Int J Softw Tools Technol Transfer (2008) 10:455–471
DOI 10.1007/s10009-008-0083-0

REGULAR CONTRIBUTION

FTSyn: a framework for automatic synthesis of fault-tolerance

Ali Ebnenasir · Sandeep S. Kulkarni · Anish Arora

Published online: 31 July 2008

Abstract In this paper, we present a software framework for adding fault-tolerance to existing finite-state programs. The input to our framework is a fault-intolerant program and a class of faults that perturbs the program. The output of our framework is a fault-tolerant version of the input program. Our framework provides (1) the first automated tool for the synthesis of fault-tolerant *distributed* programs, and (2) an extensible platform for researchers to develop a repository of heuristics that deal with the complexity of adding fault-tolerance to *distributed* programs. We also present a set of heuristics for polynomial-time addition of fault-tolerance to distributed programs. We have used this framework for automated synthesis of several fault-tolerant programs including a simplified version of an aircraft altitude switch, token ring, Byzantine agreement, and agreement in the presence of Byzantine and fail-stop faults. These examples illustrate that our framework can be used for synthesizing programs that tolerate different types of faults (process restarts, Byzantine and fail-stop) and programs that are subject to multiple faults (Byzantine and fail-stop) simultaneously. We have found our framework to be highly useful for pedagogical purposes, especially for teaching concepts of fault-tolerance, automatic program transformation, and the effect of heuristics.

A. Ebnenasir
Computer Science Department, Michigan Technological University, Houghton, MI 49931, USA
e-mail: aebnenas@mtu.edu

S. S. Kulkarni (✉)
Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824, USA
e-mail: sandeep@cse.msu.edu

A. Arora
Department of Computer Science and Engineering, Ohio State University, Columbus, OH 43210, USA
e-mail: anish@cse.ohio-state.edu

Keywords Fault-tolerance · Automatic addition of fault-tolerance · Formal methods · Program synthesis · Distributed programs

1 Introduction

In the initial design of a fault-tolerant program, it is often difficult to identify all the faults that may perturb the program. Thus, when new faults that affect an existing program are identified, it becomes necessary to upgrade the system to deal with those new faults. Moreover, during such addition of fault-tolerance, it is necessary to reuse the existing program as much as possible. Specifically, when the new fault *does not* occur, we expect the program to behave in the same way as it behaved before the upgrade.

It is desirable to use an automated synthesis algorithm while adding fault-tolerance to a program as the synthesized program is correct by construction, and there will be no need for its proof of correctness. To automatically synthesize a fault-tolerant program, we can begin either with its formal specification [6–9,19], or with (the transitions of) the fault-intolerant version thereof [27,28]. In the context where we need to upgrade an existing program, we follow the latter approach and reuse the existing program.

One of the difficulties in automating the addition of fault-tolerance to *distributed* programs is the complexity of such addition. In [27,31], the authors have shown that, in general, the addition of fault-tolerance to distributed programs is NP-complete in the state space of the fault-intolerant program. To deal with this complexity and to synthesize programs that have large state space, heuristic-based approaches are proposed in [21,28,29]. These heuristic-based approaches reduce the complexity of synthesis by forfeiting the completeness of adding fault-tolerance (see Sect. 4 for the description of heuristics). In other words, if heuristics are applicable then a heuristic-based algorithm will generate a fault-tolerant program in polynomial-time (in the state space of the fault-intolerant program). However, if the heuristics are not applicable then the algorithm may declare failure even though it is possible to add fault-tolerance to the given fault-intolerant program.

The development and the application of heuristics is complicated by the fact that, for a given set of heuristics, we need to determine which combinations of those heuristics are applicable in the synthesis of a fault-tolerant program. In [28], Kulkarni et al. apply their heuristics for the synthesis of a canonical version of Byzantine agreement program by careful exploration of reachable states and transitions. However, for programs with very large state space, such *manual* exploration of state space is not practical. This problem becomes even more challenging where a combination of heuristics fails to be applicable in the synthesis of a fault-tolerant program and developers need to determine what went wrong during the synthesis. To address such questions, an extensible software tool is necessary where we provide (1) the ability to apply different combinations of a set of heuristics; (2) the ability to add new heuristics, and (3) the ability for automatic generation of meaningful representation of the cases where the addition of fault-tolerance fails.

Goals. To address the above-mentioned issues, we develop a software framework, called Fault-Tolerance Synthesizer (FTSyn), for the synthesis of fault-tolerant programs from their fault-intolerant version. FTSyn has the following properties:

1. *Ability to add fault-tolerance to existing fault-intolerant programs.* One group of users who use FTSyn are the *developers of fault-tolerant programs*. For this group of users, FTSyn should provide mechanisms for the addition of fault-tolerance. Thus, at different stages in the synthesis of a fault-tolerant program, these developers should be able to interact with FTSyn in order to apply different built-in heuristics (in their desired order) depending on the program being synthesized.

   In the cases where the application of heuristics fails, developers should be able to determine the cause of failure. In other words, they should be able to query FTSyn to obtain meaningful representations for the failure cases. Towards this end, they must be able to automatically generate the intermediate versions of the program being synthesized, and automatically identify counterexamples of desired fault-tolerance properties. This way, they can determine the heuristics that should be applied next.
2. *Ability to add new heuristics.* Another group of users are the *developers of heuristics* who need to evaluate the applicability of their new heuristics in reducing the complexity of fault-tolerance addition. In order to increase the efficiency of synthesis, the developers of heuristics may need to *improve* the existing heuristics or *add* new heuristics for different tasks during synthesis. Thus, FTSyn should allow improvements or additions of heuristics with a low overhead. In other words, FTSyn should be *extensible*.
3. *Ability to change internal representations.* The internal representation of entities such as programs and faults affects the efficiency of the synthesis of fault-tolerant programs. It is difficult to determine the ideal internal representation of these entities as each representation has its own advantages and disadvantages. Moreover, depending on the user requirements at run-time, FTSyn should switch between different internal representations of a particular entity. Hence, we should be able to modify the way these entities are represented with a low overhead.

Contributions of the paper. The main contributions of the paper are as follows:

- We extend the scope of program synthesis by designing and implementing an extensible software framework, called FTSyn, for adding fault-tolerance to existing distributed programs.
- We provide the possibility of changing the internal representation of different entities in FTSyn. Towards this end, we effectively utilize some of the design patterns in [20]. We note that the identification of appropriate implementation structures for programs, faults, and specifications is a research problem that is outside the scope of this paper. Nonetheless, we provide the necessary software framework for such investigation.
- We provide the option of obtaining an intermediate version of the synthesized program in Promela [2] modeling language; this option is especially useful if the heuristics being used fail, and it becomes necessary to analyze the intermediate version of the synthesized program to identify if another heuristic could be used or how a new heuristic can be developed.
- We note that FTSyn provides a suitable platform for teaching some basic concepts of distributed and fault-tolerant systems (e.g., distribution issues, non-determinism, faults, and fault tolerance); i.e., FTSyn is used for pedagogical purposes as well.

We have used FTSyn to synthesize several fault-tolerant programs among them (1) a simplified version of an altitude switch that tolerates the corruption of altitude sensors; (2) a token ring protocol that tolerates process-restart faults; (3) an agreement protocol that tolerates Byzantine faults; (4) an agreement program that tolerates both Byzantine faults and fail-stop faults; (5) an alternating bit protocol program that tolerates message-loss faults, and (6) a Triple Modular Redundancy program that tolerates input-corruption faults. These examples illustrate the potential of FTSyn in adding tolerance against faults of different natures (e.g., input-corruption, Byzantine, message loss, etc.).

Organization of the paper. In Sect. 2, we present preliminary concepts. Then, in Sect. 3, we provide an overview of how developers of fault-tolerance can interact with FTSyn to add fault-tolerance to programs. We also use an example program to demonstrate the input and the output of FTSyn. Subsequently, in Sect. 4, we present the theory behind the internal working of FTSyn, where we present heuristics applied during synthesis. In Sect. 5, we show how one can integrate new heuristics in FTSyn and can change the internal representation of the components of FTSyn. In Sect. 6, we present a simplified version of an altitude switch synthesized using FTSyn. We discuss scalability issues related to our framework in Sect. 7. We make concluding remarks and discuss future work in Sect. 8. We note that a detailed user manual including the source code of FTSyn is available in [1].

2 Preliminaries

In this section, we present the theoretical background on which FTSyn is based. We present basic concepts in Sect. 2.1 and then, in Sect. 2.2, we recall the problem statement of adding fault-tolerance to programs. Finally, in Sect. 2.3, we represent a non-deterministic algorithm from [27,28] for adding fault-tolerance to distributed programs.

2.1 Basic concepts

In this section, we give the definitions of programs, problem specifications, state predicates, faults, and fault-tolerance. The programs are defined in terms of their state space and their transitions. The definition of specifications is adapted from Alpern and Schneider [3]. The definition of faults and fault-tolerance is adapted from Arora and Gouda [4] and Kulkarni [26]. The issues of modeling distributed programs are adapted from Kulkarni and Arora [27], and Attie and Emerson [7].

Program. A program p is defined by a finite set of variables, say $V = \{v_1, \ldots, v_u\}$, and a finite set of processes, say $P = \{P_1, \ldots, P_n\}$, where u and n are positive integers. Each variable is associated with a finite domain of values. Let $v_1, v_2, \ldots, v_u$ be variables of p, and let $D_1, D_2, \ldots, D_u$ be their respective domains. A state of p is obtained by assigning each variable a value from its respective domain. Thus, a state s of p has the form: $\langle l_1, l_2, \ldots, l_u \rangle$ where $\forall i : 1 \le i \le u : l_i \in D_i$. The state space of p, S_p, is the set of all possible states of p.

