Constraint Based Automated Synthesis of Nonmasking and Stabilizing Fault-Tolerance *

Fuad Abujarad    Sandeep S. Kulkarni
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
Email: {abujarad,sandeep}@cse.msu.edu
Web: http://www.cse.msu.edu/~{abujarad,sandeep}

Abstract

We focus on constraint-based automated addition of nonmasking and stabilizing fault-tolerance to hierarchical programs. We specify legitimate states of the program in terms of constraints that should be satisfied in those states. To deal with faults that may violate these constraints, we add recovery actions while ensuring interference freedom among the recovery actions added for satisfying different constraints. Since the constraint-based manual design of fault-tolerance is well-known to be applicable in the manual design of nonmasking fault-tolerance, we expect our approach to have a significant benefit in automation of fault-tolerant programs. We illustrate our algorithms with three case studies: stabilizing mutual exclusion, stabilizing diffusing computation, and a data dissemination problem in sensor networks. With experimental results, we show that the complexity of synthesis is reasonable and that it can be reduced using the structure of the hierarchical systems.

To our knowledge, this is the first instance where automated synthesis has been successfully used in synthesizing programs that are correct under fairness assumptions. Moreover, in two of the case studies considered in this paper, the structure of the recovery paths is too complex to permit existing heuristic based approaches for adding recovery.

1 Introduction

At design time, it may not be possible to predict all faults that a distributed program may be subjected to. Thus, it is necessary to add fault-tolerance to existing programs while ensuring that their functional properties continue to be satisfied. One way to preserve the existing properties is to add fault-tolerance properties to distributed programs by utilizing automated program synthesis. This approach is capable of deriving programs that are correct-by-construction while ensuring that existing properties continue to be satisfied.

In this work, we focus our attention on automated addition of nonmasking fault-tolerance to fault-intolerant programs. Intuitively, a nonmasking fault-tolerant program ensures that if it is perturbed by faults to an illegitimate state, then it would eventually recover to its legitimate states. However, safety may be violated during recovery. Therefore, nonmasking fault-tolerance is useful to tolerate a temporary perturbation of the program state. After recovery is completed, a nonmasking fault-tolerant program satisfies both the safety and liveness in the subsequent computation. Nonmasking fault-tolerance is an ideal solution to add fault-tolerance to the programs that organize network nodes in specified topology or a predefined logical structure [6].

There are several reasons that make the design of nonmasking fault-tolerance attractive. For one, the design of masking fault-tolerant programs, where both safety and liveness are preserved during recovery, is often expensive or impossible even though the design of nonmasking fault-tolerance is easy [7]. Also, the design of nonmasking fault-tolerance can assist and simplify the design of masking fault-tolerance [8, 19]. Moreover, in several applications nonmasking fault-tolerance is more desirable than solutions that provide fail-safe fault-tolerance (where in the presence of faults the program reaches to “safe” states from where it does not satisfy liveness requirements). This is especially true for networking related applications such as routing and tree maintenance.

A special case of nonmasking fault-tolerance is stabilization [10, 11], where, starting from an arbitrary state, the program is guaranteed to reach a legitimate state. Stabilizing systems are especially useful in handling unexpected transient faults. Moreover, this property is often critical in long-lived applications where faults are difficult to predict. Furthermore, it is recognized that verifying stabilizing systems is especially hard [16]. Hence,
techniques for automated synthesis are expected to be useful for designing stabilizing systems.

There are several issues that complicate the design of nonmasking fault-tolerance [3]. One such issue is the complexity of designing and analyzing the recovery actions needed to ensure that the program recovers to legitimate states. Another issue is that to verify correctness of the nonmasking fault-tolerant program, one needs to consider all possible concurrent executions of the original program, recovery actions, and fault actions. Yet another issue is that most nonmasking algorithms assume that faults can keep happening (although they will eventually stop for a long enough time to permit recovery) even during recovery, thereby, complicating the recovery to legitimate states.

Adding nonmasking fault-tolerance to an existing program is achieved by performing three steps. The first step is to identify the set of legitimate states of the fault-intolerant program. This set defines the constraints that should be true in the legitimate states. The second step is to identify a set of convergence actions that recover the program from illegitimate states to a legitimate state. This can be done by finding actions that satisfy one or more constraints. The last step consists of ensuring that the convergence actions do not interfere with each other. In other words, the collective effect of all recovery actions should, eventually, lead the program to legitimate states.

In this paper, we automate the last two steps by identifying the necessary actions to ensure that the constraints are satisfied and that the recovery actions do not interfere with each other. As discussed in the conclusion, even the first step can be partially automated. However, since it requires at least some manual efforts, this issue is not considered in this work.

Contribution of the paper.

- We propose an automated synthesis algorithm for constraint-based synthesis of nonmasking fault-tolerant programs. We illustrate our algorithm with three case studies. We note that the structure of the recovery actions in the first two case studies is too complex to permit previous approaches to achieve synthesis of the corresponding fault-tolerant programs [9]. Our algorithm, however, synthesizes the corresponding fault-tolerant program in a short time. We also show that the structure of the hierarchical system can be effectively used to generalize programs with a small number of processes while preserving correct-by-construction property of the synthesized program.
- To our knowledge, this is the first instance where programs that require fairness assumptions have been synthesized with automated techniques. Particularly, in our first case study, it is straightforward to observe that stabilizing fault-tolerance cannot be added without some fairness among all processes. Hence, the previous algorithms (e.g. [9]) will declare failure in adding fault-tolerance.

