On the Complexity of Adding Convergence

Alex Klinkhamer, Ali Ebnenasir

On the Complexity of Adding Convergence *

Alex Klinkhamer and Ali Ebnenasir

Department of Computer Science
Michigan Technological University
Houghton, MI 49931, U.S.A.
{apklinkh,aebnenas}@mtu.edu

Abstract. This paper investigates the complexity of designing Self-Stabilizing (SS) distributed programs, where an SS program meets two properties, namely closure and convergence. *Convergence* requires that, from *any* state, the computations of an SS program reach a set of legitimate states (a.k.a. *invariant*). Upon reaching a legitimate state, the computations of an SS program remain in the set of legitimate states as long as no faults occur; i.e., *Closure*. We illustrate that, in general, the problem of augmenting a distributed program with convergence, i.e., *adding convergence*, is NP-complete (in the size of its state space). An implication of our NP-completeness result is the hardness of adding non-masking fault tolerance to distributed programs, which has been an open problem for the past decade.

Keywords: Self-Stabilization, Convergence, NP-Completeness

1 Introduction

Today's distributed programs are subject to a variety of transient faults due to their inherent complexity, human errors and environmental factors, where transient faults perturb program state without causing any permanent damage (e.g., bad initialization, loss of coordination, soft errors). Distributed applications should guarantee service availability even in the presence of faults. However, providing global recovery in distributed programs is difficult in part due to (1) no central point of control/administration; (2) lack of knowledge about the global state of the program by program processes/components, and (3) the need for global recovery using only the local actions of processes. To design programs that recover from any arbitrary configuration/state without human intervention, Dijkstra [1] proposed self-stabilization as a property of distributed programs. A Self-Stabilizing (SS) program meets two requirements, namely closure and convergence. *Convergence* requires that, from *any* state, the computations of an SS program reach a set of legitimate states (a.k.a. program *invariant*). Upon reaching an invariant state, the computations of an SS program remain in its

* This work was sponsored in part by the NSF grant CCF-1116546 and a grant from Michigan Technological University.

invariant as long as no faults occur; i.e., *Closure*. Indeed, self-stabilization is a special case of *nonmasking* fault tolerance [2,3], where instead of providing recovery from any state, designers identify a subset of the state space from where recovery to invariant can be provided. While there are several approaches in the literature for the design of SS algorithms for specific problems, we are not aware of a general case complexity analysis of designing SS programs.

Several researchers have investigated the problem of adding nonmasking fault tolerance to programs [1,2,4,3,5,6]. For instance, Liu and Joseph [4] present a method for the transformation of an intolerant program to a fault-tolerant version thereof by going through a set of refinement steps. Arora and Gouda [2,3] use the notions of closure and convergence to define three levels of fault tolerance based on the extent to which safety and liveness specifications [7] are satisfied in the presence of faults. In their setting, a *failsafe* fault-tolerant program ensures its safety at all times even if faults occur, whereas, in the presence of faults, a *nonmasking* program provides recovery to its invariant; no guarantees on meeting safety during recovery. A *masking* fault-tolerant program is both failsafe and nonmasking. Arora *et al.* [5] design nonmasking fault tolerance by creating a dependency graph of the local constraints of program processes, and by illustrating how these constraints should be satisfied so global recovery is achieved. Kulkarni and Arora [6] demonstrate that adding failsafe/nonmasking/masking fault tolerance to high atomicity programs can be done in polynomial-time in program's state space, where a *high atomicity* program can read/write all program variables in an atomic step. Nonetheless, they illustrate that adding masking fault tolerance to *low atomicity* programs – where processes have read/write restrictions with respect to variables of other processes – is NP-complete (in the size of the state space).[1] Moreover, Kulkarni and Ebnenasir [8] show that adding failsafe fault tolerance to low atomicity programs is also an NP-complete problem. Nonetheless, while adding nonmasking fault tolerance is known to be in NP, no polynomial-time algorithms are known for efficient design of nonmasking fault tolerance for low atomicity programs; neither has there been a proof of NP-hardness!

In this paper, we illustrate that adding nonmasking fault tolerance to low atomicity programs is NP-complete (in the size of the state space). Our hardness proof is based on a reduction from the 3-SAT problem [9] to the problem of adding convergence to non-stabilizing programs. Since adding convergence is a special case of adding nonmasking fault tolerance, it follows that, in general, it is unlikely that adding nonmasking fault tolerance to low atomicity programs can be done efficiently (unless $P = NP$). The significance of our NP-hardness proof is multi-fold. First, our proof provides a solution for a problem that has been open for more than a decade. Second, illustrating the NP-completeness of adding convergence is particularly important since the proof requires the construction of the entire state space of the instance of the problem of adding convergence, yet such a mapping should be polynomial in the size of the instance of the source NP-complete problem (in our case 3-SAT). Devising such a reduction has been

[1] Low atomicity programs enable the modeling of distributed programs.

another open problem in the literature. Third, our proof illustrates that even if we have a process in a program that can atomically read the global state of the program and can update its own local state, the addition of recovery still remains a hard problem. Fourth, the presented hardness proof lays the foundation for the design of a new family of synthesis algorithms inspired by the DPLL algorithm [10], which we are currently investigating. We conjecture that such synthesis algorithms will be more efficient than existing methods for SAT-based synthesis of fault tolerance [11] where one formulates the sub-problems of adding fault tolerance in terms of CNF formulas and invokes off-the-shelf SAT solvers.

Organization. Section 2 presents the basic concepts of programs, faults and fault tolerance. Section 3 formally states the problem of adding nonmasking fault tolerance and convergence. Section 4 illustrates that adding convergence in particular and adding nonmasking fault tolerance in general are NP-complete. Section 5 discusses related work. Finally, Section 6 makes concluding remarks and discusses future work.

2 Preliminaries

In this section, we present the formal definitions of programs, faults, fault tolerance and self-stabilization, and our distribution model (adapted from [6]). Programs are defined in terms of their set of variables, their transitions and their processes. The definitions of fault tolerance and self-stabilization is adapted from [1,3,12,13]. For ease of presentation, we use a simplified version of Dijkstra's token ring protocol [1] as a running example.

Programs as (non-deterministic) finite-state machines. A program in our setting is a representation of any system that can be captured by a (finite-state) non-deterministic state machine (e.g., network protocols). Formally, a *program* p is a tuple $\langle V_p, \delta_p, \Pi_p, T_p \rangle$ of a finite set V_p of variables, a set δ_p of transitions, a finite set Π_p of k processes, where $k \geq 1$, and a topology T_p. Each variable $v_i \in V_p$, for $i \in \mathbb{N}_m$ where $\mathbb{N}_m = \{0, 1, \cdots, m-1\}$ and $m > 0$, has a finite non-empty domain D_i. A *state* s of p is a valuation $\langle d_0, d_1, \cdots, d_{m-1} \rangle$ of variables $\langle v_0, v_1, \cdots, v_{m-1} \rangle$, where $d_i \in D_i$. A *transition* t is an ordered pair of states, denoted (s_0, s_1), where s_0 is the source and s_1 is the target/destination state of t. A *deadlock state* is a state with no outgoing transitions. For a variable v and a state s, $v(s)$ denotes the value of v in s. The *state space* of p, denoted S_p, is the set of all possible states of p, and $|S_p|$ denotes the size of S_p. A *state predicate* is any subset of S_p specified as a Boolean expression over V_p. We say a state predicate X *holds in a state* s (respectively, $s \in X$) *if and only if (iff)* X evaluates to true at s.

