Adding Fault-Tolerance Using Pre-Synthesized Components[1]

Sandeep S. Kulkarni and Ali Ebnenasir

Department of Computer Science and Engineering
Michigan State University
48824 East Lansing, Michigan, USA
{sandeep, ebnenasi}@cse.msu.edu
http://www.cse.msu.edu/~{sandeep,ebnenasi}

Abstract. We present a hybrid synthesis method for automatic addition of fault-tolerance to *distributed* programs. In particular, we automatically specify and add pre-synthesized fault-tolerance components to programs in the cases where existing heuristics fail to add fault-tolerance. Such addition of pre-synthesized components has the advantage of *reusing* pre-synthesized fault-tolerance components in the synthesis of different programs, and as a result, reusing the effort put in the synthesis of one program for the synthesis of another program. Our synthesis method is sound in that the synthesized fault-tolerant program satisfies its specification in the absence of faults, and provides desired level of fault-tolerance in the presence of faults. We illustrate our synthesis method by adding pre-synthesized components with linear topology to a token ring program that tolerates the corruption of all processes. Also, we have reused the same component in the synthesis of a fault-tolerant alternating bit protocol. Elsewhere, we have applied this method for adding presynthesized components with hierarchical topology.

Keywords: Automatic addition of fault-tolerance, Formal methods, Detectors, Correctors, Distributed programs

1 Introduction

Automatic synthesis of fault-tolerant distributed programs from their fault-intolerant versions is desirable in variety of disciplines (e.g., safety-critical systems, embedded systems, network protocols) since such automated synthesis (i) generates a program that is correct by construction, and (ii) has the potential to preserve the properties of the fault-intolerant program. However, the exponential complexity of synthesis is one of the important obstacles in such automated synthesis. Thus, it is desirable to reuse the effort put in the synthesis of one program for the synthesis of another program. In this paper, we concentrate on the identification and the addition of pre-synthesized fault-tolerance components to

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

fault-intolerant programs so that we can reuse those components in the synthesis of different programs.

In the previous work on automatic transformation of fault-intolerant programs to fault-tolerant programs, Kulkarni and Arora [1] present polynomial time algorithms (in the state space of the fault-intolerant program) for the synthesis of fault-tolerant programs in the high atomicity model – where each process of the program can read/write all program variables in an atomic step. However, for the synthesis of fault-tolerant distributed programs, they show that the complexity of synthesis is exponential. Techniques presented in [2–4] reduce the complexity of synthesis by using heuristics and by identifying classes of programs and specifications for which efficient synthesis is possible. However, these approaches cannot apply the lessons learnt in synthesizing one fault-tolerant program while synthesizing another fault-tolerant program. As we encounter new problems, it is desirable to reuse synthesis techniques that we have already used during the synthesis of other problems. Hence, if we recognize the *patterns* that we often apply in the synthesis of fault-tolerant distributed programs then we can organize those patterns in terms of fault-tolerance components and reuse them in the synthesis of new problems.

To investigate the use of pre-synthesized fault-tolerance components in the synthesis of fault-tolerant distributed programs, we use *detectors* and *correctors* identified in [5]. Specifically, in [5], it is shown that detectors and correctors suffice in the *manual* design of a rich class of *masking* fault-tolerant programs – where the fault-tolerant program satisfies its safety and liveness specification even in the presence of faults. To achieve our goal, we present a synthesis method that adds pre-synthesized detectors and correctors to a given fault-intolerant program in order to synthesize its fault-tolerant version. Using our synthesis method, we identify (i) the representation of the pre-synthesized detectors and correctors; (ii) when and where the synthesis algorithm should use a detector or a corrector, and (iii) how to ensure the correctness of the fault-tolerant program and pre-synthesized detectors and correctors in the presence of each other.

Contributions. The contributions of this paper are as follows: (i) we develop a synthesis method for reusing pre-synthesized fault-tolerance components in the synthesis of different programs; (ii) we reduce the chance of failure of the synthesis algorithm by using pre-synthesized fault-tolerance components in the cases where existing heuristics fail; (iii) we present a systematic approach for expanding the state space of the program being synthesized in the cases where synthesis fails in the original state space, and finally (iv) we present a systematic method for adding new variables to programs for the sake of adding fault-tolerance.

As an illustration of our synthesis method, we add pre-synthesized components with linear topology to a token ring program that is subject to process-restart faults. The masking fault-tolerant (token ring) program can recover even from the situation where every process is corrupted. We note that the previous approaches that added fault-tolerance to the token ring program presented in this paper fail to synthesize a fault-tolerant program when all processes are corrupted. We have also synthesized a fault-tolerant alternating bit protocol by

reusing the same pre-synthesized fault-tolerance component used in the synthesis of the token ring program (cf. [6] for this synthesis). Elsewhere [7], we have used this method for synthesizing a fault-tolerant diffusing computation where the added component is hierarchical in nature. This example also demonstrates the addition of multiple components. Thus, the synthesis method presented in this paper can be used for adding fault-tolerance components (i) on different topologies, and (ii) for different types of faults.

Note. The notion of program in this paper refers to the abstract structure of a program. The abstract structure of a program is an abstraction of the parts of the program code that execute inter-process synchronization tasks.

The organization of the paper. In Section 2, we present preliminary concepts. In Section 3, we formally state the problem of adding fault-tolerance components to fault-intolerant programs. Then, in Section 4, we present a synthesis method that identifies when and how the synthesis algorithm decides to add a component. Subsequently, in Section 5, we describe how we formally represent a fault-tolerance component. In Section 6, we show how we automatically specify a required component and add it to a program. We discuss issues related to our synthesis method in Section 7. Finally, we make concluding remarks and discuss future work in Section 8.

2 Preliminaries

In this section, we give formal definitions of programs, problem specifications, faults, and fault-tolerance. The programs are specified in terms of their state space and their transitions. We have adapted the definition of (i) specifications from Alpern and Schneider [8], and (ii) faults and fault-tolerance from Arora and Gouda [9] and Kulkarni and Arora [10]. The issues of modeling distributed programs is adapted from [1, 11].

Program. A program p is a finite set of variables and a finite set of processes. Each variable is associated with a finite domain of values. A state of p is obtained by assigning each variable a value from its respective domain. The state space of p, S_p, is the set of all possible states of p.

A process, say P_j, 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. Also, process P_j consists of a set of transitions of the form (s_0, s_1) where $s_0, s_1 \in S_p$. The set of the transitions of p is 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 iff (if and only if) $\forall s_0, s_1 : (s_0, s_1) \in p : (s_0 \in S \Rightarrow s_1 \in S)$. A sequence of states, $\langle s_0, s_1, ... \rangle$, is a computation of p iff the following two conditions are satisfied: (1) $\forall j : j > 0 : (s_{j-1}, s_j) \in p$, and (2) if $\langle s_0, s_1, ... \rangle$ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in p$. A sequence of states, $\langle s_0, s_1, ..., s_n \rangle$, is a computation prefix of p iff $\forall j : 0 < j \leq n : (s_{j-1}, s_j) \in p$, i.e., a computation prefix need not be maximal. The projection of program p on state predicate S, denoted as $p|S$, consists of transitions $\{(s_0, s_1) : (s_0, s_1) \in p \ \wedge \ s_0, s_1 \in S\}$.