Organization of the paper. In Section 2, we define the notion of distributed programs, invariant, faults, and the problem of the automated addition of nonmasking fault-tolerance. In Section 3, we outline the algorithm for automated synthesis of nonmasking fault-tolerance. In Section 4, we present our results in the contexts of three case studies: stabilizing mutual exclusion, stabilizing diffusing computation, and data dissemination protocol for sensor networks. In Section 5 we show that the structure of the hierarchical system can be used to reduce complexity of synthesis. We present the related work in Section 6. The conclusion and future work are described in Section 7.

2 Programs and Specifications

In this section we define the notion of distributed programs, faults, and the problem statement for adding nonmasking fault-tolerance. Those definitions are based on the ones given by Arora and Gouda [5]. We also identify how the notion of fairness can be modeled for automated addition of nonmasking fault-tolerance.

For the following definitions of enabled and fairness let $S_0$ be a set of states. A transition over $S_0$ is of the form $(s_0, s_1)$, where $s_0, s_1 \in S_0$. Let $\alpha, \alpha_1, \alpha_2, \alpha_3, ... \alpha_m$ be sets of transitions over $S_0$. In other words, $\alpha, \alpha_1, \alpha_2, \alpha_3, ... \alpha_m$ are subsets of $S_0 \times S_0$. $\alpha$ is enabled in state $s_0$ iff there exists a state $s_1$ such that $(s_0, s_1) \in \alpha$.

- **Enabled.** Intuitively, $\alpha$ is enabled in $s_0$ if $\alpha$ contains some transitions that begins in $s_0$. Formally, $\alpha$ is enabled in a state $s_0$ iff there exists a state $s_1$, such that $(s_0, s_1) \in \alpha$.

- **Fairness.** Intuitively, if a sequence is fair with respect to $(\alpha_1, \alpha_2, \alpha_3, ... \alpha_m)$ and $\alpha_i$ is continuously enabled in that sequence then that sequence includes a transition in $\alpha_i$. Formally, an infinite sequence $(s_0, s_1, s_3, ...)$ is fair with respect to $(\alpha_1, \alpha_2, \alpha_3, ... \alpha_m)$ iff for each $i$, $k$ the following condition is satisfied:

$$\alpha_i \text{ is enabled in each state } s_k, s_{k+1}, ... \Rightarrow (\exists l : i \geq k : (s_1, s_{l+1}) \in \alpha_i).$$

Note that this definition is equivalent to weak fairness from [1, 2, 13].

- **Program.** A program $p$ is specified in terms of its state space, $S_p$ and $(\alpha_1, \alpha_2, \alpha_3, ... \alpha_m)$, where for each $i$, $\alpha_i \in S_p \times S_p$. The transitions of $p, \delta_p$, are equal to $\alpha_1 \cup \alpha_2 \cup \alpha_3 \cup ... \cup \alpha_m$. We use the notion $(S_p, (\alpha_1, \alpha_2, \alpha_3, ... \alpha_m))$ to denote such programs. Whenever it is clear from the context, we use
p and its transitions $\delta_p$ interchangeably. A sequence of states, $\sigma = (s_0, s_1, \ldots)$ is a computation of $p$ iff (1) $(\forall j : 0 < j < \text{length}(\sigma) : (s_j-1, s_j) \in \delta_p)$, that is, in each step of this sequence, a transition of $p$ is executed, (2) if the sequence is finite and terminates in $s_j$ then $\forall s' :: (s_j, s') \notin p$, i.e., a computation is finite only if it reaches a state from where the program does not have any outgoing transition, and (3) if the sequence is infinite then it is fair with respect to $(\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_m)$.

A state predicate $C$ of program $p$ is a subset, say $S_C$, of $S_p$. In our BDD based implementation, we represent it using an equivalent function $fC$ with domain $S_p$ and range $\{\text{true}, \text{false}\}$ where $f_C(s) = \text{true}$ iff $s \in S_C$. Let $C1$ and $C2$ be two state predicates represented with sets $S_{C1}$ and $S_{C2}$, respectively. Let $f_{C1}$ and $f_{C2}$ be corresponding functions. Observe that the function corresponding to $S_{C1} \cap S_{C2}$ is $f_{C1 \cap C2}$ where $f_{C1 \cap C2}(s) = f_{C1}(s) \land f_{C2}(s)$. In other words, the intersection of two state predicates corresponds to the conjunction of corresponding functions. Likewise, disjunction corresponds to union, and so on. Hence, throughout the rest of the paper, we use these boolean operators for constructing different state predicates, as this directly corresponds to our BDD based implementation. Likewise, in our implementation, to represent a set of transitions over state space $S_p$, we use a function with domain $S_p \times S_p$ and range $\{\text{true}, \text{false}\}$. Thus, a conjunction of such formula is equivalent to the intersection of corresponding sets of transitions and so on.

**Invariant.** Legitimate states of a program, say $p$, are characterized by a set of constraints $C_1, C_2, \ldots, C_m$, where each $C_i$ is a subset of the state space $S_p$. Thus, predicate $S = C_1 \land C_2 \land \cdots \land C_m$, denoted as invariant of $p$, identifies all legitimate states of $p$. In other words, if a computation of $p$ begins in a state that is in $S$ then (1) $S$ is true at all states in that computation and (2) the computation is correct. Note that the notion of this correctness has to deal with the fault-intolerant program that is assumed to be correct. We assume that each transition of $p$ preserves each constraint in the invariant, i.e. for each $i$, if $(s_0, s_1)$ is transition of $p$ and $s_0 \in C_i$ then $s_1 \in C_i$.

**Faults.** Let $f$ be the class of faults to which tolerance is to be added. Faults for program $p$ are specified as a subset of $S_p \times S_p$. Note that this allows modeling of different types of faults, such as transients, Byzantine, crash faults, etc.

**Fault-span.** A fault-span of program $p$, say $T$, with respect to faults from invariant $S$ is the set of states reached by starting from a state in $S$ and executing transitions from $p/f$. Just like the invariant captured the boundary up to which the program can reach in the absence of faults, fault-span is the boundary up to which the program can reach in the presence of faults.