Read/Write model. We adopt a shared memory model [14] since reasoning in a shared memory setting is easier, and several (correctness-preserving) transformations [15,16] exist for the refinement of shared memory fault-tolerant programs to their message-passing versions. We model the topological constraints (denoted T_p) of a program p by a set of read and write restrictions imposed on variables that identify the locality of each process. Specifically, we consider a subset of variables in V_p that a process P_j ($j \in \mathbb{N}_k$) can write, denoted w_j, and a subset of variables that P_j is allowed to read, denoted r_j. We assume that for

each process P_j, $w_j \subseteq r_j$; i.e., if a process can write a variable, then it can also read that variable.

Impact of read/write restrictions. Every transition of a process P_j belongs to a *group* of transitions due to the inability of P_j in reading variables that are not in r_j. Consider two processes P_0 and P_1 each having a Boolean variable that is not readable for the other process. That is, P_0 (respectively, P_1) can read and write x_0 (respectively, x_1), but cannot read x_1 (respectively, x_0). Let $\langle x_0, x_1 \rangle$ denote a state of this program. Now, if P_0 writes x_0 in a transition $(\langle 0,0\rangle, \langle 1,0\rangle)$, then P_0 has to consider the possibility of x_1 being 1 when it updates x_0 from 0 to 1. As such, executing an action in which the value of x_0 is changed from 0 to 1 is captured by the fact that a group of two transitions $(\langle 0,0\rangle, \langle 1,0\rangle)$ and $(\langle 0,1\rangle, \langle 1,1\rangle)$ is included in P_0. In general, a transition is included in the set of transitions of a process *iff* its associated group of transitions is included. Formally, any two transitions (s_0, s_1) and (s'_0, s'_1) in a group of transitions formed due to the read restrictions of a process P_j, denoted r_j, meet the following constraints: $\forall v : v \in r_j : (v(s_0) = v(s'_0)) \wedge (v(s_1) = v(s'_1))$ and $\forall v : v \notin r_j : (v(s_0) = v(s_1)) \wedge (v(s'_0) = v(s'_1))$.

Due to read/write restrictions, a *process* P_j ($j \in \mathbb{N}_k$) includes a set of transition groups $P_j = \{g_{j0}, g_{j1}, \cdots, g_{j(max-1)}\}$ created due to read restrictions r_j, where $max \geq 1$. Due to write restrictions w_j, no transition group g_{ji} ($i \in \mathbb{N}_{max}$) can have a transition (s_0, s_1) that updates a variable $v \notin w_j$. Thus, the set of transitions δ_p of a program p is equal to the union of the transition groups of its processes; i.e., $\delta_p = \cup_{j=0}^{k-1} P_j$. (It is known that the total number of groups is polynomial in $|S_p|$ [6]). We use p and δ_p interchangeably.

To simplify the specification of δ_p for designers, we use Dijkstra's guarded commands language [17] as a shorthand for representing the set of program transitions. A guarded command (action) is of the form $grd \rightarrow stmt$, and includes a set of transitions (s_0, s_1) such that the predicate grd holds in s_0 and the atomic execution of the statement $stmt$ results in state s_1. An action $grd \rightarrow stmt$ is *enabled* in a state s iff grd holds at s. A process $P_j \in \Pi_p$ is *enabled* in s iff there exists an action of P_j that is enabled at s.

Example: Token Ring (TR). The Token Ring (TR) program (adapted from [1]) includes three processes $\{P_0, P_1, P_2\}$ each with an integer variable x_j, where $j \in \mathbb{N}_3$, with a domain $\{0, 1, 2\}$. The process P_0 has the following action (addition and subtraction are in modulo 3):

$$A_0 : \quad (x_0 = x_2) \qquad \longrightarrow \qquad x_0 := x_2 + 1$$

When the values of x_0 and x_2 are equal, P_0 increments x_0 by one. We use the following parametric action to represent the actions of processes P_j, for $1 \leq j \leq 2$:

$$A_j : \quad (x_j \neq x_{(j-1)}) \qquad \longrightarrow \qquad x_j := x_{(j-1)}$$

Each process P_j copies x_{j-1} only if $x_j \neq x_{j-1}$, where $j = 1, 2$. By definition, process P_j *has a token* iff $x_j \neq x_{j-1}$. Process P_0 *has a token* iff $x_0 = x_2$. We define a state predicate I_{TR} that captures the set of states in which only one token exists, where I_{TR} is

$$((x_0 = x_1) \wedge (x_1 = x_2)) \ \vee \ ((x_1 \neq x_0) \wedge (x_1 = x_2)) \ \vee \ ((x_0 = x_1) \wedge (x_1 \neq x_2))$$

Each process P_j $(1 \leq j \leq 2)$ is allowed to read variables x_{j-1} and x_j, but can write only x_j. Process P_0 is permitted to read x_2 and x_0 and can write only x_0. Thus, since a process P_j is unable to read one variable (with a domain of three values), each group associated with an action A_j includes three transitions. For a TR protocol with n processes and with n values in the domain of each variable x_j, each group includes n^{n-2} transitions. $\lhd$

Computations. Intuitively, a computation of a program $p = \langle V_p, \delta_p, \Pi_p, T_p \rangle$ is an *interleaving* of its actions. Formally, a *computation* of p is a sequence $\sigma = \langle s_0, s_1, \cdots \rangle$ of states that satisfies the following conditions: (1) for each transition (s_i, s_{i+1}) in σ, where $i \geq 0$, there exists an action $grd \rightarrow stmt$ in some process $P_j \in \Pi_p$ such that grd holds at s_i and the execution of $stmt$ at s_i yields s_{i+1}, and (2) σ is *maximal* in that either σ is infinite or if it is finite, then σ reaches a state s_f where no action is enabled. A *computation prefix* of a program p is a *finite* sequence $\sigma = \langle s_0, s_1, \cdots, s_z \rangle$ of states, where $z > 0$, such that each transition (s_i, s_{i+1}) in σ $(i \in \mathbb{N}_z)$ belongs to some action $grd \rightarrow stmt$ in some process $P_j \in \Pi_p$. The *projection* of a program p on a non-empty state predicate X, denoted as $\delta_p|X$, is the program $\langle V_p, \{(s_0, s_1) : (s_0, s_1) \in \delta_p \wedge s_0, s_1 \in X\}, \Pi_p, T_p \rangle$. In other words, $\delta_p|X$ consists of transitions of p that start in X and end in X.

