Hierarchical Presynthesized Components for Automatic Addition of Fault-Tolerance: A Case Study[1]

Ali Ebnenasir Sandeep S. Kulkarni
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing MI 48824 USA

Abstract

We present a case study of automatic addition of fault-tolerance to distributed programs using presynthesized distributed components. Specifically, we extend the scope of automatic addition of fault-tolerance using presynthesized components to the case where we automatically add hierarchical *components to fault-intolerant programs, whereas in our previous work, we have shown the addition of linear presynthesized components to programs. Also, our case study provides an example for the cases where multiple components are simultaneously added to a program. Towards this end, we present an automatically generated diffusing computation program that provides nonmasking fault-tolerance – where, in the presence of faults, the nonmasking program guarantees recovery to states from where it satisfies its safety and liveness specifications. Since presynthesized components provide reuse in the synthesis of fault-tolerant distributed programs, we expect that our method will pave the way for automatic addition of fault-tolerance to large-scale programs.*

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

1 Introduction

In this paper, we present a case study for automatic addition of presynthesized fault-tolerance components to distributed programs using a software framework called Fault-Tolerance Synthesizer (FTSyn) [1]. Specifically, we use FTSyn to add components with *hierarchical* topology to a diffusing computation program to provide recovery in the presence of faults. Presynthesized fault-tolerance components provide *reuse* in the synthesis of fault-tolerant distributed programs from their fault-intolerant version. Such reuse is particularly beneficial in dealing with the exponential complexity of synthesis [2,3]. Also, fault-tolerance components provide an abstraction that simplifies the reasoning about the fault-tolerance and functional concerns.

[1]Email: ebnenasi@cse.msu.edu, sandeep@cse.msu.edu
Web: `http://www.cse.msu.edu/~{ebnenasi,sandeep}`
Tel: +1-517-355-2387, Fax: +1-517-432-1061
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.

The FTSyn framework incorporates the results of [4] where the synthesis algorithm automatically specifies and adds presynthesized fault-tolerance components, namely *detectors* and *correctors*, to fault-intolerant programs during the synthesis of their fault-tolerant version. It is shown in the literature [5] that such components are necessary and sufficient for the *manual* design of fault-tolerant programs. As a result, we expect to benefit from their generality in *automatic* addition of fault-tolerance as well.

In [4], the synthesis algorithm is applied to programs where the underlying communication topology between processes is linear. In this paper, we show how we add *hierarchical* presynthesized components to distributed programs. Specifically, we add tree-like structured components to a diffusing computation program where processes are arranged in an outtree, where the indegree of each node is at most one. A diffusing computation starts at the root and propagates throughout the tree, and then, reflects back up to the root of the tree. The fault-intolerant program is subject to faults that perturb the state of the diffusing computation and the topology of the program (i.e., the parenting relationship amongst processes).

This case study shows that the synthesis method presented in [4] handles presynthesized components (respectively, distributed programs) with different topologies as we have already reused a particular linear component in the synthesis of a token ring program and an alternating bit protocol [4]. Also, we extend the scope of the synthesis to the case where we simultaneously add multiple presynthesized components to the program being synthesized. Moreover, the use of presynthesized components provides an abstraction that modularizes the reasoning about the correctness of each individual component and the composition of components and program.

Note. The notion of program in this paper refers to the abstract structure of a program that 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. Subsequently, in Section 3, we describe how we formally represent a hierarchical fault-tolerance component. In Section 4, we show how we automatically add a hierarchical component to a diffusing computation program. Finally, we make concluding remarks and discuss future work in Section 5.

2 Preliminaries

In this section, first, we present basic concepts in Section 2.1. Then, in Section 2.2, we represent the formal problem statement of adding fault-tolerance components to programs (adapted from [4]). In Section 2.3, we give an informal overview of the synthesis method presented in [4] as our presentation in the rest of the paper follows that synthesis method.

2.1 Basic Concepts

In this subsection, 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. The definition of specifications is adapted from Alpern and Schneider [6]. The definition of faults and fault-tolerance is adapted from Arora and Gouda [7] and Kulkarni and Arora [2]. The issues of modeling distributed programs is adapted from [2, 8].