The goal of an algorithm that adds nonmasking fault-tolerance is to begin with a fault-intolerant program $p$, its invariant $S$, and faults $f$, and to derive the nonmasking fault-tolerant program, say $p'$, such that in the presence of faults, $p'$ eventually converges to $S$. Furthermore, computations of $p'$ that begin in $S$ must be the same as that of $p$.

Based on this discussion, we define the problem of adding self-stabilization fault-tolerance as follows: **Problem statement 2.1** Given $p, S$, and $f$, identify $p'$ such that:

- Transitions within the invariant remain unchanged
  
  $s_0 \in S \Rightarrow (\forall s_1 :: (s_0, s_1) \in p \iff (s_0, s_1) \in p')$

- There exists a state predicate $T$ (fault-span) such that
  
  - $S \subseteq T$
  - $(s_0, s_1) \in (p' \lor f) \land (s_0 \in T) \Rightarrow s_1 \in T$
  - $s_0 \in T \land (s_0, s_1, \ldots)$ is a computation of $p'$
  
  $\Rightarrow (\exists j : j \geq 0 : s_j \in S)$

The above statement is equivalent to the definition of nonmasking fault-tolerance from [6]. Stabilizing fault-tolerance is a special instance of this problem statement with the requirement that $T=S_p$, i.e. the fault-span equals the set of all states.

3 Synthesis Algorithm

In this section, we describe our approach to add nonmasking fault-tolerance to fault-intolerant programs based on [6]. The goal of nonmasking fault-tolerance is to ensure that starting from any state in the fault-span, say $T$, the program eventually reaches a state in $S$ where, $S = C_1 \land C_2 \land \cdots \land C_m$. Faults perturb the program to a state in $(T-S)$. Hence, in the presence of $f$, one or more of the constraints from $C_1, C_2, \ldots, C_m$ is violated. Our goal is to automatically synthesize the recovery actions such that when faults stop occurring, the constructed recovery actions in conjunction with the original program actions will, eventually, converge the program to a state where $S$ holds.

Since we focus on the design of distributed programs, for brevity, we specify the state space of a program in terms of its variables. Thus, the state space of the program is obtained by assigning each variable each possible value from its domain. Furthermore, we specify the transitions of the program in terms of a set of processes, where each process can read and write a subset of the program variables. Transitions of a process are obtained by considering how that process updates the program variables. And, finally, the transitions of the program is the union of the transitions of its processes.

To concisely describe the transitions of the processes of a program we use guarded command notation: $(guard) \rightarrow (statement)$, where guard is state
predicate, that is a subset of the state space. The statement describes how program state is updated and it always terminates. A guarded command of the form \( g \rightarrow st \) corresponds to transitions of the form \( \{ (s_0, s_1) \mid g(s_0) = true \text{ and } s_1 \text{ is obtained by executing } st \text{ from } s_0 \} \).

### 3.1 Constraint Satisfier

Our algorithm for adding nonmasking fault-tolerance is as shown in Algorithm 1. The input for the algorithm includes the constraint array \( C \), fault-span \( T \), and program \( p \).

**Algorithm 1 ConstraintSatisfier**

**Input:** constraint array \( C \), fault-span \( T \), and program transitions predicate \( p \).

**Output:** recovery transitions \( rec \).

1. \( rec, temp := false; \)
2. for \( i := 0 \) to \( SizeOf(C) - 1 \) do
   3. \( temp := Group((T - C[i]) \land (C[i]')); \)
      // \( temp \) is the transitions that start in a state \( \{ s \in T - C(i) \text{ and reach } C(i) \} \)
      // ensure that \( T \) and \( S \) are not violated by \( temp \)
   4. for \( j := 0 \) to \( i - 1 \) do
      5. \( temp := Group(C[i] \land temp \land (C[i]')); \)
   6. end for
      // Combine current recovery transitions
      // with the new recovery transition.
5. \( rec := rec \lor temp; \)
8. end for
   // return the recovery transition.
9. return \( rec \);

In our algorithm, we satisfy the constraints from the constraint array one after another. More specifically, to satisfy constraint \( i \), we construct transitions that start in a state where \( (T - C[i]) \) is true and reach a state where \( C[i] \) is true (Line 3). Furthermore, we need to ensure that added transitions are consistent. In particular, with respect to the read/write restrictions in the underlying system model, if a transition \( (s_0, s_1) \) is to be added as a transition of a process \( j \) then we must also include transitions of the form \( (s_0', s_1') \) where \( s_0 \) and \( s_0' \) (respectively \( s_1 \) and \( s_1' \)) are indistinguishable for \( j \), i.e., they differ only in terms of the variables that \( j \) cannot read. We use the function \( Group \) to include these additional transitions. Since the exact implementation of \( Group \) is not critical for the algorithm, in this paper we omit it.

Notice that, before we add \( temp \) (c.f. Line 3) to the recovery transitions \( rec \), we ensure that none of the transitions in \( temp \) interfere with earlier constraints. Therefore, for all \( j \) where \( (0 \leq j < x) \) we check if any of the transitions in \( temp \) originate from a state where \( C[j] \) is \( true \) and end in state where \( C[j] \) is \( false \) (Line 7). If any are found, we remove the transition \( group \) that contains such transitions.

We continue repeating steps 2–8 until we find all the recovery actions that satisfy all the constraints in the array \( C \). Finally, we return the recovery actions of the nonmasking fault-tolerant program \( p \).

**Theorem 3.1:**

- Given are fault-intolerant program \( p \), constraints \( C_1, C_2 \ldots C_m \), and faults \( f \).
- Let \( S = C_1 \land C_2 \land \ldots \land C_m \).
- Let \( T = \{ s \in S : (s_0, s_1) \in rec \} \).