Closure and invariant. A state predicate X is *closed in an action* $grd \rightarrow stmt$ *iff* executing $stmt$ from any state $s \in (X \wedge grd)$ results in a state in X. We say a state predicate X is *closed in a program* p *iff* X is closed in every action of p. In other words, *closure* [13] requires that every computation of p starting in X remains in X. We say a state predicate I is an *invariant* of p *iff* I is closed in p.

TR Example. Starting from a state in the predicate I_{TR}, the TR protocol generates an infinite sequence of states, where all reached states belong to I_{TR}. $\lhd$

Faults. Intuitively, we capture the impact of faults on a program as state perturbations. Formally, a class of *faults* f for a program $p = \langle V_p, \delta_p, \Pi_p, T_p \rangle$ is a subset of $S_p \times S_p$. We use $p[]f$ to denote the transitions obtained by taking the union of the transitions in δ_p and the transitions in f. We say that a state predicate T is an f-span (read as *fault-span*) of p from a state predicate I *iff* the following two conditions are satisfied: (1) $I \subseteq T$, and (2) T is closed in $p[]f$. Observe that for all computations of p that start in I, the state predicate T is a superset of I in S_p up to which the state of p may be perturbed by the occurrence of f transitions. We say that a sequence of states, $\sigma = \langle s_0, s_1, ... \rangle$ is a *computation of* p *in the presence of* f *iff* the following conditions are satisfied: (1) $\forall j : j > 0 : (s_{j-1}, s_j) \in (p[]f)$; (2) if σ is finite and terminates in state s_l, then there is no state s such that $(s_l, s) \in \delta_p$, and (3) $\exists n : n \geq 0 : (\forall j : j > n : (s_{j-1}, s_j) \in \delta_p)$. The first requirement captures that in each step, either a program transition or a fault transition is executed. The second requirement captures that faults do not have to execute. That is, if the only transition that starts from s_l is a fault transition (s_l, s_f) then as far as the program is concerned, s_l is still a deadlock state because the program does not have control over the execution of (s_l, s_f); i.e., (s_l, s_f) may or may not be executed. Finally, the third requirement captures that the number of fault occurrences in a computation is finite. This requirement is the same as

that made in previous work (e.g., [1,18,3,19]) to ensure that eventually recovery can occur. The same way we use guarded commands to represent program transitions, we use them to specify fault transitions. That is, the impact of faults can be captured as a set of actions that update program variables.

TR Example. The TR protocol is subject to transient faults that can perturb its state to an arbitrary state. For instance, the following action captures the impact of faults on x_0, where | denotes non-deterministic assignment of values to x_0:

$$F_0: \quad \text{true} \qquad \longrightarrow \qquad x_0 := 0|1|2;$$

The impact of faults on x_1 and x_2 are captured with two actions F_1 and F_2 symmetric to F_0. ◁

Nonmasking fault-tolerance and self-stabilization. Let I be a state predicate closed in a program p and f be a class of faults. We say that p is nonmasking f-tolerant from I *iff* there exists an f-span of p from I, denoted T, such that T *converges* to I in p. That is, from any state $s_0 \in T$, *every* computation of p that starts in s_0 reaches a state where I holds. We say that p is *self-stabilizing* from I *iff* p is nonmasking f-tolerant from I, where $T = true$. That is, the f-span of p is equal to S_p, and convergence to I is guaranteed from any state in S_p. Notice that, to design recovery, one has to ensure that no deadlock states exist in $T-I$, and no non-progress cycles exist in $\delta_p \mid (T-I)$. A *non-progress cycle* (a.k.a. *livelock*) in $\delta_p \mid (T-I)$ is a sequence of states $\sigma = \langle s_0, s_1, \cdots, s_m, s_0 \rangle$, where $m \geq 0$, $(s_i, s_{i\oplus 1}) \in \delta_p$ and $s_i \in (T-I)$, for $i \in \mathbb{N}_{m+1}$ and $\oplus$ denotes addition modulo $m+1$.

Proposition 1. *A program p is nonmasking f-tolerant from I with a f-span T iff there are no deadlock states in $T-I$ and no non-progress cycles in $\delta_p \mid (T-I)$.*

Note. In this paper, we analyze the complexity of adding convergence under the assumption of no fairness.

3 Problem Statement

In this section, we represent the problem of adding nonmasking fault-tolerance from [6]. Consider a fault-intolerant program $p = \langle V_p, \delta_p, \Pi_p, T_p \rangle$, a class of faults f, and a state predicate I, where I is closed in p. Our objective is to generate a revised version of p, denoted p', such that p' is nonmasking f-tolerant from an invariant I'. To separate fault tolerance from functional concerns, we would like to preserve the behaviors of p in the absence of f during the addition of fault tolerance. For this reason, during the synthesis of p' from p, no states (respectively, transitions) are added to I (respectively, $\delta_p|I$). Thus, we have $I' \subseteq I$ and $p'|I' \subseteq p|I'$. This way, if p' starts in I' in the absence of faults, then p' will preserve the correctness of p; i.e., the added recovery does not interfere with normal functionalities of p in the absence of faults. Moreover, if p' starts in a state outside I', then only recovery to I' will be provided by p'. Thus, we formally state the problem as follows:

Problem 1. **Adding Nonmasking Fault Tolerance**

- **Input**: (1) A program $p = \langle V_p, \delta_p, \Pi_p, T_p \rangle$; (2) A class of faults f; (3) A state predicate I such that I is closed in p, and (4) topological constraints of p captured by read/write restrictions.
- **Output**: A program $p' = \langle V_{p'}, \delta_{p'}, \Pi_{p'}, T_{p'} \rangle$ and a state predicate I' such that the following constraints are met: (1) I' is non-empty and $I' \subseteq I$; (2) $\delta_{p'}|I' \subseteq \delta_p|I'$; (3) Π_p and $\Pi_{p'}$ have the same number of processes and $T_p = T_{p'}$, and (4) p' is nonmasking f-tolerant from I'. □

We state the corresponding decision problem as follows:

Problem 2. **Decision Problem of Adding Nonmasking Fault Tolerance**

- INSTANCE: (1) A program $p = \langle V_p, \delta_p, \Pi_p, T_p \rangle$; (2) A class of faults f; (3) A state predicate I such that I is closed in p, and (4) topological constraints of p captured by read/write restrictions.
- QUESTION: Does there exist a program $p' = \langle V_{p'}, \delta_{p'}, \Pi_{p'}, T_{p'} \rangle$ and a state predicate I' such that the constraints of Problem 1 are met? □

A special case of Problem 2 is where f denotes a class of transient faults, $I = I'$, $\delta_{p'}|I' = \delta_p|I'$, and p' is self-stabilizing from I.