A process, say P_j $(1 \le j \le n)$, in p is associated with a set of program variables, say r_j, that P_j can read and a set of variables, say w_j, that P_j can write. We assume that $w_j \subseteq r_j$, i.e., P_j cannot *blindly* write any variable. Process P_j consists of a set of transitions δ_j; each transition is of the form (s_0, s_1) where $s_0, s_1 \in S_p$. Later in this section, we address the effect of read/write restrictions on δ_j. The set of transitions of p, δ_p, is equal to the union of the transitions of its processes.

A state predicate of p is any subset of S_p. A state predicate S is closed in the program p (respectively, δ_p) iff (if and only if) the following condition holds:

$$\forall s_0, s_1 :: (((s_0, s_1) \in \delta_p) \wedge (s_0 \in S)) \Rightarrow (s_1 \in S).$$

A sequence of states, $\sigma = \langle s_0, s_1, \ldots \rangle$ with $len(\sigma)$ states, is a computation of p iff the following two conditions are satisfied: (1) $\forall j : 0 < j < len(\sigma) : (s_{j-1}, s_j) \in \delta_p$ and (2) if σ is finite and terminates in a state s_l then there does not exist a state s such that $(s_l, s) \in \delta_p$ (note that $len(\sigma)$ could be infinity). A finite sequence of states, $\langle s_0, s_1, \ldots, s_k \rangle$, is a computation prefix of p iff $\forall j : 0 < j \le k : (s_{j-1}, s_j) \in \delta_p$, where k is a positive integer.

The projection of program p on a state predicate S, denoted as $p|S$, is the set of transitions $\{(s_0, s_1) : (s_0, s_1) \in \delta_p \wedge s_0, s_1 \in S\}$; i.e., $p|S$ consists of the transitions of p that start in S and end in S.

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

Specification. A specification is a set of infinite sequences of states that is suffix-closed and fusion-closed. Suffix closure of a set means that if a state sequence σ is in that set then so are all the suffixes of σ. Fusion closure of a set means that if state sequences $\langle \alpha, s, \gamma \rangle$ and $\langle \beta, s, \delta \rangle$ are in that set then so are the state sequences $\langle \alpha, s, \delta \rangle$ and $\langle \beta, s, \gamma \rangle$, where α and β are finite prefixes of state sequences, γ and δ are suffixes of state sequences, and s is a program state.

Following Alpern and Schneider [3], we rewrite a specification as a conjunction of a safety specification and a liveness specification. Since the specification is suffix-closed and fusion-closed, it is possible to represent the safety specification of a program as a set of bad transitions that the program is not allowed to execute (see P. 26, Lemma 3.6 of [26] for proof). Thus, for program p, its safety specification is a subset of $S_p \times S_p$. We do not explicitly specify the liveness specification as we show that the fault-tolerant program satisfies the liveness specification (in the absence of faults) iff the fault-intolerant program satisfies the liveness specification.

Given a program p, a state predicate S, and a specification $spec$, we say that p satisfies $spec$ from S iff (1) S is closed in p, and (2) every computation of p that starts in a state where S is true is in $spec$. If p satisfies $spec$ from S and $S \neq \{\}$, we say that S is an invariant of p for $spec$. For a finite sequence (of states) α, we say that α maintains (does not violate) $spec$ iff there exists an infinite sequence of states β such that $\alpha\beta \in spec$. We say that p maintains (does not violate) $spec$ from S iff (1) S is closed in p, and (2) every computation prefix of p that starts in a state in S maintains $spec$. Note that the definition of *maintains* focuses on finite sequences of states, whereas the definition of *satisfies* concentrates on infinite sequences of states. Since a specification is a set of infinite sequences of states, if p satisfies $spec$ from S then all computations of p that start in S must be infinite. However, p may deadlock if it starts in a state that is not in S. Moreover, notice that p is allowed to contain a self-loop of the form (s_0, s_0) inside its invariant S; i.e., it is permissible for p to reach s_0 and remain there forever.

Notation. Let $spec$ be a specification. We use the term safety of $spec$ to mean the smallest safety specification that includes $spec$. Whenever the specification is clear from the context, we will omit it; thus, S is an invariant of p abbreviates S is an invariant of p for $spec$.

Distribution model. We identify how read/write restrictions on a process affect its transitions. Given a transition (s_0, s_1), it is straightforward to determine the variables that need to be changed in order to transition from state s_0 to s_1. Specifically, if $x(s_0)$ denotes the value of x in state s_0 and $x(s_1)$ denotes the value of x in state s_1 then we say that (s_0, s_1) writes the value of x iff $x(s_0) \neq x(s_1)$. Thus, the write restrictions amount to ensuring that the transitions of a process only modify those variables that it can write. More specifically, if process P_j can only write the variables in w_j and the value of a variable other than that in w_j is changed in the transition (s_0, s_1) then (s_0, s_1) cannot be used in obtaining the transitions of P_j. In other words, if P_j can write only the variables in w_j then P_j cannot use the transitions in $nw(w_j)$, where $nw(w_j) = \{(s_0, s_1) : (\exists x : x \notin w_j : x(s_0) \neq x(s_1))\}$.

Read restrictions require us to *group* transitions and ensure that the entire group is included or the entire group is excluded (the idea of grouping has also appeared in previous work [7,27]). As an example, consider a program consisting of variables a and b and let their domain be $\{0, 1\}$. Moreover, consider a process that cannot read the variable a. We can think of the transition from the state $\langle a = 0, b = 0\rangle$ to the state $\langle a = 0, b = 1\rangle$ as an atomic *if* statement 'if a is 0 and b is 0 then set b to 1'. In this case, the process must read a. However, if we also include the transition from the state $\langle a = 1, b = 0\rangle$ to the state $\langle a = 1, b = 1\rangle$ then these two transitions can be thought of as 'if b is 0 then set b to 1'. In other words, the inability to read causes the transitions $(\langle a=0, b=0\rangle, \langle a=0, b=1\rangle)$ and $(\langle a=1, b=0\rangle, \langle a=1, b=1\rangle)$ to be grouped. In the set of transitions of a process, we need to include all transitions in this group or exclude all of them.

More generally, consider the case where r_j is the set of variables that P_j can read, w_j is the set of variables that P_j can write, and $w_j \subseteq r_j$. Process P_j can include the transition (s_0, s_1) iff P_j also includes the transition (s'_0, s'_1) where s_0 (respectively, s_1) and s'_0 (respectively, s'_1) are identical as far as the variables in r_j are concerned, and s_0 (respectively, s'_0) and s_1 (respectively, s'_1) are identical as far as the variables *not* in r_j are considered. We define these transitions as $group(r_j)_{(s_0,s_1)}$ for the case $w_j \subseteq r_j$, where

$$\begin{aligned} group(r_j)_{(s_0,s_1)} = \{(s'_0, s'_1) :& \\ &(\forall x : x \in r_j : x(s_0) = x(s'_0) \;\; \wedge x(s_1) = x(s'_1)) \;\wedge \\ &(\forall x : x \notin r_j : x(s'_0) = x(s'_1) \;\wedge\; x(s_0) = x(s_1)) \quad \} \end{aligned}$$

The grouping introduced by the read restrictions increases the complexity of synthesizing distributed programs [27,31]. To clarify this, we consider the case where transitions (s_0, s_1) and (s'_0, s'_1) are grouped together, (s_0, s_1) is a desirable transition (e.g., because it is used to satisfy the specification in the absence of faults), and (s'_0, s'_1) should never be executed (e.g., because it violates the safety specification). In this scenario, we are faced with two choices (1) include this group and ensure that s'_0 is never reached, or (2) exclude this group and lose the useful transition (s_0, s_1). Thus, we need to perform a tradeoff between states and transitions. The authors of [27] have used this crucial fact to show that adding fault-tolerance to distributed programs is NP-hard. One approach to deal with such exponential complexity is to design heuristics that deterministically identify which transition (respectively, state) can be included in the fault-tolerant program (see Sect. 4 for examples of such heuristics).

Faults. We systematically represent the faults that a program is subject to by a set of transitions. Thus, a class of fault f for program p is a subset of the set $S_p \times S_p$. We use $p[]f$ to denote the transitions obtained by taking the union of the transitions in p and the transitions in f. We say that a state predicate T is an f-span (read as fault-span) of p from S iff the following two conditions are satisfied: (1) $S \subseteq T$ (equivalently, $S \Rightarrow T$), and (2) T is closed in $p[]f$. Thus, at each state where an invariant S of p is true, an f-span T of p from S is also true. The state predicate T, similar to S, is closed in p. Moreover, if any transition in f is executed in a state where T is true, then T is also true in the resulting state. It follows that for all computations of p that start at states where S is true, T is a boundary in the state space of p to which (but not beyond which) the state of p may be perturbed by the occurrence of the transitions in f.

As we defined a computation of p, we say that a sequence of states, $\sigma = \langle s_0, s_1, \ldots\rangle$ with $len(\sigma)$ states, is a computation of p in the presence of f iff the following three conditions are satisfied: (1) $\forall j : 0 < j < len(\sigma) : (s_{j-1}, s_j) \in (\delta_p \cup f)$, (2) if σ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in \delta_p$, and (3) $\exists n : n \geq 0 : (\forall j : j > n : (s_{j-1}, s_j) \in \delta_p)$. The first requirement captures that in each step, either a program transition or a fault transition is executed. The second requirement captures that faults do not have to execute, i.e., if the program reaches a state where only a fault transition can be executed, it is not required that the fault transition be executed. It follows that fault transitions cannot be used to deal with deadlocked states. Finally, the third requirement captures that the number of fault occurrences in a computation is finite. Such requirement also appears in previous work [4,5,13,34] in order to ensure that eventually recovery can occur.