Program. A program p is specified by a finite set of variables, say $V = \{v_0, v_2, .., v_q\}$, and a finite set of processes, say $P = \{P_0, \cdots, P_n\}$, where q and n are positive integers. Each variable is associated with a finite domain of values. Let $v_0, v_2, .., v_q$ be variables of p, and let $D_0, D_2, .., D_q$ 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 is of the form: $\langle l_0, l_2, .., l_q \rangle$ where $\forall i : 0 \leq i \leq q : l_i \in D_i$. The state space of p, S_p, is the set of all possible states of p.

A process, say P_j ($0 \leq j \leq n$), 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. 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 finite 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 of p that start in S and end in S. We represent $p|S$ as the set 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 transitions that write (i.e., update) 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, hence, we do not need to 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 [6], we let the specification consist of a safety specification and a liveness specification. For a suffix-closed and fusion-closed specification, the safety specification can be specified as a set of bad transitions [5], 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 its invariant from where it *satisfies* its 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 for S 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.

Nonmasking fault-tolerance. Given a program p, its invariant, S, its specification, $spec$, and a class of faults, f, we say p is nonmasking 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, and every computation of $p[]f$ that starts from a state in T has a state in S.

Remark on the program representation. We use Dijkstra's

guarded commands [9] to represent the set of program (respectively, fault) transitions. A guarded command (action) is of the form $grd \rightarrow st$, where grd is a state predicate and st is an assignment statement that updates 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 $s_1\}$.

2.2 Problem Statement

In this subsection, we represent the problem statement of adding fault-tolerance components to fault-intolerant programs adapted from [4]. 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 presynthesized fault-tolerance components to p in order to synthesize a fault-tolerant program p' with the new invariant S'. Since each component has its own set of variables, we expand the state space of p by adding a fault-tolerance component to it. The new state space, say $S_{p'}$, is actually the state space of the synthesized fault-tolerant program p'.

To map the states and the transitions of p' to the states and the transitions of p, we define an onto function $H : S_{p'} \rightarrow S_p$, which can be applied on the domain of states, state predicates, transitions, and groups of transitions. The function H provides a mapping from the state space of the composition of the program and the fault-tolerance component to the state space of the fault-intolerant program. In other words, we use function H as a projection from the new state space to the old state space.

Now, since we require p' not to include new behaviors in the absence of faults, the invariant S' cannot contain states s'_0 whose image $H(s'_0)$ is not in S. Otherwise, in the absence of faults, p' will include computations in the new state space $S_{p'}$ that do not have corresponding computations in p. Hence, we have $H(S') \subseteq S$. Likewise, we require p' not to contain 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 $H(s'_1) = s_1$). Otherwise, p' may create a new way for satisfying $spec$ in the absence of faults. Therefore, the problem of adding fault-tolerance components to programs is as follows:

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 nonmasking f-tolerant for $spec$ from S'. □

2.3 The Synthesis Method

In this subsection, we present an informal overview of the synthesis method presented in [4]. We note that the presentation of this subsection suffices for this paper, however, the interested reader may refer to [4] for a formal presentation.

The essential task in the synthesis of nonmasking fault-tolerant programs is to ensure that the program recovers to its invariant when faults perturb it outside its invariant. Towards this end, the synthesis algorithm must ensure that there exist (i) no deadlock states, and (ii) no non-progress cycles outside the invariant. As a result, the recovery to the invariant will be guaranteed. The complexity of resolving deadlocks and non-progress cycles depends on the program model. In the high atomicity model, where processes can read/write all program variables in an atomic step, the addition of nonmasking fault-tolerance can be done in polynomial time [2].

For distributed programs, where processes have read/write restrictions with respect to program variables, adding nonmasking fault-tolerance is non-trivial since removing non-progress cycles and resolving deadlock states become conflicting tasks due to grouping issues (i.e., read restrictions). For example, when the synthesis algorithm adds a recovery transition t_{rec} from a deadlock state outside the invariant, the transitions that are grouped with t_{rec} due to read restrictions may create non-progress cycles with other transitions.