Problem 3. **Decision Problem of Adding Convergence**

- INSTANCE: (1) A program $p = \langle V_p, \delta_p, \Pi_p, T_p \rangle$; (2) A state predicate I such that I is closed in p, and (3) topological constraints captured by read/write restrictions.
- QUESTION: Does there exist a program p_{ss} with an invariant I_{ss} such that $I = I_{ss}$, $\delta_{p_{ss}}|I_{ss} = \delta_p|I_{ss}$, and p_{ss} is self-stabilizing from I_{ss}? □

4 Hardness Results

In this section, we illustrate that adding convergence to low atomicity programs is NP-complete (in the size of the state space). This hardness result implies the hardness of general case addition of nonmasking fault tolerance to low atomicity programs (i.e., Problem 2). Specifically, we demonstrate that, for a given intolerant program p with an invariant I, adding convergence from S_p to I is an NP-hard problem. Section 4.1 presents a polynomial-time mapping from 3-SAT to an instance of Problem 3. Section 4.2 shows that the instance of 3-SAT is satisfiable *iff* a self-stabilizing version of the instance of Problem 3 exists.

Problem 4. The 3-SAT decision problem.

- INSTANCE: A set $\mathcal{V}$ of n propositional variables $(v_0, \cdots, v_{n-1})$ and k clauses $(C_0, \cdots, C_{k-1})$ over $\mathcal{V}$ such that each clause is of the form $(l_q \vee l_r \vee l_s)$, where $q, r, s \in \mathbb{N}_k$ and $\mathbb{N}_k = \{0, 1, \cdots, k-1\}$. Each l_r denotes a literal, where a literal is either $\neg v_r$ or v_r for $v_r \in \mathcal{V}$. We assume that $\neg(q = r = s)$ holds for all clauses; otherwise, the 3-SAT instance can efficiently be transformed to a formula that meets this constraint.

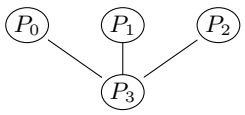
– QUESTION: Is there a satisfying truth-value assignment for the variables in $\mathcal{V}$ such that each C_i evaluates to *true*, for all $i \in \mathbb{N}_k$?

Notation. We say l_r is a *negative* (respectively, *positive*) literal *iff* it has the form $\neg v_r$ (respectively, v_r), where $v_r \in \mathcal{V}$. Consider a clause $C_i = (l_q \vee l_r \vee l_s)$. We use a binary variable b^i_j, where $i \in \mathbb{N}_k$ and $j \in \mathbb{N}_3$, to denote the sign of the first, second and the third literal in C_i. For example, if $l_q = \neg v_q, l_r = v_r$ and $l_s = \neg v_s$, then we have $b^i_0 = 0, b^i_1 = 1$ and $b^i_2 = 0$. Accordingly, for each clause C_i, we define a tuple $B^i = \langle b^i_0, b^i_1, b^i_2 \rangle$. Notice that, the binary variable b^i_j is independent from the indices of the literals in clause C_i and represents only the positive/negative form of the three literals in C_i.

4.1 Polynomial Mapping

In this section, we present a polynomial-time mapping from an instance of 3-SAT to the instance of Problem 3, denoted $p = \langle V_p, \delta_p, \Pi_p, T_p \rangle$. That is, corresponding to each propositional variable and clause, we illustrate how we construct a program p, its processes Π_p, its variables V_p, its read/write restrictions and its invariant I. We shall use this mapping in Section 4.2 to demonstrate that the instance of 3-SAT is satisfiable *iff* convergence from S_p can be added to p.

Processes, variables and read/write restrictions. We consider three processes, P_0, P_1, and P_2 in p. Each process P_j ($j \in \mathbb{N}_3$) has two variables x_j and y_j, where the domain of x_j is equal to $\mathbb{N}_n = \{0, 1, \cdots, n-1\}$ and y_j is a binary variable. (Notice that n denotes the number of propositional variables in the 3-SAT instance.) The process P_j can read both x_j and y_j, but can write only y_j. We also consider a fourth process P_3 that can read all variables and write a binary variable $sat \in \mathbb{N}_2$. The variable sat can be read by processes P_0, P_1 and P_2, but not written. That is, we have $r_j = \{x_j, y_j, sat\}$, $w_j = \{y_j\}$ for $j \in \mathbb{N}_3$, and $r_3 = \{x_0, y_0, x_1, y_1, x_2, y_2, sat\}$ and $w_3 = \{sat\}$ (see Figure 1). We also have $V_p = \{x_0, y_0, x_1, y_1, x_2, y_2, sat\}$, $\Pi_p = \{P_0, P_1, P_2, P_3\}$.



For all $i \in \mathbb{N}_3$:

$$P_i : \begin{cases} x_i \in \mathbb{N}_n & \text{(read-only)} \\ y_i \in \mathbb{N}_2 & \text{(read \& write)} \\ sat & \text{(read from } P_3\text{)} \end{cases}$$

$$P_3 : \begin{cases} sat \in \mathbb{N}_2 & \text{(read \& write)} \\ x_i, y_i & \text{(read from } P_i\text{)} \end{cases}$$

Invariant: $I_{ss} \equiv \mathit{Iden} \wedge \mathit{Clauses} \wedge (sat = 1)$

Fig. 1: Instance of Problem 3.

Invariant/legitimate states. Inspired by the form of the 3-SAT instance and its requirements, we define a state predicate I_{ss} that denotes the invariant of p.

- Corresponding to each clause $C_i = (l_q \vee l_r \vee l_s)$, we construct a state predicate $PredC_i \equiv (x_0 = q \implies y_0 = b_0^i) \vee (x_1 = r \implies y_1 = b_1^i) \vee (x_2 = s \implies y_2 = b_2^i)$. In other words, we have $PredC_i \equiv ((x_0 = q) \wedge (x_1 = r) \wedge (x_2 = s)) \implies ((y_0 = b_0^i) \vee (y_1 = b_1^i) \vee (y_2 = b_2^i))$. This way, we construct a state predicate $Clauses \equiv (\forall i \in \mathbb{N}_k : PredC_i)$. Notice that, we check the value of each x_j with respect to the index of the literal appearing in position j in C_i, where $j \in \mathbb{N}_3$. This is due to the fact that the domain of x_j is equal to the range of the indices of propositional variables (i.e., $\mathbb{N}_n$).
- A literal l_r may appear in positions i and j in distinct clauses of 3-SAT, where $i, j \in \mathbb{N}_3$ and $i \neq j$. Since each propositional variable $v_r \in \mathcal{V}$ gets a unique truth-value in 3-SAT, the truth-value of l_r is independent from its position in the 3-SAT formula. Given the way we construct the state predicate *Clauses*, it follows that, in the instance of Problem 3, whenever $x_i = x_j$ we should have $y_i = y_j$. Thus, we construct the state predicate $Iden \equiv (\forall i, j \in \mathbb{N}_3 : (x_i = x_j \implies y_i = y_j))$, which is conjoined with the predicate *Clauses*.
- In the instance of Problem 3, we require that $(sat = 1)$ holds in all invariant states.

