from where the second player can move [43]. To the contrary, in our work, fault-tolerance should be provided against faults that can execute from any state.

Game theoretic methods are based on the theory of tree automata [44]. Such an automaton represents the specification of a system. A synthesis algorithm checks the non-emptiness of the automaton, i.e., whether there exists a tree acceptable by the tree automaton. If the tree automaton is indeed nonempty, then the specification is called *realizable* and there exists a model of the synthesized program.

Pnueli and Rosner address the problem of synthesizing synchronous open reactive modules in [41]. They generalize their method in [45], by proposing a technique for synthesizing asynchronous reactive modules. In particular, they investigate the problem of synthesizing an asynchronous reactive module that include only one process and interacts with a non-deterministic environment through Boolean variables.

While symbolic model checking has been studied extensively (e.g., [8, 18, 46]), little work has been done on symbolic synthesis and especially on performance analysis of synthesis methods. Wallmeier, Hütten, and Thomas [43] introduce an algorithm for synthesizing finite state controllers by solving infinite games over finite state spaces. They model the winning constraint by safety conditions and a set of request-response properties as liveness conditions. They transform this game into a Büchi game which inevitably involves an exponential blow-up. Moreover, the approach in [43] does not address the issue of distribution. The reported maximum number of variables in their experiments is 23, which is far less than the number of variables that we have handled using our symbolic algorithms.

We emphasize that similar to discrete controller synthesis, game theoretic approaches do not address the issue of addition of recovery. Also, in game theory, the notion of distribution is modelled by partial observability.

11.3 Automated Addition of Fault-Tolerance

Algorithms for automatic addition of fault-tolerance [6, 47–49] add fault-tolerance concerns to existing untimed or real-time programs in the presence of faults, and guarantee the addition of no new behaviors to the original program in the absence of faults. In the seminal work in this area, Kulkarni and Arora [6] introduce synthesis methods for automated addition of fault-tolerance to untimed centralized and distributed programs. In particular, they introduce polynomial-time sound and complete algorithms for adding all levels of fault-tolerance (failsafe, nonmasking, and masking) to centralized programs. The input to these algorithms is a fault-intolerant centralized program, safety specification, and a set of fault transitions. The algorithms generate a fault-tolerant program along with an invariant predicate. The authors also show that the problem of adding masking fault-tolerance to distributed programs is NP-complete in the size of the input program's state space.

In [47], Kulkarni and Ebnenasir address the problem of automated synthesis of *untimed multitolerant* programs, i.e., programs that tolerate multiple classes of faults and provide a (possibly) different level of fault-tolerance to each class. They show that if one needs to add failsafe (respectively, nonmasking) fault-tolerance with respect to one class of faults and masking fault-tolerance with respect to another class of faults, then such addition can be done in polynomial-time in the size of the state space of the fault-intolerant program. They, however, show that if one needs to add failsafe fault-tolerance with respect to one class of faults and nonmasking fault-tolerance with respect to another class of faults, then the problem is NP-complete.

Ebnenasir [23] develops a divide-and-conquer method synthesis problem that scales up. In an application of this approach for safety properties, Ebnenasir develops an algorithm that statically analyzes (and possibly revises) program instructions on separate machines in a parallel/distributed platform. Based on this method, the author implements a distributed framework that exploits the computational resources of wide area networks for program synthesis. Using this approach, it is possible to synthesize failsafe Byzantine agreement with 40 processes on a cluster of three machines in 353 seconds. Using our BDD-based approach, the same program can be synthesized in 22 seconds using one machine only.

The problem of online fault *detection* in timed automata is studied by Tripakis [50]. The author proposes a polynomial-space online algorithm for designing a diagnoser that detects faults in behaviors of a given timed automaton after they occur. In this work, it is assumed that (1) the given system is in the synchronous model, and (2) faults and failures are identical events. Thus, this model does not capture situations where the occurrence of faults (although undesirable) is common and expected, but may *lead* to a system with failures. Bouyer, Chevalier, and D'Souza [51] address the same problem where the diagnoser is realizable as a deterministic timed automaton or an event record automaton.

12 Concluding Remarks

In this paper, we focused on the problem of BDD-based automated addition of masking fault-tolerance to distributed programs. We showed that although the synthesis problem is NP-complete in the size of the input program's state space, the high complexity can be overcome through devising efficient heuristics and effective implementation. In particular, we demonstrated that synthesizing moderate-sized distributed programs (reachable states of size 10^{50} and beyond) is feasible in reasonable amount of time and space. Our analysis also shows that the growth of the time complexity is sublinear in the state space. Moreover, we demonstrated that through incorporating efficient heuristics, the growth of time complexity is comparable to that of model checking.