To deal with such difficulties of adding recovery, the synthesis algorithm presented in [4] provides a hybrid approach where it uses heuristics (developed in [10, 11]) along with presynthesized fault-tolerance components. Specifically, the algorithm of [4] first uses heuristics under distribution restrictions to add recovery from a specific deadlock state s_d. If the heuristics fail then the synthesis algorithm adds presynthesized detectors (respectively, correctors) to resolve the deadlock state s_d (cf. Section 3 for a formal definition of detectors/correctors). To add a presynthesized component (i.e., detectors/correctors), the synthesis algorithm automatically (i) specifies the required component; (ii) extracts the necessary component from an existing component library; (iii) ensures that the components do not *interfere* with the program execution, i.e., the program and the presynthesized components satisfy their specifications in the presence of each other, and (iv) adds the components.

To automatically specify and add the required components during the synthesis of a distributed program p with n processes $\{P_1, \cdots, P_n\}$, the synthesis algorithm of [4] introduces a high atomicity processes P_{high_i} corresponding to each P_i $(1 \leq i \leq n)$. Each P_{high_i} is allowed to read all program variables and has the write abilities of P_i. At the outset of the synthesis, process P_{high_i} does not have any action to execute. For a specific deadlock state s_d, the synthesis algorithm determines whether there exists a high atomicity process P_{high_i} that can add recovery from s_d, given its high atomicity abilities.

If P_{high_k}, for some $1 \leq k \leq n$, succeeds in adding high atomicity recovery from s_d then the synthesis algorithm automatically specifies and extracts the desired detectors (respectively, correctors) for the refinement of the added high atomicity recovery actions. If the presynthesized detectors (respectively, correctors) do not *interfere* with program execution then the refinement will be successful. Otherwise, the synthesis algorithm of [4] fails to add recovery to s_d.

3 Specifying Hierarchical Components

In this section, we describe the specification and the representation of hierarchical 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 process. We have adapted the specification of detectors from [5].

The Specification of Detectors. We recall the specification of a detector component presented in [5]. 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 [5].

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 $X_0, \cdots, X_{n-1}$ where the compositional detection of $X_0, \cdots, X_{n-1}$ 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* as well.

The detection predicates that we encounter represent deadlock states. Thus, 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 $0 \leq i < n$.

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 execute. Thus, the following set of transitions represents the safety specification of a detector.

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

The Representation of Hierarchical Detectors. We focus on the representation of a detector with a tree-like structure as a special case of hierarchical detectors. The hierarchical detector d consists of n elements d_i ($0 \leq i < n$), its specification $spec_d$, its variables, and its invariant U. We introduce a relation $\preceq$ on the elements d_i that represents the parenting relation between the nodes of the tree; e.g., $i \preceq j$ means d_i is the parent of d_j.

The element d_0 is placed at the root of the tree and other elements of the detector are placed in other nodes of the tree. Each node d_i has its own detection predicate X_i and witness predicate Z_i. The siblings of a node can detect their detection predicate in parallel. However, the truth-value of the detection predicate of each node depends on the truth-value of its children. In other words, node d_i can witness if all its children have already witnessed.

Each element d_i, $0 \leq i < n$, of the detector has a Boolean variable y_i that represents its witness predicate; i.e., the witness predicate of each d_i, say Z_i, is equal to $(y_i = true)$. Also, the element d_i can read/write the y values of its children and its parent ($0 \leq i < n$). Moreover, each element d_i is allowed to read the variables that P_i can read. Now, we present the *template* action of the detector d_i as follows ($(0 \leq i, j, k < n) \wedge \ (j < k) \wedge (\forall r : j \leq r \leq k : i \preceq r)$):

$$DA_i : \ (LC_i) \ \wedge \ (y_j \wedge \cdots \wedge y_k) \ \wedge \ (y_i = false) \longrightarrow \ y_i := true;$$

Using action DA_i ($0 \leq i < n$), each element d_i of the hierarchical 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 children $d_j, \cdots, d_k$ have already witnessed ($(0 \leq j, k < n) \wedge (j < k)$). The detection predicate X_i for element d_i is equal to $(LC_i \ \wedge \ LC_j \ \wedge \ \cdots \ \wedge \ LC_k)$. Therefore, d_0 detects the global detection predicate $LC_0 \wedge \cdots \wedge LC_{n-1}$.