Distribution issues. We model distribution by identifying how read/write restrictions on a process affect its transitions. A process P_j cannot include transi-

tions that write a variable x, where $x \notin w_j$. In other words, the write restrictions identify the set of transitions that a process P_j can execute. Given a single transition (s_0, s_1), it appears that all the variables must be read to execute that transition. For this reason, read restrictions require us to group transitions and ensure that the entire group is included or the entire group is excluded. As an example, consider a program consisting of two variables a and b, with domains $\{0, 1\}$. Suppose that we have a process that cannot read b. Now, observe that the transition from the state $\langle a = 0, b = 0\rangle$ to $\langle a = 1, b = 0\rangle$ can be included iff the transition from $\langle a = 0, b = 1\rangle$ to $\langle a = 1, b = 1\rangle$ is also included. If we were to include only one of these transitions, we would need to read both a and b. However, when these two transitions are grouped, the value of b is irrelevant, and we do not need read it.

Specification. A specification is a set of infinite sequences of states that is suffix closed and fusion closed. Suffix closure of the set means that if a state sequence σ is in that set then so are all the suffixes of σ. Fusion closure of the set means that if state sequences $\langle \alpha, s, \gamma\rangle$ 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 [8], we rewrite the specification as the intersection of a safety specification and a liveness specification. For a suffix-closed and fusion-closed specification, the safety specification can be specified [10] as a set of bad transitions that must not occur in program computations, that is, for program p, its safety specification is a subset of $S_p \times S_p$.

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 in S is in *spec*. If p satisfies *spec* from S and $S \neq \{\}$, we say that S is an invariant of p for spec.

We do not explicitly specify the liveness specification in our algorithm; the liveness requirements for the synthesis is that the fault-tolerant program eventually recovers to states from where it satisfies its safety and liveness specification.

Faults. The faults that a program is subject to are systematically represented by transitions. A fault f for a program p with state space S_p, is a subset of the set $S_p \times S_p$. A sequence of states, $\sigma = \langle s_0, s_1, ...\rangle$, is a computation of p in the presence of f (denoted $p[]f$) iff the following three conditions are satisfied: (1) every transition $t \in \sigma$ is a fault or program transition; (2) if σ is finite and terminates in s_l then there exists no program transition originating at s_l, and (3) the number of fault occurrences in σ is finite.

We say that a state predicate T is an f-span (read as fault-span) of p from S iff the following two conditions are satisfied: (1) $S \Rightarrow T$ and (2) T is closed in $p[]f$. Observe that for all computations of p that start at states where S is true, T is a boundary in the state space of p up to which (but not beyond which) the state of p may be perturbed by the occurrence of the transitions in f.

Fault-tolerance. Given a program p, its invariant, S, its specification, *spec*, and a class of faults, f, we say p is masking f-tolerant for *spec* from S iff the following two conditions hold: (i) p satisfies *spec* from S; (ii) there exists a state

predicate T such that T is an f-span of p from S, $p[]f$ satisfies *spec* from T, and every computation of $p[]f$ that starts from a state in T has a state in S.

Program representation. We use Dijkstra's guarded commands [12] to represent the set of program transitions. 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 $s_1\}$.

3 Problem Statement

In this section, we formally define the problem of adding fault-tolerance components to a fault-intolerant program. We identify the conditions of the addition problem by which we can verify the correctness of the synthesized fault-tolerant program after adding fault-tolerance components.

Given a fault-intolerant program p, its state space S_p, its invariant S, its specification *spec*, and a class of faults f, we add pre-synthesized fault-tolerance components to p in order to synthesize a fault-tolerant program p' with the new invariant S'. When we add a fault-tolerance component to p, we also add the variables associated with that component. As a result, we expand the state space of p. The new state space, say $S_{p'}$, is actually the state space of the synthesized fault-tolerant program p'.

After the addition, we require the fault-tolerant program p' to behave similar to p in the absence of faults f. In the presence of faults f, p' should satisfy *masking* fault-tolerance. To ensure the correctness of the synthesized fault-tolerant program in the new state space, we need to identify the conditions that have to be met by the synthesized program, p'. Towards this end, we define a projection from $S_{p'}$ to S_p using onto function $H : S_{p'} \rightarrow S_p$. We apply H on states, state predicates, transitions, and groups of transitions in $S_{p'}$ to identify their corresponding entities in S_p.

Let the invariant of the synthesized program be $S' \subseteq S_{p'}$. If there exists a state $s'_0 \in S'$ where $H(s'_0) \notin S$ then in the absence of faults p' can start at s'_0 whose image, $H(s'_0)$, is outside S. As a result, in the absence of faults, p' will include computations in the new state space $S_{p'}$ that do not have corresponding computations in p. These new computations resemble new behaviors in the absence of faults, which is not desirable. Therefore, we require that $H(S') \subseteq S$. Also, if p' contains a transition (s'_0, s'_1) in $p'|S'$ that does not have a corresponding transition (s_0, s_1) in $p|H(S')$ (where $H(s'_0) = s_0$ and $H(s'_1) = s_1$) then p' can take this transition and create a new way for satisfying *spec* in the absence of faults. Therefore, we require that $H(p'|S') \subseteq p|H(S')$. Now, we present the problem of adding fault-tolerance components to p.

The Addition Problem.

Given p, S, *spec*, f, with state space S_p such that p satisfies *spec* from S,

$S_{p'}$ is the new state space due to adding fault-tolerance components to p,

$H : S_{p'} \rightarrow S_p$ is an onto function,

Identify p' and $S' \subseteq S_{p'}$ such that

$H(S') \subseteq S$,

$H(p'|S') \subseteq p|H(S')$, and
p' is masking f-tolerant for *spec* from S'. □

4 The Synthesis Method

In this section, we present a synthesis method to solve the addition problem. In Section 4.1, we present a high level description of our synthesis method and express our approach for combining heuristics from [2] (cf. Section 4.2 for an example heuristic) with pre-synthesized components. Then, in Section 4.2, we illustrate our synthesis method using a simple example, a token ring program with 4 processes. We use the token ring program as a running example in the rest of the paper, where we synthesize a token ring program that is masking fault-tolerant to process-restart faults.

4.1 Overview of Synthesis Method

Our synthesis method takes as its input a fault-intolerant program p with a set of processes $P_0 \cdots P_n$ $(n > 1)$, its specification *spec*, its invariant S, a set of read/write restrictions $r_0 \cdots r_n$ and $w_0 \cdots w_n$, and a class of faults f to which we intend to add fault-tolerance. The synthesis method outputs a fault-tolerant program p' and its invariant S'.