Fault-tolerance. We say that p is masking f-tolerant to $spec$ from S iff the following two conditions hold: (1) in the absence of f, p satisfies $spec$ from S, and (2) there exists T such that (a) T is an f-span of p from S, (b) $p[]f$ maintains $spec$ from T, and (c) every computation of $p[]f$ that starts from a state in T has a state in S.

2.2 Problem statement for addition of fault-tolerance

For a given class of faults f, the objective of the addition of fault-tolerance to an existing fault-intolerant program p is to ensure no new behaviors are added in the absence of f and to add the necessary fault-tolerance behaviors in the presence of f. Let S be an invariant of p from where p satisfies its specification $spec$. Also, let p' be the program derived by adding fault-tolerance to p and let S' be the invariant of p'. If S' includes states that are not in S then, in the absence of faults, the computations of p' may reach such states and generate new ways for satisfying $spec$. Also, a similar case will occur if $p' \mid S'$ includes transitions that do not belong to $p \mid S'$. Hence, we require that $S' \subseteq S$ and $p'|S' \subseteq p|S'$. Thus, the problem of fault-tolerance addition is defined as follows (from [27]):

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

2.3 Non-deterministic synthesis algorithm for distributed programs

Kulkarni and Arora [27] show that the addition of masking fault-tolerance to distributed programs is NP-complete (in program state space). They present a non-deterministic polynomial algorithm in [27,28] for the addition of fault-tolerance to distributed programs. We repeat this algorithm in Fig. 1 since the implementation of FTSyn is based on a deterministic version of this algorithm.

The Add_ft algorithm (see Fig. 1) first computes a set of states, denoted ms (i.e., *marked states*), from where safety can be violated by the execution of fault transitions alone. Thus, the fault-tolerant program should not reach a state in ms. Then, it computes a set of transitions, denoted mt (i.e., *marked transitions*), that violate safety or reach a state in ms. It follows that a fault-tolerant program should not execute a transition in mt. Then, the Add_ft algorithm non-deterministically guesses the fault-tolerant program, p', its invariant, S' and its fault-span, T'. Finally, the algorithm verifies that the guessed fault-tolerant program satisfies the three conditions of the addition problem (see Sect. 2.2). This goal is achieved by verifying the six formulas $F1$–$F6$.

3 FTSyn overview

In this subsection, we explain how developers of fault-tolerance should prepare the input to FTSyn and how FTSyn provides the output to its users. The input of FTSyn consists of the fault-intolerant program, its invariant, its safety specification, its initial states, and a class of faults (see Fig. 2).

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

    Guess $S', T'$, and $p' := \bigcup (g_i : g_i$ is chosen to be included in the fault-tolerant program);
    Verify the following
        $(F1)$ $p'|S' \subseteq p|S'$;
        $(F2)$ $S' \Rightarrow T'$; $T'$ is closed in $p'[]f$;                  // $T'$ is a fault-span of $p'$.
        $(F3)$ $T' \cap ms \ = \ \{\}$; $(p'|T') \cap mt \ = \ \{\}$;    // Safety cannot be violated from states in $T'$.
        $(F4)$ $(\forall s_0 : s_0 \in T' : (\exists s_1 :: (s_0, s_1) \in p'))$;    // $T'$ does not have deadlocks.
        $(F5)$ $S' \neq \{\}$; $S' \subseteq S$; $S'$ is closed in $p'$;          // $S'$ is an invariant of $p'$.
        $(F6)$ $p'|(T' - S')$ is acyclic;                             // $p'$ cannot stay in $(T' - S')$ forever.
}
```

Fig. 1 The non-deterministic algorithm for automatic addition of fault-tolerance to distributed programs

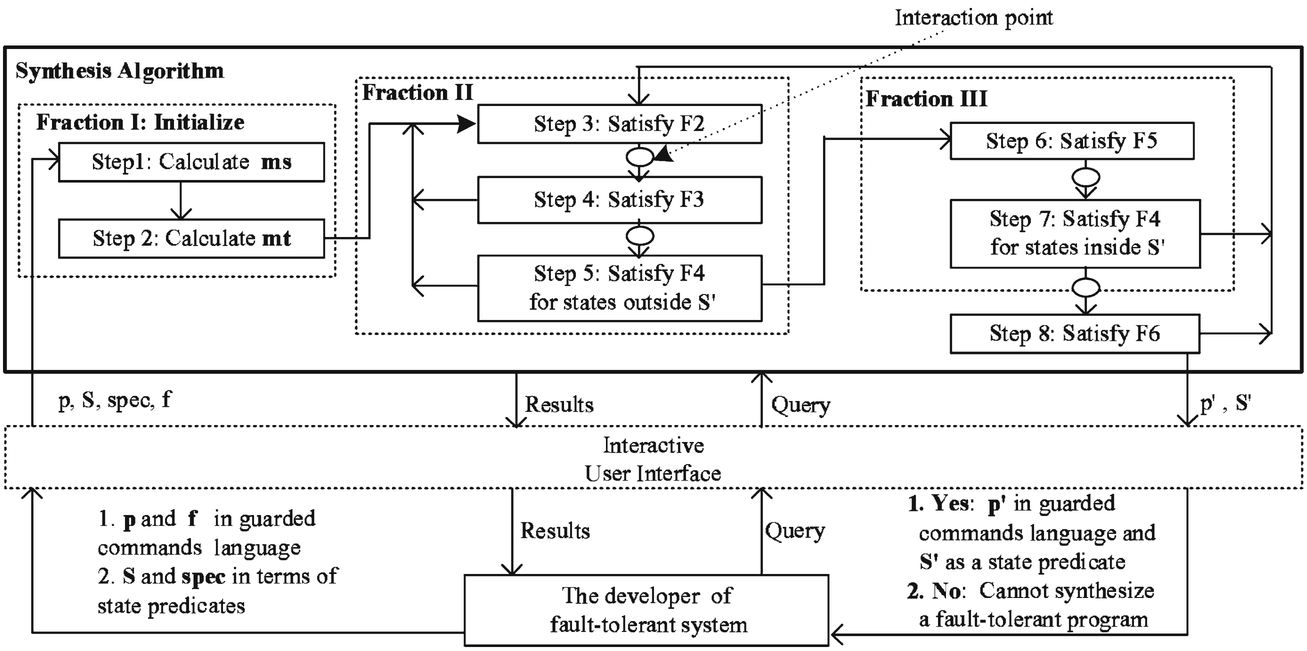
Interaction point
Synthesis Algorithm
Fraction I: Initialize
Step1: Calculate ms
Step 2: Calculate mt
Fraction II
Step 3: Satisfy F2
Step 4: Satisfy F3
Step 5: Satisfy F4 for states outside S'
Fraction III
Step 6: Satisfy F5
Step 7: Satisfy F4 for states inside S'
Step 8: Satisfy F6
p, S, spec, f
Results
Query
p' , S'
Interactive User Interface
1. p and f in guarded commands language
2. S and spec in terms of state predicates
Results
Query
The developer of fault-tolerant system
1. Yes: p' in guarded commands language and S' as a state predicate
2. No: Cannot synthesize a fault-tolerant program

Fig. 2 Deterministic execution of FTSyn

We represent the input fault-intolerant program by Dijkstra's guarded commands [14]. A guarded command (action) is of the form $grd \rightarrow st$, where grd is a state predicate and st is a statement that updates the program variables. The guarded command $grd \rightarrow st$ includes all program transitions $\{(s_0, s_1) : grd$ holds at s_0 and the *atomic* execution of st at s_0 takes the program to state $s_1\}$. In other words, we use guarded commands as a shorthand for representing the set of program transitions. The output of FTSyn is also represented by guarded commands (see Fig. 2).

We note that there exist automated techniques (e.g., [23, 24]) by which one can transform fault-intolerant programs written in common programming languages to the guarded commands language. Moreover, after the synthesis of a fault-tolerant program, there exist tolerance-preserving techniques (e.g., [12,22,33]) that allow us to refine the structure of the synthesized fault-tolerant program (represented in guarded commands).

3.1 Token ring program

In this section, we demonstrate the addition of fault-tolerance to a simple example of a token ring program to illustrate the way developers can communicate with FTSyn to add fault-tolerance. Our goal in this section is to provide an overall picture about the input/output of FTSyn.

The fault-intolerant token ring program consists of four processes P_0, P_1, P_2, and P_3 arranged in a ring. Each process P_i, $0 \leq i \leq 3$, has a variable x_i with the domain $\{-1, 0, 1\}$. We say that process P_i, $1 \leq i \leq 3$, has the token if and only if $(x_i \neq x_{i-1})$ and fault transitions have not corrupted P_i and P_{i-1}. Process P_0 has the token if $(x_3 = x_0)$ and fault transitions have not corrupted P_0 and P_3. Process P_i, $1 \leq i \leq 3$, copies x_{i-1} to x_i if the value of x_i is different from x_{i-1}. This action propagates the token to the next process. If $(x_0 = x_3)$ holds then process P_0 copies the value of $(x_3 \oplus 1)$ to x_0, where $\oplus$ denotes addition in modulo 2. Thus, if we initialize every x_i, $0 \leq i \leq 3$, with 0 then process P_0 has the token and the token circulates along the ring. In the input file of FTSyn, we specify the actions of P_0 as follows (keywords are shown in *italic*):

```
1  process  P0
2  begin
3     (x0 == x3)  ->  x0 = ((x3+1)%2);
4  read  x0, x3;
5  write x0;
6  end
```

While in the input of FTSyn we specify P_1, P_2, and P_3 separately, for the ease of presentation, we present their actions in a parameterized format as follows ($1 \leq i \leq 3$).