The above action is an abstract template that should be instantiated by the synthesis algorithm during the synthesis of a specific program in such a way that the program and the detector do not *interfere*; i.e., the program and the detector specifications are satisfied in the presence of each other. For automatic addition of nonmasking fault-tolerance, the interference-freedom of the program and the detector requires that (i) in the absence of faults, the program specification and the safety specification of detectors are satisfied, and (ii) in the presence of faults, recovery is provided by the composition of the program and the detectors.

During the detection, when d_i sets y_i to true, its children have already set their y values to true. Hence, we represent the invariant of the hierarchical detector by the predicate U, where

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

4 Diffusing Computation

In this section, we present the addition of a hierarchical presynthesized component to a fault-intolerant diffusing computation. We have adapted the diffusing computation program from [12]. First, in Subsection 4.1, we give the specification of the diffusing computation program. Then, in Subsection 4.2, we present the synthesized nonmasking fault-tolerant program before the addition of the hierarchical component, which includes high atomicity recovery actions. Finally, in Subsection 4.3, we show how we add presynthesized components to refine the high atomicity actions added during synthesis.

4.1 Diffusing Computation Program

The diffusing computation (DC) program consists of four processes $\{P_0, P_1, P_2, P_3\}$ whose underlying communica-

tion is based on a tree topology. The process P_0 is the root of the tree. Processes P_1 and P_2 are the children of P_0 (i.e., $(0 \preceq 1) \wedge (0 \preceq 2)$) and P_3 is the child of P_2 (i.e., $2 \preceq 3$).

Starting from a state where every process is green, P_0 initiates a diffusing computation throughout the tree by propagating the red color towards the leaves. The leaves reflect the diffusing computation back to the root by coloring the nodes green. Afterwards, when all processes become green again, the cycle of diffusing computation repeats.

Each process P_j ($0 \leq j \leq 3$) has a variable c_j that represents its color and whose domain is $\{0, 1\}$, where 0 represents the red and 1 represents the green. Also, process P_j has a Boolean variable sn_j that represents the session number of the diffusing computation where P_j is currently participating. Thus, we use sn_j to distinguish the case where P_j *has not started to participate in the current diffusing computation* from the case where P_j *has completed the current session of diffusing computation*. Moreover, each process has a variable par_j that represents the parent of P_j. The domain of par_j is equal to $\{0, 1, 2, 3\}$. The value of par_j identifies the node from where there exists an edge to P_j in the out-tree. For example, since the parent of P_0 is itself, we have $par_0 = 0$.

Program actions. The actions of the process P_j ($0 \leq j < 4$) are as follows:

$$
\begin{array}{l}
DC_{j1}: \ (c_j = 1) \ \wedge \ (par_j = j) \quad \longrightarrow \ c_j := 0; \ sn_j = \neg sn_j; \\
DC_{j2}: \ (c_j = 1) \ \wedge \ (c_{par_j} = 0) \ \wedge \ (sn_j \not\equiv sn_{par_j}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \longrightarrow \ c_j := c_{par_j}; \ sn_j = sn_{par_j}; \\
DC_{j3}: \ (c_j = 0) \ \wedge \ (\forall k :: (par_k = j) \Rightarrow (c_k = 1 \wedge sn_j \equiv sn_k)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \longrightarrow c_j := 1;
\end{array}
$$

Read/write restrictions. Each process P_j is allowed to read/write the variables of its children and its parent. For example, process P_0 can read/write its local variables and the local variables of P_1 and P_2. However, P_0 is not allowed to read/write the variables of P_3. Also, P_3 cannot read/write the variables of P_0 and P_1.