Thus, the invariant of p is equal to the state predicate I_{ss}, where

$$I_{ss} \equiv Iden \wedge Clauses \wedge (sat = 1)$$

Notice that, the size of the state space of p is equal to $2\,(2n)^3$; i.e., $|S_p|$ is polynomial in the size of the 3-SAT instance.

Example 1. Example Construction

Let us consider the 3-SAT formula $\phi \equiv (v_0 \vee v_1 \vee v_2) \wedge (\neg v_1 \vee \neg v_1 \vee \neg v_2) \wedge (\neg v_1 \vee \neg v_1 \vee v_2) \wedge (v_1 \vee \neg v_2 \vee \neg v_0)$. Since there are three propositional variables and four clauses, we have $n = 3$ and $k = 4$. Moreover, based on the mapping described before, we have $C_0 \equiv (v_0 \vee v_1 \vee v_2)$, $C_1 \equiv (\neg v_1 \vee \neg v_1 \vee \neg v_2)$, $C_2 \equiv (\neg v_1 \vee \neg v_1 \vee v_2)$ and $C_3 \equiv (v_1 \vee \neg v_2 \vee \neg v_0)$. We have $B^0 = \langle 1, 1, 1\rangle$, $B^1 = \langle 0, 0, 0\rangle$, $B^2 = \langle 0, 0, 1\rangle$ and $B^3 = \langle 1, 0, 0\rangle$. The state predicate *Iden* is defined as before and *Clauses* is the conjunction of each $PredC_i$ $(i \in \mathbb{N}_4)$ defined as follows:

$$\begin{aligned}
PredC_0 &\equiv (x_0 = 0 \wedge x_1 = 1 \wedge x_2 = 2) \implies (y_0 = 1 \vee y_1 = 1 \vee y_2 = 1)\\
PredC_1 &\equiv (x_0 = 1 \wedge x_1 = 1 \wedge x_2 = 2) \implies (y_0 = 0 \vee y_1 = 0 \vee y_2 = 0)\\
PredC_2 &\equiv (x_0 = 1 \wedge x_1 = 1 \wedge x_2 = 2) \implies (y_0 = 0 \vee y_1 = 0 \vee y_2 = 1)\\
PredC_3 &\equiv (x_0 = 1 \wedge x_1 = 2 \wedge x_2 = 0) \implies (y_0 = 1 \vee y_1 = 0 \vee y_2 = 0)
\end{aligned}$$

4.2 Correctness of Reduction

In this section, we illustrate that Problem 3 is NP-complete. Specifically, we show that the instance of 3-SAT is satisfiable *iff* convergence from S_p to I_{ss} can be added to the instance of Problem 3, denoted p.

Lemma 1. *If the instance of 3-SAT has a satisfying valuation, then convergence from S_p can be added to the instance of Problem 3; i.e., there is a self-stabilizing version of p, denoted p_{ss}.*

Let there be a truth-value assignment to the propositional variables in $\mathcal{V}$ such that every clause evaluates to *true*; i.e., $\forall i : i \in \mathbb{N}_k : C_i$. Initially, $\delta_p = \emptyset$ and $\delta_p = \delta_{p_{ss}}$. Based on the value assignments to propositional variables, we include a set of transitions (represented as convergence actions) in p_{ss}. Then, we illustrate that the following three properties hold: the invariant $I_{ss} \equiv \mathit{Clauses} \wedge \mathit{Iden} \wedge (\mathit{sat} = 1)$ remains closed in p_{ss}, deadlock-freedom in $\neg I_{ss}$ and livelock-freedom in $p_{ss}|\neg I_{ss}$.

- If a propositional variable v_r (where $r \in \mathbb{N}_n$) is assigned *true*, then we include the following action in each process P_j, where $j \in \mathbb{N}_3$: $x_j = r \wedge y_j = 0 \wedge \mathit{sat} = 0 \rightarrow y_j := 1$.
- If a propositional variable v_r (where $r \in \mathbb{N}_n$) is assigned *false*, then we include the following action in each process P_j, where $j \in \mathbb{N}_3$: $x_j = r \wedge y_j = 1 \wedge \mathit{sat} = 0 \rightarrow y_j := 0$.
- We include the following actions in P_3: $(\mathit{Iden} \wedge \mathit{Clauses}) \wedge \mathit{sat} = 0 \rightarrow \mathit{sat} := 1$ and $\neg(\mathit{Iden} \wedge \mathit{Clauses}) \wedge \mathit{sat} = 1 \rightarrow \mathit{sat} := 0$.

Now we illustrate that closure, deadlock-freedom and livelock-freedom hold. That is, the resulting program is self-stabilizing to I_{ss}.

Closure. Since the first three processes can execute an action only in states where $\mathit{sat} = 0$, their actions are disabled where $\mathit{sat} = 1$. Thus, the first three processes exclude any transition that starts in I_{ss}; i.e., preserving the closure of I_{ss} and ensuring $p_{ss}|I_{ss} \subseteq p|I_{ss}$. Moreover, P_3 takes an action only when its guards are enabled; i.e., in illegitimate states. Therefore, none of the included actions violates the closure of I_{ss}, and the second constraint of the output of Problem 1 holds.

Livelock Freedom. To show livelock-freedom, we illustrate that the included actions have no circular dependencies. That is, no set of actions can enable each other in a cyclic fashion. Due to read/write restrictions, none of the three processes P_0, P_1 and P_2 executes based on the local variables of another process. Moreover, each process can update only its own y value. Once any one of the processes P_0, P_1 and P_2 updates its y value, it disables itself. Thus, the actions of one process cannot enable/disable another process. Moreover, since each action disables itself, there are no self-loops either. The guards of the actions of P_3 cannot be simultaneously *true*. Moreover, once one of them is enabled, the other one is certainly disabled, and the execution of one cannot enable another (because they only update the value of *sat*). Only processes P_0, P_1 and P_2 can make the predicate $(\mathit{Iden} \wedge \mathit{Clauses})$ *true* when $\mathit{sat} = 0$. Once P_3 sets *sat* to 1 from states $(\mathit{Iden} \wedge \mathit{Clauses}) \wedge (\mathit{sat} = 0)$, a state in I_{ss} is reached. Therefore, there are no cycles that start in $\neg I_{ss}$ and exclude any state in I_{ss}.

Deadlock Freedom. We illustrate that, in every state in $\neg I_{ss} \equiv (\neg(\mathit{Iden} \wedge \mathit{Clauses}) \vee (\mathit{sat} = 0))$, there is at least one action that is enabled.

- **Case 1**: $((\mathit{Iden} \wedge \mathit{Clauses}) \wedge (\mathit{sat} = 0))$ holds. In these states, the first action of P_3 is enabled. Thus, there are no deadlocks in this case.

