Automated constraint-based addition of nonmasking and stabilizing fault-tolerance

F. Abujarad \( ^{a, \ast} \), S.S. Kulkarni \( ^b \)

\( ^a \) Department of Emergency Medicine, Yale University, New Haven, CT 06519, USA
\( ^b \) Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824, USA

**Abstract**

We focus on the 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, we expect our approach to have a significant benefit in automating the addition of fault-tolerance. We illustrate our algorithm with four case studies: stabilizing mutual exclusion, stabilizing diffusing computation, a data dissemination problem in sensor networks, and tree maintenance. With experimental results, we show that the complexity of our algorithm is reasonable and that it can be reduced using the structure of the hierarchical systems.

We also reduced the time complexity of the synthesis using parallelism. We consider two approaches to speedup the synthesis algorithm: first, the use of the multiple constraints that have to be satisfied during synthesis; second, the use of the distributed nature of the programs being synthesized. We show that our approaches provide significant reduction in the synthesis time.

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 three 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.

© 2011 Elsevier B.V. All rights reserved.

1. Introduction

At design time, it may not be possible to predict all faults that a distributed program may be subject 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 [1]. 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 and stabilizing 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. Hence, nonmasking fault-tolerance is useful to tolerate a temporary perturbation of the program state. After recovery...
is complete, 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 [2].

A special case of nonmasking fault-tolerance is stabilization [3,4], 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 [5]. Hence, techniques for automated synthesis are expected to be useful for designing stabilizing systems.

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 [6]. Also, the design of nonmasking fault-tolerance can assist and simplify the design of masking fault-tolerance [7,8]. Moreover, in several applications nonmasking fault-tolerance is more desirable than solutions that provide failsafe fault-tolerance (where in the presence of faults the program reaches “safe” states from where it does not satisfy liveness requirements). This is especially true for networking related applications such as routing and tree maintenance.

There are several issues that complicate the manual design of nonmasking fault-tolerance [10]. 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. For these reasons, algorithms that automate the addition of nonmasking fault-tolerance are especially valuable.

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 legitimate states. 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. Since the time complexity of the automation algorithms can be high, we also evaluate parallelization techniques to expedite addition of nonmasking and stabilizing algorithm.

Contributions of the paper

• We propose an automated synthesis algorithm for constraint-based synthesis of nonmasking fault-tolerant programs. We illustrate our algorithm with four case studies. We note that the structure of the recovery actions in the first, the third, and the fourth case studies is too complex to permit previous approaches to achieve synthesis of the corresponding fault-tolerant programs [11].
• We present a multi-core algorithm to synthesize distributed stabilizing fault-tolerant programs by partitioning the satisfaction of the constraints among available threads.
• We present a multi-core algorithm that utilizes the distributed nature of programs being synthesized.
• As a part of this work, we modify the MDD (Multi-valued Decision Diagrams) library [12] to make it reentrant and to use it in the parallel synthesis.
• We also show that the structure of the hierarchical (tree-based) 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., [11]) will declare failure in adding fault-tolerance.

Organization of the paper. The rest of the paper is organized as follows. In Section 2, we define distributed programs, faults, and specifications. Subsequently, in Section 3 we show how we model distributed programs and the read/write restrictions. We describe the algorithm for the automated addition of nonmasking and stabilizing fault-tolerance in Section 4. We present our multi-core algorithms in Section 5 and experimental results in Section 6. In Section 7, we discuss the effect of the order in which our algorithm uses to satisfy the constraints. We show that the structure of the hierarchical system can be used to reduce complicity of synthesis in Section 8. Finally, we discuss related work in Section 9 and conclude in Section 10.

2. Programs and specifications

In this section we define the notion of distributed programs, faults, and the problem statement for adding nonmasking and stabilizing fault-tolerance. Those definitions are based on the ones given by Arora and Gouda [13]. We also identify how the notion of fairness can be modeled for automated addition of stabilizing fault-tolerance.
For the following definitions of enabled and fairness let \( S_i \) be a set of states. A transition over \( S_i \) is of the form \((s_0, s_1)\), where \( s_0, s_1 \in S_i \). Let \( \alpha, \alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_m \) be sets of transitions over \( S_i \). In other words, \( \alpha, \alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_m \) are subsets of \( S_i \times S_i \).

**Enabled.** Intuitively, \( \alpha \) is enabled in \( s_0 \) if \( \alpha \) contains some transitions that begin 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 \((\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_m)\) and \( \alpha_i \) is continuously enabled in that sequence then that sequence includes a transition in \( \alpha_i \). Formally, an infinite sequence \((s_0, s_1, s_2, \ldots)\) is fair with respect to \((\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_m)\) iff for each \( i, k \) the following condition is satisfied:

\[
(\alpha_i \text{ is enabled in each state } s_k, s_{k+1}, \ldots) \Rightarrow (\exists l : l \geq k : (s_l, s_{l+1}) \in \alpha_i).
\]

Note that this definition is equivalent to weak fairness from [14–16].

**Program.** A program \( p \) is specified in terms of its state space, \( S_p \) and the transition sets \((\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_m)\), where for each \( i, \alpha_i \subseteq S_p \times S_p \). The transitions of \( p \), \( \delta_p \), are equal to \( \alpha_1 \cup \alpha_2 \cup \alpha_3 \cup \cdots \cup \alpha_m \). We use the notation \((\delta_p, (\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_m))\) 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 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 MDD [12] based implementation, we represent it using an equivalent function \( f_C \) with domain \( S_p \) and range \{True, False\} where \( f_C(s) = \text{True} \iff s \in S_C \). Let \( C_1 \) and \( C_2 \) be two state predicates represented by sets \( S_{C_1} \) and \( S_{C_2} \), respectively. Let \( f_{C_1} \) and \( f_{C_2} \) be corresponding functions. Observe that the function corresponding to \( S_{C_1} \cap S_{C_2} \) is \( f_{C_1} \land f_{C_2} \) where \( f_{C_1} \land f_{C_2}(s) = f_{C_1}(s) \land f_{C_2}(s) \). 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 MDD-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 \{True, 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 \( I = C_1 \land C_2 \land \cdots \land C_m \), denoted as invariant \( I \), identifies all legitimate states of \( p \). In other words, if a computation of \( p \) begins in a state that is in \( I \) then (1) \( I \) is True at all states in that computation and (2) the computation is 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 \( I \) is the set of states reached by starting from a state in \( I \) and executing transitions from \( p \forall f \).

The goal of an algorithm that adds nonmasking fault-tolerance is to begin with a fault-intolerant program \( p \), its invariant \( I \), and faults \( f \), and to derive the nonmasking fault-tolerant program, say \( p' \), such that in the presence of faults, \( p' \) eventually converges to \( I \). Furthermore, computations of \( p' \) that begin in \( I \) must be the same as that of \( p \). Based on this discussion, we define the problem of adding nonmasking fault-tolerance as follows:

**Problem statement 2.1** Given \( p, I, \) and \( f \), identify \( p' \) such that:

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

- There exists a state predicate \( T \) (fault-span) such that:
  \(- I \subseteq T \),
  \(- (s_0, s_1) \in (p' \forall f) \land (s_0 \in T) \Rightarrow s_1 \in T \),
  \(- s_0 \in T \land (s_0, s_1, \ldots) \text{ is a computation of } p' \Rightarrow (\exists j : j \geq 0 : s_j \in I) \).