Invariant. In each session of diffusing computation, every process P_j meets one of the following requirements: (i) P_j and P_{par_j} have both started participating; (ii) P_j and P_{par_j} have both completed the current session of diffusing computation; (iii) P_j has not started participating in the current session whereas P_{par_j} has, and (iv) P_j has completed participating in the current session whereas P_{par_j} has not. Hence, the invariant of the program contains all state where S_{DC} holds, where

$$
\begin{array}{l}
S_{DC} = (\forall j : (0 \leq j \leq 3) : \\
\quad ((c_j = c_{par_j} \ \wedge \ sn_j \equiv sn_{par_j}) \ \vee \ (c_j = 1 \ \wedge \ c_{par_j} = 0))) \ \wedge \\
\qquad\qquad (par_0 = 0 \wedge par_1 = 0 \wedge par_2 = 0 \wedge par_3 = 2)
\end{array}
$$

Faults. Fault transitions can perturb the values of c_j and sn_j ($0 \leq j \leq 3$), and the underlying communication topology of the program. We represent the fault transitions by the following actions:

$$
\begin{array}{lll}
F_{j_1}: & (true) & \longrightarrow \ c_j = 0|1; \\
F_{j_2}: & (true) & \longrightarrow \ sn_j = false|true; \\
F_0: & (true) & \longrightarrow \ par_0 = 0|1|2;
\end{array}
$$

The actions F_{j_1} and F_{j_2} represent the fault transitions that perturb a process P_j whereas action F_0 only affects P_0. The class of faults F_0 perturbs the parenting relationship by changing the value of par_0 to one of the values $\{0, 1, 2\}$. We have included fault-class F_0 since it perturbs the DC program to states where we can demonstrate the advantages of using presynthesized components in dealing with deadlock states.

4.2 Intermediate Nonmasking Program

In this subsection, we present the intermediate nonmasking fault-tolerant program that includes high atomicity recovery actions. We have synthesized this intermediate program using our software framework FTSyn [1].

The faults may perturb the state of the DC program outside S_{DC} where the program may fall in a non-progress cycle or reach a deadlock state. For example, faults F_0 may perturb the program to states where the condition $T_{deadlock} \equiv ((c_0 = 1) \wedge (c_1 = 1) \wedge (c_2 = 1) \wedge (c_3 = 1)) \wedge (par_0 \neq 0)$ holds. The state predicate $T_{deadlock}$ represents states from where no program action is enabled; i.e., deadlock states. Now, to add recovery from a state in $T_{deadlock}$, FTSyn assigns a high atomicity process P_{high_j} to each process P_j ($0 \leq j < 4$).

To illustrate our approach of adding hierarchical presynthesized detectors (respectively, correctors) and for reasons of space, we only focus on one of the high atomicity recovery actions added by process P_{high_0}. The actions of other high atomicity processes in the intermediate nonmasking program and the refined program are available at [1]. The action HAC is as follows:

$$
\begin{array}{l}
HAC: (c_0 = 1) \wedge (c_1 = 1) \wedge (c_2 = 1) \wedge (c_3 = 1) \wedge (sn_0 = 1) \wedge \\
\qquad\qquad ((par_0 = 2) \vee (par_0 = 1)) \wedge \\
\qquad ((sn_3 = 0) \vee (sn_1 = 0) \vee (sn_2 = 0)) \quad \longrightarrow \quad sn_0 := 0;
\end{array}
$$

The guard of HAC identifies a subset of $T_{deadlock}$ for which HAC provides recovery to states from where recovery to S_{DC} has been already established. The write action performed by HAC is a local write operation in process P_0, whereas the guard of HAC is a global state predicate that should be refined in the distributed program. Thus, we only need to add detectors for the refinement of the guard of HAC. In the next subsection, we show how FTSyn uses the guard of HAC to automatically specify the required detectors.

4.3 Adding Presynthesized Detectors

In this subsection, we show how we add hierarchical presynthesized detectors to refine the guard of the high atomicity action HAC added to the intermediate DC program.

To refine the guard of HAC, the synthesis algorithm of [4] automatically identifies the interface of the required component. The component interface is a triple $\langle X, R, i \rangle$, where X is the detection predicate of the required component, R is a relation that represents the topology of the required component, and i is the index of the process that performs the local write action after the detection of X. For example, for action HAC, X is equal to the state predicate X_0 as we describe next in this section, R is a set of pairs where each pair represents the existence of a communication link between two processes, and i is equal to 0 since P_0 should perform the local write action.

