On the Hardness of Adding Nonmasking Fault Tolerance

Alex Klinkhamer and Ali Ebnenasir

Abstract—This paper investigates the complexity of adding nonmasking fault tolerance, where a nonmasking fault-tolerant program guarantees recovery from states reached due to the occurrence of faults to states from where its specifications are satisfied. We first demonstrate that adding nonmasking fault tolerance to low atomicity programs—where processes have read/write restrictions with respect to the variables of other processes—is NP-complete (in the size of the state space) on an unfair or weakly fair scheduler. Then, we establish a surprising result that even under strong fairness, addition of nonmasking fault tolerance remains NP-hard! The NP-hardness of adding nonmasking fault tolerance is based on a polynomial-time reduction from the 3-SAT problem to the problem of designing self-stabililizing programs from their non-stabilizing versions, which is a special case of adding nonmasking fault tolerance. While it is known that designing self-stabilization under the assumption of strong fairness is polynomial, we demonstrate that adding self-stabilization to non-stabilizing programs is NP-hard under weak fairness.

Index Terms—Fault tolerance, distributed programs, NP-hardness

1 INTRODUCTION

Today’s distributed programs are subject to a variety of types of faults (e.g., node crash, process restart, transient faults, message loss) due to their inherent complexity, human errors and environmental factors (e.g., soft errors). Such programs should guarantee service availability even in the presence of faults. Nonetheless, designing and verifying recovery of distributed programs is a difficult task in part due to the limitations of distribution and the need for global recovery by a coordination of local actions. This paper investigates the complexity of augmenting an existing distributed program with nonmasking fault tolerance (i.e., adding nonmasking fault tolerance), where a nonmasking program ensures recovery from a subset of states reached due to the occurrence of faults to states from where its specifications are satisfied. A special case of nonmasking nonmasking fault tolerance is self-stabilization where recovery should be provided from any state.

Several researchers have investigated the problem of adding nonmasking fault tolerance to programs [1], [2], [3], [4], [5], [6]. For instance, Liu and Joseph [4] present a method for the transformation of a fault-intolerant program to a fault-tolerant version thereof by going through a set of refinement steps—where a fault-intolerant program provides no guarantees when faults occur. They model faults by state perturbation, where program actions are executed in an interleaving with fault actions. Arora and Gouda [2], [3] provide a unified theory for the formulation of fault tolerance functionalities in terms of closure and convergence, where closure means that, in the absence of faults, a fault-tolerant program remains in a set of legitimate states, called its invariant, and convergence specifies that the state of the program is recovered to its invariant from a superset of the invariant reached due to the occurrence of faults, called a fault-span. 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. 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. In a shared memory model, Kulkarni and Arora [6] demonstrate that adding failsafe/nonmasking/masking fault tolerance to high atomicity programs can be done in polynomial time in the size of the state space (under no fairness), where the processes of a high atomicity program can read/write all program variables in an atomic step. Nonetheless, they show that, for distributed programs, adding masking fault tolerance is NP-complete (in the size of the state space) on an unfair scheduler. The authors of [6] model distribution in a low atomicity shared memory model, where each process has read and write restrictions with respect to the local variables of other processes. Kulkarni and Ebnenasir [8], [9] show that adding failsafe fault tolerance to distributed programs is also an NP-complete problem. However, the complexity of adding nonmasking fault tolerance has remained an open problem for more than a decade. While this problem is known to be in NP, no polynomial-time algorithms are known for efficient design of nonmasking fault tolerance for low atomicity programs; nor has there been a proof of NP-completeness.
In this paper, we prove that adding nonmasking fault tolerance to low atomicity programs is NP-complete under no fairness, weak, and strong fairness assumptions (see Fig. 1). A weakly fair scheduler infinitely often executes any program action that is continuously enabled (i.e., ready for execution), whereas a strongly fair scheduler infinitely often executes any transition that is enabled infinitely often. Our hardness proof is based on a reduction from the 3-SAT problem [10] to the problem of adding self-stabilization to non-stabilizing programs under no fairness. Since self-stabilization is a special case of 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 \( P = \mathsf{NP} \)). We also show that even under weak fairness the addition of stabilization to low atomicity programs remains an NP-complete problem (see Fig. 1), which implies the NP-completeness of adding nonmasking fault tolerance under weak fairness in general. Moreover, we present a surprising result that, while adding stabilization under the assumption of strong fairness is known to be polynomial (in the state space) [11], [12], [13], the general case complexity of adding nonmasking fault tolerance under strong fairness remains NP-complete!