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. Based on this discussion, we define the problem of adding stabilizing fault-tolerance as follows:

**Problem statement 2.2** Given \( p, I, \) and \( f \), identify \( p' \) such that:

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

- All program transitions eventually converge to the invariant:
  \(- s_0 \in S_p \land (s_0, s_1, \ldots) \text{ is a computation of } p' \Rightarrow (\exists j : j \geq 0 : s_j \in I) \).
Note that since each constraint is preserved by the original program $p$, closure property of the stabilizing fault-tolerant program $p'$ is satisfied from the first constraint of the problem statement. Hence, it is not explicitly specified above.

3. Modeling distributed programs

In this section, we present our approach for modeling programs. For brevity, we specify the state space of a program in terms of its variables. The state space of the program is obtained by assigning each variable each possible value from its domain. Furthermore, since we focus on distributed programs, we specify the transitions of the program in terms of a set of processes. 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 are the union of the transitions of its processes.

A process in a distributed program has a partial view of the program variables, which introduces write/read restrictions.

Therefore, when a new program transition is added/removed, we need to add/remove a group of transitions based on the variables that cannot be read/written by that process. Let $k$ be a process, let $R_k$ be the set of variables that $k$ can read, and let $W_k$ be the set of variables that $k$ can write. Also, let $v_a(s_0)$ denote the value of variable $v_a$ in the state $s_0$. Then, write/read restrictions of the process are defined as follows.

3.1. Write restrictions of distributed programs

If $k$ can only write the subset of variables $W_k$ and the value of a variable other than that in $W_k$ is changed in the transition $(s_0, s_1)$ then that transition cannot be used in synthesizing the transitions of $k$. In other words, being able to write the subset $W_k$ is equivalent to providing a set of transitions $write_k(W_k)$ that $k$ can use in the synthesis algorithm, where

\[
write_k(W_k) = \{(s_0, s_1) : (\forall v : v \notin W_k : v(s_0) = v(s_1))\}.
\]

3.2. Read restrictions of distributed programs

If $t = (s_0, s_1)$ is a transition that $k$ can execute, then the group of transitions associated with $t$ must also include transitions of the form $(s_2, s_3)$ where $s_0$ and $s_2$ (respectively $s_1$ and $s_3$) are undistinguishable for $k$, i.e., they differ only in terms of the variables that $k$ cannot read. The synthesis algorithm uses the function $Group$ to include these additional transitions. The group itself is given by the following formula:

\[
\text{group}_k(t) = \bigvee_{(s_2, s_3)} \left( \bigwedge_{v \notin R_k} (v(s_0) = v(s_1) \land v(s_2) = v(s_3)) \land \bigwedge_{v \in R_k} (v(s_0) = v(s_2) \land v(s_1) = v(s_3)) \right).
\]

Observe that for the transition $t$, $\text{group}_k(t)$ can be executed by process $k$ while respecting its read/write restrictions. Let $tr_k$ denote a set of transitions. Now, based on the notion of read/write restrictions, $tr_k$ can be included as transitions of process $k$ iff there exist transitions $t_1, t_2, \ldots, t_x$ such that $tr_k = \text{group}_k(t_1) \cup \text{group}_k(t_2) \cup \cdots \cup \text{group}_k(t_x)$. Furthermore, let $p$ be a program whose transitions are specified with the processes $1, 2, \ldots, x$. Also, let $tr_p$ denote a set of transitions that can be included as transitions of $p$ iff there exists a set of transitions $tr_1, tr_2, \ldots, tr_x$ such that $\forall k : 1 \leq k \leq x, tr_k$ can be included as transitions of process $k$.

Remark. Note that the $\text{group}_k(t)$ is defined only if $t$ does not violate write restrictions of process $k$. However, for brevity, we do not specify this whenever it is clear from the context.

The way we use this group operation is as follows: When we compute a set of transitions say $tr$ that we need to either add or remove, we ensure that $tr$ can be implemented using read/write restrictions of the synthesized program. Hence, often, we cannot add/remove $tr$ as is. Instead, we need to revise $tr$ so that it respects the read/write restrictions of the program being synthesized. One operation we utilize for this is called $Group$, where $Group_{\text{max}}(tr)$ returns a superset, say $tr_{\text{larg}}$, that can be included as transitions of the synthesized program. The intuition of $Group_{\text{max}}$ operation is as follows (c.f. Algorithm 3.1): Given a set of transitions, say $tr$, we use a loop that traverses through all the processes. While traversing process $k$, it computes subset of transitions, say $tr_k$, in $tr$ such that each transition in $tr_k$ satisfies the write restrictions of process $k$. Then, for each subset in $tr_k$, it applies the group operation described above to compute other transitions that must be included. (Note that with the use of MDDs [12], we do not have to actually evaluate each transition in $tr_k$ separately to compute the corresponding group. However, the details of how this can be achieved with MDDs are outside the scope of the goal of this paper.) Finally, it takes a union of all transitions obtained thus to compute $Group_{\text{max}}(tr)$. Another operation we utilize is $Group_{\text{min}}$. This operation takes a set of transitions $tr$ as input and returns $tr_{\text{small}}$ such that $tr_{\text{small}}$ can be included as transitions of the synthesized program and $tr_{\text{small}} \subseteq tr$. The operation $Group_{\text{min}}$ is implemented in a similar fashion to that of $Group$ by traversing through all processes.

Remark. Since $Group_{\text{max}}$ is the operation that is used most frequently in our algorithms, for simplicity of presentation, we drop the subscript and call it $Group$. 

\[ \text{Group}_{\text{max}}(tr) \]

\[ \text{Group}_{\text{min}}(tr) \]
Algorithm 3.1 Group

Input: transitions set trans.
Output: transitions group transg.

1: MDD tPred := MDD[numberOfProcesses];
2: for j := 0 to numberOfProcesses do
3:   tPred[j] := trans & writej(Wj);
4:   tPred[j] := FindGroup(tPred[j], Rj);
5: end for
6: MDD transg := false;
7: for j := 0 to numberOfProcesses do
8:   transg := transg ∨ tPred[j];
9: end for
10: return transg;

Based on the above description, the sequential implementation of the Group is as shown in Algorithm 3.1. This algorithm (taken from [29]) takes a transition set, trans, as an input and computes the transition group, transg, as an output. Specifically, it creates an array, tPred[], with number of elements equal to the number of processes such that tfPred[j] holds the part of the group transitions associated with the process j (Line 1). Now, based on Wj (i.e. the set of variables the process j is allowed to write) the group algorithm uses the function writej(Wj) to find the set of all transition which process j is permitted to execute. Then, it uses this set to find which of the transitions in trans process j is responsible for (Line 3). Later, it uses the tPred[j] and Rj in the function FindGroup to account for all variables that process j cannot read and computes the transitions that cannot be distinguished by, j (Line 4). Once the steps in Lines 3 and 4 are completed for all processes, the algorithm collects the transitions of the group in transg (Lines 7–9) and returns.

4. Synthesis algorithm of the nonmasking and stabilizing fault-tolerance

In this section, we present our approach for adding nonmasking and stabilizing fault-tolerance to fault-intolerant programs based on [2]. The goal of nonmasking and stabilizing fault-tolerance is to ensure that after faults occur, the program eventually reaches one of the legitimate states in the invariant I, we focus on the instance of the problem where \( I = C_1 \land C_2 \land \cdots \land C_m \), 1 ≤ i ≤ m, and Ci is a constraint on the variables of the program. Faults, f, perturb the program to a state in (¬I). Hence, in the presence of f, one or more of the constraints from C1, C2 . . . Cm are violated. The goal of our algorithm 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 I holds.