The heuristics in [2] (i) *add safety* to ensure that the masking fault-tolerant program never violates its safety specification, and (ii) *add recovery* to ensure that the masking fault-tolerant program never deadlocks (respectively, livelocks). Moreover, while adding recovery transitions, it is necessary to ensure that all the groups of transitions included along that recovery transition are safe unless it can be guaranteed (with the help from heuristics) that those transitions cannot be executed. Thus, adding recovery transitions from deadlock states is one of the important issues in adding fault-tolerance. Hence, the method presented in this paper, focuses on adding pre-synthesized components for resolving such deadlock states, say s_d.

Now, in order to resolve s_d using our hybrid approach, we proceed as follows: First, for each process P_i in the given fault-intolerant program, we introduce a high atomicity pseudo process PS_i. Initially, PS_i has no action to execute, however, we allow PS_i to read all program variables and write only those variables that P_i can write. Using these special processes, we present the *ResolveDeadlock* routine (cf. Figure 1) that is the core of our synthesis method. The input of *ResolveDeadlock* consists of the deadlock state that needs to be resolved, s_d, and the set of high atomicity pseudo processes PS_i $(0 \leq i \leq n)$.

First, in Step 1, we invoke a heuristic-based routine *Add_Recovery* to add recovery from s_d under the distribution restrictions (i.e., in the low atomicity model) – where program processes have read/write restrictions with respect to the program variables. *Add_Recovery* explores the ability of each process P_i to add recovery transition from s_d under the distribution restrictions. If *Add_Recovery* fails then we will choose to add a fault-tolerance component in Steps 2 and 3.

In Steps 2 and 3, we identify a fault-tolerance component and then add it to p in order to resolve s_d. To add a fault-tolerance component, the synthesis

algorithm should (i) specify the required component; (ii) retrieve the specified component from a given library of components; (iii) ensure the interference freedom of the component and the program, and finally (iv) add the extracted component to the program. As a result, adding a pre-synthesized component is a costly operation. Hence, we prefer to add a component during the synthesis only when available heuristics for adding recovery fail in Step 1.

```
Resolve_Deadlock(s_d: state, PS_0, ..., PS_n: high atomicity pseudo process)
{
  Step 1. If Add_Recovery (s_d) then return true.
  Step 2. Else non-deterministically choose a PS_index, where 0 ≤ index ≤ n and PS_index
          adds a high atomicity recovery action grd → st
  Step 3. If (there exists a PS_index) and (there exists a detector d in the component
          library that suffices to refine grd → st without interfering with the program)
          then add d to the program, and return true.
          else return false.
              // Subsequently, we remove some transitions to make s_d unreachable.
}
```

Fig. 1. Overview of the synthesis method.

To identify the required fault-tolerance components, we use pseudo process PS_i that can read all program variables and write w_i (i.e., the set of variables that P_i can write). In other words, we check the ability of each PS_i to add high atomicity recovery – where we have no read restrictions – from s_d. If no PS_i can add recovery from s_d then our algorithm fails to resolve s_d. If there exist one or more pseudo processes that add recovery from s_d then we non-deterministically choose a process PS_{index} with high atomicity action $ac : grd \rightarrow st$. Since we give PS_{index} the permission to read all program variables for adding recovery from s_d, the guard grd is a global state predicate that we need to refine. If there exists a detector that can refine grd without interfering with the program execution then we will add that detector to the program. (The discussion about how to specify the required detector d and how to add d to the fault-intolerant program is in Sections 5 and 6.)

In cases where *Resolve_Deadlock* returns *false*, we remove some transitions to make s_d unreachable. If we fail to make s_d unreachable then we will declare failure in the synthesis of the masking fault-tolerant program p'. Observe that by using pre-synthesized components, we increase the chance of adding recovery from s_d, and as a result, we reduce the chance of reaching a point where we declare failure to synthesize a fault-tolerant program.

4.2 Token Ring Example

Using our synthesis method (cf. Figure 1), we synthesize a token ring program that is masking fault-tolerant for the case where all processes are corrupted.

The token ring program. The fault-intolerant program consists of four processes P_0, P_1, P_2, and P_3 arranged in a ring. Each process P_i has a variable x_i $(0 \leq i \leq 3)$ with the domain $\{\bot, 0, 1\}$. Due to distribution restrictions, process P_i can read x_i and x_{i-1} and can only write x_i $(1 \leq i \leq 3)$. P_0 can read x_0 and x_3 and can only write x_0. We say, a process P_i $(1 \leq i \leq 3)$ has the token iff

$x_i \neq x_{i-1}$ and fault transitions have not corrupted P_i and P_{i-1}. And, P_0 has the token iff $x_3 = x_0$ and fault transitions have not corrupted P_0 and P_3. A 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}. Also, if $x_0 = x_3$ then process P_0 copies the value of $(x_3 \oplus 1)$ to x_0, where $\oplus$ is addition in modulo 2. This way, a process passes the token to the next process.

We denote a state s of the token ring program by a 4-tuple $\langle x_0, x_1, x_2, x_3 \rangle$. Each element of the 4-tuple $\langle x_0, x_1, x_2, x_3 \rangle$ represents the value of x_i in s $(0 \leq i \leq 3)$. Thus, if we start from initial state $\langle 0, 0, 0, 0 \rangle$ then process P_0 has the token and the token circulates along the ring. We represent the transitions of the fault-intolerant program TR by the following actions $(1 \leq i \leq 3)$.

$TR_0 : \ (x_0 = 1) \ \wedge \ (x_3 = 1) \quad \longrightarrow x_0 := 0;$
$TR_0' : \ (x_0 = 0) \ \wedge \ (x_3 = 0) \quad \longrightarrow x_0 := 1;$
$TR_i : \ (x_i = 0) \ \wedge \ (x_{i-1} = 1) \quad \longrightarrow x_i := 1;$
$TR_i' : \ (x_i = 1) \ \wedge \ (x_{i-1} = 0) \quad \longrightarrow x_i := 0;$

Faults. Faults can restart a process P_i. Thus, the value of x_i becomes unknown. We use $\perp$ to model the unknown value of x_i.

Specification. The problem specification requires that the corrupted value of one process does not affect a non-corrupted process, and there is only one process that has the token.

Invariant. The invariant of the above program includes states $\langle 0, 0, 0, 0 \rangle$, $\langle 1, 0, 0, 0 \rangle$, $\langle 1, 1, 0, 0 \rangle$, $\langle 1, 1, 1, 0 \rangle$, $\langle 1, 1, 1, 1 \rangle$, $\langle 0, 1, 1, 1 \rangle$, $\langle 0, 0, 1, 1 \rangle$, and $\langle 0, 0, 0, 1 \rangle$.