In addition to demonstrate feasibility of synthesizing moderate-sized fault-tolerant distributed programs, we made the following observation through conducting several case studies:

1. Although the growth of the time complexity is sublinear in the state space and is comparable to that of model checking, in general, synthesis algorithms tend to run out of time before they run out of space. Thus, we believe sacrificing a little bit of space in order to achieve speed-ups is a reasonable way to remedy the time complexity of synthesis decision procedures.
2. We observed that the state explosion problem by itself is not the sole obstacle in program synthesis. In particular, we identified different bottlenecks of synthesis depending upon the input program's structure. These bottlenecks are namely, deadlock resolution (i.e., state elimination), cycle detection and resolution, fault-span generation, and identifying recovery computations. Thus, depending upon the structure of the input program, our synthesis algorithm may suffer from a subset of the aforementioned bottlenecks.
3. We also observed that small human knowledge can dramatically improve the performance of our synthesis algorithm. For instance, we showed that adding a simple condition to the safety specification in order to constrain the problem space or enforcing a particular recovery mechanism reduces the total synthesis time drastically.

Based on the lessons learned from our experiments, we categorize open problems and suggest a comprehensive roadmap for further research as follows:

- In our implementation, the Procedure FWReachableStates is implemented simply by next-state relation. This approach is efficient for cases where the size of BDDs are small. However, as soon as the size of BDDs become larger, next-state reachability analysis can be as bad as enumerative methods. Hence, we are planning to incorporate more recent symbolic techniques such as partitioning [19], clustering [52], and saturation-based reachability analysis [53,54] in our current implementation. These techniques will certainly improve computation of state predicates such as program invariant and fault-span.

  Since we add and remove transitions and states during the course of synthesis, 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, another open problem is to develop algorithms that reuse the existing fault-span from previous iterations and revise it by removing unreachable states.
- Another future work is to develop more efficient methods for deadlock resolution, which turns out to be one of the most serious bottlenecks of addition of fault-tolerance. Note that deadlock resolution (in the sense presented in Section 5) is a problem that exists in the context of program synthesis and transformation and, hence, has not been addressed by the model checking community. Dealing with bottlenecks of course includes developing efficient algorithms for other issues such as cycle detection and resolution as well.
- Distributed programs often consist of processes with the same or similar structure. Thus, an interesting problem is to exploit the symmetry in distributed programs to reduce the synthesis time using symmetry reduction techniques [55–57].
- 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 modelled as a simple satisfiability problem [23], we expect that integrating our implementation with a SAT or SMT (satisfiability modulo theories) solver is beneficial. In SMT solvers (e.g., Yices [58]), 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.
- The BDD data structure 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 BDD 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 so that it satisfies a desired property.

In other words, since the structure of a program changes during synthesis, reordering the variables of BDDs dynamically may be beneficial. Nevertheless, there is a trade-off between the time spent to reorder variables 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 variable reordering is beneficial.