4.1. Constraint satisfier

Our algorithm for adding nonmasking and stabilizing fault-tolerance is shown in Algorithm 4.1. The input for the algorithm is the constraint array C, fault-span T,1 and program p. In this algorithm, the constraints from the constraint array are satisfied one after another. The algorithm starts by computing the invariant as the intersection of all constraints in the constraint array (Line 3).

Then, the algorithm computes the recovery transitions to satisfy C[i]. Let Tr denote transitions that begin in the fault-span, T, and in a state where C[i] is false and end in a state where C[i] is true. Unfortunately, we cannot add Tr as is, since Tr may not be implementable using read/write constraints on the processes due to the distributed nature of the program. The algorithm adds a subset of Tr, say Tr1, such that Tr1 can be implemented using the read/write restrictions of one or more processes. We denote this by the function Groupmin (see Line 6).2 This ensures that the only transitions added are those that start from a state where C[i] is false and reach a state where C[i] is true. These transitions are denoted by temp on Line 6.

Subsequently, the algorithm removes transitions from temp that violate the closure of the fault-span T. Thus, it computes a subset of transitions in temp that begin in a state in T and reach a state in ¬T. Again, we need to ensure that the removed transitions are consistent with read/write restrictions of processes. The algorithm achieves this by applying function Groupmax to temp; this computes a superset of temp such that one or more processes can execute it. Subsequently, it removes this superset from temp (Line 7). This ensures that all transitions that violate closure of T are removed. Likewise, it removes transitions that violate the closure of I (Line 8).

The algorithm needs to ensure that none of the transitions used to satisfy the constraint say C[i] violate the pre-satisfied constraints C[0] to C[i − 1]. Hence, it lets V include the transitions that originate from a state where C[i − 1] is true and end in a state where C[i − 1] is false as well as similar transitions for the constraints C[0] to C[i − 2] (Line 11). The transitions in V

---

1 Fault-span, T, could be computed to be the states reached in the presence of faults. However, this is not always a requirement. T can be any predicate that satisfies the constraints of the fault-span. For example, T could be set to True when dealing with stabilization.

2 \((X \land (Y'))\) refers to the transitions that start in a state in X and reach Y.
Algorithm 4.1 ConstraintSatisfier

Output: recovery transitions $rec$.

1: $temp, V := false, false$;
2: $m := \text{SizeOf}(C) - 1$; // $m$ is the number of constraints
3: $I := \bigwedge_{i=0}^{m} C[i]$; //Compute $I$ (invariant) as the intersection of all constraints
4: for $i := 0$ to $m$ do
5: //temp is the transitions that start in a state in $T - C(i)$ and reach $C(i)$
6: $temp := \text{Group}_{\text{max}}((T - C[i]) \land (C[i])')$;
7: //ensure that no recovery transitions violate $T$
8: $temp := temp - \text{Group}(temp \land (T \land \neg I))$;
9: //ensure that no recovery transitions violate $I$
10: $temp := temp - V$;
11: // Compute, $V$, the set of the transitions that violating the constraints
12: $V := V \land \text{Group}(C[i] \land (\neg C[i])')$;
13: end for
14: // return the recovery transition.
15: return $rec$;

are used to ensure that recovery transitions do not violate other pre-satisfied constraints. The algorithm ensures that none of the transitions in $temp$ interfere with earlier constraints. Therefore, it removes the transitions in $V$ from $temp$ (Line 9). At this point the algorithm collects all recovery transitions in $rec$ (Line 10). Steps 4–12 are repeated until all the recovery actions that satisfy all the constraints in the array $C$ are found. Finally, it returns the recovery actions of the program $p$.

Theorem 1. Given are fault-intolerant program $p$, constraints $C_1, C_2 \ldots C_m$, and faults $f$.
- Let $I = C_1 \land C_2 \land \ldots \land C_m$.
- Let $T = \text{set of states reached in the execution of } p \lor f \text{ that start from any state in } I$.
- Let $rec = \text{ConstraintSatisfier}(C, T, p)$.

If $\forall s_0 : s_0 \in T - I : (\exists s_1 : s_1 \in T : (s_0, s_1) \in rec)$
Then $(S_p, (rec, \delta_p))$ solves the constraints in Problem statement 2.1.

Proof. To prove Theorem 1 we show that the $(S_p, (rec, \delta_p))$ solves the constraints of the problem statement 2.1.

- By the construction of the transitions in $rec$, it is straightforward to see that $rec$ does not introduce any new transitions in $I$. Therefore, the transitions within the invariant remain unchanged.
- By the construction of $T$, it is clear that $I \subseteq T$ since $T$ includes all the states in $I$ as well as the states reachable from $I$ by $(p \lor f)$.
- From Line 7 in the algorithm ConstraintSatisfier, the transitions in $rec$ does not include any transition that violates $T$.
- Since $rec$ does not include any of the transitions from $V$ (Lines 9 and 11), none of the transitions in $rec$ violate pre-satisfied constraints. Therefore, there will be no cycles between the recovery transitions themselves. Hence, the constraint $(s_0 \in T \land \langle s_0, s_1, \ldots \rangle)$ is a computation of $p' \Rightarrow (\exists j : j \geq 0 : s_j \in I)$ is satisfied. □

4.2. Algorithm illustration

To illustrate the algorithm ConstraintSatisfier, consider the program described in Fig. 1. In this program, we have three constraints $C_1, C_2$, and $C_3$ and $I = C_1 \land C_2 \land C_3$. Since $C_1$ is the first to be satisfied, we construct all possible recovery actions that start from any state in $T - C_1$ and reach a state in $C_1$. We proceed to satisfy $C_2$ in the same manner. However, after constructing the recovery actions that satisfy $C_2$, we need to exclude actions that violate the constraint $C_1$. In particular, we exclude actions like $rec_1$ (c.f. Fig. 1) since it starts from a state, $s_0$, where $C_1$ is True and ends in a state, $s_1$, where $C_1$ is false. On the other hand, we keep transitions like $rec_2$ and $rec_3$. We continue to construct the recovery actions that establish $C_3$ provided that they preserve $T, C_1$, and $C_2$.

5. Using parallelism in synthesis

In Section 4, we described the sequential approach (i.e., single thread) for synthesizing nonmasking and stabilizing fault-tolerant distributed programs from fault-intolerant versions. After reviewing Algorithm 4.1, we can see that there are two main bottlenecks. The first is the main loop (Lines 4–12) where the number of iterations is determined by the number of constraints. The second is the Group operation in Lines 6, 7, 8 and 11. Next, we describe our choices in parallelizing these
5.1. Design choices for constraint satisfaction

One way to partition the computation of recovery transitions is to split the recovery computation among multiple threads by allowing them to work on satisfying separate constraints. However, Algorithm 4.1 uses the computation of $V$, transitions that violate preceding constraints (Line 11). Clearly, one possibility is to compute all possible values taken by $V$ during the computation upfront and utilize them appropriately for computing valid recovery transitions. Computing the possible values taken by $V$ also requires a computation that utilizes a loop that requires $\text{sizeOf}(C)$ iterations, which can be parallelized using standard techniques from parallel computing.