A heuristic for adding recovery. In the presence of faults, the program TR may reach states where there exists at least a process P_i $(0 \leq i \leq 3)$ whose x_i is corrupted (i.e., $x_i = \perp$). In such cases, processes P_i and $P_{((i+1) \ mod \ 4)}$ cannot take any transition, and as a result, the propagation of the token stops (i.e., the whole program deadlocks).

In order to recover from the states where there exist some corrupted processes, we apply the heuristic for single-step recovery from [2] in an iterative fashion. Specifically, we identify states from where single-step recovery to a set of states *RecoverySet* is possible. The initial value of *RecoverySet* is equal to the program invariant. At each iteration, we include a set of states in *RecoverySet* from where single-step recovery to *RecoverySet* is possible.

In the first iteration, we search for deadlock states where there is only one corrupted process in the ring. For example, consider a state $s_0 = \langle 1, \perp, 1, 0 \rangle$. In the state s_0, P_1 and P_2 cannot take any transitions. However, P_3 can copy the value of x_2 and reach $s_2 = \langle 1, \perp, 1, 1 \rangle$. Subsequently, P_0 changes x_0 to 0, and as a result, the program reaches state $s_3 = \langle 0, \perp, 1, 1 \rangle$. The state s_3 is a deadlock state since no process can take any transition at s_3. To add recovery from s_3, we allow P_1 to correct itself by copying the value of x_0, which is equal to 0. Thus, by copying the value of x_0, P_1 adds a recovery transition to an invariant state $\langle 0, 0, 1, 1 \rangle$. Therefore, we include s_3 in the set of states *RecoverySet* in the first iteration. Note that this recovery transition is added in low atomicity in that all the transitions included in action $(x_0 = 0) \wedge (x_1 = \perp) \rightarrow x_1 := 0$ can be included in the fault-tolerant program without violating safety.

In the second and third iterations, we follow the same approach and add recovery from states where there are two or three corrupted processes to states

that we have already resolved in the previous iterations. Adding recovery up to the fourth iteration of our heuristic results in the intermediate program ITR ($1 \leq i \leq 3$).

$$
\begin{array}{lll}
ITR_0: & ((x_0 = 1) \vee (x_0 = \bot)) \wedge (x_3 = 1) & \longrightarrow x_0 := 0; \\
ITR_0': & ((x_0 = 0) \vee (x_0 = \bot)) \wedge (x_3 = 0) & \longrightarrow x_0 := 1; \\
ITR_i: & ((x_i = 0) \vee (x_i = \bot)) \wedge (x_{i-1} = 1) & \longrightarrow x_i := 1; \\
ITR_i': & ((x_i = 1) \vee (x_i = \bot)) \wedge (x_{i-1} = 0) & \longrightarrow x_i := 0;
\end{array}
$$

Using above heuristic, we can only add recovery from the states where there exists at least one uncorrupted process. If there exists at least one uncorrupted process P_j ($0 \leq j \leq 3$) then $P_{((j+1) \bmod 4)}$ will initiate the token circulation throughout the ring, and as a result, the program recovers to its invariant. However, in the fourth iteration of the above heuristic, we reach a point where we need to add recovery from the state where all processes are corrupted; i.e., $s_d = \langle \bot, \bot, \bot, \bot \rangle$. In such a state, the program ITR deadlocks as an action of the form $(x_0 = \bot) \wedge (x_1 = \bot) \rightarrow x_1 := 0$ cannot be included in the fault-tolerant program. Such an action can violate safety if x_2 and x_3 are not corrupted. In fact, no process can add safe recovery from s_d in low atomicity. Thus, *Add_Recovery* returns false for $\langle \bot, \bot, \bot, \bot \rangle$.

Adding the actions of the high atomicity pseudo process. In order to add masking fault-tolerance to the program ITR, a process P_{index} ($0 \leq index \leq 3$) should set its x value to 0 (respectively, 1) when all processes are corrupted. Hence, we follow our synthesis method (cf. Figure 1), where the pseudo process PS_0 takes the high atomicity action HTR and recovers from s_d. Thus, the actions of the masking program MTR are as follows ($1 \leq i \leq 3$).

$$
\begin{array}{lll}
MTR_0: & ((x_0 = 1) \vee (x_0 = \bot)) \wedge (x_3 = 1) & \longrightarrow x_0 := 0; \\
MTR_0': & ((x_0 = 0) \vee (x_0 = \bot)) \wedge (x_3 = 0) & \longrightarrow x_0 := 1; \\
MTR_i: & ((x_i = 0) \vee (x_i = \bot)) \wedge (x_{i-1} = 1) & \longrightarrow x_i := 1; \\
MTR_i': & ((x_i = 1) \vee (x_i = \bot)) \wedge (x_{i-1} = 0) & \longrightarrow x_i := 0; \\
HTR: & (x_0 = \bot) \wedge (x_1 = \bot) \wedge (x_2 = \bot) \wedge (x_3 = \bot) & \longrightarrow x_0 := 0;
\end{array}
$$

In order to refine the high atomicity action HTR, we need to add a detector that detects the state predicate $(x_0 = \bot) \wedge (x_1 = \bot) \wedge (x_2 = \bot) \wedge (x_3 = \bot)$. In Section 5, we describe the specification of fault-tolerance components, and we show how we use a distributed detector to refine high atomicity actions.

Remark. Had we non-deterministically chosen to use PS_i ($i \neq 0$) as the process that adds the high atomicity recovery action then the high atomicity action HTR would have been different in that HTR would write x_i. (We refer the reader to [13] for a discussion about this issue.)

5 Specifying Pre-Synthesized Components

In this section, we describe the specification of fault-tolerance components (i.e., detectors and correctors). Specifically, we concentrate on detectors and we consider a special subclass of correctors where a corrector consists of a detector and a write action on the local variables of a single process.

5.1 The Specification of Detectors

We recall the specification of a detector component presented in [10,14]. Towards this end, we describe detection predicates, and witness predicates. A detector, say d, identifies whether or not a global state predicate, X, holds. The global state predicate X is called a *detection* predicate in the global state space of a distributed program [10, 14].

It is often difficult to evaluate the truth value of X in an atomic action. Thus, we (i) decompose the detection predicate X into a set of smaller detection predicates $X_0 \cdots X_n$ where the compositional detection of $X_0 \cdots X_n$ leads us to the detection of X, and (ii) provide a state predicate, say Z, whose value leads the detector to the conclusion that X holds. Since when Z becomes true its value witnesses that X is true, we call Z a *witness* predicate. If Z holds then X will have to hold as well. If X holds then Z will eventually hold and continuously remain *true*. Hence, corresponding to each detection predicate X_i, we identify a witness predicate Z_i such that if Z_i is *true* then X_i will be *true*.