Then \( \{ S_p, (rec, \delta_p) \} \) solves the constraints in Problem statement 2.1.

**Proof.** The proof is similar to Theorem 1 from [6]. Note that to ensure that the transitions from \( rec \) are executed, we need fairness between original program actions and recovery actions.

### 3.2 Choosing Ordering Among Constraints

To apply Theorem 3.1, we need to identify an order among the constraints. In our case studies, we attempted several orderings and most were successful in synthesizing the nonmasking fault-intolerant program. Hence, choosing the “right” order does not appear to be very crucial. Also, [6] identifies several heuristics that can assist in identifying the right order among constraints.

Finally, it is possible to consider different combinations as part of the synthesis algorithm. With such an approach, \( O(n^2) \) combinations suffice for most examples. In particular, to identify an ordering, we can utilize an algorithm similar to insert-sort as follows: first consider only constraints \( C_1 \) and \( C_2 \) and attempt both orderings between them. If both orderings fail, then adding nonmasking fault-tolerance cannot be achieved using the constraint-based approach that uses constraints \( C_1 \) and \( C_2 \). If both succeed, then we can choose any order. Without loss of generality, let the order be \( C_1 \) and \( C_2 \). Then, we consider constraint \( C_3 \) in conjunction with \( C_1 \) and \( C_2 \). There are three possible combinations to insert \( C_3 \) without affecting the order between \( C_1 \) and \( C_2 \). We can evaluate all three options and then consider \( C_4 \) and so on. It follows that the number of such runs will be \( O(n^3) \). In all the case studies in this paper as well as several other algorithms in the literature, the above approach would succeed in identifying the right order of constraints. It follows that one does not need to consider all possible \((n!) \) orderings among the constraints.
4 Case Studies

In Subsections 4.1-4.3, we describe and analyze three case studies, namely the Stabilizing Mutual Exclusion [20], the Stabilizing Diffusing Computation [6], and the Data Dissemination Problem in Sensor Networks [18]. Of these, the first case study is of a classic problem from distributed computing and illustrates the feasibility of algorithms that require fairness constraints for correctness. Also, the first two case studies demonstrate the applicability of our approach in synthesizing programs where previous heuristic based approaches [9] fail. In the last case study we demonstrate the applicability of our approach on a real world problem in the field of the sensor networks.

Throughout this section, all case studies are run on a MacBook Pro with 2.6 Ghz Intel Core 2 Duo processor and 4 GB RAM. The OBDD representation of the Boolean formulae has been done using the C++ interface to the CUDD package developed at the University of Colorado [21].

4.1 Case Study 1: Stabilizing Mutual Exclusion

Mutual exclusion is one of the fundamental problems in distributed/concurrent programs. One of the classical solutions to this problem is the token based solution due to Raymond [20]. In this solution, the processes form a directed rooted tree, holder tree, in which there is a unique token held at the tree root. If a process wants to access the critical section, it must first acquire the token. Our goal in this case study is to add stabilizing fault-tolerance to the program in [8]. When faults occur and perturb the holder tree, the new program will self-stabilize and reconstruct a correct holder tree within a finite number of steps under weak fairness assumption.

4.1.1 Fault-Intolerant Program

In Raymond’s algorithm, the processes are organized in a logical tree, denoted as a parent. The holder tree is superimposed on top of the parent tree such that the root of the holder tree is the process that has the token. For example, Figure 1.a represents the undirected parent tree and Figure 1.b shows the holder tree when \( c \) has the token. In the fault-intolerant program, each process \( j \) has a variable \( h.j \). If \( h.j = j \) then \( j \) has the token. Otherwise, \( h.j \) contains the process number of one of \( j \)’s neighbors. The holder variable forms a directed path from any process in the tree to the process currently holding the token.

In this program, a process can send the token to one of its neighbors. For example, Figure 1.c shows the case where process \( c \) sends the token to \( e \). In particular, if \( j \) and \( k \) are adjacent (in the parent tree), then the action by which \( k \) sends the token to \( j \) is as follows:

\[
NM1 :: (h.k = k \land j \in Adj.k) \land (h.j = k) \\
\quad \rightarrow h.k, h.j := j, j;
\]

![Figure 1. The holder tree](image)

4.1.2 Constraints

Recall from Section 2 that we define the invariant to be a set of constraints on the program state space. In this case study, this set is the conjunction of the constraints \( S1, S2, \) and \( S3 \), described next. Moreover, each of these constraints is specified for each process separately. Therefore, if \( n \) is the number of processes then we have \( 3n \) constraints to satisfy. Constraint \( S1 \) requires that \( j \)’s holder can either be \( j \)’s parent, \( j \) itself, or one of \( j \)’s children. \( S2 \) requires that the holder tree conforms to the parent tree. Finally, \( S3 \) requires that there are no cycles in the holder relation. Thus, predicates \( S1, S2, \) and \( S3 \) are as follows:

\[
(S1) \quad \forall j : (h.j = P.j) \lor (h.j = j) \\
\quad \lor (\exists k : (P.k = j) \land (h.j = k))
\]

\[
(S2) \quad \forall j : (P.j \neq j) \Rightarrow (h.j = P.j) \lor (h.(P.j) = j)
\]

\[
(S3) \quad \forall j : (P.j \neq j) \Rightarrow \neg((h.j = P.j) \land (h.(P.j) = j))
\]

4.1.3 Faults

Since we focus on stabilizing fault-tolerance, we consider faults that perturb the holder relation of all processes to an arbitrary value. Thus the fault action is as follows:

\[
(F1) \quad true \rightarrow \{h.j = \text{any arbitrary value from its domain}\};
\]