After computation of $V$, we can partition the iterations (Lines 4–12 in Algorithm 4.1) among several threads. We considered several approaches for this. One approach we considered was dynamic partitioning. In particular, in this approach, a pool of uncompleted iterations is maintained. Each thread picks an iteration from this pool and computes the recovery transitions for that iteration. Subsequently, it picks another iteration from the pool and so on. We found that this dynamic partitioning approach, however, resulted in a high overhead thereby reducing the speedup. Hence, we considered static partitioning where each thread was given fixed iterations. Even here, we tried different options. One option was to partition the iterations in an alternating manner (e.g., thread 1 gets iterations 0, 2, 4, . . . and thread 2 gets iterations 1, 3, 5, . . . ). It was expected that this would leave the size of MDDs used in each thread to be evenly balanced. However, we found that this approach and the approach of partitioning where thread 1 got iterations 0, 1, . . . ($\text{sizeOf}(C)/2$ − 1) and thread 2 got iterations $\text{sizeOf}(C)/2$, . . . $\text{sizeOf}(C) − 1$ had almost identical performance in the case studies. We have used the latter in our experiments. However, we believe that the choice of partitioning could play a role in other case studies.

Thus, our algorithm for satisfying the constraints in parallel is as shown in Algorithm 5.1. This algorithm begins with the array of constraints to be satisfied $C$, fault-intolerant program $p$, fault-span $T$, and the number of worker threads to be spawned $n$. The goal of this algorithm is to discover the set of recovery transitions $\text{recAll}$ such that all the constraints in $C$ are satisfied in a way that enables the fault-tolerant program to recover to its legitimate states. Initially, the algorithm starts by computing the invariant $I$ as the intersection of all constraints (Lines 2). Now, the algorithm constructs the array $V$ such that $V[i]$ includes the transitions that start from a state where $C[i]$ is True and end in a state where $C[i]$ is False as well as the similar transitions for the constraints $C[j]$, where $0 \leq j \leq n − 1$ (Lines 3–8).

A more efficient way to do this computation is by letting the master thread use the worker threads such that each worker thread computes its share of $V$ elements such that $V[i]$ contains the transitions that starts from $C[i]$ and end in $\neg C[i]$. Once all threads are done, the master thread updates the array $V$ such that $V[i] = V[i] \lor V[i]$. In other words, $V[i]$ contains all transitions that violate the constraint $C[0]$ to $C[i]$.

After constructing the array $V$, the algorithm proceeds to evenly distribute the elements of the arrays $C$ and $V$ among the worker threads (Lines 9–12). Specifically, $C_{p}[i]$ includes the array of constraints assigned to the thread $i$, and $V_{p}[i]$ includes the array of corresponding constraints violating transitions. Note that the availability of the array $V_{p}$ enables each worker thread to work independently without interfering with the other threads. To compute the respective recovery transitions, each worker thread (Lines 13–15) calls the algorithm $P\text{ConstraintSatisfier}$, which is similar to the Algorithm 4.1 except that in addition to $C_{p}$ and $p$ it also takes $V_{p}$ and $I$ as an input. Once all worker threads complete their jobs (Line 16), the master thread collects all the recovery transitions returned by worker threads in $\text{recAll}$ (Lines 17–19) and returns the overall recovery transitions.

5.2. Design choices for utilizing distributed nature

When the recovery algorithm adds new transitions (or removes transitions that violate earlier constraints), we have to add the corresponding group of transitions based on the distributed nature of the program. Moreover, with symbol
Algorithm 5.1 ParallelConstraintsSatisfaction [Master Thread]

**Input:** constraint array $C$, program transitions $p$, fault-span $T$, and number of threads $n$.

**Output:** recovery transitions $recAll$.

```
1: $gAll := false$;
2: $l := \bigwedge_{i=0}^{m} C[i]$;
3: for $i := 1$ to $n - 1$ do
4:     SpawnThread $\rightsquigarrow$ ComputeViolate($i$);
5: end for
6: for $i := 1$ to $SizeOf(C) - 1$ do
7:     $V[i] := V[i - 1] \lor V[i]$;
8: end for
9: for $i := 0$ to $n - 1$ do
10:    $C_p[i] = Split(i, C)$;
11:    $V_p[i] = Split(i, V)$;
12: end for
13: for $i := 1$ to $n - 1$ do
14:    $rec[i] :=$ SpawnThread $\rightsquigarrow$ PConstraintSatisfier($C_p[i], p, fault-span T, V_p[i], I$);
15: end for
16: ThreadJoin(0 . . . $n - 1$);
17: $recAll := \bigvee_{i=0}^{n-1} rec[i] ;// Merging the results from all threads$
18: return $recAll$;
```

approach, we add (or remove) a set of transitions at a time. This set may include transitions that could be executed by several processes. Therefore, for a given set of transitions that are added, we need to consider read/write restrictions of each of these processes to determine the group for that set of transitions. We can utilize this feature to parallelize the group computation itself by having each thread compute the group corresponding to a subset of processes.

Again, similar to the parallelization with constraints, we considered several approaches. It turned out that even for this approach, the overhead of dynamic partitioning was more than its benefit. Thus, we utilized static approaches. Since several approaches considered for partitioning resulted in a similar speedup, in this paper we utilize the simple approach where each thread obtains a subset of processes and computes the corresponding group for those processes. Thus, the algorithm for parallelization of the group computation requires us to parallelize the loop on Lines 2–5 in Algorithm 3.1. Since the parallelization of this algorithm is similar to that in Algorithm 4.1, we omit it.

Finally, in the group parallelization, the actual computation involved in the group itself is small. Hence, we found that the overhead of creating and terminating threads for each group computation was very high. For this reason, we created the threads upfront and used mutexes to determine when they will be active.

5.3. Design choices for parallelizing the MDD (multi-valued decision diagrams) library

Since we are using MDD-based symbolic synthesis [1], the constraints are characterized by Boolean formulas involving the variables in the program being synthesized. The MDD library [12] is not designed to be reentrant and assumes that at most one MDD package is active at any given time. Multiple threads cannot operate on the same MDD package simultaneously. Also, different threads cannot access different MDD packages simultaneously. We considered two approaches to solve this problem: (1) utilize a reentrant version of the MDD package, or (2) utilize multiple independent MDD packages. Since, reentrant MDD package is not available, we followed the second approach. Also, the MDD package had several macros and functions, which assume that only one MDD package is used. We modified the MDD package code such that multiple instances of the MDD package can exist simultaneously. We also added a Transfer function to transfer an MDD object from one MDD package to a different MDD package. Hence, during the parallel algorithms, a master thread spawns several worker threads each running on a different core/socket in parallel with an instance of its own MDD package. The instance of the MDD package assigned to each worker thread is initialized using MDDs (e.g., program transitions MDD) transferred from the MDD package of the master thread.

6. Case studies

In Section 4, we presented our approach for constraint-based automated addition of nonmasking and stabilizing fault-tolerance. In Section 5, we presented different approaches to exploit parallelism. In Sections 6.1–6.4, we describe and analyze four case studies, namely the Stabilizing Mutual Exclusion [17], the stabilization of Data Dissemination Problem in Sensor Networks [18], the Stabilizing Diffusing Computation [2] and the Stabilizing Distance Based Spanning Tree. Of these, the first and the third case study are classic problems from distributed computing and illustrate the feasibility of algorithms that add stabilizing fault-tolerance. In the second case study we demonstrate the applicability of our approach on a model of a real world problem, particularly, in the field of sensor networks. In all of these case studies, we find that our approach
for constraint-based automated addition of nonmasking and stabilizing fault-tolerance was successful in synthesizing the nonmasking fault-intolerant programs. Furthermore, we find that parallelism significantly reduces the total synthesis time.

