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

[Extended Abstract]

Ali Ebnenasir
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing MI 48824 USA
ebnenasi@cse.msu.edu

Sandeep S. Kulkarni
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing MI 48824 USA
sandeep@cse.msu.edu

ABSTRACT

We present a case study for 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 for the case where we automatically add *hierarchical* components to fault-intolerant programs. Towards this end, we present an automatically generated diffusing computation program that provides nonmasking fault-tolerance. 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

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) [5]. Specifically, we use FTSyn to add distributed 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 [7]. Also, fault-tolerance components provide an abstraction that simplifies the reasoning about the fault-tolerance and functional concerns.

The FTSyn framework incorporates the results of [9] 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 [6] 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 [9], 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 out-tree, where the indegree of each node is at most one.

This case study shows that the synthesis method presented in [9] handles presynthesized components (respectively, distributed programs) with different topologies. Also, we extend the scope of synthesis for the case where we simultaneously add multiple presynthesized components to the program being synthesized. Moreover, the use of presynthesized components provides a theoretical foundation for automated development of component-based systems where we reason about the correctness of each individual component and the composition of components.

The organization of the paper. In Section 2, we present preliminary concepts. In Section 3, we describe how we formally represent a hierarchical fault-tolerance component. Subsequently, 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 Subsection 2.1. Then, in Subsection 2.2, we represent the formal

[*]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.

problem statement of adding fault-tolerance components to programs (adapted from [9]). In Subsection 2.3, we give an informal overview of the synthesis method presented in [9].

2.1 Basic Concepts

We specify programs in terms of their state space and their transitions. The definition of specifications is adapted from Alpern and Schneider [1]. The definition of faults and fault-tolerance is adapted from Arora and Gouda [2] and Kulkarni and Arora [7]. The issues of modeling distributed programs is adapted from [7, 4].

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 v_i is associated with a finite domain of values D_i $(1 \leq i \leq q)$. A state 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 $s_0, s_1 \in S_p$.

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 $\{(s_0, s_1) : (s_0, s_1) \in p \ \wedge \ s_0, s_1 \in S\}$.

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 a variable x, where $x \notin w_j$. 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. For example, in a program with two Boolean variables a and b and a process P_r that cannot read b, 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. The grouping of these two transitions makes the value of b irrelevant for P_r.

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 [1], 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 [6] that a program is not allowed to execute, 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. A class of faults 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 (i.e., transitions) 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 $p[]f$.

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.

2.2 Problem Statement

In this subsection, we adapt the problem statement presented in [9] where the authors add presynthesized fault-tolerance components to a program p, with state space S_p, invariant $S \subseteq S_p$, specification *spec*, and faults f, in order to synthesize a fault-tolerant program p' with the new invariant S' in the new state space $S_{p'}$. Since each component has its own set of variables, we expand the state space of p to $S_{p'}$ by adding a fault-tolerance component to it.

To create a projection from 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.

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'. $\square$

2.3 The Synthesis Method

In this subsection, we present an informal overview of the synthesis method presented in [9]. We note that the presentation of this subsection suffices for this paper, however, the interested reader may refer to [9] for a formal presentation.

To deal with the exponential complexity [7] of synthesizing distributed programs, the synthesis algorithm presented in [9] provides a hybrid approach where it uses heuristics (developed in [8]) along with presynthesized fault-tolerance components. Specifically, the algorithm of [9] 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 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 [9] 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} has no actions to execute, where an *action* atomically updates program variables when a specific condition holds. 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. Since high atomicity processes have read access to all program variables, they may add recovery actions whose guards are global state predicates; i.e., *high atomicity actions*.

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 for the refinement of the added high atomicity recovery actions. If the presynthesized detectors do not *interfere* with program execution then the refinement will be successful. Otherwise, the synthesis algorithm of [9] 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 [6].

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 $spec_d$ of a detector as a set of transitions that a detector is not allowed to execute.