4.1.4 Fault-Tolerant Program

To add stabilizing fault-tolerance to the above program, we used the synthesis algorithm as follows. The fault intolerant program for each process is specified by actions \( NM1 \); the faults are specified by the fault action \( F1 \); and the constraints are from \( S1, S2, \) and \( S3 \). We specified these constraints in the following order: first, we specified constraints \( S1 \) for the root, then its children, then its grandchildren and so on. Subsequently, we specified constraint \( S2 \)
likewise. Finally, we specified constraint $S3$ in the reverse order. (We have also tried other orderings of parts of $S1$, $S2$, and $S3$. In most cases, the synthesis succeeded although the derived program was different than the one described next.) The recovery actions computed by our algorithm are as follows:

$$(R1) \neg((h.j = P.j) \lor (h.j = j))$$
$$\lor (\exists k: (P.k = j) \land (h.j = k)))$$
$$\rightarrow h.j := j \land h.j := P.j$$
$$h.j := \{\text{child of } j\};$$

$$\rightarrow h.j := j \land (h.(P.j) := j);$$

$$(R2) \neg((P.j \neq j) \Rightarrow (h.j = P.j) \lor (h.(P.j) = j))$$
$$\Rightarrow h.j := P.j \lor (h.(P.j) := j);$$

$$(R3) \neg((P.j \neq j) \Rightarrow \neg((h.j = P.j) \land (h.(P.j) = j))$$
$$\Rightarrow h.j := j \land h.(P.j) := P.j$$
$$h.(P.j) := P.(P.j);$$

Figure 2 shows the results of synthesizing the Stabilizing Mutual Exclusion program with various number of processes organized in linear topology. It shows the time needed, in seconds, to add recovery, validate the recovery transitions (against presup-satisfied constraints), generate the fault-span, and the total synthesis time. It also shows the amount of memory usage in megabytes needed by our algorithm in terms of the number of processes being synthesized. Figure 3 shows the result of a similar case study where the processes are arranged in a binary tree. We can clearly see the feasibility of adding nonmasking fault-tolerance using automated synthesis.

![Figure 2. Stabilizing Mutual Exclusion with linear topology.](image)

Figure 3 illustrates that given the same state space, the complexity is higher in the tree topology than the linear topology. This is due to the following reason: the constraints of a process compares its variables with that of its neighbors. To model this effectively, the process variables and the variables of its neighbors need to be close to each other in the BDD ordering. This can be achieved easily on a linear topology. However, for a tree topology, this is not possible for all the processes. Hence, computing recovery transitions for those cases is more expensive.

### 4.2 Case Study 2: Stabilizing Diffusing Computation

In distributed systems, diffusing computation is used to inquire about (e.g. termination detection) or establish (e.g. distributed reset) a system global state. We consider a diffusing computation on a system where processes are arranged in a logical tree. The root initiates a diffusing computation and propagates it to its children and the children forward it to their children and so on until it reaches all processes. Once the computation reaches a leaf, it marks the leaf as completed and reflects back to the parent. When all children of a process are marked completed, that process marks itself completed and reflects the computation to its parent. The diffusing computation ends when it marks the root as completed.

#### 4.2.1 Fault-Intolerant Program

The fault-intolerant program in this case study is the diffusing computation program from [6]. Each process $j$ has two Boolean variables $c.j$ (color) and $sn.j$ (session number) and an integer variable $P$ (the parent of $j$). A new diffusing computation can start if the root is colored green ($c.root = green$) and the session number of the root is the same as its children. To start a new diffusing computation, the root sets $c.root = red$ and flips $sn.root$. When a green process finds that its parent is red, it copies its parent color and session number. Moreover, if a process has no children or all its children switched colors from red to green, the process then switches its color to green. The program for the diffusing computation consists of three actions. The first action starts the diffusing computation at the root $(DC1)$. The second action propagates the diffusing computation to the children $(DC2)$. The third action completes the diffusing computation when all the children complete computation $(DC3)$. The program actions are described below:

$$(DC1) :: (c.root = green)$$
$$\rightarrow c.root := red, sn.root := \neg sn.root;$$

$$(DC2) :: c.j = green \land c.(P.j) = red \land sn.j \neq sn.(P.j)$$
$$\rightarrow c.j, sn.j = c.(P.j), sn.(P.j);$$

$$(DC3) :: (c.j = red) \land$$
$$(\forall k : P.k = j \Rightarrow (c.k = green \land sn.j = sn.k))$$
$$\rightarrow c.j := green;$$

#### 4.2.2 Constraints

The first disjunction of $(S1)$ states that $j$’s parent has participated in a diffusing computation while $j$ did not participate yet. The second disjunction of $(S1)$ states that $j$ and its parent are participating
in a computation or they both have completed a computation.

\((S1)\) \(\forall j: (c.j = \text{green} \land c.(P.j) = \text{red}) \lor (c.j = c.(P.j) \land sn.j = sn.(P.j))\)

### 4.2.3 Faults

We now consider the faults that change the values of \(c.j\) and \(sn.j\) to an arbitrary value. The fault actions are as follows:

\((F1)\) \(\text{true} \rightarrow c.j := \text{red} \mid \text{green};\)

\((F2)\) \(\text{true} \rightarrow sn.j := \text{true} \mid \text{false};\)

### 4.2.4 Fault-Tolerant Program

To construct the nonmasking fault-tolerant program of the fault-intolerant program of Diffusing Computation, we used our algorithm with program actions \((DC1 - DC3)\), and the constraint \((S1)\) with the fault actions \((F1, F2)\) as an input. The synthesized program has the actions \((DC1 - DC3)\) in addition to the following recovery actions:

\((R1)\) \((c.j = \text{red}) \land (sn.j \neq sn.(P.j)) \rightarrow c.j := \text{green}, sn.j := sn.(P.j);\)