**Contributions.** We present

- a proof of \( \mathsf{NP} \)-completeness of adding self-stabilization to distributed programs under no fairness and weak fairness assumptions;
- a proof of \( \mathsf{NP} \)-completeness of adding nonmasking fault tolerance to distributed programs under any fairness assumption, thereby solving a decade-old problem, and
- a proof of \( \mathsf{NP} \)-completeness of adding self-stabilization even in special cases where (i) a process can atomically read the global state of the distributed program and can update its own local state, and (ii) processes have self-disabling actions—where the execution of an action disables itself.

**Organization.** Section 2 presents the basic concepts of programs, faults and fault tolerance. Section 3 formally states the problem of adding nonmasking fault tolerance. Section 4 illustrates that adding nonmasking fault tolerance to low atomicity programs is in general \( \mathsf{NP} \)-complete (on an unfair, weakly or strongly fair scheduler). 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, specifications, our distribution model (adapted from [6]), faults and fault tolerance (adapted from [1], [3], [11], [14]). For ease of presentation, we use a simplified version of Dijkstra’s token ring (TR) protocol [1] as a running example.

**Programs.** A program in our setting is a representation of any system that can be captured by a (non-deterministic) finite-state machine (e.g., network protocols). Formally, a program \( p \) is a tuple \((V_p, \delta_p, \Pi_p, T_p)\) of a finite set \( V_p \) of variables, a set \( \delta_p \) of transitions, a finite set \( \Pi_p \) of \( N \) processes, where \( N \geq 1 \), and a topology \( T_p \). Each variable \( v_i \in V_p \) for \( i \in \mathbb{N} \), where \( \mathbb{N} = \{0, 1, \ldots, m-1\}\) and \( m > 0 \), has a finite non-empty domain \( D_i \). A state \( s \) of \( p \) is a valuation \( \langle d_0, d_1, \ldots, d_{m-1} \rangle \) of variables \( \langle v_0, v_1, \ldots, v_{m-1} \rangle \), 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 process \( P_j \in \Pi_p \) is a triple \((d_j, r_j, w_j)\), where \( 0 \leq j \leq N - 1 \) and \( d_j \subseteq S_{p_j} \times S_{p_j} \) denotes the set of transitions of \( P_j \). The set of transitions of the program \( p \), denoted \( \delta_p \), is the union of the sets of transitions of its processes; i.e., \( \delta_p = \bigcup_{j=0}^{N-1} \delta_j \). 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 \( (v)X \) evaluates to true at \( s \). For simplicity, we misuse the notations \( p \) and \( \delta_p \) interchangeably.

To simplify the specification of \( \delta_p \) for designers, we use Dijkstra’s guarded commands language [15] as a shorthand for representing the set of program transitions. A guarded command (a.k.a. action) is of the form \( \text{grd} \rightarrow \text{stmt} \), and includes a set of transitions \((s_0, s_1)\) such that the predicate \( \text{grd} \) holds in \( s_0 \) and the atomic execution of the statement \( \text{stmt} \) results in state \( s_1 \). An action \( \text{grd} \rightarrow \text{stmt} \) is enabled in a state \( s \) iff \( \text{grd} \) 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 \( s \).