```
1  process  Pi
2  begin
3      (xi != x(i-1))  ->  xi = x(i-1);
4  read  xi, x(i-1);
5  write xi;
6  end
```

Read/write restrictions. Each process P_i, $1 \leq i \leq 3$, is only allowed to read x_{i-1} and x_i, and is allowed to write x_i. Process P_0 is allowed to read x_3 and x_0, and to write x_0. We specify the read/write restrictions of a process by *read* and *write* keywords inside the body of the process (see Lines 4 and 5 in the body of P_i).

Faults. Faults are also modeled as a set of guarded commands that change the values of program variables. In the case of the token ring program, faults may corrupt at most three processes. In this example, faults are detectable in that a process that is corrupted can detect if it is in a corrupted state. Hence, we model the fault at process P_i by setting $x_i = -1$. Thus, one of the fault actions that corrupts x_0 is represented as follows:

```
fault  TokenCorruption
begin
 (((x0!=-1)&&(x1!=-1)) || ((x0!=-1)&&(x2!=-1)) ||
  ((x0!=-1)&&(x3!=-1)) || ((x1!=-1)&&(x2!=-1)) ||
  ((x1!=-1)&&(x3!=-1)) || ((x2!=-1)&&(x3!=-1)) )
                                         -> x0 = -1;
end
```

The above fault action stipulates that faults may occur if there exist at least two uncorrupted processes. Note that there exist no read/write restrictions for the fault transitions because we assume that fault transitions can read and write arbitrary program variables.

Safety specification. The safety specification of the fault-intolerant program is represented as a Boolean expression over program variables. In the token ring program, the safety specification stipulates that no non-faulty process is allowed to copy a corrupted value from its predecessor. Note that, in this example, only program transitions may violate safety after faults perturb the state of the program. In the input of FTSyn, we represent the safety specification as follows.

```
(((x1s!=-1)&&(x1d==-1))||((x2s!=-1)&&(x2d==-1))||
 ((x3s!=-1)&&(x3d==-1))||((x3s==-1)&&(x0s!=x0d)))
```

Note that we have added a suffix “s” (respectively, suffix “d”) to variable names that stands for *source* (respectively, *destination*). Since the above condition specifies the set of transitions t_{spec} using their source and destination states, we need to distinguish between the value of a specific variable xi in the source state of t_{spec} (i.e., xis denotes the value of xi in the source state of t_{spec}) and in the destination state of t_{spec} (i.e., xid denotes the value of xi in the destination state of t_{spec}).

Invariant. The invariant is also specified as a Boolean expression over program variables. The invariant of the token ring program consists of the states where no process is corrupted and there exists only one token in the ring.

```
invariant
((x0==1)&&(x1==0)&&(x2==0)&&(x3==0)) ||
((x0==1)&&(x1==1)&&(x2==0)&&(x3==0)) ||
((x0==1)&&(x1==1)&&(x2==1)&&(x3==0)) ||
((x0==1)&&(x1==1)&&(x2==1)&&(x3==1)) ||
((x0==0)&&(x1==0)&&(x2==0)&&(x3==0)) ||
((x0==0)&&(x1==0)&&(x2==0)&&(x3==1)) ||
((x0==0)&&(x1==0)&&(x2==1)&&(x3==1)) ||
((x0==0)&&(x1==1)&&(x2==1)&&(x3==1))
```

Initial states. We also specify some initial states in the input of FTSyn. While these initial states are included in the invariant of the fault-intolerant program, we find that explicitly listing them assists in adding fault-tolerance. The initial states of the token ring program are as follows (*init* and *state* are keywords):

```
init
state x0 = 0; x1 = 0; x2 = 0; x3 = 0;
state x0 = 1; x1 = 1; x2 = 1; x3 = 1;
```

The output fault-tolerant program. The output of FTSyn is also generated in guarded commands. For the token ring program, the actions of process P_0 in the synthesized fault-tolerant program are as follows:

```
(x0==-1) && (x3==1) -> x0 := 0;
 |
(x0==1) && (x3==1)  -> x0 := 0;
 |
(x0==0) && (x3==0)  -> x0 := 1;
 |
(x0==-1) && (x3==0) -> x0 := 1;
```

The above actions mean that P_0 can copy the value of $(x_3 \oplus 1)$ to x_0 as long as $x_3 \neq -1$. We present the actions of other processes in a parameterized format.