To concisely describe the transitions of the program we use guarded command notation: \((\text{guard}) \rightarrow (\text{statement})\), where the guard is a Boolean expression over program variables and the statement describes how program variables are updated and it always terminates. A guarded command of the form \(g \rightarrow \text{st}\) corresponds to transitions of the form \([(s_0, s_1)]\) \(g\) evaluates to True in \(s_0\) and \(s_1\) is obtained by executing \(\text{st}\) from \(s_0\). Throughout this section, all experiments are run on a Sun x4275 with 4 x Quad-core Intel Xeon E5520 (2.27 GHz w/ 8 MB cache) processors with 24 GB RAM. The MDD representation of the Boolean formulas has been done using a modified version of the MDD/BDD Glu 2.1 package [12] developed at the University of Colorado [12]. We allowed each run to take at most 1 h. If the time needed to complete the run was more than one hour, we terminated the corresponding run.

6.1. Case study 1: stabilizing mutual exclusion program

Mutual exclusion is one of the fundamental problems in distributed/concurrent programs. One of the classic solutions to this problem is the token-based solution due to Raymond [17]. In this solution, the processes form a directed rooted tree, a 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 stabilization to the fault-intolerant program in [8]. When faults occur and perturb the holder tree, the new program will stabilize and reconstruct a correct holder tree within a finite number of steps under weak fairness assumption.

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, Fig. 2(a) represents the undirected parent tree and Fig. 2(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, Fig. 2(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:

\[ A1 \triangleq (h.k = k \land j \in \text{Adj.k}) \land (h.j = k) \rightarrow h.k, \ h.j := j, \ j; \]

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:

\[
\begin{align*}
S1 & \triangleq \forall j : (P.j = P.j) \lor (h.j = j) \lor (\exists k : (P.k = j) \land (h.j = k)) \\
S2 & \triangleq \forall j : (P.j \neq j) \Rightarrow (h.j = P.j) \lor (h.(P.j) = j) \\
S3 & \triangleq \forall j : (P.j \neq j) \Rightarrow \neg((h.j = P.j) \land (h.(P.j) = j)).
\end{align*}
\]

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:

\[
(F1) \quad \text{true} \rightarrow \{h.j := \text{any arbitrary value from its domain}\};
\]
(a) Linear topology. (b) Binary tree topology.

Fig. 3. Stabilizing mutual exclusion.

| No. of Processes | Time(s) | | Total |
|------------------|---------|------------------|
|                  | constraint satisfaction | Validation |        |
| 30               | 19      | 21               | 40     |
| 40               | 78      | 74               | 153    |
| 50               | 217     | 238              | 457    |
| 60               | 505     | 509              | 1020   |
| 70               | 1110    | 1103             | 2238   |

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 $A_1$; the faults are specified by the fault action $F_1$; and the constraints are from $S_1, S_2,$ and $S_3$. We specified these constraints in the following order: first, we specified constraints $S_1$ for the root, then its children, then its grandchildren and so on. Subsequently, we specified constraint $S_2$ likewise. Finally, we specified constraint $S_3$ in the reverse order. The recovery actions computed by the synthesis algorithm are as follows:

$$R_1 :: \neg((h.j = P.j) \lor (h.j = j) \lor (\exists k : (P.k = j) \land (h.j = k)))$$
$$\quad \rightarrow h.j := j \mid h.j := P.j \mid h.j := \text{child of } j;$$

$$R_2 :: \neg((P.j \neq j) \Rightarrow (h.j = P.j) \lor (h.(P.j) = j))$$
$$\quad \rightarrow h.j := P.j \mid h.(P.j) := j;$$

$$R_3 :: \neg((P.j \neq j) \Rightarrow ((h.j = P.j) \land (h.(P.j) = j)))$$
$$\quad \rightarrow h.j := j \mid h.(P.j) := P.j \mid h.(P.j) := P.(P.j);$$

Analysis of experimental results. Fig. 3(a) shows the results of synthesizing the Stabilizing Mutual Exclusion program with various numbers of processes organized in linear topology. It shows the time needed, in seconds, to add recovery, validate the recovery transitions (i.e., insures previous constraints are not violated), and the total synthesis time in terms of the number of processes being synthesized. Fig. 3(b) shows the result of a similar case study where the processes are arranged in a binary tree topology. This result is even more evident in the third case study.

Fig. 3(b) 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 compare 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 MDD variable ordering. This can be achieved easily on a linear topology. However, for a tree topology, this is not possible for all the processes. Therefore, computing recovery transitions for those cases is more expensive.

Fig. 4 shows the results of using parallelism during constraints satisfaction in synthesizing the stabilizing Mutual Exclusion program. The table illustrates the results for various numbers of processes organized in linear topology (results for tree topology are similar) using different numbers of processors/cores. It shows the time needed, in seconds, to satisfy the constraints, and the total synthesis time. It also shows the amount of memory in megabytes. As we can see from this figure, using parallelism has substantially reduced the time needed for the synthesis. As a concrete example, observe that the time required to synthesize a stable mutual exclusion program with 50 processes dropped from 457 s, using the sequential algorithm, to 374 s when two cores were used, and to 178 s when four cores were used.

Fig. 5 shows the results of exploiting the distributed nature of the program being synthesized (i.e., Group parallelism) in synthesizing the stabilizing Mutual Exclusion program. It shows the time needed, in seconds, to compute the group, and the total synthesis time. It also shows the amount of memory in megabytes needed by our algorithm.
In this section, we consider faults that lose a message. To model such faults for the base station, we add action (Faults).

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).

We can clearly see the feasibility of adding stabilizing fault-tolerance using automated synthesis. Both time and space complexity are reasonable and proportional to the reachable state space. Furthermore, as specified in Section 8, utilizing the hierarchical structure can reduce the complexity for a larger number of processes.

### 6.2. Case study 2: 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. We note that this algorithm could be synthesized with a single conjunctive predicate. However, our goal, in this case study, is to illustrate how model based constraints (see MT1–MT6 described later in this section) can be handled in nonmasking fault-tolerance.