The detection predicate X is either the conjunction of X_i $(0 \leq i \leq n)$ or the disjunction of X_i. Since the detection predicates that we encounter represent deadlock states, they are inherently in conjunctive form where each conjunct represents the valuation to program variables at some process. Hence, in the rest of the paper, we consider the case where X is a conjunction of X_i, for $0 \leq i \leq n$.

Specification. Let X and Z be state predicates. Let 'Z detects X' be the problem specification. Then, 'Z detects X' stipulates that

- (*Safety*) When Z holds, X must hold as well.
- (*Liveness*) When the predicate X holds and continuously remains *true*, Z will eventually hold and continuously remain *true*. □

We represent the safety specification of a detector as a set of transitions that a detector is not allowed to take. Thus, the following set of transitions represents the safety specification of a detector.

$$spec_d = \{(s_0, s_1) : (Z(s_1) \ \wedge \ \neg X(s_1))\}$$

Notation. The predicate $Z(s_1)$ denotes the truth value of Z at state s_1.

5.2 The Representation of Detectors

In this section, we describe how we formally represent a distributed detector. While our method allows one to use detectors of different topologies (cf. Section 6.1), in this section, we comprehensively describe the representation of a linear (sequential) detector as such a detector will be used in our token ring example.

The composition of detectors. A detector, say d, with the detection predicate $X \equiv X_0 \wedge \ldots \wedge X_n$ is obtained by composing d_i, $0 \leq i \leq n$, where d_i is responsible for the detection of X_i using a witness predicate Z_i $(0 \leq i \leq n)$. The elements of d can execute in parallel or in sequence. More specifically, parallel detection of X requires $d_0 \cdots d_n$ to execute concurrently. As a result, the state predicate $(Z_0 \wedge \cdots \wedge Z_n)$ is the witness predicate for detecting X.

A sequential detector requires the detectors $d_0 \cdots d_n$ to execute one after another. For example, given a linear arrangement $d_n \cdots d_0$, a detector d_i $(0 \leq$

$i < n$) detects its detection predicate, using Z_i, after d_{i+1} witnesses. Thus, when Z_i becomes *true*, it shows that Z_{i+1} already holds. Since when Z_i becomes *true* X_i must be also *true*, it follows that the detection predicates $X_n \cdots X_i$ hold. Therefore, we can atomically check the witness predicate Z_0 in order to identify whether or not $X \equiv (X_n \wedge \cdots \wedge X_0)$ holds.

The detection of global state predicates of programs that have a hierarchical topology (e.g., tree-like structures) requires parallel and sequential detectors. For brevity, we demonstrate our method in the context of a linear detector. As such a detector suffices for the example considered in this paper, we refer the reader to [7] for an illustration of this method for hierarchical components.

A linear detector. We consider a detector d with linear topology. The detector d consists of $n+1$ elements ($n > 0$), its specification $spec_d$, its variables, and its invariant U. Since the structure of the detector is linear, without loss of generality, we consider an arrangement $d_n \cdots d_0$ for the elements of the distributed detector, where the left-most element is d_n and the right-most element is d_0.

Component variables. Each element d_i, $0 \leq i \leq n$, of the detector has a Boolean variable y_i.

Read/write restrictions. Element d_i can read y_i and y_{i+1}, and can only write y_i ($0 \leq i < n$). d_n reads and writes y_n. Also, d_i is allowed to read r_i; i.e., the set of variables that are readable for a process P_i with which d_i is composed.

Witness predicates. The witness predicate of each d_i, say Z_i, is equal to $(y_i = true)$.

The detector actions. The actions of the linear detector are as follows ($0 \leq i < n$).

$$DA_n : (LC_n) \wedge (y_n = false) \longrightarrow y_n := true;$$
$$DA_i : (LC_i) \wedge (y_i = false) \wedge (y_{i+1} = true) \longrightarrow y_i := true;$$

Using action DA_i ($0 \leq i < n$), each element d_i of the linear detector witnesses (i.e., sets the value of y_i to *true*) whenever (i) the condition LC_i becomes *true*, where LC_i represents a local condition that d_i atomically checks (by reading the variables of P_i), and (ii) its neighbor d_{i+1} has already witnessed. The detector d_n witnesses (using action DA_n) when LC_n becomes true.

Detection predicates. The detection predicate X_i for element d_i is equal to $(LC_n \wedge \cdots \wedge LC_i)$ ($0 \leq i \leq n$). Therefore, d_0 detects the global detection predicate $LC_n \wedge \cdots \wedge LC_0$.

Invariant. During the detection, when an element d_i sets y_i to true, the elements d_j, for $i < j \leq n$, have already set their y values to true. Hence, we represent the invariant of the linear detector by the predicate U, where

$$U = \{s : (\forall i : (0 \leq i \leq n) : (y_i(s) \Rightarrow (\forall j : (0 \leq j \leq n) \wedge (j > i) : LC_j))\}$$

Faults. We model the fault transitions that affect the linear detector using the following action (cf. Section 7 for a discussion about the way that we have modeled the faults).

$$F : true \longrightarrow y_i := false;$$

Theorem 5.1 The linear detector is masking F-tolerant for 'Z detects X' from U. (cf. [13] for proof.) □

5.3 Token Ring Example Continued

In Section 4.2, we added the following high atomicity action to the token ring program ITR that is executed by the pseudo process PS_0.

$$HTR: \quad (x_0 = \bot) \wedge (x_1 = \bot) \wedge (x_2 = \bot) \wedge (x_3 = \bot) \longrightarrow \quad x_0 := 0$$

In order to synthesize a distributed program (that includes low atomicity actions), we need to refine the guard of the above action. The read/write restrictions of the processes in the token ring program identify the underlying communication topology of the fault-intolerant program, which is a ring. Hence, we select a linear detector, d, so that we can organize its elements, d_3, d_2, d_1, d_0, in the ring. Each detector d_i is responsible to detect whether or not the local conditions LC_3 to LC_i hold ($LC_i \equiv (x_i = \bot)$), for $0 \leq i \leq 3$. Thus, the detection predicate X_i is equal to $((x_3 = \bot) \wedge \cdots \wedge (x_i = \bot))$, for $0 \leq i \leq 3$. As a result, the global detection predicate of the linear detector is $((x_3 = \bot) \wedge (x_2 = \bot) \wedge (x_1 = \bot) \wedge (x_0 = \bot))$. The witness predicate of each d_i, say Z_i, is equal to $(y_i = true)$, and the actions of the sequential detector are as follows ($0 \leq i \leq 2$).

$$DA_3: (x_3 = \bot) \wedge (y_3 = false) \longrightarrow y_3 := true;$$
$$DA_i: (x_i = \bot) \wedge (y_i = false) \wedge (y_{i+1} = true) \longrightarrow y_i := true;$$

Note that we replace (LC_i) with $(x_i = \bot)$ in the above actions. During the synthesis, after the synthesis algorithm acquires the actions of its required component, it replaces each (LC_i) with the appropriate condition in order to create the transition groups corresponding to each action of the component.