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

References

1. Lee, E.A.: Cyber-physical systems - are computing foundations adequate? In: Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap (2006)
2. Stankovic, J.A., Lee, I., Mok, A.K., Rajkumar, R.: Opportunities and obligations for physical computing systems. IEEE Computers **38**(11), 23–31 (2005)
3. Kulkarni, S.S., Arora, A., Chippada, A.: Polynomial time synthesis of Byzantine agreement. In: Symposium on Reliable Distributed Systems (SRDS), pp. 130–140 (2001)
4. Ebnenasir, A., Kulkarni, S.S., Arora, A.: FTSyn: a framework for automatic synthesis of fault-tolerance. International Journal of Software Tools for Technology Transfer (STTT) **10**(5), 455–471 (2008)
5. Bonakdarpour, B., Kulkarni, S.S.: Revising distributed UNITY programs is NP-complete. In: Principles of Distributed Systems (OPODIS), pp. 408–427 (2008)
6. Kulkarni, S.S., Arora, A.: Automating the addition of fault-tolerance. In: Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT), pp. 82–93 (2000)
7. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers **35**(8), 677–691 (1986)
8. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Information and Computation **98**(2), 142–170 (1992)
9. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems (TOPLAS) **4**(3), 382–401 (1982)
10. Arora, A., Kulkarni, S.S.: Component based design of multitolerant systems. IEEE Transactions on Software Engineering **24**(1), 63–78 (1998)
11. Schlichting, R.D., Schneider, F.B.: Fail-stop processors: An approach to designing fault-tolerant computing systems. ACM Transactions on Computers **1**(3), 222–238 (1983)
12. Kulkarni, S.S., Arumugam, M.: Infuse: A TDMA based data dissemination protocol for sensor networks. International Journal on Distributed Sensor Networks (IJDSN) **2**(1), 55–78 (2006)
13. Bonakdarpour, B., Kulkarni, S.S.: SYCRAFT: A tool for synthesizing fault-tolerant distributed programs. In: Concurrency Theory (CONCUR), pp. 167–171 (2008)
14. Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters **21**, 181–185 (1985)
15. Arora, A., Gouda, M.G.: Closure and convergence: A foundation of fault-tolerant computing. IEEE Transactions on Software Engineering **19**(11), 1015–1027 (1993)
16. Tripakis, S.: Undecidable problems of decentralized observation and control on regular languages. Information Processing Letters **90**(1), 21–28 (2004)
17. Kulkarni, S.S., Ebnenasir, A.: Adding fault-tolerance using pre-synthesized components. In: European Dependable Computing Conference (EDCC), pp. 72–90 (2005)
18. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
19. Burch, J., Clarke, E., Long, D.: Symbolic model checking with partitioned transition relations. In: International Conference on Very Large Scale Integration, pp. 49–58 (1991)
20. Fisler, K., Fraer, R., Kamhi, G., Vardi, M.Y., Yang, Z.: Is there a best symbolic cycle-detection algorithm? In: In Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 420–434 (2001)
21. Emerson, E.A., Lei, C.L.: Efficient model checking in fragments of the propositional model mu-calculus. In: Logic in Computer Science (LICS), pp. 267–278 (1986)
22. Somenzi, F.: CUDD: Colorado University Decision Diagram Package. `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`
23. Ebnenasir, A.: DiConic addition of failsafe fault-tolerance. In: Automated Software Engineering (ASE), pp. 44–53 (2007)
24. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NUSMV: A new symbolic model checker. Software Tools for Technology Transfer (STTT) **2**(4), 410–425 (2000)
25. Bonakdarpour, B., Kulkarni, S.S.: Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In: IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 3–10 (2007)
26. Ramadge, P.J., Wonham, W.M.: The control of discrete event systems. Proceedings of the IEEE **77**(1), 81–98 (1989)
27. Cho, K.H., Lim, J.T.: Synthesis of fault-tolerant supervisor for automated manufacturing systems: A case study on photolithography process. IEEE Transactions on Robotics and Automation **14**(2), 348–351 (1998)
28. Girault, A., Rutten, É.: Automating the addition of fault tolerance with discrete controller synthesis. Formal Methods in System Design (FMSD) **35**(2), 190–225 (2009)
29. Gohari, P., Wonham, W.M.: On the complexity of supervisory control design in the RW framework. IEEE Transactions on Systems, Man, and Cybernetics **30**(5), 643–652 (2000)
30. Bouyer, P., D'Souza, D., Madhusudan, P., Petit, A.: Timed control with partial observability. In: Computer Aided Verification (CAV), pp. 180–192 (2003)
31. D'Souza, D., Madhusudan, P.: Timed control synthesis for external specifications. In: Symposium on Theoretical Aspects of Computer Science (STACS), pp. 571–582 (2002)
32. Asarin, E., Maler, O.: As soon as possible: Time optimal control for timed automata. In: Hybrid Systems: Computation and Control (HSCC), pp. 19–30 (1999)
33. Asarin, E., Maler, O., Pnueli, A., Sifakis, J.: Controller synthesis for timed automata. In: IFAC Symposium on System Structure and Control, pp. 469–474 (1998)
34. Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems. In: 12th Annual Symposium on Theoretical Aspects of Computer Science (STACS), pp. 229–242 (1995)
35. Tripakis, S., Altisen, K.: On-the-fly controller synthesis for discrete and dense time systems. In: Formal Methods 1999 (FM), pp. 233–252 (1999)
36. Lin, F., Wonham, W.M.: Decentralized control and coordination of discrete-event systems with partial observation. IEEE Transactions On Automatic Control **35**(12) (1990)