\((R2)\) \((c.(P.j) = \text{green}) \land (c.j = \text{red}) \rightarrow c.j := \text{green};\)

\((R3)\) \((c.(P.j) = c.j) \land (sn.j \neq sn.(P.j)) \rightarrow sn.j := sn.(P.j);\)

\((R4)\) \((c.(P.j) = \text{red}) \land (c.j = \text{red}) \land (sn.j \neq sn.(P.j)) \rightarrow c.j := \text{green};\)

<table>
<thead>
<tr>
<th>No. of Processes</th>
<th>Process Space</th>
<th>Process Memory</th>
<th>Processor Space</th>
<th>Processor Memory</th>
<th>Total Space</th>
<th>Total Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>(10^8)</td>
<td>6.64</td>
<td>1</td>
<td>2</td>
<td>&lt;1</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>(10^8)</td>
<td>7.73</td>
<td>1</td>
<td>2</td>
<td>&lt;1</td>
<td>4</td>
</tr>
<tr>
<td>21</td>
<td>(10^{11})</td>
<td>12.15</td>
<td>1</td>
<td>4</td>
<td>29</td>
<td>33</td>
</tr>
<tr>
<td>23</td>
<td>(10^{13})</td>
<td>16.22</td>
<td>2</td>
<td>4</td>
<td>156</td>
<td>162</td>
</tr>
</tbody>
</table>

**Figure 4. Stabilizing Diffusing Computation program with linear topology.**

<table>
<thead>
<tr>
<th>No. of Processes</th>
<th>Process Space</th>
<th>Processor Space</th>
<th>Processor Memory</th>
<th>Processor Memory</th>
<th>Total Space</th>
<th>Total Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>(10^8)</td>
<td>6.12</td>
<td>1</td>
<td>&lt;1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>(10^8)</td>
<td>6.60</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>(10^{11})</td>
<td>87.98</td>
<td>3</td>
<td>21</td>
<td>21</td>
<td>25</td>
</tr>
<tr>
<td>23</td>
<td>(10^{13})</td>
<td>91.88</td>
<td>4</td>
<td>199</td>
<td>199</td>
<td>205</td>
</tr>
</tbody>
</table>

**Figure 5. Stabilizing Diffusing Computation program with binary tree topology.**

Figure 4 shows the results for synthesizing a stabilizing diffusing computation program with a various number of processes organized in a linear topology. Figure 5 shows the result where the processes are arranged in a binary tree. One can notice that most of the time is spent in computing the fault-span\(^1\). This behavior is caused by the depth of the transition graph and a huge number of fault transitions that occur in this example.

\(^1\)As we can observe, most of the time is spent in calculating the fault-span. For stabilizing programs, the fault-span

### 4.3 Case Study 3: Data Dissemination in Sensor Networks

In this problem, a base station initiates a computation in which a block of data is to be sent to all sensors in the network. The data message is split into fixed size packets. Each packet is given a sequence number. The base station starts transmitting the packets to its neighbor(s) in specified time slots, in the order of the packet sequence number. Subsequently, when the neighbor(s) receive a message, they, in turn, retransmit it to their neighbors and so on. The computation ends when all sensors in the network receive all the messages.

Our goal in this case study is to synthesize a nonmasking fault-tolerant version of the data dissemination program that can tolerate a finite number of lost packets. The synthesized program is the same as Infuse [18] that is designed manually.

#### 4.3.1 Fault-Intolerant Program

In this case study, we arrange the processes in a linear topology. The base station has \(N\) packets to send to \(M\) processes (We note that similar synthesis is possible for any other fixed topology. However, for reasons of space it is omitted). The Fault-intolerant program transmits the packets in a simple pipeline. For this, each process keeps track of the messages (received/sent) using two variables \(u.j\) and \(l.j\), where \(u.j\) is the highest message sequence number received by process \(j\), and \(l.j\) is the sequence number of the message currently being transmitted by process \(j\). Process \(j\) increments \(u.j\) every time it receives a new message. It also sets \(l.j\) to be the sequence number of the message it is transmitting. The base station transmits a packet if its neighbor has received the previous packet (action \(IN1\)). A process \(j, j > 0\), receives a packet from its predecessor if its successor had received the previous packet (actions \(IN2\) and \(IN3\)). Thus, the actions of the fault-intolerant program are as follows:

Action for base station:

\((IN1)\) \(L0 := U1 \rightarrow L0 := L0 + 1;\)

Action for process \(j \in \{1..M-1\}:

\((IN2)\) \((U.j \leq U.(j + 1)) \land (U.(j) \leq U.(j - 1)) \land (L.(j - 1) = U.j + 1)\)

is the set of all states. In this implementation we did not use the fact that we are only designing a stabilizing algorithm. Hence, the fault span was computed explicitly. However, if we utilize the knowledge that we are designing a stabilization program then we can simply define fault-span to be the set of all states. With BDD based implementation, this can be achieved by setting the fault-span to a \(BDD\) (i.e. true). Hence, the time for constructing fault-span will be 0 seconds. Thus, the time to synthesize diffusing computation for 23 processes would be 6 seconds. The same reduction is possible for the first case study as well.
\[ \rightarrow U.j, L.j := U.j + 1, L.j + 1; \]

Action for process \( M \) (the last process):

(IN3) \( U.M \leq U.(M-1) \land L.(M-1) = U.M + 1 \)

\[ \rightarrow U.M, L.M := U.M + 1, L.M + 1; \]

4.3.2 Faults

In this section, we consider faults that lose a message. To model such faults for the base station, we add action (F1), where the base station increments \( L.0 \), even though its successor has not received the previous packet. To model such action for other processes, we add action (F2), where a process advances \( L.j \), even though the successor has not yet received the previous packet.