Using the interface of the required presynthesized component, the synthesis algorithm queries an existing library of presynthesized components. At this step, we have the option of supervising the synthesis algorithm in that we can observe the guard of HAC and manually identify the required components. This manual intervention helps in minimizing the number of components added to the program since each component adds its associated variables to the program and expands the state space.

For example, in the case of action HAC, the synthesis algorithm automatically identifies one component corresponding to each deadlock state in the set of states represented by the guard of HAC, whereas by manual intervention, we observe that the only variables that are not readable for P_0 are c_3 and sn_3. Hence, we add two distributed detectors d and d' to simultaneously detect the predicates X_0 and X'_0, where

$$X_0 \equiv ((c_3 = 1) \wedge (c_0 = 1) \wedge (c_1 = 1) \wedge (c_2 = 1) \wedge (sn_0 = 1) \wedge ((par_0 = 2) \vee (par_0 = 1)))$$

$$X'_0 \equiv ((sn_3 = 0) \wedge (c_0 = 1) \wedge (c_1 = 1) \wedge (c_2 = 1) \wedge (sn_0 = 1) \wedge ((par_0 = 2) \vee (par_0 = 1)))$$

The presynthesized detector d (respectively, d') includes four elements d_0, d_1, d_2, and d_3 (respectively, d'_0, d'_1, d'_2, and d'_3), where d_i (respectively, d'_i) is composed with P_i ($0 \leq i \leq 3$). Thus, the topologies of the distributed detectors d and d' are similar to the topology of the DC program. Also, the parenting relationship (respectively, read/write restrictions) between d_0, d_1, d_2, d_3 (respectively, d'_0, d'_1, d'_2, and d'_3) follows the parenting relationship (respectively, read/write restrictions) of P_0, P_1, P_2, and P_3.

The synthesis algorithm automatically instantiates an instance of the template action presented in Section 3 with the appropriate local condition. The local conditions are automatically identified based on the set of readable variables of each process. For example, the part of X_0 that is readable for detector d_3 is identified as $LC_3 \equiv ((c_3 = 1) \wedge (c_2 = 1))$. Thus, the instantiation of the template action for detector d_3 results in the following action:

$$D_{31} : (c_3 = 1) \wedge (c_2 = 1) \wedge (y_3 = false) \longrightarrow y_3 := true;$$

Likewise, the part of X'_0 that is readable for detector d'_3 is automatically identified as $LC'_3 \equiv ((sn_3 = 0) \wedge (c_2 = 1))$. Hence, the action of d'_3 is as follows:

$$D'_{31} : (sn_3 = 0) \wedge (c_2 = 1) \wedge (y'_3 = false) \longrightarrow y'_3 := true;$$

The detector d_3 (respectively, d'_3) sets y_3 (respectively, y'_3) to *true* if the local condition LC_3 (respectively, LC'_3) holds and y_3 (respectively, y'_3) is *false*. The predicate $Z_3 \equiv (y_3 = true)$ (respectively, $Z'_3 \equiv (y'_3 = true)$) is the witness predicate of d_3 (respectively, d'_3), and the predicate $X_3 \equiv LC_3$ (respectively, $X'_3 \equiv LC'_3$) constructs the detection predicate of d_3 (respectively, d'_3). Note that since d_3 (respectively, d'_3) is the leaf of the tree, it does not have any children to wait for before it witnesses. Next, we present the actions of d_2 and d'_2 (i.e., actions D_{21} and D'_{21}) as follows:

$$D_{21} : (y_3 = true) \wedge (c_2 = 1) \wedge (sn_0 = 1) \wedge (c_0 = 1) \wedge ((par_0 = 2) \vee (par_0 = 1)) \wedge (y_2 = false) \longrightarrow y_2 := true;$$

$$D'_{21} : (y'_3 = true) \wedge (c_2 = 1) \wedge (sn_0 = 1) \wedge (c_0 = 1) \wedge ((par_0 = 2) \vee (par_0 = 1)) \wedge (y'_2 = false) \longrightarrow y'_2 := true;$$

The local condition of the action D_{21} (i.e., LC_2) is equal to $(c_2 = 1) \wedge (sn_0 = 1) \wedge (c_0 = 1) \wedge ((par_0 = 2) \vee (par_0 = 1))$. Thus, the detection predicate of d_2 is equal to $X_2 \equiv (LC_2 \wedge LC_3)$ and its witness predicate Z_2 is equal to $(y_2 = true)$. The local condition of the action D'_{21} (i.e., LC'_2) is also equal to LC_2. Hence, the detection predicate of d'_2 is equal to $X'_2 \equiv (LC'_2 \wedge LC'_3)$ and its witness predicate Z'_2 is equal to $(y'_2 = true)$.