```
(xi==1) && (x(i-1)==0)   -> xi := 0;
 |
(xi==-1) && (x(i-1)==0)  -> xi := 0;
 |
(xi==0) && (x(i-1)==1)   -> xi := 1;
 |
(xi==-1) && (x(i-1)==1)  -> xi := 1;
```

The above actions state that each process P_i, for $1 \leq i \leq 3$, can copy the value of x_{i-1} to x_i if $((x_{i-1} \neq -1) \wedge (x_i \neq x_{i-1}))$ holds (i.e., P_{i-1} is not corrupted). We would like to note that the token ring program that we have *automatically* synthesized using FTSyn is the same as the program that was *manually* designed in [26].

3.2 User interactions

Although FTSyn can automatically synthesize a fault-tolerant program without user intervention, there are some situations where (1) user intervention can help to speed up the synthesis of fault-tolerant programs, or (2) a fully automatic approach fails. In this subsection, we present the nature of the interactions that fault-tolerance developers can have with FTSyn.

FTSyn permits developers to semi-automatically supervise the synthesis procedure. In such *supervised synthesis*, fault-tolerance developers interact with FTSyn and apply their insights during the synthesis. In order to achieve this goal, we have devised some *interaction points* (see Fig. 2) where the developers can stop the synthesis algorithm and query it.

At each interaction point, the users can make the following kinds of *queries*: (1) apply a specific heuristic for a particular task; (2) apply some heuristics in a particular order; (3)

view the incoming program (respectively, fault) transitions to a particular state; (4) view the outgoing program (respectively, fault) transitions from a particular state; (5) check the membership of a particular state (respectively, transition) to a specific set of states (respectively, set of transitions); e.g., check the membership of a given state s in the set of ms states, and finally (6) view the intermediate representation of the program that is being synthesized. Since the goal of the paper is to focus on the technical details of FTSyn and its application in adding fault-tolerance, we omit the details about the user interface of FTSyn. We refer the reader to the tutorial about using FTSyn in [1].

While we expect that the queries included in this version will be sufficient for a large class of programs, we also provide an alternative for the cases where the heuristics fail and these queries are insufficient. Specifically, in such cases, the users of FTSyn need to determine what went wrong during synthesis. The answer to this question is very difficult without the help of automated techniques, especially for programs with large state space. To address this issue, developers of fault-tolerance can obtain the corresponding intermediate program in Promela modeling language [2]. This program can then be checked by the SPIN model checker to determine the exact scenario where the intermediate program does not provide the required fault-tolerance property. The counterexamples generated by SPIN help the users to identify the appropriate heuristics that should be applied in the subsequent steps of synthesis.

4 Theoretical background

In this section, we present the underlying theory behind FTSyn. Specifically, we present a set of heuristics integrated in FTSyn (see Fig. 2). We demonstrate how we employ such heuristics while adding fault-tolerance to distributed programs. We use the token ring program introduced in Sect. 3 as a running example throughout this section.

Recall that in the discussion about read restrictions in Sect. 2.1, we argued that while synthesizing distributed fault-tolerant programs, we are faced with the following choice: either ensure that (1) some state, say s_0', is not reached, or (2) some transition, say (s_0, s_1), is not included in the fault-tolerant program. To make the suitable choice, we develop a set of heuristics by considering whether (1) the transition being excluded is in the fault-intolerant program, (2) the state being excluded is in the invariant of the fault-intolerant program, or (3) recovery is possible from the state that is to be excluded. By default, we prefer to exclude a transition (and its corresponding group) over excluding a state. This is due to the fact that if we choose to exclude a state, we need to exclude all transitions that reach that state. We also consider the invariant states to be valuable, i.e., every attempt is made to ensure that a state in the invariant of the fault-intolerant program is not excluded. Based on these preferences, we develop four heuristics, and use them in our synthesis algorithm.

The synthesis algorithm implemented in FTSyn comprises eight steps organized in four fractions (see Fig. 2), namely *initialize, preserve invariant, modify invariant*, and *resolve cycles*. The first fraction, as in Add_ft, computes ms and mt respectively (see fraction (I) in Fig. 2). The remaining steps proceed to identify the fault-tolerant program, p', its invariant, S', and its fault-span, T', so that formulae $F1$–$F6$ (see Fig. 1) are true. Specifically, all steps ensure that $F1$ is always true. The third step in Fig. 2 computes a new fault-span and ensures that the formula $F2$ is true. The fourth step then removes transitions and/or states from the fault-span computed in Step 3 so that formula $F3$ becomes true. Since Step 4 may violate $F2$, we redo Steps 3 and 4. When no *progress* is made in the last repetition of Steps 3–4, we continue to Step 5 where we deal with deadlock states that are outside S' (also, to prevent an infinite loop, we keep an upper bound on how often each iteration may continue). After Step 5, we repeat Steps 3–5 to re-satisfy $F2$ and $F3$. Throughout Steps 3–5, we do not modify the invariant (see fraction (II) in Fig. 2). This requirement is based on the premise that states in the invariant are valuable and should not be removed prematurely. When there is no progress from Steps 3–5, we continue to Step 6 where we recalculate the invariant while ensuring that $F5$ stays true (i.e., the invariant is closed). In Step 7, we deal with states in S' where $F4$ is false (i.e., deadlock states). Finally, in Step 8, we deal with cases where formula $F6$ is false (i.e., non-progress cycles). The eight steps of our algorithm are as follows:

Step 1. Identifying a set of states from where execution of faults alone can violate safety. Consider a transition (s_0, s_1), which is a fault transition and violates safety. We must ensure that the fault-tolerant program never reaches state s_0. Also, in this scenario, if (s_{-1}, s_0) is a fault transition then we must ensure that the program never reaches s_{-1}. Hence, we identify the set of states, ms (i.e., *marked states*), from where execution of one or more fault actions violate safety.

Application in the token ring program. In the case of the token ring program, safety is violated when a process propagates a corrupted value from its predecessor. Thus, fault transitions do not directly violate safety, and as a result, the set of ms states is empty.

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

Application in the token ring program. Since ms is empty for the token ring program, the set of mt transitions is equal to the set of program transitions that directly violate safety. Let $\langle x_0, x_1, x_2, x_3\rangle$ denote a state of the token ring program. Then, as an example, the transition that process P_1 takes from state $b = \langle -1, 0, 0, 0\rangle$ to $c = \langle -1, -1, 0, 0\rangle$ violates the safety of specification. Thus, (b, c) belongs to the set of mt transitions.

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

Reasoning behind Heuristic 1. If (s_0', s_1') is a transition such that $s_0' \in ms$, then (s_0', s_1') may be included in the transitions of the fault-tolerant program. This heuristic is based on the premise that the synthesis algorithm will ensure that state s_0' will never be reached. This heuristic is useful when (s_0', s_1') is grouped with some other transition that is desirable in the fault-tolerant program (thus, in the scenario discussed at the start of the section, we can choose to include the group that contains (s_0', s_1') and ensure that state s_0' is not reached).

Application in the token ring program. Since ms is empty for the token ring program, Heuristic 1 is not applicable for this program.

Step 3. Identifying the fault-span of the fault-intolerant program. In order to determine the fault-span of the fault-intolerant program, we identify the set of reachable states by the computations of the fault-intolerant program in the presence of faults. Such computations start in a state in the invariant of the fault-intolerant program and may reach states outside the invariant by a mixture of fault and program transitions.

Application in the token ring program. The state space of the token ring program presented in Sect. 3.1 includes 81 states. For example, starting from an invariant state $a = \langle 0, 0, 0, 0\rangle$, fault transitions may perturb the program to $b = \langle -1, 0, 0, 0\rangle$, where process P_0 is corrupted. From b, process P_1 copies the corrupted value and the fault-intolerant program reaches state $c = \langle -1, -1, 0, 0\rangle$. Thus, starting from the invariant, a combination of program and fault transitions can take the state of the program to any possible state in the whole state space. Therefore, for the token ring program, its fault-span is equal to its entire state space with 81 states.

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

Application in the token ring program. Since the fault-span of the token ring program is equal to its state space, all states are reachable.

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

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

In order to ensure that $F1$ also remains true in Step 4, if we add a transition that originates from the invariant, we ensure that the resulting state is also in the invariant. During the first iteration of Step 4, this is straightforward as the invariant of the fault-intolerant program is closed in the fault-intolerant program. However, this check is done explicitly after the invariant is recalculated in Steps 6–7.

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

Application in the token ring program. In the case of the token ring program, we must exclude all mt transitions; i.e., the set of program transitions that directly violate safety. Such transitions are those during which an uncorrupted process copies the corrupted value of its predecessor. After removing such transitions (and their corresponding group), we include the remaining transitions in the fault-tolerant program. For example, the group associated with the transition (b, c), where $b = \langle -1, 0, 0, 0\rangle$ and $c = \langle -1, -1, 0, 0\rangle$, contains nine transitions $\langle -1, 0, y, z\rangle \rightarrow \langle -1, -1, y, z\rangle$, where $-1 \leq y, z \leq 1$, that must be excluded from the set of transitions of the fault-tolerant program. Note that since in the source states of these nine transitions at least one process (i.e., P_0) is corrupted, none of these transitions starts in the invariant.

Repeat Steps 3 and 4. After completing Step 4, we recalculate the fault-span with the revised program to determine if any additional transitions of the fault-intolerant program may be included. This repetition can proceed until there are no more changes. If there are no changes in Steps 3-4 then F2 and F3 have become true.

Application in the token ring program. Since in the token ring program faults can at most corrupt three processes, all the incoming transitions to the state $\langle -1, -1, -1, -1\rangle$ are program transitions that violate safety. Since such transitions will not be included in the fault-tolerant program, the new fault-span of the token ring program does not include state $\langle -1, -1, -1, -1\rangle$. Also, the removal of safety-violating transitions causes 16 other states to become unreachable: $\langle -1, 1, 0, 1\rangle$, $\langle -1, 0, 1, 0\rangle$, $\langle 0, 1, -1, 0\rangle$, $\langle 1, 0, -1, 1\rangle$, $\langle 0, 1,$

$0, y\rangle$, $\langle 0, y, 1, 0\rangle$, $\langle 1, y, 0, 1\rangle$, and $\langle 1, 0, 1, y\rangle$, where $-1 \leq y \leq 1$. Thus, the recalculated fault-span includes 64 states.

Step 5. Resolving deadlocks. State s_d is deadlocked if there is no program transition that originates in s_d. Note that such states cause violation of formula $F4$. If s_d is in the invariant of the fault-intolerant program and s_d is a deadlocked state in the fault-intolerant program then in this step we ignore the deadlock at s_d. We deal with remaining deadlocks states (in the fault-span, outside the invariant) using the following heuristic.

> *Heuristic 4:* Given a deadlocked state s_d that does not belong to the invariant, either include a recovery transition from s_d to an invariant state, or make s_d unreachable from the invariant without eliminating any invariant states.
>
> (*Step 5.1*) If it is possible to add a transition from s_d to a state in the invariant (i.e., single-step recovery) then we add such a transition. Note that in distributed programs, we must add the group corresponding to that transition. We require that the added group satisfies the following two conditions: (1) no transition in that group is in mt (except as permitted by Heuristics 1-3), and (2) if any transition in that group originates in the invariant of the fault-intolerant program, then it satisfies the second condition of the transformation problem, i.e., that transition is a transition of the fault-intolerant program.
>
> (*Step 5.2*) If such a (recovery) group cannot be added, we consider whether s_d can be reached from the invariant with the execution of faults alone.
>
>> If yes (*Step 5.2.1*), we leave s_d as is.
>>
>> If no (*Step 5.2.2*), we ensure that the fault-tolerant program does not reach s_d by removing some program transitions.

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

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

Application in the token ring program. To synthesize a fault-tolerant token ring program, in Step 5, we identify deadlock states created due to removing mt transitions. Recall that since ms is empty, mt only includes program transitions that directly violate safety. For example, after we remove the transition (b, c), where $b = \langle -1, 0, 0, 0\rangle$ and $c = \langle -1, -1, 0, 0\rangle$, state b becomes a deadlock state. In the case of the token ring program, since no mt transitions (and their grouped transitions) originate in the invariant, removing mt transitions does not create any deadlock state inside the invariant. Thus, all deadlock states are outside the invariant of the token ring program.

In Step 5 (see fraction (II) in Fig. 2), the synthesis algorithm adds single-step recovery transitions from deadlock states to the invariant by allowing a corrupted process to copy an uncorrupted value from its predecessor. For example, from state $\langle 0, -1, 0, 0\rangle$, process P_1 can copy the value of x_0, and as a result, the program recovers to the state $\langle 0, 0, 0, 0\rangle$ inside its invariant. However, such addition of single-step recovery transitions is not possible from states where more than one process is corrupted (e.g., $\langle 0, -1, 0, -1\rangle$). Such states are directly reachable from the invariant by fault transitions alone. Thus, the synthesis algorithm fails to eliminate such states in Step 5.

Repeat Step 3–5. After the completion of Step 5, we repeat Steps 3–5. Let p_r be the revised program obtained from Steps

Fig. 3 Making deadlock states unreachable without removing invariant states