- **Case 2**: $(\neg(\mathit{Iden} \wedge \mathit{Clauses}) \wedge (\mathit{sat} = 1))$ holds. In this case, the second action of P_3 is enabled. Thus, there are no deadlocks in this case.
- **Case 3**: $(\neg(\mathit{Iden} \wedge \mathit{Clauses}) \wedge (\mathit{sat} = 0))$ holds. None of the actions of P_3 are enabled in this case. Nonetheless, since $\neg(\mathit{Iden} \wedge \mathit{Clauses})$ holds, either $\neg \mathit{Iden}$ or $\neg \mathit{Clauses}$, or both are *true*. When $\neg \mathit{Clauses}$ holds, there must be some state predicate PredC_i $(i \in \mathbb{N}_k)$ that is *false*. (Recall that, the invariant I_{ss} includes a state predicate $\mathit{PredC}_i \equiv (x_0 = q \implies y_0 = b_0^i) \vee (x_1 = r \implies y_1 = b_1^i) \vee (x_2 = s \implies y_2 = b_2^i)$ corresponding to each clause $C_i \equiv (l_q \vee l_r \vee l_s)$ in the instance of 3-SAT.) This means that the following three state predicates are *false*: $(x_0 = q \implies y_0 = b_0^i), (x_1 = r \implies y_1 = b_1^i)$ and $(x_2 = s \implies y_2 = b_2^i)$. Since the instance of 3-SAT is satisfiable, at least one of the literals l_q, l_r or l_s must be true. As a result, based on the way we have included the actions depending on the truth-values of the propositional variables, at least one of the following actions must have been included in p_{ss}: $(x_0 = q \wedge y_0 \neq b_0^i \wedge \mathit{sat} = 0) \rightarrow y_0 := b_0^i$, $(x_1 = r \wedge y_1 \neq b_1^i \wedge \mathit{sat} = 0) \rightarrow y_1 := b_1^i$, and $(x_2 = s \wedge y_2 \neq b_2^i \wedge \mathit{sat} = 0) \rightarrow y_2 := b_2^i$. Thus, there is some action that is enabled when $\neg \mathit{Clauses}$ holds. A similar reasoning implies that there exists some action that is enabled when $\neg \mathit{Iden}$ holds. Thus, there are no deadlocks in Case 3.

Based on the closure of the invariant I_{ss} in all actions, deadlock-freedom in $\neg I_{ss}$ and lack of non-progress cycles in $p_{ss}|\neg I_{ss}$, it follows that the resulting program p_{ss} is self-stabilizing to I_{ss}. □

Example 2. Example Construction
In the example discussed in this section, the formula ϕ has a satisfying assignment for $v_0 = 1$, $v_1 = 0$, $v_2 = 0$. Using this value assignment, we include the following actions in the first three processes P_r where $r \in \mathbb{N}_3$:

$$x_r = 0 \wedge y_r = 0 \wedge \mathit{sat} = 0 \rightarrow y_r := 1$$
$$x_r = 1 \wedge y_r = 1 \wedge \mathit{sat} = 0 \rightarrow y_r := 0$$
$$x_r = 2 \wedge y_r = 1 \wedge \mathit{sat} = 0 \rightarrow y_r := 0$$

The actions of P_3 are as follows:

$$(\mathit{Iden} \wedge \mathit{Clauses}) \wedge \mathit{sat} = 0 \rightarrow \mathit{sat} := 1$$
$$\neg(\mathit{Iden} \wedge \mathit{Clauses}) \wedge \mathit{sat} = 1 \rightarrow \mathit{sat} := 0$$

Figure 2 illustrates the transitions of the stabilizing program p_{ss} originating in the state predicate $(x_0 = 0 \wedge x_1 = 1 \wedge x_2 = 2)$. Each state is represented by four bits which signify the respective values of $y_0, y_1, y_2, \mathit{sat}$. Invariant states are depicted by ovals, and the label on each transition denotes the executing process.

Lemma 2. *If there is a self-stabilizing version of the instance of Problem 3, then the corresponding 3-SAT instance has a satisfying valuation.*

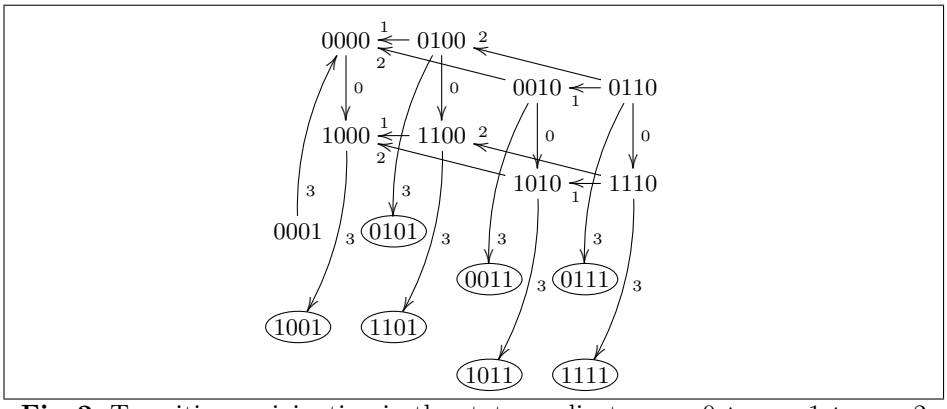


Fig. 2: Transitions originating in the state predicate $x_0 = 0 \wedge x_1 = 1 \wedge x_2 = 2$

By assumption, we consider a program p_{ss} to be a self-stabilizing version of p from I_{ss}. That is, p_{ss} satisfies the requirements of Problem 3.

Only P_3 can correct $(sat = 0)$. Clearly, p_{ss} must preserve the closure of I_{ss}, and should not have any deadlocks or livelocks in the states in $\neg I_{ss} \equiv (\neg(\mathit{Iden} \wedge \mathit{Clauses}) \vee (sat = 0))$. Thus, p_{ss} must include actions that correct $\neg(\mathit{Iden} \wedge \mathit{Clauses})$ and $(sat = 0)$. Since p_{ss} must adhere to the read/write restrictions of p, only P_3 can correct $(sat = 0)$ to $(sat = 1)$. For the same reason, P_3 cannot contribute to correcting $\neg(\mathit{Iden} \wedge \mathit{Clauses})$; only P_0, P_1 and P_2 have the write permissions to do so by updating their own y values.

The rest of the reasoning is as follows: We first illustrate that P_0, P_1 and P_2 in p_{ss} must not execute in states where $(sat = 1)$. Then, we draw a correspondence between actions included in p_{ss} and how propositional variables get unique truth-values in 3-SAT and how the clauses are satisfied.