37. Rudie, K., Wonham, W.M.: Think globally, act locally: Decentralized supervisory control. IEEE Transactions On Automatic Control **37**(11), 1692–1708 (1992)
38. Kumar, R., Garg, V.K.: Optimal supervisory control of discrete event dynamicalsystems. SIAM Journal on Control and Optimization **33**(2), 419–439 (1995)
39. Cassez, F., David, A., Fleury, E., Larsen, K., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Concurrency Theory (CONCUR), pp. 66–80 (2005)
40. Bournai, P., Borgne, M.L., Guernic, P.L.: Synthesis of discrete-event controllers based on the signal environment. In: Discrete Event Dynamic System: Theory and Applications, pp. 325–346 (2000)
41. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Principles of Programming Languages (POPL), pp. 179–190 (1989)
42. Thomas, W.: On the synthesis of strategies in infinite games. In: Theoretical Aspects of Computer Science (STACS), pp. 1–13 (1995)
43. Wallmeier, N., Hütten, P., Thomas, W.: Symbolic synthesis of finite-state controllers for request-response specifications. In: Implementation and Application of Automata (CIAA), pp. 11–22 (2003)
44. Thomas, W.: Handbook of Theoretical Computer Science, vol. B, chap. 4: Automata on Infinite Objects, pp. 133–192. Elsevier Science Publishers B. V., Amsterdam (1990)
45. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: International Colloqium on Automata, Languages, and Programming (ICALP), pp. 652–671 (1989)
46. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Information and Computation **111**(2), 193–244 (1994)
47. Kulkarni, S.S., Ebnenasir, A.: Automated synthesis of multitolerance. In: International Conference on Dependable Systems and Networks (DSN), pp. 209–219 (2004)
48. Bonakdarpour, B., Kulkarni, S.S.: Incremental synthesis of fault-tolerant real-time programs. In: International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), LNCS 4280, pp. 122–136 (2006)
49. Bonakdarpour, B., Kulkarni, S.S.: Masking faults while providing bounded-time phased recovery. In: International Symposium on Formal Methods (FM), pp. 374–389 (2008)
50. Tripakis, S.: Fault diagnosis for timed automata. In: Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT), pp. 205–224 (2002)
51. Bouyer, P., Chevalier, F., D'Souza, D.: Fault diagnosis using timed automata. In: Foundations of Software Science and Computation Structure, pp. 219–233 (2005)
52. Ranjan, R., Aziz, A., Brayton, R., Plessier, B., Pixley, C.: Efficient BDD algorithms for FSM synthesis and verification. In: IEEE/ACM International Workshop on Logic Synthesis (1995)
53. Ciardo, G., Lüttgen, G., Siminiceanu, R.: Saturation: An efficient iteration strategy for symbolic state-space generation. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 328–342 (2001)
54. Ciardo, G., Yu, A.J.: Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In: Correct Hardware Design and Verification Methods (CHARME), pp. 146–161 (2005)
55. Clarke, E.M., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. In: Computer Aided Verification (CAV), pp. 450–462 (1993)
56. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. Formal Methods in System Design: An International Journal **9**(1/2), 105–131 (1996)
57. Attie, P., Emerson, E.A.: Synthesis of concurrent systems with many similar processes. ACM Transactions on Programming Languages and Systems (TOPLAS) **20**(1), 51–115 (1998)
58. Yices: An SMT Solver. `http://yices.csl.sri.com`

Appendix

A Summary of Notations

V	set of variables
D	domain of variables
s	state
$\overline{s}$	computation
$\mathcal{S}$	state space
T_p	transition predicate of process p
W_p	set of variables that process p can write
R_p	set of variables that process p can read
$\mathcal{P}$	distributed program
$\Pi_{\mathcal{P}}$	processes of program $\mathcal{P}$
$T_{\mathcal{P}}$	transition predicate of program $\mathcal{P}$
I	invariant predicates
S	fault-span
F	fault transition predicate
$T\|S$	projection of T on S
$\mathcal{BA}$	Byzantine agreement
$\mathcal{BAFS}$	Byzantine agreement with fail-stop faults
$\mathcal{TR}$	token ring
$\mathcal{IF}$	Infuse

B Output of SYCRAFT for Byzantine Agreement

In this appendix, we include the output of the tool SYCRAFT where the input is the fault-intolerant Byzantine agreement program with three non-general processes. These processes are labeled as 0, 1, and 2. The variable naming conforms with that of in Section 6.

```
----------------------------------------------------------------
UNCHANGED ACTIONS:
----------------------------------------------------------------
1-((d0==2) & !(f0==1)) & !(b0==1)         -->  (d0 := dg)
----------------------------------------------------------------
REVISED ACTIONS:
----------------------------------------------------------------
2-(b0==0) & (d0==1) & (d1==1) & (f0==0)   -->  (f0 := 1)
3-(b0==0) & (d0==0) & (d2==0) & (f0==0)   -->  (f0 := 1)
4-(b0==0) & (d0==0) & (d1==0) & (f0==0)   -->  (f0 := 1)
5-(b0==0) & (d0==1) & (d2==1) & (f0==0)   -->  (f0 := 1)
----------------------------------------------------------------
NEW RECOVERY ACTIONS:
----------------------------------------------------------------
6-(b0==0)&(d0==0)&(d1==1)&(d2==1)&(f0==0) -->  (d0 := 1)
7-(b0==0)&(d0==1)&(d1==0)&(d2==0)&(f0==0) -->  (d0 := 0)
8-(b0==0)&(d0==0)&(d1==1)&(d2==1)&(f0==0) -->  (d0 := 1),(f0 := 1)
9-(b0==0)&(d0==1)&(d1==0)&(d2==0)&(f0==0) -->  (d0 := 0),(f0 := 1)
----------------------------------------------------------------
```