6 Using Pre-Synthesized Components

In this section, we describe how we perform the second and the third step of our synthesis method presented in Figure 1. In particular, in Section 6.1, we show how we automatically specify the required components during the synthesis. Then, in Section 6.3, we show how we ensure that no interference exists between the program and the fault-tolerance component. Afterwards, we present an algorithm for the addition of fault-tolerance components. In Sections 6.2 and 6.4, we respectively present the algorithmic specification and the algorithmic addition of a linear detector to the token ring program.

6.1 Algorithmic Specification of the Fault-Tolerance Components

We present the Component_Specification algorithm (cf. Figure 2) that takes a deadlock state s_d, the distribution restrictions (i.e., the read/write restrictions) of the program being synthesized, and the set of high atomicity pseudo processes PS_i ($0 \leq i \leq n$). First, the algorithm searches for a high atomicity process PS_{index} that is able to add a high atomicity recovery action, $ac : grd \rightarrow st$, from s_d to a state in the state predicate S_{rec}, where S_{rec} represents the set of states from where there exists a safe recovery path to the invariant. Also, we verify the closure of $S_{rec} \cup s_d$ in the computations of $p[]f$. If there exists such a process PS_{index} then the algorithm returns a triple $\langle X, R, index \rangle$, where (i) X is the detection predicate that should be refined in the refinement of the action ac; (ii) R is a relation that represents the topology of the program, and (iii) the $index$ is an integer that identifies the process that should detect grd and execute st.

The Component_Specification algorithm constructs the state predicate X using the LC_i conditions. Each LC_i condition is by itself a conjunction that consists

of the program variables readable for process P_i. Therefore, the predicate X will be the conjunction of LC_i conditions $(0 \leq i \leq n)$.

The relation $R \subseteq (P \times P)$ identifies the communication topology of the distributed program, where P is the set of program processes. We represent R by a finite set $\{\langle i,j\rangle : (0 \leq i \leq n) \wedge (0 \leq j \leq n) : w_i \subseteq r_j\}$ that we create using the read/write restrictions among the processes. The presence of a pair $\langle i,j\rangle$ in R shows that there exists a communication link between P_i and P_j. Since we internally represent R by an undirected graph, we consider the pair $\langle i,j\rangle$ as an unordered pair.

The interface of the fault-tolerance components. The format of the interface of each component is the same as the output of the Component_Specification algorithm, which is a triple $\langle X, R, index\rangle$ as described above. We use this interface to extract a component from the component library using a pattern-matching algorithm. Towards this end, we use existing specification-matching techniques [15]. For reasons of space, we omit the details of the component extraction from the library of components.

Component_Specification(s_d: state, S_{rec}: state predicate, $PS_0, \cdots, PS_n$: high atomicity pseudo process, *spec*: safety specification, $r_0, \cdots, r_n$: read restrictions, $w_0, \cdots, w_n$: write restrictions)
{ // n is the number of processes.
 if ($\exists index : 0 \leq index \leq n : (\exists s : s \in S_{rec} : (s_d, s) \in PS_{index} \wedge$
 $((s_d, s)$ does not violate *spec*$) \wedge (\forall x : (x(s_d) \neq x(s)) : x \in w_{index}))$)
 then $X := \wedge_{i=0}^{n}(LC_i)$, where $LC_i = (\wedge^{|r_i|}(x = x(s_d)))$;
 $R = \{\langle i,j\rangle : (0 \leq i \leq n) \wedge (0 \leq j \leq n) : w_i \subseteq r_j\}$;
 return X, R, *index*;
 else return $false$, $\emptyset$, -1;
}

Fig. 2. Automatic specification of a component.

The output of the component library. Given the interface $\langle X, R, index\rangle$ of a required component, the component library returns the witness predicate, Z, the invariant, U, and the set of transition groups, $gd_0 \cup \cdots \cup gd_k \cup g_{index}$, of the pre-synthesized component ($k \geq 0$). The group of transitions g_{index} represents the low atomicity write action that should be executed by process P_{index}.

Complexity. Since the algorithm Component_Specification checks the possibility of adding a high atomicity recovery action to each state of S_{rec}, its complexity is polynomial in the number of states of S_{rec}.

6.2 Token Ring Example Continued

We trace the algorithm of Figure 2 for the case of token ring program. First, we non-deterministically identify PS_0 as the process that can read every program variable and can add a high atomicity recovery transition from the deadlock state $s_d = \langle \bot, \bot, \bot, \bot\rangle$. Thus, the value of *index* will be equal to 0. Second, we construct the detection predicate X, where $X \equiv ((x_0 = \bot) \wedge (x_1 = \bot) \wedge (x_2 = \bot) \wedge (x_3 = \bot))$. Finally, using the read/write restrictions of the processes in the token ring program, the relation R will be equal to $\{\langle 0,1\rangle, \langle 1,2\rangle, \langle 2,3\rangle, \langle 3,0\rangle\}$.

6.3 Algorithmic Addition of The Fault-Tolerance Components

In this section, we present an algorithm for adding a fault-tolerance component to a fault-intolerant distributed program to resolve a deadlock state s_d. Before the addition, we ensure that no interference exists between the program and the fault-tolerance component; i.e., the execution of one of them does not violate the (safety or liveness) specification of the other one. We show that our addition algorithm is sound; i.e., the synthesized program satisfies the requirement of the addition problem (cf. Section 3).

We represent the transitions of p by the union of its groups of transitions (i.e., $\cup_{i=0}^{m} g_i$). We also assume that we have extracted the required pre-synthesized component, c, as described in Section 6.1. The component c consists of a detector d that includes a set of transition groups $\cup_{i=0}^{k} gd_i$, and the write action of the pseudo process PS_{index} represented by a group of transitions g_{index} in the low atomicity.

The state space of the composition of p and d is the new state space $S_{p'}$. We introduce an onto function $H_1 : S_{p'} \rightarrow S_p$ (respectively, $H_2 : S_{p'} \rightarrow S_d$) that maps the states in the new state space $S_{p'}$ to the states in the old state space S_p (respectively, S_d where S_d is the state space of the detector d). Now, we show how we verify the interference-freedom of the composition of c and p.

Interference-freedom. In order to ensure that no interference exists between p and c, we have to ensure the following three conditions hold in the new state space $S_{p'}$: (i) transitions of p do not interfere with the execution of d; (ii) transitions of d do not interfere with the execution of p, and (iii) the low atomicity write action associated with c does not interfere with the execution of p and d.