**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.) 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 predecessor had received the previous packet (actions \( IN2 \) and \( IN3 \)). Thus, the actions of the fault-intolerant program are as follows:

Action for base station:

\[(IN1) \quad (L0 = U1) \implies L0 := L0 + 1; \]

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

\[(IN2) \quad (Uj \leq U.(j + 1)) \land (Uj \leq U.(j - 1)) \land (L.(j - 1) = U.j + 1) \implies U.j, Lj := U.j + 1, L.j + 1; \]

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

\[(IN3) \quad U.M \leq U.(M - 1) \land L.(M - 1) = U.M + 1 \implies U.M, L.M := U.M + 1, L.M + 1; \]

**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.

\[(F1) \quad true \implies L0 := L0 + 1; \]

\[(F2) \quad (U.j \leq U.(j - 1)) \land (L.(j - 1) = U.(j + 1)) \implies U.j, Lj := U.j + 1, L.j + 1; \]

**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).

\[(S1) \quad (U.0 = N) \]

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

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

\[(S4) \quad (L.0 \leq U.1 + 1) \]

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

---

### Table 1: Reusable program: Data dissemination in sensor networks

<table>
<thead>
<tr>
<th>No. of Processes</th>
<th>reachable states</th>
<th>Sequential 2 threads</th>
<th>Mem (MB)</th>
<th>Sequential 4 threads</th>
<th>Mem (MB)</th>
<th>Sequential 8 threads</th>
<th>Mem (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Grp t(s)</td>
<td>Syn t(s)</td>
<td>Grp t(s)</td>
<td>Syn t(s)</td>
<td>Grp t(s)</td>
<td>Syn t(s)</td>
</tr>
<tr>
<td>20</td>
<td>(10^{26})</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>30</td>
<td>(10^{54})</td>
<td>40</td>
<td>40</td>
<td>13</td>
<td>36</td>
<td>37</td>
<td>32</td>
</tr>
<tr>
<td>40</td>
<td>(10^{54})</td>
<td>152</td>
<td>153</td>
<td>14</td>
<td>105</td>
<td>106</td>
<td>40</td>
</tr>
<tr>
<td>50</td>
<td>(10^{54})</td>
<td>455</td>
<td>457</td>
<td>15</td>
<td>320</td>
<td>322</td>
<td>44</td>
</tr>
<tr>
<td>60</td>
<td>(10^{106})</td>
<td>1014</td>
<td>1020</td>
<td>16</td>
<td>679</td>
<td>686</td>
<td>45</td>
</tr>
<tr>
<td>70</td>
<td>(10^{129})</td>
<td>2213</td>
<td>2238</td>
<td>17</td>
<td>2110</td>
<td>2135</td>
<td>47</td>
</tr>
</tbody>
</table>

---

**Fig. 5.** Stabilizing Mutual Exclusion using Group threading. Grp t(s): Total time spent in Group computation in seconds. Syn t(s): Total synthesis time in seconds. Mem (MB): Memory usage in MB.
Algorithm transitions. Notice that specifically, these constraints identify the transitions that the synthesized algorithm is not allowed to use as recovery transitions. Unlike the other three 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 4.1 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:

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

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.

\[ \begin{align*}
(R1) \quad & (U.j > U.(j+1)) \land (L.j > U.(j+1) + 1) \land (U.j + 1 = L.(j-1)) \\
& \quad \rightarrow U.j : = L.(j-1), L.j : = U.(j+1) + 1; \\
(R2) \quad & (U.j > U.(j+1) + 1) \land (L.j > U.(j+1) + 1) \\
& \quad \rightarrow L.j : = U.(j+1) + 1; \\
\end{align*} \]

Fig. 6 shows the results of synthesizing the data dissemination protocol with various numbers of processes. One can notice that most of the total synthesis time was spent on adding recovery, while a smaller 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).

Fig. 7 shows the results of synthesizing the data dissemination protocol with various numbers of processes by partitioning the constraints among available threads. Note that, in the case of the data dissemination problem, there were only 5 constraints to satisfy. Hence, when the synthesis is launched with 8 threads, we are only utilizing 5 of them. As can be seen from Fig. 7 if the number of constraints is not large enough then the speedup gained from portioning the constraints is limited.

Fig. 8 shows the results of synthesizing the data dissemination protocol with various numbers of processes by exploiting the distributed nature of this program.

6.3. Case study 3: 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
The first disjunction of (S1) states that constraints. Faults. We now consider the faults that change the values of processes organized in a lineartopology.

Fault-intolerant program. The fault-intolerant program in this case study is the diffusing computation program from [2]. 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:

\[ DC1 :: (c.root = green) \rightarrow c.root := red, sn.root := \neg sn.root; \]
\[ DC2 :: c.j = green \land c.(P.j) = red \land sn.j \neq sn.(P.j) \rightarrow c.j, sn.j = c.(P.j), sn.(P.j); \]
\[ DC3 :: (c.j = red) \land (\forall k : P.k = j \Rightarrow (c.k = green \land sn.j = sn.k)) \rightarrow c.j := green; \]

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.

\[ (S1) \forall j : (c.j = green \land c.(P.j) = red) \lor (c.j = c.(P.j) \land sn.j = sn.(P.j)). \]

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:

\[ (F1) \text{true} \rightarrow c.j := \text{red} \mid \text{green}; \]
\[ (F2) \text{true} \rightarrow sn.j := \text{true} \mid \text{false}; \]

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:

\[ (R1) \text{(c.j = red)} \land (sn.j \neq sn.(P.j)) \rightarrow c.j := \text{green}, sn.j := sn.(P.j); \]
\[ (R2) \text{(c.(P.j) = green)} \land (c.j = \text{red}) \rightarrow c.j := \text{green}; \]
\[ (R3) \text{(c.(P.j) = c.j)} \land (sn.j \neq sn.(P.j)) \rightarrow sn.j := sn.(P.j); \]
\[ (R4) \text{(c.(P.j) = \text{red})} \land \text{(c.j = \text{red})} \land (sn.j \neq sn.(P.j)) \rightarrow c.j := \text{green}; \]

Fig. 9(a) shows the results for synthesizing a stabilizing diffusing computation program with various numbers of processes organized in a lineartopology. Fig. 9(b) shows the result where the processes are arranged in a binary tree. Note that, similar to the first case study, the complexity of the tree topology is higher than the linear topology. Moreover,
the time needed to run the experiment with 34 processes or more was more than 1 hour and the run was terminated. Fig. 10 shows the results of synthesizing the diffusing computation program with various numbers of processes by exploiting the distributed nature of this program. Fig. 11 shows the results of synthesizing the diffusing computation program with various numbers of processes by partitioning the constraints among available threads.

### 6.4. Case study 4: stabilizing distance based spanning tree

In this case study, we demonstrate another use of our algorithm in the context of synthesizing stabilizing fault-tolerant programs from their specification. We present a simple algorithm for constructing a spanning tree over a group of processes connected in grid topology. This system consists of $m \times n$ processes connected in grid topology. All processes run the same program except the root, $P_{0,0}$. Each process, $P_{j,k}$, has a variable, $d_{j,k}$, which identifies the distance between $P_{j,k}$ and the root process $P_{0,0}$. Each process continuously maintains the value for the shortest distance, $d$, between the process and the root. A process can only communicate with its immediate neighbors. More specifically, a process $P_{j,k}$ can only read its own $d$ value and the $d$ value of the processes $P_{j-1,k}$ and $P_{j,k-1}$. Furthermore, a process can only write its own $d$ value. In this case, the only initialized and fixed $d$ value is the one of the process $P_{0,0}$. The rest of the $d$ values should be computed dynamically. Thus, let $d_{j,k} = x$ then either $d_{j-1,k} = x - 1$ or $d_{j,k-1} = x - 1$. Our goal in this case study is to automatically generate the stabilizing fault-tolerant program that satisfies the above requirements.

We note that although we are denoting these processes as $d_{j,k}$ during synthesis, the processes do not know the actual values of $j$ and $k$. They only know identities of the neighbors closer to the root. It is possible to modify the synthesis algorithm to use the approach in [19] where inversion factor is used to deal with cases when neighbors closer to the fail. However, this issue is out of the scope of this paper.