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

**Properties and specifications.** For a program \( p \), a safety property \( \text{sprop} \) stipulates that nothing bad ever happens (e.g., no two processes access a shared resource simultaneously). Formally, we follow [7], [16] to specify a safety property \( \text{sprop} \) as a set of bad transitions that must not be executed by \( p \); i.e., \( \text{sprop} \subseteq S_p \times S_p \). A computation \( \sigma = \ll s_0, s_1, \ldots \gg \) of \( p \) satisfies its safety property \( \text{sprop} \) from \( s_0 \) iff no transition \((s_i, s_{i+1})\), where \( i \geq 0 \), is in \( \text{sprop} \); i.e., \( \sigma \) includes no bad transitions. A liveness property, denoted \( \text{lprop} \), specifies some good things that should eventually occur (e.g., a process will eventually access some shared resource). Formally, liveness is captured as a set of sequences of states. A computation \( \sigma = \ll s_0, s_1, \ldots \gg \) of \( p \)
satisfies its liveness property \(lprop\) from \(s_0\) iff \(\sigma\) has a suffix in \(lprop\). Following Alpern and Schneider [7], we define a specification \(spec\) as a set of safety and liveness properties. A computation \(\sigma\) of \(p\) satisfies its specification \(spec\) from \(s_0\) iff \(\sigma\) satisfies the safety and liveness of \(spec\) from \(s_0\). A program satisfies its specification \(spec\) from a state predicate \(I\) iff every computation of \(p\) starting in \(I\) satisfies \(spec\).

**Read/Write model.** We adopt a shared memory model [17] since reasoning in a shared memory setting is easier, and several (correctness-preserving) transformations [18], [19] exist for the refinement of shared memory programs to their message-passing versions. To model the topological constraints (denoted \(T_p\)) of a program \(p\), we consider a subset of variables in \(V_p\) that each process \(P_j\) \((j \in \mathbb{N})\) 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. A process \(P_j\) is not allowed to update a variable \(v \notin r_j\).

**Impact of read 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 \(x_0, x_1\) denote a state of this program. Now, if \(P_0\) writes \(x_0\) in a transition \((0,0), (1,0)\), 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 \((0,0), (1,0))\) and \((0,1), (1,1))\) is included in \(P_0\). In general, a transition is included in the set of transitions of a process \(I\) 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 \(P_j\) meet the following constraints: \(\forall v \in r_j: (v(s_0) = v(s_0')) \land (v(s_1) = v(s_1'))\) and \(\forall v \notin r_j: (v(s_0) = v(s_0')) \land (v(s_1) = v(s_1'))\). (It is known that the total number of groups is polynomial in \(|S_p|\) [6].)

**Example: Token Ring.** The token ring 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 (\(\otimes\) and \(\otimes\) respectively denote addition and subtraction in modulo 3):

\[
A_0 : \quad (x_0 = x_2) \quad \rightarrow \quad x_0 := x_2 \pm 1
\]

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 \(P_j\), for \(1 \leq j \leq 2\):

\[
A_j : \quad (x_j \neq x_{j+1}) \quad \rightarrow \quad x_j := x_{j+1}
\]

Each process \(P_j\) copies \(x_{j+1}\) only if \(x_j \neq x_{j+1}\), where \(j = 1, 2\). By definition, process \(P_0\) 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

\[
((x_0 = x_1) \land (x_1 = x_2)) \lor ((x_1 + 1 = x_0) \land (x_1 = x_2))
\]

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 includes three transitions.

**Closure and invariant.** A state predicate \(X\) is closed in an action \(grd \rightarrow stmt\) iff executing \(stmt\) from any state \(s \in (X \land \neg \neg \neg grd)\) 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 in \(X\) requires that every computation of \(p\) starting in \(X\) remains in \(X\) [11]. A state predicate \(I\) is an invariant of \(p\) iff \(I\) is closed in \(p\) and \(p\) satisfies its spec from \(I\).

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

**Faults.** We capture the impact of faults on a program as state perturbations. Formally, a class of faults \(f\) for a program \(p = (V_p, \delta_p, \Pi_p, T_p)\) 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 \(\delta_p\) 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 \(I \subseteq T\) and \(T\) is closed in \(p[f]\). Observe that for all computations of \(p\) that start in \(I, T\) is a boundary in the state space of \(p\) to which (but not beyond which) the state of \(p\) may be perturbed by the occurrence of \(f\). The same way we use guarded commands to represent program transitions, we use them to specify fault transitions. While we concentrate on transient faults that can perturb the state of a program without causing any permanent damage, the notion of state perturbation is appropriate for modeling other types of faults. Liu and Joseph [4] use state perturbation to model failstop failures. Chen and Kulkarni [20] show that 20 out of 31 categories of faults classified by Avizienis et al. [21] can be modeled by state perturbation.