P_0, P_1 and P_2 can be enabled only when $(sat = 0)$. We observe that no process P_j $(j \in \mathbb{N}_3)$ can have a transition that starts in the invariant I_{ss}; otherwise, the constraint $\delta_{p_{ss}}|I_{ss} \subseteq \delta_p|I_{ss}$ would be violated. We also show that no recovery action of P_0, P_1 and P_2 can include a transition that starts in a state where $sat = 1$. By contradiction, assume that some P_j $(j \in \mathbb{N}_3)$ includes a transition (s_0, s_1) where $s_0 \in \neg I_{ss}$ and $sat(s_0) = 1$ for some fixed values of x_j and y_j. Since P_j cannot read x_i and y_i of other processes P_i, where $(i \in \mathbb{N}_3) \wedge (i \neq j)$, the transition (s_0, s_1) has a groupmate (s'_0, s'_1), where $x_i(s'_0) = x_j(s'_0)$ and $y_i(s'_0) = y_j(s'_0)$ for all $i \in \mathbb{N}_3$ where $(i \neq j)$. Thus, *Iden* is true at s'_0. Moreover, due to the form of the 3-SAT instance, no clause $(l_q \vee l_r \vee l_s)$ exists such that $(q = r = s)$. Thus, *Clauses* holds at s'_0 as well, thereby making s'_0 an invariant state. As a result, (s_0, s_1) is grouped with a transition that starts in I_{ss}, which again violates the constraint $\delta_{p_{ss}}|I_{ss} \subseteq \delta_p|I_{ss}$. Hence, P_0, P_1 and P_2 can be enabled only when $(sat = 0)$.

Actions of P_3. We show that P_3 must set sat to 0 when $\neg(\mathit{Iden} \wedge \mathit{Clauses}) \wedge sat = 1$ and may only assign sat to 1 when $(\mathit{Iden} \wedge \mathit{Clauses}) \wedge sat = 0$. As shown above, P_0, P_1, and P_2 cannot act when $sat = 1$, forcing P_3 to execute from

$\neg(\mathit{Iden} \wedge \mathit{Clauses}) \wedge (\mathit{sat} = 1)$. P_3 must therefore have the action $\neg(\mathit{Iden} \wedge \mathit{Clauses}) \wedge \mathit{sat} = 1 \rightarrow \mathit{sat} := 0$. Consequently, P_3 cannot assign *sat* to 1 when $\neg(\mathit{Iden} \wedge \mathit{Clauses}) \wedge \mathit{sat} = 0$; otherwise, it would create a livelock with the previous action. From states where $(\mathit{Iden} \wedge \mathit{Clauses}) \wedge \mathit{sat} = 0$ holds, P_3 is the only process which can change *sat* to 1, thereby reaching an invariant state. Thus, P_3 must include the actions $\neg(\mathit{Iden} \wedge \mathit{Clauses}) \wedge \mathit{sat} = 1 \rightarrow \mathit{sat} := 0$ and $(\mathit{Iden} \wedge \mathit{Clauses}) \wedge \mathit{sat} = 0 \rightarrow \mathit{sat} := 1$.

Each P_j, for $j \in \mathbb{N}_3$ must have exactly one action for each unique value of x_j. When $\mathit{sat} = 0$, fixing the value of x_j to some $a \in \mathbb{N}_n$ reduces the possible local states for process P_j to 2, where $y_j = 0$ or $y_j = 1$ for $j \in \mathbb{N}_3$. (Notice that, both of these states are illegitimate since $\mathit{sat} = 0$.) Thus, when $(x_j = a \wedge \mathit{sat} = 0)$ holds, process P_j has 4 possible actions: $y_j = 0 \rightarrow y_j := 0$, $y_j = 0 \rightarrow y_j := 1$, $y_j = 1 \rightarrow y_j := 0$, and $y_j = 1 \rightarrow y_j := 1$. It is clear that the first and last of those actions are self-loops and cannot be included. Thus, P_j can have either action $y_j = 0 \rightarrow y_j := 1$ or $y_j = 1 \rightarrow y_j := 0$, but not both without creating a livelock. That is, P_j cannot have more than 1 action. Additionally, to make *Iden* true, P_j must include some action. By contradiction, assume that P_j has no actions. Another process P_i ($i \in \mathbb{N}_3$, $i \neq j$) can be in the state where $x_i = x_j$. There are two possibilities for the y values in this non-invariant state, $y_j = 0 \wedge y_i = 1$ or $y_j = 1 \wedge y_i = 0$. P_i can resolve either scenario with an action but cannot resolve both as this would require 2 actions. That is, to resolve both cases P_i needs the cooperation of P_j. Thus, P_j must have some action. Since P_j cannot have more than one action, it follows that P_j has exactly one action.

Truth-value assignment to propositional variables. Based on the above reasoning, for each value $a \in \mathbb{N}_n$, if a process P_j includes the action $x_j = a \wedge y_j = 0 \wedge \mathit{sat} = 0 \rightarrow y_j := 1$, then we assign *true* to the propositional variable v_a. If P_j includes the action $x_j = a \wedge y_j = 1 \wedge \mathit{sat} = 0 \rightarrow y_j := 0$, then we assign *false* to v_a. Let P_j include the action $x_j = a \wedge y_j = 0 \wedge \mathit{sat} = 0 \rightarrow y_j := 1$. By contradiction, if another process P_i, where $i \in \mathbb{N}_3 \wedge i \neq j$, includes the action $x_i = a \wedge y_i = 1 \wedge \mathit{sat} = 0 \rightarrow y_i := 0$, then *Iden* would be violated and p_{ss} would never recover from the state $x_j = a \wedge x_i = a \wedge y_j = 1 \wedge y_i = 0 \wedge \mathit{sat} = 0$; i.e., a deadlock state, which is a contradiction with p_{ss} being self-stabilizing. Thus, each propositional variable gets a unique truth-value assignment and these value assignments are logically consistent.

Satisfying the clauses. Since p_{ss} is self-stabilizing from I_{ss}, eventually I_{ss} becomes *true*. This means that every PredC_i in the *Clauses* predicate becomes *true*. The one-to-one correspondence created by the mapping between each state predicate PredC_i and each clause C_i implies that PredC_i *holds iff at least one literal in* C_i *holds*. Therefore, all clauses are satisfied with the truth-value assignment based on the actions of p_{ss}. □

Theorem 1. *Adding convergence to low atomicity programs is NP-complete.*

Proof. The NP-hardness of adding convergence follows from Lemmas 1 and 2. The NP membership of adding convergence has already been established in [6]; hence the NP-completeness. □

Corollary 1. *Adding nonmasking fault tolerance to low atomicity programs is NP-complete.*

Corollary 1 follows from Theorem 1 and the fact that Problem 3 is a special case of Problem 2.

5 Discussion

This section discusses extant work in three most related categories: algorithmic design of fault tolerance in general, algorithmic design of self-stabilization in particular, and complexity of algorithmic design. Several researchers have investigated the problem of algorithmic design of fault-tolerant systems [6,20,21,22,23], where a specific level of fault tolerance (e.g., failsafe, nonmasking or masking) is systematically incorporated in an existing program. Kulkarni and Arora [6] present a family of polynomial-time algorithms for the addition of different levels of fault tolerance to high atomicity programs, while demonstrating that adding masking fault tolerance to low atomicity programs is NP-complete. In our previous work [20], we establish a foundation for the addition of fault tolerance to low atomicity programs using efficient heuristics and component-based methods. Jhumka *et. al* [21,22] investigate the addition of failsafe fault tolerance under efficiency constraints. Bonakdarpour and Kulkarni [23] exploit symbolic techniques to increase the scalability of the addition of fault tolerance.