Likewise, the synthesis algorithm identifies the detection (respectively, the witness) predicate of d_1 based on identifying $LC_1 \equiv (c_1 = 1)$. We omit the details of the actions of d_1 as it is straightforward and similar to the actions of d_2 and d_3. The local condition LC_0 of detector d_0 is equal to $(c_1 = 1) \wedge LC_2$. Also, the local condition LC'_0 of detector d'_0 is equal to $(c_1 = 1) \wedge LC'_2$. Thus, the actions of detectors d_0 and d'_0 are as follows:

$$D_{01} : (y_1 = true) \wedge (y_2 = true) \wedge LC_0 \wedge (y_0 = false) \longrightarrow y_0 := true;$$

$$D'_{01} : (y'_1 = true) \wedge (y'_2 = true) \wedge LC'_0 \wedge (y'_0 = false) \longrightarrow y'_0 := true;$$

The truth-value of y_0 witnesses the truth-value of X_0 and y'_0 witnesses the truth-value of X'_0. Now, we add a recovery action that only reads the local variables of P_0 and d_0 and writes the local variables of P_0. The recovery action is as follows:

$$Rec : (y_0 = true) \wedge ((y'_0 = true) \vee (sn_1 = 0) \vee (sn_2 = 0)) \longrightarrow sn_0 := 0; y_0 := false; y'_0 := false; y_2 := false; y'_2 := false;$$

When the program executes the above recovery action, the predicates X_0 and X_2 (respectively, X'_0 and X'_2) no longer hold. Thus, the witness predicates of d_0 and d_2 (respectively, d'_0 and d'_2) must be falsified; i.e., y_0 and y_2 (respectively, y'_0 and y'_2) should become $false$.

The composition of the DC program and the presynthesized detectors. For the ease of presentation, we only present the actions of process P_0 of the nonmasking DC program that is a composition of the actions of the presynthesized detector and the actions of P_0 in the intermediate fault-intolerant program. We refer the reader to [1] for the actions of other processes.

$$DC_{01} : (c_0 = 1) \wedge (par_0 = 0) \longrightarrow c_0 := 0; y_0 := false; y'_0 := false;$$

$$DC_{02} : (c_0 = 1) \wedge (c_{par_0} = 0) \wedge (sn_0 \not\equiv sn_{par_0}) \longrightarrow c_0 := c_{par_0}; sn_0 = sn_{par_0}; \text{if } ((c_0 = 0) \wedge (y_0 = true)) \text{ then } y_0 := false; y'_0 := false;$$

$$DC_{03} : (c_0 = 0) \wedge (\forall k :: (par_k = 0) \Rightarrow (c_k = 1 \wedge sn_0 \equiv sn_k)) \longrightarrow c_0 := 1; \text{if } (((y_1 = false) \vee (y_2 = false)) \wedge (y_0 = true)) \text{ then } y_0 := false; \text{if } (((y'_1 = false) \vee (y'_2 = false)) \wedge (y'_0 = true)) \text{ then } y'_0 := false;$$

D_{01} : // The same as action D_{01} defined earlier
D'_{01} : // The same as action D'_{01} defined earlier
Rec : // The same as action Rec defined earlier

The actions of process P_0 are composed with the actions of detectors d_0 and d'_0 (i.e., D_{01} and D'_{01}) and the recovery action Rec presented in this section. Observe that the statement of each action of P_0 is composed with assignments that falsify the witness predicates of the detectors d_0 and d_2. Such falsification of the witness predicates is necessary so that program execution preserves the safety of detectors.