**Constraints.** In this case, the constraints describe the conditions imposed on the processes to maintain correct values for the distance variable $d$. Clearly, the value of the distance variable of the root is zero. The constraints for the rest of the processes ensure that each process maintains the shortest distance to the root. Hence, the constraints are formally described in $S_1$ and $S_2$ as follows.

1. $d_{0,0} = 0$
2. $(d_{j,k} = d_{j-1,k} + 1) \lor (d_{j,k} = d_{j,k-1} + 1)$.

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

$$ (F) \quad true \rightarrow (d_{j,k} := \text{any arbitrary value from its domain}); $$

**Fault-tolerant program.** To automatically synthesize a stabilizing fault-tolerant program, we used the synthesis algorithm as follows. The fault-intolerant program for each process is specified with no actions; the faults are specified by the fault action $F$; and the constraints are from $S_1$ and $S_2$. We specified these constraints in the following order: first, we specified

<table>
<thead>
<tr>
<th>No. of Processes</th>
<th>reachable states</th>
<th>Sequential</th>
<th>2 threads</th>
<th>4 threads</th>
<th>8 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Grp t(s)</td>
<td>Syn t(s)</td>
<td>Mem (MB)</td>
<td>Grp t(s)</td>
</tr>
<tr>
<td>50</td>
<td>$10^{30}$</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>$10^{60}$</td>
<td>31</td>
<td>32</td>
<td>14</td>
<td>24</td>
</tr>
<tr>
<td>150</td>
<td>$10^{90}$</td>
<td>110</td>
<td>113</td>
<td>15</td>
<td>74</td>
</tr>
<tr>
<td>200</td>
<td>$10^{120}$</td>
<td>275</td>
<td>282</td>
<td>15</td>
<td>177</td>
</tr>
</tbody>
</table>
constraints $S_1$ for the root, and then we specified $S_2$ for the root children, then its grandchildren and so on. The recovery actions computed by the synthesis algorithm are as follows:

$$R_1:: \neg (d_{0, 0} = 0) \implies d_{0, 0} := 0;$$

$$R_2:: \neg ((d_{j, k} = d_{j-1, k} + 1) \lor (d_{j, k} = d_{j, k-1} + 1)) \implies d_{j, k} := d_{j-1, k} + 1 \lor d_{j, k} = d_{j, k-1} + 1;$$

**Analysis of experimental results.** Fig. 12 shows the results of synthesizing a fault-tolerant distance based spanning tree with various numbers of processes. In this experiment, we arranged the process in $3 \times n$ grid, where $3 \leq n \leq 12$. One can notice that amount of time required to synthesis a stabilizing program for various numbers of processes is reasonable and proportional to the number of processes used in the grid. The running time for 42 processors, however, was more than 1 h.

**Memory usage.** Notice that the amount of memory needed during synthesis is proportional to the number of threads being used. It is approximately the amount of memory used by the sequential algorithm multiplied by the number of cores being used. Clearly, this is expected since for every thread used, we create a new MDD package. We argue that using extra memory to gain a speedup is acceptable, since in the automated synthesis, time complexity is a far more serious barrier than space complexity.

### 7. Choosing ordering among constraints

To apply Theorem 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 and stabilizing fault-tolerant program. Hence, choosing the “right” order does not appear to be very crucial. Also, [2] identifies several heuristics that can assist in identifying the right order among constraints.

One possible approach is to consider different combinations as part of the synthesis algorithm. With such an approach, $O(n^2)$ combinations suffice. 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 $O(n^2)$. 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.

Another approach is to allow the synthesis algorithm to choose a random ordering for satisfying the constraints. If the synthesis algorithm fails to find a solution using a given constraints ordering, then it chooses a different random order. The synthesis algorithm keeps trying different random ordering for the constraints until it finds a solution or it exhausts all possible combinations. Although, the worst case complexity of this approach is $O(n!)$, we found that the average case complexity in our case studies was close to $O(1)$. Hence, we do not present results related to the $O(n^2)$ solutions.

We implemented this approach. We found that depending on the program being synthesized the time required to complete the synthesis may vary significantly. More specifically, in the case of the Stabilizing Mutual Exclusion from Section 6.1, the order of the constraints is almost always irrelevant and the synthesis algorithm found a solution using any order it tried. Fig. 13, shows the results of 10 experiments, in each experiment the synthesis algorithm randomly chose an order for the constraints and tried to synthesize using that order. In all cases the synthesis completed successfully for any order and in the first attempted ordering. The time needed to complete the synthesis was almost identical to that of the case where the constraints were manually ordered (c.f. Fig. 3(a)). However, this was not always the case. For example, Fig. 14 shows the results of synthesizing the Stabilizing Diffusing Computation from Section 6.3. In this case, the order in which

<table>
<thead>
<tr>
<th>No. of Processes</th>
<th>Space reachable states (MB)</th>
<th>Memory constraint satisfaction total (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Recovery</td>
</tr>
<tr>
<td>12</td>
<td>$10^{10}$</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>$10^{12}$</td>
<td>6</td>
</tr>
<tr>
<td>18</td>
<td>$10^{14}$</td>
<td>15</td>
</tr>
<tr>
<td>21</td>
<td>$10^{16}$</td>
<td>14</td>
</tr>
<tr>
<td>24</td>
<td>$10^{18}$</td>
<td>15</td>
</tr>
<tr>
<td>27</td>
<td>$10^{20}$</td>
<td>17</td>
</tr>
<tr>
<td>30</td>
<td>$10^{22}$</td>
<td>23</td>
</tr>
<tr>
<td>33</td>
<td>$10^{24}$</td>
<td>31</td>
</tr>
<tr>
<td>36</td>
<td>$10^{26}$</td>
<td>41</td>
</tr>
<tr>
<td>39</td>
<td>$10^{28}$</td>
<td>55</td>
</tr>
</tbody>
</table>

Fig. 12. Stabilizing distance based spanning tree using grid topology.
the synthesis algorithm satisfies the constraints is significantly important. More specifically, the synthesis algorithm has to try different orderings (on average 3–4 times) before it successfully synthesized the stabilizing fault-tolerant program. Moreover, the time required to complete the synthesis, in this case, was higher than that when the constraints were manually ordered (c.f. Fig. 9(a)).

8. 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 to 4 (i.e., from 7 to 15 processes), the synthesis time increased from 5 to 72 s. This is expected since the state space increases from $10^5$ 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 and binary tree topology.

**Linear topology.** Consider the case where the system is as shown in Fig. 15(a). 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 Fig. 15(a) be $C_A$, $C_B$, $C_D$. Furthermore, let the added recovery actions be $\text{rec}_A$, $\text{rec}_B$, $\text{rec}_D$.

**Theorem 2.** If $(\text{rec}_A \lor \text{rec}_B \lor \text{rec}_D)$ form the recovery actions for the program in Fig. 15(a) then $(\text{rec}_A \lor \text{rec}_B \lor \text{rec}_C \lor \text{rec}_D)$ form the recovery actions for the program in Fig. 15(b) where $\text{rec}_C$ is obtained by replacing $B$ by $C$ and (then) replacing $A$ by $B$ from $\text{rec}_D$ and $\text{rec}_D'$ is obtained by replacing $B$ by $C$ in $\text{rec}_D$.