```
Eliminate($s_d$ : state, $S$ : state predicate, $p$ : transitions) {
    1) If $s_d$ was considered earlier for elimination then   return;
    2) Remove transitions of the form $(s', s_d)$ from $p$;
    3) If there exists a fault transition $(s', s_d)$ then   Eliminate($s', S, p$);
    4) If all the transitions from some state, say $s_0$, are removed then
        - Add the transitions of $p$ that start from $s_0$;
        - If $s_0$ is unreachable from $S$ by execution of faults alone then
            Eliminate($s_0, S, p$);   }
```

4 and 5. In Step 3, we use the transitions of p_r to identify the fault-span. However, while computing the fault-span, we do not explore states that were not eliminated in Step 5.2.2 (if we explore these states, we will get the same deadlocked states which we were trying to eliminate in Step 5.2.2). Then, in Step 4, we consider transitions of p_r that can still be used. We also determine if transitions from the original fault-intolerant program can be added; this may occur if the fault-span recalculated in Step 3 is different. Subsequently, we resolve the deadlocks as mentioned in Step 5. While repeating Step 5, additional recovery transitions could be added due to the revised fault-span. We continue this until a fixpoint is reached (alternatively, we could stop after certain iterations and continue to Step 6).

Application in the token ring program. Repeating Steps 3–5 will not change the fault-span recalculated in repeating Steps 3 and 4 right after Step 4. Thus, with a fault-span that includes some unresolved deadlock states, the synthesis algorithm moves to Step 6.

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

Reasoning behind Step 6. Since repetitions of Steps 3–5 have reached a fixpoint, all deadlocked states fall in category 5.2.1 or 5.2.2. This suggests that there are some offending states in the invariant which should not be in the invariant of the fault-tolerant program.

Application in the token ring program. At this step, the fault-span of the token ring program contains a set of states D_1 with only one corrupted process in which a single-step recovery has already been included in Step 5, and a set of deadlock states D_2 (with more than one corrupted processes in each state) whose states are reachable from the invariant directly by a sequence of fault transitions. Thus, to make the deadlock states D_2 unreachable, the synthesis algorithm needs to remove all invariant states.

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

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

ConstructInvariant eliminates the state s_0 from S if there is no transition of the form (s_0, s_1) such that (s_0, s_1) is a transition of p and s_1 is in S (i.e., s_0 is a deadlock state). Step 6 can produce such state s_0 if it eliminates the state s_1. After computing the invariant using ConstructInvariant, we recalculate the program transitions to ensure that the revised invariant is closed in the program. Towards this end, if the program obtained in Step 6 contains a transition of the form (s_0, s_1) where s_0 is in the revised invariant but s_1 is not in the revised invariant then we remove the transition (s_0, s_1) (and the group associated with it). With the revised program, there may be new deadlock states created within the invariant. Hence, we apply ConstructInvariant again with the revised program. We continue this until the revised invariant is closed in the program transitions or the revised invariant is empty. In the latter case, we declare that synthesis fails.

Application in the token ring program. Since the algorithm removes all states in the invariant of the token ring program, the resulting invariant will be empty. Thus, the synthesis fails in this step. In Sect. 5, we present a heuristic for adding multiple-step recovery that results in synthesizing the fault-tolerant token ring program presented in Sect. 3.1.

Repeat Steps 3–7. After completing Step 7, we redo Steps 3–7, i.e., with this reduced invariant, we compute the new fault-span. Then, we decide which transitions of the fault-intolerant program may be used in Step 4. Since the synthesis of the token ring program failed in Step 7, this case is not applicable for this program.

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

Repeat Steps 3–8. If the program obtained after Step 8 does not satisfy the formulas in Add_ft, we repeat Steps 3–8 with the program obtained in Step 8. If after some predetermined number of iterations, a fault-tolerant program is not found, our algorithm declares failure in finding a fault-tolerant program.

Comment on the heuristics. Since the problem of adding fault-tolerance to distributed programs is NP-complete [27, 31] (in the state space of the fault-intolerant program), we cannot design a sound and complete polynomial-time synthesis algorithm unless $P = NP$. Thus, although the heuristics that we have presented in this section are sound (i.e., if they result in the synthesis of a fault-tolerant program then the synthesized program meets the requirements of the addition problem), they may fail to synthesize a fault-tolerant program in some cases (e.g., the single-step recovery heuristic presented in Step 5.1 failed to synthesize a masking fault-tolerant token ring program). As a result, there is a well-defined need for a repository of heuristics available to *developers of fault-tolerance* that can be extended by *developers of heuristics*. In Sect. 5.1, we show how we provide an extensible design for FTSyn to achieve this goal.

5 An extensible and changeable design for FTSyn

In this section, we first give an overview of the design of FTSyn. Then, in Sect. 5.1, we show how developers can extend the design of FTSyn by adding new heuristics. Subsequently, in Sect. 5.2, we illustrate how one can change the implementation of the components of FTSyn without a significant overhead. Additional details of the design of FTSyn that may be of interest to users interested in extending FTSyn are included in [1].

In the conceptual (object-oriented) design of FTSyn, we model each one of the entities (i.e., Program, Process, Fault, SafetySpecification, Invariant, and InitialStates) involved in the problem of adding fault-tolerance as a class. Using the initial states and program/fault transitions, we generate the fault-span of the fault-intolerant program as a set of reachable states; i.e., the reachability graph of the fault-intolerant program. Hence, we regard the fault-span of the fault-intolerant program as an input entity and we model it as a class in the design of FTSyn. After taking the input entities, FTSyn instantiates an object corresponding to each one of the design classes. Subsequently, FTSyn executes the synthesis algorithm on the reachability graph of the fault-intolerant program to generate a reachability graph of the fault-tolerant program. The output entities (i.e., fault-tolerant program and its invariant) are also instances of the existing classes in the design of FTSyn. Next, we demonstrate how one can integrate new heuristics into the design of FTSyn.

5.1 Extending FTSyn: illustration of integrating new heuristics for resolving deadlocks

In this section, we illustrate how we have developed two new heuristics for adding recovery from deadlock states, and have integrated these new heuristics in FTSyn.

Heuristic 5: Adding multi-step recovery. The Step 5.1 of Heuristic 4 (presented in Sect. 4) only adds single-step recovery from deadlock states to the invariant. As a result, it fails in cases where single-step recovery is not possible. For example, Heuristic 4 failed to add recovery to states where there is more than one corrupted process (e.g., $\langle 0, -1, -1, -1\rangle$) in the token ring program. The idea behind our new heuristic is that we provide recovery from a deadlock state, say s'_d, via another deadlock state, say s_d, from where we have already added a recovery transition.

Likewise Heuristic 4, Heuristic 5 also consists of two passes. In the first pass, we conduct a fixpoint computation that searches through the deadlock states outside the invariant in the fault-span. In the first iteration of the fixpoint computation, we find all *deadlock* states from where single-step recovery to the invariant is possible. In the second iteration, we find all *deadlock* states from where single-step recovery is possible to recovery states explored in the previous iteration. Continuing thus, we reach an iteration of the fixpoint computation where either no more deadlock states exist or no more recovery is possible. In the latter case, we choose to deal with the remaining deadlock states in the second pass. In the former case, at the end of the fixpoint computation, we will have a set of states, *RecoveryStates*, from where there exists a multi-step recovery path to the invariant (notice that adding a recovery transition in a distributed program requires the satisfaction of the grouping requirements).

In the second pass, we try to remove s_d if s_d is directly reachable by fault transitions from the invariant and no recovery can be added from s_d. If the removal of s_d requires the removal of one or more invariant states then we remove those offending invariant states. During deadlock resolution, if the invariant becomes empty then we declare that the synthesis algorithm failed to synthesize a fault-tolerant program.

Application in the token ring program. Heuristic 4 adds recovery to states where there is only one corrupted process;

e.g., $d = \langle 0, 0, 0, -1\rangle$. Using our new heuristic, we add recovery from states where there exist exactly two corrupted processes, e.g., $e = \langle 0, 0, -1, -1\rangle$, to states where there exists only one corrupted process. Likewise, we add recovery from states where there exist exactly three corrupted processes, e.g., $g = \langle 0, -1, -1, -1\rangle$, to states where there exist exactly two corrupted processes. In this case, a recovery from the state g to the invariant contains three steps where (1) P_1 corrects itself by copying the value of x_0 and reaching state e; (2) P_2 corrects itself by copying the value of x_1 and reaching state d, and (3) P_3 corrects itself by copying the value of x_2 and reaching the *invariant* state $\langle 0, 0, 0, 0\rangle$.

In order to provide extensibility in FTSyn, we employ a set of design patterns [20] in the object-oriented design of FTSyn. For example, we have applied the Strategy design pattern [20] to the DeadlockResolver method of the ReachabilityGraph class in the design of FTSyn that implements deadlock resolution schemes. The application of the Strategy pattern to this method allows us to easily extend the design of FTSyn upon developing new heuristics for adding recovery to deadlock states. We have integrated the above heuristic in FTSyn without any changes in the existing design of FTSyn. Using this new heuristic, we have automatically synthesized the masking fault-tolerant token ring program presented in Sect. 3.1.

Heuristic 6: The strategy of Heuristic 6 is similar to that in Heuristic 5, except that the domain of the fixpoint computation includes all the states outside the invariant in the fault-span (i.e., $(T' - S')$) instead of just resolved deadlock states. In other words, Heuristic 6 is more general than Heuristic 5 (likewise, Heuristic 5 is more general than Heuristic 4). We have also used Heuristic 6 for enhancing the fault-tolerance of nonmasking programs to masking fault-tolerance [29], where a nonmasking program only guarantees recovery to the invariant, but does not guarantee safety during recovery. The integration of Heuristic 6 was fairly simple. We integrated Heuristic 6 as an alternative strategy of deadlock resolution in the DeadlockResolver method.

The application of heuristics. The Heuristic 5 suffices for the synthesis of the fault-tolerant token ring program presented in Sect. 3.1. However, in the synthesis of an agreement protocol in the presence of arbitrary faults [32], we applied Heuristic 6 since Heuristic 5 failed (see [1]). Given a particular problem, the developers can either use their insight to choose the appropriate heuristic or they can rely on FTSyn to make that choice. The former choice provides more efficiency whereas the latter choice allows more automation.

5.2 Changing the Implementation of FTSyn

As we mentioned in the Introduction, it is difficult to determine a priori the internal representation that one should use for different components of FTSyn, namely Program, Fault, Specification, and Invariant, involved in the synthesis of fault-tolerant programs. Thus, it is necessary to provide the ability to modify the internal representation of these components while reusing the remaining parts of FTSyn. In fact, there are situations where one needs to use one internal representation while executing in one fraction of FTSyn (see Fig. 2), and a different internal representation for the same component while executing in another fraction of FTSyn.

In the conceptual design of FTSyn, we consider a class SafetySpecification that models the safety specification of programs. We have two different implementations for the class SafetySpecification: (1) linked-list, and (2) symbolic. The linked-list implementation contains a list of elements where each element represents a set of safety-violating transitions. Thus, during synthesis, to verify the safety of an individual transition t, we traverse the linked-list to verify the membership of t to the set of safety-violating transitions. The symbolic implementation directly uses the *specification* predicates that represent the safety specification in the input of FTSyn (for an example, see Sect. 6 or 3.1). Afterwards, the symbolic implementation substitutes the values of program variables at the source and the destination of t in the *specification* predicate to verify the safeness of t.

Reasoning about a query. The symbolic implementation helps to improve the efficiency of the synthesis when we need to automatically synthesize a fault-tolerant program without any user intervention. Specifically, the symbolic implementation reduces the problem of checking the safety of a transition to an instance of the satisfiability problem, where only a yes/no answer is provided. However, when users interact with FTSyn, they may need to know why and how a transition violates the safety specification. To fulfill users' requirements, in the symbolic implementation, we can only provide the values of program variables in the source and target states of the safety-violating transitions, which are difficult to interpret. On the other hand, in the SafetySpecification linked-list, different scenarios of violating safety specification are represented as different sets of transitions. Thus, the linked-list structure can provide a better intuition as to why a particular transition violates safety. Therefore, to provide reasoning about the violation of safety, FTSyn switches the implementation of the SafetySpecification class from the symbolic to the linked-list structure.

6 Example: altitude switch controller

In this section, we demonstrate how we used FTSyn to synthesize a simplified version of an altitude switch (ASW) used in aircraft altitude controller. We have adapted this example from [10] and the output program of FTSyn is the same as the fault-tolerant program that is manually designed in [10]. This example illustrates the applicability of FTSyn in automatic synthesis of practical applications.

The fault-intolerant altitude switch (ASW). The ASW program monitors a set of input variables and generates an output. There exist four internal variables, a mode variable that determines the operating mode of the program, and four input variables that represent the state of the altitude sensors. The internal variables are as follows: (1) $AltBelow$ is equal to 1 if the altitude is below a specific threshold, otherwise, it is equal to 0; (2) $ActuatorStatus$ is equal to 1 if the actuator is powered on, otherwise, it is equal to 0; (3) $Inhibit$ is equal to 1 when the actuator power-on is inhibited, otherwise, it is equal to 0, and (4) $Reset$ is equal to 0 if the system is being reset.

The ASW program can be in three different modes: (1) the *Initialization* mode when the ASW system is initializing; (2) the *Await-Actuator* mode if the system is waiting for the actuator to power on, and (3) the *Standby* mode. We use an integer variable $Status$ to represent the system modes in the program where (1) $Status = -1$ if the system is in the initialization mode; (2) $Status = 0$ if the system is in the Await-Actuator mode, and (3) $Status = 1$ if the system is in the Standby mode.

Moreover, we model the signals that come from the input (analog and digital) altitude sensors to indicate the occurrence of faults using the following variables: (1) $AltFail$ is equal to 1 when altitude sensors are failed; (2) if the system fails in the Initialization mode then the variable $InitFailed$ will be set to 1, otherwise, $InitFailed$ remains 0; (3) if the altitude sensors fail (i.e., $AltFail = 1$) and do not recover in a certain number of built-in reset attempts then the variable $AltFailOver$ will be equal to 1, otherwise, $AltFailOver$ remains 0, and (4) if the Actuator fails in the Await-Actuator mode then the variable $AwaitOver$ will be equal to 1, otherwise, $AwaitOver$ remains 0.

The output of the ASW program is identified based on the system mode. The ASW program has an output integer variable $WakeupActuator$ that is equal to 1 if the system is in the Await-Actuator mode and is equal to 0 otherwise. The domain of all variables except $Status$ is equal to {0, 1}.

The fault-intolerant program consists of only one process, called Controller. In the input of FTSyn, we specify the Controller process as follows:

```
process  Controller
begin

 ((Status == -1) && (InitFailed == 0))
                  ->Status = 1;
 |
 ((Status == 1) && (Reset == 0))
                  -> Status = -1; Reset = 1;
 |
 ((Status == 1) && (AltBelow == 0) &&
  (Inhibit == 0) && (ActuatorStatus ==0))
                  -> Status = 0; AltBelow = 1;
 |
 ((Status == 0) && (ActuatorStatus == 0))
                  -> Status = 1;
                     ActuatorStatus = 1;
 |
 ((Status == 0) && (Reset == 0))
                  -> Status = -1; Reset = 1;

read  AltBelow, ActuatorStatus, Inhibit, Reset,
       AltFail, InitFailed, AltFailOver,
       AwaitOver, WakeupActuator, Status;

write WakeupActuator, AltBelow, ActuatorStatus,
       Inhibit, Reset,  Status;
end
```

The ASW program changes its mode from Initialization to Standby. The program transitions to the Initialization mode when it is either in Standby or in Await-Actuator mode and the reset signal is received. If the program is in the Standby mode, the altitude is not below a pre-determined threshold, the actuator power-on is not inhibited and the actuator is not powered on, then the program goes to Await-Actuator mode. In the Await-Actuator mode, the program either powers on the actuator and goes to the standby mode, or transitions to the Initialization mode upon receiving the reset signal.

Read/Write restrictions. The Controller process can read all program variables and can write only a subset of variables.

Faults. The malfunction of the altitude sensors may perturb the state of the program to a faulty state. We introduce a new mode, where $Status = 2$, that represents the system is in a faulty state. We represent the fault actions as follows:

```
fault  Malfunction
begin

  (InitFailed == 1)  -> InitFailed = 0;
                        Status = 2;
  |
  (AltFailOver == 1) -> AltFailOver = 0;
                         Status = 2;
  |
  (AwaitOver == 1)   -> AwaitOver = 0;
                         Status = 2;

end
```

Note that the guards of the above actions represent conditions under which the program *detects* the occurrence of faults and switches to the faulty mode. We could have added the following actions to the list of fault actions to model the effect of faults.

```
  (InitFailed == 0)  -> InitFailed = 1;
  |
  (AltFailOver == 0) -> AltFailOver = 1;
  |
  (AwaitOver == 0)   -> AwaitOver = 1;
```

The above actions perturb the program to states where at least one of the variables $InitFailed$, $AltFailOver$, and $AwaitOver$ is equal to one; i.e., shows the occurrence of faults. Since in this case there exist only two values 0 and 1 in the domain of these variables, we have adopted a simpler approach where we set the values of these variables to 1 in the initial states of the program. In this example, the structure of the synthesized fault-tolerant program remains the same.

Safety specification. The problem specification requires that the program does not change its mode from Standby to Await-Actuator if the altitude sensors are failed; i.e., $AltFail$ is equal to 1. Also, from the faulty state, the program can only go to the Initialization mode. Moreover, in the faulty state, the program can recover if it is not reset. In the input file, we represent the specification as a state predicate.

```

((AltFails == 1) && (Statuss == 1) &&
                         (Statusd == 0))    ||
((Statuss == 2) &&
     ((Statusd == 1) || (Statusd == -1))) ||
((Statuss == 2) && (Resets == 0))
```

As we described in subsect. 3.1, to distinguish the value of a variable (e.g., $AltFail$) at the source of a transition from its value at the destination, we append the variable names with suffixes ‘s’ and ‘d’ (e.g., $AltFails$ and $AltFaild$).

Invariant. The invariant of the program consists of the states where the program is not in the faulty state; i.e., $Status \neq 2$. We specify the invariant as follows:

```
invariant

(Status != 2)
```

Initial states. We specify the initial state as follows:

```
init

state
   WakeupActuator = 0;
   AltBelow = 1;
   ActuatorStatus = 0;
   Inhibit = 0;
   Reset = 0;
   AltFail = 0;
   InitFailed = 1;
   AwaitOver = 1;
   AltFailOver = 1;
   Status = -1;
```

Fault-tolerant program. FTSyn automatically generates the following fault-tolerant program (**Bold** fonts represent the code updates). We present the actions of the Controller process as follows:

```
1   ((Status == -1) && (InitFailed == 0))
2               -> Status = 1;
3   |
4   ((Status == 1) && (Reset == 0))
5               -> Status = -1; Reset = 1;
6   |
7   ((Status == 1) && (AltBelow == 0) &&
8    (Inhibit == 0) && (ActuatorStatus ==0) &&
9         (AltFail == 0))
10              -> Status = 0; AltBelow = 1;
11  |
12  ((Status == 0) && (ActuatorStatus == 0))
13              -> Status = 1; ActuatorStatus = 1;
14  |
15  ((Status == 0) && (Reset == 0))
16              -> Status = -1; Reset = 1;
17  |
18
19  ((Status == 2) && (Reset == 0))
20              ->  Status = -1; Reset = 1;
```

The fault-tolerant program has a new recovery action (see Lines 19–20 above), where it recovers to the initialization mode from faulty state (i.e., states where $Status = 2$ holds). Moreover, a new constraint has been added to the third action (see Line 9) where the program is allowed to change its state to the Await-Actuator mode only when the input sensors are not corrupted; i.e., the condition ($AltFail = 0$) holds.

7 Discussion

In this section, we discuss some theoretical, practical, and pedagogical aspects of FTSyn.

Complexity. In principal, the problem of adding fault tolerance to distributed programs is NP-complete in program state space [27,31]. The complexity of synthesis, however, can be reduced to polynomial time if we use appropriate heuristics and the heuristics are applicable. Thus, one of the important problems in synthesis is to identify heuristics that will keep the complexity of synthesis manageable. The FTSyn framework proposed in this paper is especially useful for testing and developing such heuristics.

Scalability. While the initial version of FTSyn adds fault tolerance to small programs (with reachable states in the scale of a few millions of states), the (space/time) efficiency of FTSyn is certainly comparable to that of early model checkers. The largest program that we have synthesized using the initial version of FTSyn is an agreement program that is simultaneously perturbed by Byzantine and fail-stop faults (1.3 millions of states) [1,30]. To our knowledge, this program is the first automatically synthesized agreement program that simultaneously tolerates both Byzantine and fail-stop faults (see [1]). Researchers were using early versions of model checkers for checking small protocols and verifying the correctness of operating system kernels [15,25]. The state space of the models checked in early 90s was approximately 500,000 states [25], which is comparable to our initial results with FTSyn. We have recently developed a symbolic version [11] and a distributed version of FTSyn [16] that adds fault tolerance to programs that have about 2^{120} reachable states.

While space and time efficiency of FTSyn are important issues, other design goals of FTSyn (such as the ability to check the effectiveness of heuristics) are orthogonal to complexity issues. For example, the complexity of determining whether or not a specific group of transitions violates safety is independent from the heuristics that determine a set of groups that should be included in a program so that the program recovers to its invariant. To illustrate this, we have implemented a SAT-based version of FTSyn where one can either take advantage of SAT solvers to verify the safety of a group of transitions [17], or exhaustively verify every transition of a given group of transitions.

Educational applications. We have used FTSyn in graduate classes where students used the automated approach to obtain the fault-tolerant programs that have already been synthesized in FTSyn. Subsequently, they focused on interactive synthesis of the same fault-tolerant programs. During this interactive synthesis, they applied different heuristics and observed the intermediate programs. This allowed them to evaluate different heuristics.

8 Concluding remarks and future work

In this paper, we presented a software framework, called Fault-Tolerance Synthesizer (FTSyn), for adding fault-tolerance to existing fault-intolerant (distributed) programs. Since the problem of adding fault-tolerance to distributed programs is NP-complete [27,31] in the state space of the fault-intolerant program, we presented sound heuristics for polynomial-time addition of fault-tolerance. In the cases where heuristics are applicable, FTSyn synthesizes a fault-tolerant program in polynomial-time using a set of built-in heuristics that can be used by *developers of fault-tolerant programs* to automatically add fault-tolerance. Moreover, FTSyn is extensible in that *developers of heuristics* can easily integrate new heuristics in FTSyn without a significant overhead.

We demonstrated how one can use FTSyn to automatically add fault-tolerance to a token ring program that is subject to process-restart faults, and an altitude switch controller that is subject to input faults. Several other examples are available at [1] among which an agreement program that simultaneously tolerates Byzantine and fail-stop faults. To our knowledge, this program is the first automatically synthesized distributed program that simultaneously tolerates Byzantine and fail-stop faults.

There are several future directions to this work. In [31], we have identified a class of specifications and programs for which failsafe fault-tolerance can be added in polynomial time (in the state space of the fault-intolerant program), where a failsafe fault-tolerant program guarantees to satisfy its safety specification in the presence of faults. Using the results of [31], we have developed heuristics that can study the structure of programs (respectively, specifications) to determine if these conditions are met [18]. Another extension of the framework is to take advantage of the structural similarity of the processes [6,8] in order to reduce the complexity of adding fault-tolerance to a fault-intolerant program.

Acknowledgments We would like to thank Constance Heitmeyer at Naval Research Laboratory for her comments and suggestions on the altitude switch example.

References

1. A framework for automatic synthesis of fault-tolerance. http://www.cse.msu.edu/~sandeep/software/Code/synthesis-framework/
2. Spin language reference. http://spinroot.com/spin/Man/promela.html
3. Alpern, B., Schneider, F.B.: Defining liveness. Inform. Process. Lett. **21**, 181–185 (1985)
4. Arora, A., Gouda, M.G.: Closure and convergence: a foundation of fault-tolerant computing. IEEE Trans. Software Eng. **19**(11), 1015–1027 (1993)
5. Arora, A., Kulkarni, S.S.: Designing masking fault-tolerance via nonmasking fault-tolerance. IEEE Trans. Software Eng. **24**(6), 435–450 (1998) (A preliminary version appears in the Proceedings of the Fourteenth Symposium on Reliable Distributed Systems, Bad Neuenahr, 174–185, 1995)
6. Attie, P.: Synthesis of large concurrent programs via pairwise composition. In: CONCUR'99: 10th International Conference on Concurrency Theory, pp. 130–145 (1999)
7. Attie, P., Emerson, A.: Synthesis of concurrent programs for an atomic read/write model of computation. ACM TOPLAS (a preliminary version of this paper appeared in PODC96) **23**(2) (2001)
8. Attie, P., Emerson, E.: Synthesis of concurrent systems with many similar processes. ACM Trans. Programming Lang. Syst. **20**(1), 51–115 (1998)
9. Attie, P.C., Arora, A., Emerson, E.A.: Synthesis of fault-tolerant concurrent programs. ACM Transactions on Programming Languages and Systems (TOPLAS). (A preliminary version of this paper appeared in PODC 1998.) **26**(1), 125 – 185 (2004)
10. Bharadwaj, R., Heitmeyer, C.: Developing high assurance avionics systems with the SCR requirements method. In: Proceedings of the 19th Digital Avionics Systems Conference, Philadelphia, PA (2000)
11. Bonakdarpour, B., Kulkarni, S.S.: Exploiting symbolic techniques in automated synthesis of distributed programs. In: IEEE International Conference on Distributed Computing Systems, pp. 3–10 (2007)
12. Demirbas, M., Arora, A.: Convergence refinement. In: International Conference on Distributed Computing Systems, pp. 589–597 (2002)
13. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM **17**(11) (1974)
14. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs, NJ, USA (1990)
15. Duval, G., Julliand, J.: Modeling and verification of rubis micro-kernel with spin. The First SPIN Workshop (1995). Available at http://spinroot.com/spin/Workshops/ws95/papers.html
16. Ebnenasir, A.: Diconic addition of failsafe fault-tolerance. In: Proceedings of the 22nd IEEE/ACM international conference on Automated Software Engineering, pp. 44–53 (2007)
17. Ebnenasir, A., Kulkarni, S.S.: SAT-based synthesis of fault-tolerance. In: Fast Abstracts of International Conference on Dependable Systems and Networks, Palazzo dei Congressi, Florence, Italy (2004)

18. Ebnenasir, A., Kulkarni, S.S.: Efficient synthesis of failsafe fault-tolerant distributed programs. Tech. Rep. MSU-CSE-05-13, Computer Science and Engineering, Michigan State University, East Lansing, MI (2005)
19. Emerson, E., Clarke, E.: Using branching time temporal logic to synthesize synchronization skeletons. Sci Comput Program **2**(3), 241–266 (1982)
20. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, USA (1995)
21. Gärtner, F.C., Jhumka, A.: Automating the addition of failsafe fault-tolerance: Beyond fusion-closed specifications. Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT), Grenoble, France, LNCS 3253, pp. 183–198 (2004)
22. Gouda, M., McGuire, T.: Correctness preserving transformations for network protocol compilers. Prepared for the Workshop on New Visions for Software Design and Productivity: Research and Applications (2001)
23. Havelund, K., Pressburger, T.: Model checking java programs using java pathfinder. Int. J. Software Tools Technol. Transf. (STTT) **2**(4), 366–381 (2000)
24. Holzmann, G.J.: From code to models. In: Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD'01), pp. 3–10 (2001)
25. Joesang, A.: Security protocol verification using spin. The First SPIN Workshop (1995). Available at http://spinroot.com/spin/Workshops/ws95/papers.html
26. Kulkarni, S.S.: Component-based design of fault-tolerance. Ph.D. Thesis, Ohio State University (1999)
27. Kulkarni, S.S., Arora, A.: Automating the addition of fault-tolerance. In: Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, pp. 82–93 (2000)
28. Kulkarni, S.S., Arora, A., Chippada, A.: Polynomial time synthesis of Byzantine agreement. Symposium on Reliable Distributed Systems, pp. 130–139 (2001)
29. Kulkarni, S.S., Ebnenasir, A.: Enhancing the fault-tolerance of nonmasking programs. In: Proceedings of the 23rd International Conference on Distributed Computing Systems, pp. 441–449 (2003)
30. Kulkarni, S.S., Ebnenasir, A.: A framework for automatic synthesis of fault-tolerance. Tech. Rep. MSU-CSE-03-16, Computer Science and Engineering, Michigan State University, East Lansing MI 48824, Michigan (2003)
31. Kulkarni, S.S., Ebnenasir, A.: Complexity issues in automated synthesis of failsafe fault-tolerance. IEEE Trans. Depend. Secure Comput. **2**(3), 201–215 (2005)
32. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Trans. Programming Lang. Syst. **4**(3), 382–401 (1982)
33. Nesterenko, M., Arora, A.: Stabilization-preserving atomicity refinement. J. Parallel Distrib. Comput. **62**(5), 766–791 (2002)
34. Varghese, G.: Self-stabilization by local checking and correction. Ph.D. Thesis, MIT/LCS/TR-583 (1993)