(F1) \( \text{true} \rightarrow L0 + 1; \)

(F2) \( (U.j \leq U.(j-1)) \land (L.(j-1) = U.(j+1)) \)

\[ \rightarrow U.j, L.j := U.j + 1, L.j + 1 \]

4.3.3 Constraints

The constraints that define the invariant in the case of the data dissemination program are as follows. The first constraint states that initially the base station has all the packets (S1). A process cannot receive a packet if its predecessor has not received it (S2), and cannot transmit a packet that it does not have (S3). A process transmits a packet that is expected by its successor (S4 and S5).

(S1) \( U.0 = N \)

(S2) \( \forall j : 0 < j \leq M : (U.j = U.(j-1)) \)

(S3) \( \forall j : 0 \leq j \leq M : (L.j \leq U.j) \)

(S4) \( (L.0 \leq U.1 + 1) \)

(S5) \( \forall j : 0 < j \leq (M - 1) : (L.j \leq U.(j - 1) + 1) \land (L.j \leq U.(j + 1) + 1) \)

Unlike the other two case studies, the data dissemination program has a set of constraints imposed by the model. More specifically, these constraints identify the transitions that the synthesized algorithm is not allowed to use as recovery transitions. Notice that Algorithm 1 is slightly modified to consider such transitions; these transitions are removed from temp right before Step 4. This set is specified by predicates imposed on the current and the next state. In particular, the model requires that the reception of a packet cannot be reversed (MT1), packets can only be received in sequence (MT2), a process can only receive one packet at a time, it can only receive a packet sent by its predecessor (MT3 and MT4), a process cannot transmit a packet unless it has received it (MT5), and a process should not transmit a packet unless it is potentially needed by its successor (MT6). Thus, the set of transitions disallowed by the model are as follows:

\begin{align*}
MT1 : & \ (\exists j : 0 < j \leq M : U.j' < U.j) \\
MT2 : & \ (\exists j : 0 < j \leq M : U.j' < (U.j) + 1) \\
MT3 : & \ (\exists j : 0 < j \leq M : (U.j' = (U.j) + 1) \land (U.j' \neq L.(j - 1)) \land (U.j' \neq L.(j + 1))) \\
MT4 : & \ U.M' = (U.M) + 1 \land U.M' \neq L.(M - 1) \\
MT5 : & \ (\exists j : 0 \leq j \leq M : (U.j' < L.j')) \\
MT6 : & \ (\exists j : 0 \leq j \leq M - 1 : (L.j > U.(j + 1) + 1) \land (L.j' < U.(j + 1) + 1))
\end{align*}

4.3.4 Fault-Tolerant program

Using the program actions (IN1-IN3) for each process, the faults (F1-F2), the constraints (S1-S5), and prohibited transitions (MT1-MT6) the output was a nonmasking fault-tolerant program with the following recovery actions added to it.

(R1) \( (U.j > U.(j + 1)) \land (L.j > U.(j + 1) + 1) \land (U.j + 1 = L.(j - 1)) \)

\[ \rightarrow U.j := L.(j - 1), L.j := U.(j + 1) + 1; \]

(R2) \( (U.j > U.(j + 1) + 1) \land (L.j > U.(j + 1) + 1) \)

\[ \rightarrow L.j := U.(j + 1) + 1; \]

Figure 6 shows the results of synthesizing the data dissemination protocol with a various number of processes. One can notice that most of the total synthesis time was spent on adding recovery, while a negligible amount of time was spent in validating the recovery transitions. The main reason for this behavior is that the structure of the fault-span in this case study is simpler: if a message is lost on one link, then until it is recovered, that message cannot be sent again (it is possibly lost on subsequent links).

<table>
<thead>
<tr>
<th>No. of Processes</th>
<th>Reachable states</th>
<th>Space (MB)</th>
<th>Recovery time (s)</th>
<th>Total time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>10^{18}</td>
<td>3.40</td>
<td>3.44</td>
<td>1105</td>
</tr>
<tr>
<td>150</td>
<td>10^{16}</td>
<td>22.64</td>
<td>130</td>
<td>451</td>
</tr>
<tr>
<td>100</td>
<td>10^{14}</td>
<td>14.28</td>
<td>48</td>
<td>1146</td>
</tr>
<tr>
<td>80</td>
<td>10^{12}</td>
<td>11.7</td>
<td>40</td>
<td>63</td>
</tr>
<tr>
<td>50</td>
<td>10^{10}</td>
<td>9.12</td>
<td>6</td>
<td>24</td>
</tr>
<tr>
<td>30</td>
<td>10^{8}</td>
<td>7.61</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 6. Nonmasking with linear topology data dissemination program.

5 Reducing the Complexity with Hierarchical Structure

Based on the case studies, we can observe that as the number of nodes in the hierarchy increases, the time complexity can increase substantially. For example, in the first case study, when we increased the height of the binary tree from 3 (7 processes) to 4 (15 processes), the synthesis time increased from 5 to 72 seconds. This is expected since the state space increases from \( 10^6 \) to \( 10^{16} \) states. Thus, a natural question in this area is whether the structure of the hierarchical system can assist in reducing the complexity. We show that the answer to this question is affirmative. For simplicity, we illustrate this in the context of the linear topology: the illustration can be trivially extended to the binary tree topology.
Consider the case where the system is as shown in Figure 7.a and let the constraints used during synthesis be $\forall j : C_j$, where the quantification is over the set of all processes in the system. Let $C_j$ be a constraint that depends on the variables of process $j$, $j - 1$ (if it exists) and $j + 1$ (if it exists). Furthermore, assume that constraints for intermediate processes are identical except for the renaming variables. Let the order of predicates added for system in Figure 7.a be $C_A, C_B, C_D$. Furthermore, let the added recovery actions be $rec_A, rec_B, rec_D$.