**Proof.** Based on the order of constraints and the rules used in constructing recovery actions, constraints $C_A$ and $C_B$ will be satisfied even for the network in Fig. 15(b). Since recovery actions do not execute after the corresponding constraint is satisfied, eventually, the recovery actions in $\text{rec}_C'$ and $\text{rec}_D'$ (and the fault-intolerant program) will execute. Since $C_D$ only depends on the variables of $D$ and its predecessor and they correct a predicate involving $D$ and its predecessor, if actions in $\text{rec}_D'$ execute then they will correct $C_D$. Moreover, if actions in $\text{rec}_D'$ execute then they terminate (after satisfying $C_D$). Hence, given the fairness assumption, actions in $\text{rec}_C'$ will execute. Observe that $\text{rec}_C'$ is obtained from $\text{rec}_B$ by replacing $B$ by $C$ and $A$ by $B$. Furthermore, based on the definitions of the constraints, $C_C$ is obtained from $C_B$ by replacing $B$ by $C$ and $A$ by $B$. Thus, $\text{rec}_C'$ will correct $C_C$. Note that $\text{rec}_C'$ can violate $C_D'$. However, it will be corrected again by $\text{rec}_D'$. □
Binary tree topology. Consider the case where the system is as shown in Fig. 16(a). 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 \( j \), \( j \)'s parent (if it exists) and \( j \)'s children (if they exist). Furthermore, assume that constraints for intermediate processes (respectively the leafs) are identical except for the renaming variables. Let the order of predicates added for system in Fig. 16(a) be \( C_A, C_B, C_C, C_D, C_E, C_I, \) and \( C_O \). Furthermore, let the added recovery actions be \( \text{rec}_A, \text{rec}_B, \text{rec}_C, \text{rec}_D, \text{rec}_E, \text{rec}_I, \) and \( \text{rec}_O \).

**Theorem 3.** If \((\text{rec}_A \lor \text{rec}_B \lor \text{rec}_C \lor \text{rec}_D \lor \text{rec}_E \lor \text{rec}_H \lor \text{rec}_I)\) form the recovery actions for the program in Fig. 16(a) then \((\text{rec}_A \lor \text{rec}_B \lor \text{rec}_C \lor \text{rec}_D \lor \text{rec}_E \lor \text{rec}_H \lor \text{rec}_I \lor \text{rec}_J \lor \text{rec}_K \lor \text{rec}_L \lor \text{rec}_M \lor \text{rec}_N \lor \text{rec}_O)\) form the recovery actions for the program in Fig. 16(b) where:

1. \( \text{rec}_B \) is used generate \( \text{rec}'_B \) by:
   - (a) replacing \( D \) by \( H \) and \( E \) by \( I \),
   - (b) replacing \( B \) by \( D \), and (then)
   - (c) replacing \( A \) by \( B \),
2. \( \text{rec}'_H \) is obtained by replacing \( D \) by \( H \) and (then) by replacing \( B \) by \( D \) in \( \text{rec}_D \).
3. \( \text{rec}'_I \) is obtained by replacing \( D \) by \( I \) and (then) by replacing \( B \) by \( D \) in \( \text{rec}_D \).

\( \text{rec}'_E, \text{rec}'_C, \text{rec}'_J, \text{rec}'_K, \text{rec}'_L, \text{rec}'_M, \text{rec}'_N, \) and \( \text{rec}'_O \) are generated by using steps similar to the Steps 1–3.

**Proof.** The proof of Theorem 3 is similar to that of Theorem 2. \( \square \)

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. Clearly, constructing and verifying the recovery action which satisfy the conditions of Theorems 2 and 3 is syntactical and requires a minimal amount of time to complete.

9. Related work

Our approach for adding nonmasking and stabilizing fault-tolerance is based on satisfying constraints that should be \textit{True} in legitimate states. An orthogonal approach is to utilize primitives such as distributed reset [20] 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 [20,21]. Our approach can be utilized to design the distributed reset protocol itself.

In masking fault-tolerance, when faults occur, the program cannot violate the safety properties during recovery therefore this approach will not be able to synthesize nonmasking fault-tolerant programs where safety can be violated during recovery. Furthermore, while our algorithm accounts for weak fairness among program actions and allows for recovery actions to be added under this assumption, the heuristic based approach does not account for fairness assumptions. In fact, heuristic based approaches will not add the recovery actions required to make the program fault-tolerant.

Katz and Perry [20] proposed an algorithm to extend an arbitrary asynchronous distributed message-passing system into a self-stabilizing system. They also gave a formal definition of the self-stabilizing extension of a non-stabilizing program and they defined the set of properties that must be maintained by the new extension. Their algorithm superimposes a control program on the original non-stabilizing program. The control program repeatedly takes a global snapshot and then checks if the snapshot indicates an illegal state. If an illegal state is found the control program resets the memory of each process to a legal default state.

Arora [2] proposed a manual approach to design nonmasking fault-tolerant programs. In this approach, a program is intended to satisfy a set of constraints during normal operation (i.e. no faults). Program actions are categorized into “closure” actions and “convergence” actions. When faults occur and violate one or more of the program constraints convergence actions are responsible for correcting program behavior and reestablishing those constraints again. This method, however, does not address the issue of automated addition of nonmasking fault-tolerance to existing fault-intolerant programs.

![Fig. 16. Complexity and hierarchy for the binary tree topology.](image-url)
The model checking community has studied the use of parallelization in expediting the symbolic reachability from different perspectives. In particular, the saturation-based generation of state space was subject to different parallelization approaches [22–24] in an effort to speedup the reachability analysis. However, in [23], the authors show that in order to gain speedup in saturation-based parallel symbolic verification, one has to pay a penalty for memory usage of up to 10 times that of the sequential algorithm. Other efforts range from simple approaches that essentially implement BDDs as two-tiered hash tables [25,26], to sophisticated approaches relying on slicing BDDs [27] and techniques for workstealing [28]. However, the resulting implementations show only limited speedup. The use of parallel group computation has also been shown to be effective for resolving deadlock states in the synthesis of masking fault-tolerance programs [29].

Ebnesasir [9] presents a divide-and-conquer method for synthesizing failsafe fault-tolerant distributed programs. In failsafe fault-tolerance, the program is not required to recover to its legitimate states. Thus, a respective synthesis algorithm does not need to discover any recovery actions.

10. Conclusion

In this paper, we focussed on the automated addition of nonmasking and stabilizing fault-tolerance to hierarchical distributed 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 were 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 $C_0, C_1 \ldots C_{i-1}$ 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 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.

Also, we focussed on improving the synthesis of fault-tolerant programs from their fault-intolerant version. We showed that the use of multi-core technology to parallelize the synthesis algorithm reduces the synthesis time substantially. We parallelized constraint satisfaction by: (1) partitioning the constraints and (2) utilizing the nature of distributed programs. We showed that parallelism provides a substantial benefit in reducing the time needed in synthesis.

We illustrated our approach with four case studies: stabilizing mutual exclusion, stabilizing diffusing computation, a data dissemination problem for sensor networks, and the stabilizing distance based spanning tree. The complexity analysis demonstrated that automated synthesis in these case studies was feasible and achieved in a reasonable time speedup in all case studies.

Furthermore, since our work is structured on constraint-based (manual) design of nonmasking and stabilizing fault-tolerance from [2] that has been found to be useful in deriving several protocols manually (e.g. [30,31,21]), 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 [17] 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 [11].

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 [8,7]. One of the future works in this area is to identify how the approach from [7] can be implemented using MDDs 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.

References