Existing methods for the algorithmic design of convergence include constraint-based methods [24] and sound heuristics [25,26]. Abujarad and Kulkarni [24] consider the program invariant as a conjunction of a set of local constraints, each representing the set of local legitimate states of a process. Then, they synthesize convergence actions for correcting the local constraints without corrupting the constraints of neighboring processes. Nonetheless, they do not explicitly address cases where local constraints have cyclic dependencies (e.g., maximal matching on a ring), and their case studies include only acyclic topologies. In our previous work [25,26], we present a method where we partition the state space to a hierarchy of state predicates based on the length of the shortest computation prefix from each state to some state in the invariant. Then, we systematically explore the space of all candidate recovery transitions that could contribute in recovery to the invariant without creating non-progress cycles outside the invariant.

Most hardness results [6,8,27] presented for the addition of fault tolerance lack the additional constraint of *recovery from any state*, which we have in the addition of convergence. The proof of NP-hardness of adding failsafe fault tolerance presented in [8] is based on a reduction from 3-SAT, nonetheless, a failsafe fault-tolerant program does not need to recover to its invariant when faults occur. While a masking fault-tolerant program is required to recover to its invariant in the presence of faults, the problem of adding masking fault tolerance relies on finding a subset of the state space from where such recovery is possible; no need to provide recovery from any state. As such, the hardness proof presented in [6] is based on a reduction in which such a subset of state space is identified along with corresponding convergence actions if and only if the instance of 3-SAT is satisfiable. This means that some states are allowed to be excluded from the fault span; this is not an option in the case of adding convergence. The essence of the

proof in [27] also relies on the same principle where Bonakdarpour and Kulkarni illustrate the NP-hardness of designing progress from one state predicate to another for low atomicity programs.

6 Conclusions and Future Work

This paper illustrates that adding convergence to low atomicity programs is an NP-complete problem, where convergence guarantees that from any state program computations recover to a set of legitimate states; i.e., invariant. In other words, we demonstrated that designing self-stabilizing programs from their non-stabilizing versions is NP-complete in the size of the state space. Since self-stabilization is a special case of nonmasking fault tolerance, it follows that adding nonmasking fault tolerance to intolerant distributed programs is also NP-complete. When faults occur, a nonmasking program guarantees recovery from states reached due to the occurrence of faults to its invariant. In the absence of faults, the computations of a nonmasking program remain in its invariant. Thus, this paper solves a decade-old open problem [6]. As an extension of this work, we will investigate special cases where the addition of convergence/nonmasking can be performed efficiently. That is, *for what programs, classes of faults and invariants the addition of convergence/nonmasking can be done in polynomial time*? Moreover, while we analyzed the general case complexity of adding convergence/nonmasking tolerance under no fairness assumption, it would be interesting to investigate the impact of different fairness assumptions on the complexity of adding convergence.

References

1. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
2. A. Arora. *A foundation of fault-tolerant computing.* PhD thesis, The University of Texas at Austin, 1992.
3. A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.
4. Zhiming Liu and Mathai Joseph. Transformation of programs for fault-tolerance. *Formal Aspects of Computing*, 4(5):442–469, 1992.
5. A. Arora, M. Gouda, and G. Varghese. Constraint satisfaction as a basis for designing nonmasking fault-tolerant systems. *Journal of High Speed Networks*, 5(3):293–306, 1996.
6. S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 82–93, London, UK, 2000. Springer-Verlag.
7. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
8. Sandeep S. Kulkarni and Ali Ebnenasir. Complexity issues in automated synthesis of failsafe fault-tolerance. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2(3):201–215, July-September 2005.
9. Michael R. Gary and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

10. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Communications of the ACM*, 7(3):201–215, July 1960.
11. Ali Ebnenasir and Sandeep S. Kulkarni. SAT-based synthesis of fault-tolerance, 2004. Fast Abstracts of the International Conference on Dependable Systems and Networks.
12. M. Gouda. The triumph and tribulation of system stabilization. In Jean-Michel Helary and Michel Raynal, editors, *Distributed Algorithms, (9th WDAG'95)*, volume 972 of *Lecture Notes in Computer Science (LNCS)*, pages 1–18. Springer-Verlag, Le Mont-Saint-Michel, France, September 1995.
13. M. Gouda. The theory of weak stabilization. In *5th International Workshop on Self-Stabilizing Systems*, volume 2194 of *Lecture Notes in Computer Science*, pages 114–123, 2001.
14. L. Lamport and N. Lynch. *Handbook of Theoretical Computer Science: Chapter 18, Distributed Computing: Models and Methods*. Elsevier Science Publishers B. V., 1990.
15. M Nesterenko and A Arora. Stabilization-preserving atomicity refinement. *Journal of Parallel and Distributed Computing*, 62(5):766–791, 2002.
16. M. Demirbas and A. Arora. Convergence refinement. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 589–597, Washington, DC, USA, July 2002. IEEE Computer Society.
17. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1990.
18. Anish Arora and Sandeep S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Transactions on Software Engineering*, 24(6):435–450, 1998.
19. G. Varghese. *Self-stabilization by local checking and correction*. PhD thesis, MIT/LCS/TR-583, 1993.
20. Ali Ebnenasir. *Automatic Synthesis of Fault-Tolerance*. PhD thesis, Michigan State University, May 2005.
21. Arshad Jhumka. *Automated design of efficient fail-safe fault tolerance*. PhD thesis, Darmstadt University of Technology, 2004.
22. Arshad Jhumka, Felix C. Freiling, Christof Fetzer, and Neeraj Suri. An approach to synthesise safe systems. *International Journal of Security and Networks*, 1(1/2):62–74, 2006.
23. B. Bonakdarpour and S. S. Kulkarni. Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, pages 3–10, Washington, DC, USA, June 2007. IEEE Computer Society.
24. Fuad Abujarad and Sandeep S. Kulkarni. Automated constraint-based addition of nonmasking and stabilizing fault-tolerance. *Theoretical Computer Science*, 412(33):4228–4246, 2011.
25. Aly Farahat and Ali Ebnenasir. A lightweight method for automated design of convergence in network protocols. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7(4):38:1–38:36, December 2012.
26. Ali Ebnenasir and Aly Farahat. Swarm synthesis of convergence for symmetric protocols. In *Proceedings of the Ninth European Dependable Computing Conference*, pages 13–24, 2012.
27. Borzoo Bonakdarpour and Sandeep S. Kulkarni. Revising distributed UNITY programs is NP-complete. In *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS)*, pages 408–427, 2008.