**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 transient faults on \(x_0\) where \(|\cdot|\) denotes the non-deterministic assignment of values to \(x_0\):

\[
F_0 : \quad true \quad \rightarrow \quad x_0 := 0|1|2
\]

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

We say that a sequence of states, \(\sigma = \langle s_0, s_1, \ldots \rangle\) 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 \(\sigma\) is finite and terminates in state \(s_t\), then there is no state \(s\) such that \((s_t, s) \in \delta_p\), and (3) \(\forall n : n \geq 0 : \forall j : j > n : (s_{j+1}, s_j) \in \delta_p\). The first requirement captures that in each step, either a program transition or a fault transition is executed. The second requirement states that if the only transition that starts from \(s_t\) is a fault transition \((s_t, s_j)\) then as far as the program is concerned, \(s_t\) is still a deadlock state because the program does not have control over the execution of \((s_t, s_j)\); i.e., \((s_t, s_j)\) 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], [3], [22], [23]) to ensure that eventually recovery can occur.
3 PROBLEM STATEMENT

In this section, we present the problem of adding nonmasking fault tolerance under different fairness assumptions. Consider a fault-intolerant program \( p = \langle V_p, \delta_p, \Pi_p, T_p \rangle \), its invariant \( I \), its specification \( spec \), a class of faults \( f \), and a fairness assumption \( \mathcal{F} \in \{ \text{unfair, weak, strong} \} \). Our objective is to generate a revised version of \( p \), denoted \( p' \), such that \( p' \) is nonmasking \( f \)-tolerant from an invariant \( I' \) under the fairness assumption \( \mathcal{F} \). To separate fault tolerance from functional concerns, we would like to preserve the behavior of \( p \) in the absence of \( f \) during the addition of fault tolerance; i.e., in the absence of faults, \( p' \) satisfies \( spec \) and meets the fairness assumption \( \mathcal{F} \) in the definitions of weak, strong, then Problem 3.3 can be solved in polynomial time.

We state the corresponding decision problem as follows:

**Problem 3.3. Decision Problem of Adding Stabilization:**

- **INSTANCE:** A program \( p = \langle V_p, \delta_p, \Pi_p, T_p \rangle \) that meets the constraints of Problem 3.1 are met under the fairness assumption \( \mathcal{F} \).
- **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 3.1 are met under the fairness assumption \( \mathcal{F} \)?

A special case of Problem 3.1 is where (i) \( f \) denotes a class of transient faults; (ii) \( I = I' \); (iii) \( \delta_p[I'] = \delta_{p'}[I'] \), and (iv) \( p' \) is self-stabilizing from \( I' \) under \( \mathcal{F} \) fairness.

Previous work [11], [12], [13] illustrates that if \( \mathcal{F} = \text{strong} \), then Problem 3.3 can be solved in exponential time in \( |S_p| \). Stabilization under strong fairness (a.k.a. weak stabilization) requires that from any state \( s \in I \), there exists a computation prefix that includes a state in \( I \). However, the general case complexity of adding stabilization under no fairness and weak fairness assumptions have been open problems thus far. Stabilization under no fairness (a.k.a. strong stabilization) stipulates that from any state \( s \in I \), each computation prefix includes a state in \( I \). We have developed heuristics and software tools [13] that synthesize self-stabilizing programs in polynomial time. Moreover, previous research [24], [25] testifies the practical significance of adding nonmasking tolerance.