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

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 safety specification $spec_d$, its variables, and its invariant U. 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. Let $i \preceq j$ denote the parenting relation between nodes d_i and d_j, where d_i is the parent of d_j. Each node d_i has its own detection predicate X_i and witness predicate Z_i represented by a Boolean variable y_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. 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. We present the *template* action of the detector d_i as follows $((0 \leq i, j, k < n) \wedge (\forall r : j \leq r \leq k : i \preceq r))$:

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

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. The above action is an abstract action 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. We represent the invariant of the hierarchical detector by the predicate U, where

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

Note that $y_i(s)$ represents the value of y_i at the state s.

4. CASE STUDY: DIFFUSING COMPUTATION

In this section, we present an overview of synthesizing a nonmasking diffusing computation program by adding presynthesized components. The synthesized program provides the same behavior as the nonmasking diffusing computation program manually designed in [3]. For reasons of space, we omit the actions of the synthesized program and refer the reader to [10].

The diffusing computation (DC) program (adapted from [3]) consists of four processes $\{P_0, P_1, P_2, P_3\}$ whose underlying communication 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.

When the root process (i.e., the node whose parent is itself) is green, it starts a session of diffusing computation by changing its color to red and toggling its session number, which is a binary value. If a process P_j $(0 \leq j \leq 3)$ is green and its parent is red and its session number is not the same as its parent then it copies the color and the session number of its parent to propagate the wave of diffusing computation. If a process P_j $(0 \leq j \leq 3)$ is red and all its children are green and have the same session number as P_j then P_j changes its color to green to reflect back the wave of diffusing computation.

In each session of diffusing computation, every process P_j meets one of the following requirements: (i) P_j and its parent have both started participating; (ii) P_j and its parent have both completed the current session of diffusing computation; (iii) P_j has not started participating in the current session whereas its parent has, and (iv) P_j has completed participating in the current session whereas its parent has not. These requirements identify the program invariant.

Fault transitions can perturb the color and the session number of the processes. Also, faults may perturb the underlying communication topology of the program by changing the parenting relationship in the tree.

Intermediate Nonmasking Program. The faults may perturb the state of the DC program to the states where the program may fall in a non-progress cycle or reach a deadlock state. For example, faults may perturb the program to states where all processes are green and P_0 is no longer the root process. No program action will be enabled from such states; i.e., deadlock states. To add recovery from such states, FTSyn assigns a high atomicity process P_{high_j} to each process P_j $(0 \leq j < 4)$. A process P_{high_j} may add high atomicity recovery actions to resolve some deadlock states.

Adding Presynthesized Detectors. To refine the guard of high atomicity actions, FTSyn 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. Using the interface of the required presynthesized component, the synthesis algorithm queries an existing library of presynthesized components. 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.

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. Currently, FTSyn reduces the interference-freedom requirements to the satisfiability problem and automatically verifies them using SAT solvers. Although the synthesized nonmasking program is correct by construction, we verified the synthesized program using the SPIN model checker to gain more confidence on the implementation of FTSyn.

Complexity. The verification of interference-freedom and the addition of presynthesized components can be done in polynomial time in the state space of program-component composition (cf. [9] for proof).

5. CONCLUSION AND FUTURE WORK

In this paper, we presented a case study for adding presynthesized fault-tolerance components to programs using a hybrid synthesis method (presented in [9]) that combines heuristics presented in [8] with pre-synthesized detectors and correctors. Specifically, we showed how we add presynthesized detectors and correctors [6] to fault-intolerant distributed programs that have *hierarchical* topology. 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. Currently, except the extraction of the components from an existing library of presynthesized components, we automatically perform other steps of the synthesis (e.g., component specification, interference-freedom verification). As an extension to this work, we plan to apply efficient component extraction techniques where we identify the appropriate components during synthesis. Also, we plan to extend this work to programs with higher number of processes and more complicated topologies.

6. REFERENCES

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

[2] 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.

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

[4] 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.

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

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

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

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

[9] S. S. Kulkarni and A. 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.

[10] S. S. Kulkarni and A. Ebnenasir. Hierarchical presynthesized components for automatic addition of fault-tolerance: A case study. Technical Report MSU-CSE-04-41, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, September 2004.