Interference-freedom. The interference-freedom requires the synthesized program to provide recovery in the presence of faults, and satisfy the specification of the DC program in the absence of faults. In the presence of faults, if faults perturb the program outside the invariant S_{DC} then the synthesized program satisfies the requirements of nonmasking fault-tolerance; i.e., recovery to S_{DC} is guaranteed. In the absence of faults, the added detectors do not interfere with the program execution. Thus, the above program satisfies the specification of diffusing computation program in the absence of faults, and the safety of detectors is preserved.

We would like to note that when faults occur, fault transitions may directly violate the safety specification of detectors; e.g., after d_3 witnesses that $(c_3 = 1)$ holds, faults may change the value of c_3 to 0, and as a result, d_3 witnesses incorrectly; i.e., the safety of d_3 will be violated by fault transitions. Since nonmasking fault-tolerance only requires recovery to the invariant, the violation of safety does not violate the nonmasking fault-tolerance property. Thus, the only requirement is that the composition of the program and the presynthesized detectors provides recovery in the presence of faults.

Although the synthesized nonmasking program is correct by construction, we verified the interference-freedom requirements of the above program in the SPIN model checker to gain more confidence on the implementation of FTSyn. The source of the Promela model is available at [1].

5 Conclusion and Future Work

In this paper, we presented a case study for adding presynthesized fault-tolerance components to programs using the software framework Fault-Tolerance Synthesizer (FTSyn) [1]. The implementation of FTSyn integrates a hybrid synthesis method (presented in [4]) that combines heuristics presented in [10, 11] with pre-synthesized detectors and correctors. Specifically, we showed how we add presynthesized detectors and correctors [5] to fault-intolerant distributed programs that have *hierarchical* topologies. This case study extends the scope of synthesis using presynthesized components to the cases where we (i) use hierarchical components, and (ii) simultaneously add multiple components.

The synthesis method of [4] provides (i) an extra option during synthesis where heuristics fail to synthesize a fault-tolerant program, and (ii) a systematic way for adding new variables during the addition of fault-tolerance. Also, adding presynthesized components provides reuse in the synthesis of fault-tolerant programs. In other words, presynthesized detectors and correctors encapsulate the solutions to commonly encountered problems in the synthesis of fault-tolerant distributed programs. As a result, we can reuse the effort put in the synthesis of one program for the synthesis of another program. The synthesis method in [4] illustrates this reuse by reusing a linear component in the synthesis of a token ring program and an alternating bit protocol. These examples along with the example presented in this paper are available at `http://www.cse.msu.edu/~ebnenasi/research/tools/ftsyn.htm`.

Although we have used presynthesized components for the purpose of adding fault-tolerance, presynthesized detectors and correctors provide generic abstractions that simplify the synthesis and the verification of component-based systems. More specifically, presynthesized components provide correct-by-construction entities that modularize the reasoning about the behavior of the resulting composition. Also, as we illustrated in this case study, it is feasible to provide automation for the synthesis of programs constructed from presynthesized components.

As an extension to this work, we are investigating the necessary and sufficient conditions of interference-free addition of multiple components to programs. Such conditions provide efficient techniques for fast extraction of appropriate components from an existing component library. As a result, we will be able to synthesize large-scale programs that are composed of a large number of presynthesized components.

References

[1] FTSyn: A framework for automatic synthesis of fault-tolerance. `http://www.cse.msu.edu/~ebnenasi/research/tools/ftsyn.htm`.

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

[3] S. S. Kulkarni and A. Ebnenasir. The complexity of adding failsafe fault-tolerance. *International Conference on Distributed Computing Systems*, pages 337–344, 2002.

[4] 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.

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

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

[7] 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.

[8] 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.

[9] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1990.

[10] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of byzantine agreement. *Symposium on Reliable Distributed Systems*, 2001.

[11] S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. *International Conference on Distributed Computing Systems*, pages 441–450, 2003.

[12] Anish Arora, Mohamed G. Gouda, and George Varghese. Constraint satisfaction as a basis for designing nonmasking fault-tolerant systems. *Journal of High Speed Networks*, 5(3):293–306, 1996.