**Problem 3.1. Adding Nonmasking Fault Tolerance:**

- **Input:** A program \( p = \langle V_p, \delta_p, \Pi_p, T_p \rangle \) that satisfies its specification \( spec \) from an invariant \( I \); (2) a class of faults \( f \), and (3) a fairness assumption \( \mathcal{F} \in \{ \text{unfair, weak, strong} \} \).
- **Output:** A program \( p' = \langle V_p, \delta_{p'}, \Pi_{p'}, T_{p'} \rangle \) and an invariant \( I' \) such that: (1) \( I' \) is non-empty and \( I' \subseteq I \); (2) \( \delta_p[I'] = \delta_{p'}[I'] \), and (3) \( p' \) is nonmasking \( f \)-tolerant from \( I' \) under \( \mathcal{F} \) fairness.

We state the corresponding decision problem as follows:

**Problem 3.2. Decision Problem of Adding Nonmasking Fault Tolerance:**

- **INSTANCE:** A program \( p = \langle V_p, \delta_p, \Pi_p, T_p \rangle \) that satisfies its specification \( spec \) from an invariant \( I \); (2) a class of faults \( f \), and (3) a fairness assumption \( \mathcal{F} \in \{ \text{unfair, weak, strong} \} \).
- **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 3.1 are met under the fairness assumption \( \mathcal{F} \)?

A special case of Problem 3.1 is where (i) \( f \) denotes a class of transient faults; (ii) \( I = I' \); (iii) \( \delta_p[I'] = \delta_{p'}[I'] \), and (iv) \( p' \) is self-stabilizing from \( I \) under \( \mathcal{F} \) fairness.

**Problem 3.3. Decision Problem of Adding Stabilization:**

- **INSTANCE:** A program \( p = \langle V_p, \delta_p, \Pi_p, T_p \rangle \) that satisfies its specification \( spec \) from an invariant \( I \); (2) a class of transient faults \( f \), and (3) a fairness assumption \( \mathcal{F} \in \{ \text{unfair, weak, strong} \} \).
- **QUESTION:** Does there exist a program \( p' = \langle V_p, \delta_{p'}, \Pi_{p'}, T_{p'} \rangle \) such that: (1) \( I \) remains unchanged (i.e., \( I' = I \)); (2) \( \delta_p[I] = \delta_{p'}[I] \), and (3) \( p' \) is self-stabilizing from \( I \) under \( \mathcal{F} \) fairness?

Previous work [11], [12], [13] illustrates that if \( \mathcal{F} = \text{strong} \), then Problem 3.3 can be solved in polynomial time in \( |S_p| \). Stabilization under strong fairness (a.k.a. weak stabilization) requires that from any state \( s \in I \), there exists a computation prefix that includes a state in \( I \). However, the general case complexity of adding stabilization under no fairness and weak fairness assumptions have been open problems thus far. Stabilization under no fairness (a.k.a. strong stabilization) stipulates that from any state \( s \in I \), each computation prefix includes a state in \( I \). We have developed heuristics and software tools [13] that synthesize self-stabilizing programs in polynomial time. Moreover, previous research [24], [25] testifies the practical significance of adding nonmasking tolerance.
4 HARDNESS RESULTS

In this section, we illustrate that adding nonmasking fault tolerance to low atomicity programs is NP-complete under no fairness (Section 4.2), weak (Section 4.3) and strong (Section 4.4) fairness assumptions. We first state the 3-SAT decision problem.

Problem 4.1. The 3-SAT decision problem:

- INSTANCE: A set $V$ of $n$ propositional variables $(v_0, \ldots, v_{n-1})$ and $k$ clauses $(C_0, \ldots, C_{k-1})$ over $V$ such that each clause is of the form $(l_q \lor l_r \lor l_s)$, where $q, r, s \in [n]$, and $N_0 = \{0, \ldots, n-1\}$. Each $l_i$ denotes a literal, where a literal is either $-v_i$ or $v_i$, for $v_i \in V$.
- QUESTION: Is there a satisfying truth-value assignment for the variables in $V$ such that each $C_i$ evaluates to true, for all $i \in N_0$?

Notation. We say $l_i$ is a negative (respectively, positive) literal if it has the form $-v_i$ (respectively, $v_i$), where $v_i \in V$.