![Figure 7. Complexity and hierarchy](image)

**Theorem 5.1** If $(rec_A \land rec_B \land rec_D)$ form the recovery actions for the program in Figure 7.a then $(rec_A \land rec_B \land rec'_C \land rec'_D)$ form the recovery actions for the program in Figure 7.b where $rec'_C$ is obtained by replacing $B$ by $C$ and (then) replacing $A$ by $B$ from $rec_B$ and $rec'_D$ is obtained by replacing $B$ by $C$ in $rec_D$.

For reasons for space, we omit the proof of this theorem.

While the above result is straightforward and widely understood, it is especially useful for managing complexity of hierarchical systems. While results of this form have been presented in the literature, the pre-conditions that must be satisfied to apply it are often difficult to evaluate during automated synthesis. However, the conditions of the above theorem are easy to evaluate and this theorem can reduce the complexity of synthesizing systems with a larger number of nodes.

6 Related Work

Automated program synthesis is studied from different perspectives. One approach (e.g., [4]) focuses on synthesizing fault-tolerant programs from their specification in a temporal logic (e.g., CTL, LTL, etc.). Our approach is closer in spirit to an alternative approach ([14,15]) that focuses on program revision where existing program is modified to add fault-tolerance properties. The algorithms in [9,12,19] are targeted toward adding masking fault-tolerance whereas the proposed work focuses on nonmasking and stabilizing fault-tolerance. Furthermore, [9,12,19] do not take fairness into account. Hence, if applied to the first case study, they fail to synthesize the program. Moreover, the structure of the recovery needed for the second case study is too complex to be derived by existing approaches.

Our approach for adding stabilization is based on satisfying constraints that should be true in legitimate states. An orthogonal approach is to utilize primitives such as distributed reset [17] where one detects whether the system is in a consistent state and resets it to a legitimate state, if needed. Examples of these approaches include [17,22]. Our approach can be utilized to design the distributed reset protocol itself. In particular, the diffusing computation considered in Case Study 2 is from the distributed reset protocol in [6].

7 Conclusion

In this paper, we focused on automated addition of nonmasking and stabilizing fault-tolerance to hierarchical systems. In particular, we considered systems where legitimate states are specified in terms of constraints that are true in legitimate states. The goal of adding nonmasking and stabilizing fault-tolerance was to ensure that if these constraints are violated by faults then eventually the program would reach a state where all the constraints are satisfied and, hence, subsequent behavior would be correct.

Our approach was to utilize an order among constraints. With this order, we ensured that correction actions that correct constraint $C_i$ did not cause violation of any of the previous constraints $C_0, C_1..C_{i-1}$ although they may violate constraints $C_j, j > i$. In our case studies, we considered different possible orderings and in most cases, we were able to synthesize a nonmasking fault-tolerant program. Therefore, identifying an order among these predicates does not appear to be a critical concern. Moreover, as discussed in Section 3.3, the number of orderings that need to be considered for a group of $n$ constraints will be at most $O(n^2)$. Finally, we find that this approach is especially suited for synthesizing stabilizing programs, since it eliminates one of the bottlenecks of the automated synthesis (evaluating fault-span).

Based on theorem 3.1 and the fact that we consider all transitions that preserve preceding constraints, it follows that if one can identify a valid order of constraints where each constraint can be satisfied atomically without violating preceding constraints, our algorithm is guaranteed to find a nonmasking fault-tolerant program. It follows that if this algorithm fails to find a nonmasking fault-tolerant program then we can utilize the offending states (from where recovery cannot be added) to determine whether the given constraint array is inaccurate, whether ordering among constraint is incorrect, and so on. Certain requirements such as ‘the set of nodes form a ring’ cannot be expressed in terms of constraints that can be atomically satisfied. However, based on the examples from the literature, constraints encountered in a fixed hierarchical system can be easily expressed in terms of constraint array required for our algorithm.

We illustrated our approach with three case studies: stabilizing mutual exclusion, stabilizing diffusing computation and a data dissemination problem for sensor networks. The complexity analy-
sis demonstrated that automated synthesis in these case studies was feasible and achieved in a reasonable time. Furthermore, since our work is structured on constraint-based (manual) design of nonmasking and stabilizing fault-tolerance from [6] that has been found to be useful in deriving several protocols manually (e.g., [14, 15, 22]), we expect that it will be highly valuable for automatically designing various stabilizing and nonmasking programs. We also showed that the hierarchical nature of the underlying system could be effectively utilized to reduce the complexity of synthesizing programs with larger number of processes while maintaining the correct-by-construction property of programs designed by automated synthesis.

This work also advances the state-of-the-art automated synthesis in yet another way. To our knowledge, this is the first instance where automated synthesis of fault-tolerance is achieved with fairness constraints. Without fairness constraints, a stabilizing mutual exclusion algorithm based on [20] is impossible. Moreover, the structure of the recovery actions in the first two case studies is too complex to successfully utilize previous heuristic-based approaches [9].

Our approach for synthesizing nonmasking fault-tolerance can be combined with addition of safety in a two-step method to obtain a masking fault-tolerant program [8, 19]. One of the future works in this area is to identify how the approach from [19] can be implemented using BDDs and how it affects complexity of synthesis. It would also be possible to design masking fault-tolerant programs that are correct under some fairness constraints.

Although we assumed that the constraints in the invariant of the program were specified manually, it is possible to derive these constraints if the user provided initial states for the fault-intolerant program. In particular, we can identify the overall invariant and individual constraints for a given node in less than one second. However, partitioning constraints of a given process (e.g., see constraints S1, S2, and S3 in 4.1.2) requires manual effort. One future work in this area is to automate the task of constraint generation.

References