First, we ensure that the set of transitions of p do not interfere with the execution of d by constructing the set of groups of transitions I_1, where I_1 contains those groups of transitions in the new state space $S_{p'}$ that violate either the safety of d or the closure of its invariant U.

$$I_1 = \{g : (\exists g_j : (g_j \in p) \wedge (0 \leq j \leq m) : (H_1(g) = g_j) \wedge (\exists (s'_0, s'_1) : (s'_0, s'_1) \in g : ((s'_0, s'_1) \text{ violates } spec_d) \vee (H_2(s'_0) \in U \wedge H_2(s'_1) \notin U))\}$$

The transitions of p do not interfere with the liveness of d because d need not execute when p is not in the deadlock state s_d. Thus, p does not affect the liveness of d. Hence, we are only concerned with the safety of the detector d and the closure of U. When we map the transitions of p to the new state space, the mapped transitions should preserve the safety of d. Moreover, if the image of a transition (s'_0, s'_1) starts in U (i.e., $H_2(s'_0) \in U$) then the image of (s'_0, s'_1) will have to end in U (i.e., $H_2(s'_1) \in U$). The emptiness of I_1 shows that the transitions of p do not interfere with the execution of d.

Likewise, we construct the set of groups of transitions I_2 and I_3 in the new state space $S_{p'}$ to verify the second and the third conditions of interference-freedom. Since I_2 and I_3 are structurally similar to I_1, we skip their presentation (cf. [13] for details). Thus, if I_1, I_2, and I_3 are empty then we declare that no interference will happen due to the addition of c to p.

Addition. We present the Add_Component algorithm for an interference-free addition of the fault-tolerance component c to p. In the new state space $S_{p'}$, we construct a set of transition groups p_{H_1} (respectively, d_{H_2}) that includes all groups of transitions, g, whose images in S_p (respectively, S_d) belong to p (respectively, d). Besides, no transition of $(s'_0, s'_1) \in g$ violates the safety specification of d (respectively, p) or the closure of the invariant of d (respectively, p), i.e., U (respectively, S). Note that in the set d_{H_2}, the image of every group g in d and p must belong to the same process.

Add_Component(S, S_{rec}, U: state predicate, H_1, H_2: onto mapping function,
$spec, spec_d$: safety specification, $g_0, \cdots, g_m$, $gd_0, \cdots, gd_k, g_{index}$: groups of transitions)
{ // $p = g_0 \cup \cdots \cup g_m$, and $d = gd_0 \cup \cdots \cup gd_k \cup g_{index}$
// $P_0 \cdots P_n$ are the processes of p, and $d_0 \cdots d_n$ are the elements of d

$p_{H_1} = \{g : (\exists g_j : (g_j \in p) \wedge (0 \leq j \leq m) : (H_1(g) = g_j) \wedge$
$(\forall (s'_0, s'_1) : (s'_0, s'_1) \in g : ((s'_0, s'_1)$ does not violate $spec_d) \wedge (H_2(s'_0) \in U \Rightarrow H_2(s'_1) \in U))\}$

$d_{H_2} = \{gd : (\exists gd_j : (gd_j \in d) \wedge (0 \leq j \leq k) : (H_2(gd) = gd_j) \wedge$
$(\exists d_i, P_l : (0 \leq i \leq n) \wedge (0 \leq l \leq n) : (H_2(gd) \in d_i) \wedge (H_1(gd) \in P_l) \wedge (l = i)) \wedge$
$(\forall (s'_0, s'_1) : (s'_0, s'_1) \in gd : ((s'_0, s'_1)$ does not violate $spec) \wedge (H_1(s'_0) \in S \Rightarrow H_1(s'_1) \in S))\}$

$p_c := \{g : (H_2(g) = g_{index}) \wedge (\forall (s'_0, s'_1) : (s'_0, s'_1) \in g : ((s'_0, s'_1)$ does not violate $spec) \wedge$
$(H_1(s'_1) \in S_{rec}) \wedge (H_2(s'_0) \in U \Rightarrow H_2(s'_1) \in U) \wedge ((s'_0, s'_1)$ does not violate $spec_d))\}$
$S' := \{s : s \in S_{p'} : H_1(s) \in S \wedge H_2(s) \in U\}$
$p' := p_{H_1} \cup d_{H_2} \cup p_c$;
return p', S';
}

Fig. 3. The automatic addition of a component.

The set p_c includes all groups of transitions, g, whose every transition has an image in g_{index} under the mapping H_2. Further, no transition $(s'_0, s'_1) \in g$ violates the safety of *spec* or the closure of S.

The set of states of the invariant of the synthesized program, S', consists of those states whose images in S_p belong to the program invariant S and whose images in the state space of the detector, S_d, belong to the detector invariant U.

Theorem 6.1 The algorithm Add_Component is sound. (cf. [13] for proof.) □

Theorem 6.2 The complexity of Add_Component is polynomial in $S_{p'}$. (cf. [13] for proof.) □

6.4 Token Ring Example Continued

Using Add_Component, we add the detector specified in Section 6.2 to the token ring program MTR introduced in Section 4.2. The resulting program, consisting of the processes $P_0 \cdots P_3$ arranged in a ring, is masking fault-tolerant to process-restart faults. We represent the transitions of P_0 by the following actions.

MTR_0 : $((x_0 = 1) \vee (x_0 = \perp)) \wedge (x_3 = 1) \longrightarrow x_0 := 0;$
MTR'_0 : $((x_0 = 0) \vee (x_0 = \perp)) \wedge (x_3 = 0) \longrightarrow x_0 := 1;$
D_0 : $(x_0 = \perp) \wedge (y_0 = false) \wedge (y_1 = true) \longrightarrow y_0 := true;$
C_0 : $(y_0 = true) \longrightarrow x_0 := 0;\ y_0 := false;$

The actions MTR_0 and MTR'_0 are the same as the actions of the MTR program presented in Section 4.2. The action D_0 belongs to the sequential detector

that sets the witness predicate Z_0 to true. The action C_0 is the recovery action that P_0 executes whenever the witness predicate ($y_0 = true$) becomes *true*. Now, we present the actions of P_3.

$$\begin{array}{lll} MTR_3: & ((x_3 = 0) \vee (x_3 = \bot)) \ \wedge \ (x_2 = 1) & \longrightarrow x_3 := 1; y_3 := false; \\ MTR_3': & ((x_3 = 1) \vee (x_3 = \bot)) \ \wedge \ (x_2 = 0) & \longrightarrow x_3 := 0; y_3 := false; \\ D3: & (x_3 = \bot) \ \wedge \ (y_3 = false) & \longrightarrow y_3 := true; \end{array}$$

The action D_3 belongs to the detector d_3 that sets Z_3 to *true*. We represent the transitions of P_1 and P_2 as the following parameterized actions (for $i = 1, 2$).

$$\begin{array}{lll} MTR_i: & ((x_i = 0) \vee (x_i = \bot)) \ \wedge \ (x_{i-1} = 1) & \longrightarrow x_i := 1; y_i := false; \\ MTR_i': & ((x_i = 1) \vee (x_i = \bot)) \ \wedge \ (x_{i-1} = 0) & \longrightarrow x_i := 0; y_i := false; \\ D_i: & (x_i = \bot) \ \wedge \ (y_i = false) \ \wedge \ (y_{i+1} = true) & \longrightarrow y_i := true; \end{array}$$

The above program is masking fault-tolerant for the faults that corrupt one or more processes. Note that when a process P_i ($1 \leq i \leq 3$) changes the value of x_i to a non-corrupted value, it falsifies Z_i (i.e., y_i). The falsification of Z_i is important during the recovery from $s_d = \langle \bot, \bot, \bot, \bot \rangle$ in that when x_i takes a non-corrupted value, the detection predicate X_i no longer holds. Thus, if Z_i remains true then the detector d_i witnesses incorrectly, and as a result, violates the safety of the detector. However, P_0 does not need to falsify its witness predicate Z_0 in actions MTR_0 and MTR_0' because the action C_0 has already falsified Z_0 during a recovery from s_d.

7 Discussion

In this section, we address some of the questions raised by our synthesis method. Specifically, we discuss the following issues: the model of faults considered in this paper, the fault-tolerance of the components, the choice of detectors and correctors, and pre-synthesized components with non-linear topologies.

Does the fault model used in this paper enable us to capture different types of faults?

Yes. The notion of state perturbation is general enough to model different types of faults (namely, stuck-at, crash, fail-stop, omission, timing, or Byzantine) with different natures (intermittent, transient, and permanent faults). As an illustration, we modeled the process-restart faults that affect the token ring program, presented in this paper, by state perturbation. This model has also been used in designing fault-tolerance to (i) fail-stop, omission faults (e.g., [9]), (ii) transient faults and improper initialization (e.g., [16]), and (iii) input corruption (e.g., [9]).

Can the synthesis method deal with the faults that affect the fault-tolerance components?

Yes. The added component may itself be perturbed by the fault to which fault-tolerance is added. Hence, the added component must itself be fault-tolerant. For example, in our token ring program, we modeled the effect of the process restart on the added component and ensured that the component is fault-tolerant to that fault (cf. Theorem 5.1). For the fault-classes that are commonly used, e.g., process failure, process restart, input corruption, Byzantine faults, such modeling is always possible. For arbitrary fault-classes, however, some *validation* may be required to ensure that the modeling is appropriate for that fault.

How does the choice of detectors and correctors help in the synthesis of fault-tolerant programs?

While there are several approaches (e.g., [17]) that manually transform a fault-intolerant program into a fault-tolerant program, we use detectors and correctors in this paper, based on their necessity and sufficiency for manual addition of fault-tolerance [5]. The authors of [5] have also shown that detectors and correctors are abstract enough to generalize other components (e.g., comparators and voters used in replication-based approaches) for the design of fault-tolerant programs. Hence, we expect that the synthesis method in this paper can benefit from the generality of detectors and correctors in the *automated* synthesis of fault-tolerant programs as there is a potential to provide a rich library of fault-tolerance components. Moreover, pre-synthesized detectors provide the kind of abstraction by which we can integrate efficient existing detections approaches (e.g., [18, 19]) in pre-synthesized fault-tolerance components.

Does the synthesis method support pre-synthesized components with non-linear topologies?

Yes. Using the synthesis method of this paper, we have added presynthesized components with tree-like structure to a diffusing computation program [7] where we synthesize a program that is fault-tolerant to the faults that perturb the state of the diffusing computation and the program topology.

8 Conclusion and Future Work

In this paper, we identified an approach for the synthesis of fault-tolerant programs from their fault-intolerant versions using pre-synthesized fault-tolerance components. Our approach differs from the synthesis method presented in [20] where one has to synthesize a fault-tolerant program from its temporal logic specification. Specifically, we demonstrated a hybrid synthesis method that combines heuristics presented in [2–4] with pre-synthesized detectors and correctors. We presented a sound algorithm for automatic specification and addition of pre-synthesized detectors/correctors to a distributed program. We showed how one could verify the interference-freedom of the fault-intolerant program and the added components. Using our synthesis algorithm, we showed how masking fault-tolerance is added to a token-ring program where all processes may be corrupted. By contrast, the previous synthesis algorithms fail to synthesize a fault-tolerant program when all processes are corrupted.

We also extended the problem of adding fault-tolerance to the case where new variables can be introduced while synthesizing fault-tolerant programs. By contrast, previous algorithms required that the state space of the fault-tolerant program is the same as that of the fault-intolerant program. Moreover, our synthesis method controls the way new variables are introduced; new variables are determined based on the added components. Hence, the synthesis method controls the way in which the state space is expanded.

References

1. S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, page 82, 2000.

2. S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of byzantine agreement. *Symposium on Reliable Distributed Systems*, 2001.
3. S. S. Kulkarni and A. Ebnenasir. The complexity of adding failsafe fault-tolerance. *International Conference on Distributed Computing Systems*, pages 337–334, 2002.
4. S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. *International Conference on Distributed Computing Systems*, pages 441–450, 2003.
5. S. S. Kulkarni. *Component-based design of fault-tolerance*. PhD thesis, Ohio State University, 1999.
6. Ali Ebnenasir and Sandeep S. Kulkarni. FTSyn: A framework for automatic synthesis of fault-tolerance. `http://www.cse.msu.edu/~ebnenasi/research/tools/ftsyn.htm`.
7. Ali Ebnenasir and S.S. Kulkarni. Hierarchical presynthesized components for automatic addition of fault-tolerance: A case study. *In the extended abstracts of the ACM workshop on the Specification and Verification of Component-Based Systems (SAVCBS), Newport Beach, California*, 2004.
8. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
9. A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.
10. A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *International Conference on Distributed Computing Systems*, pages 436–443, May 1998.
11. P. Attie and A. Emerson. 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), March 2001.
12. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1990.
13. S. S. Kulkarni and Ali Ebnenasir. Adding fault-tolerance using pre-synthesized components. *Technical report MSU-CSE-03-28, Department of Computer Science, Michigan State University, East Lansing, Michigan, USA. A revised version is available at* `http://www.cse.msu.edu/~sandeep/auto_component_techreport.ps`, 2003.
14. A. Arora and S. S. Kulkarni. Component based design of multi-tolerant systems. *IEEE Transactions on Software Engineering*, 1998.
15. A. Moormann Zaremski and J.M. Wing. Specification matching of software components. *in proceedings of the 3 rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995.
16. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 1974.
17. Z. Liu and M. Joseph. Transformations of programs for fault-tolerance. *Formal Aspects of Computing*, 1992.
18. A.I. Tomlinson and V.K. Garg. Detecting relational global predicates in distributed systems. *In proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, California.*, pages 21–31, May 1993.
19. Neeraj Mittal. *Techniques for Analyzing Distributed Computations*. PhD thesis, The University of Texas at Austin, 2002.
20. A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC)*, 1998.