Consider a clause $C_i = (l_q \lor l_r \lor l_s)$. We use a binary variable $b_i$ where $i \in N_0$ and $j \in N_3$, to denote the sign of the first, second and the third literal in $C_i$. For example, if $l_q = -v_i$, $l_r = v_i$, and $l_s = -v_i$, then we have $b_0 = 0, b_1 = 1$ and $b_2 = 0$. Let the tuple $B_i = (b_0, b_1, b_2)$ denote the values of $b_i$ variables, for each clause $C_i$ where $j \in N_3$.

4.1 Intuition Behind Hardness Proofs

This section presents the intuition behind the hardness of adding nonmasking fault tolerance under different fairness assumptions.

No fairness. In Section 4.2, we show that adding stabilization under no fairness is NP-hard, thereby implying the NP-hardness of adding nonmasking fault tolerance in general. Consider a deadlock state $s_0$ outside $I$. To ensure that some state in $I$ will eventually be reached from $s_0$, we need to build a computation prefix from $s_0$ to $I$ while ensuring that non-progress cycles are not formed in $\neg I$. Let $(s_d, s)$ be a transition included in a process $P_j$ during the construction of some computation prefix. We can include $(s_d, s)$ in the set of transitions of $P_j$ if we include any transition $(s'_d, s')$ grouped with $(s_d, s)$ (due to read restrictions of $P_j$), and $(s'_d, s')$ does not create a cycle with other transitions. That is, one has to identify a subset of transition groups that construct a computation prefix from any state in $\neg I$ to $I$ without creating cycles outside $I$. Thus, deciding whether a transition group should be included in some process resembles the assignment of a truth value to a propositional variable in the instance of 3-SAT.

Weak fairness (Section 4.3). Under weak fairness, a cycle $c$ that has an action $A$ enabled in every state of $c$ is not considered a non-progress cycle because a weakly fair scheduler guarantees the execution of $A$, thereby exiting the cycle. Thus, an algorithm for the addition of stabilization need not resolve such cycles. One would think that this could simplify the design of stabilization under weak fairness, but Theorem 4.8 proves otherwise. The intuition behind this hardness result is that there might still be cycles for which there is no action similar to $A$. Thus, we have to deal with a similar combinatorial problem mentioned for stabilization under no fairness.

Strong fairness (Section 4.4). A strongly fair scheduler (as defined in Section 2) ensures that a program will eventually exit any reachable cycle $c$ that has some outgoing transition from one of its states to a state outside $c$. Thus, to design self-stabilization under strong fairness, we need to ensure that from any state there exists a computation prefix that reaches the invariant; no need to resolve cycles. This problem is known to be solvable in polynomial time [11, 13]. Since the general case problem of adding nonmasking fault tolerance should deal with cases where faults may cause permanent damage (unlike transient faults), one has to ensure that permanent faults are not activated or else the system may reach an unrecoverable state. To ensure that a distributed program does not reach such states, designers might have to guarantee cycle-freedom (despite strong fairness) in certain subsets of the fault span; otherwise, an interleaving of fault and cycle transitions may perturb the program to an unrecoverable state. Thus, designing the general case nonmasking fault tolerance under strong fairness is at least as hard as designing self-stabilization under no fairness!

4.2 Hardness Under No Fairness

In this section, we investigate the general case complexity of Problem 3.3 under no fairness. We specifically demonstrate that, for a given intolerant program $p$ with an invariant $I$, designing a revised version of $p$, denoted $p_{ss}$, such that $p_{ss}$ is self-stabilizing from $I$ is an NP-hard problem. Section 4.2.1 presents a polynomial-time mapping from 3-SAT to an instance of Problem 3.3. Section 4.2.2 shows that the instance of 3-SAT is satisfiable iff a self-stabilizing version of the instance of Problem 3.3 exists where $F = \text{unfair}$.

4.2.1 Polynomial Mapping

In this section, we present a polynomial-time mapping from an instance of 3-SAT to the instance of Problem 3.3 where $F = \text{unfair}$, denoted $p = (V_p, \delta_p, \Pi_p, T_p)$. That is, corresponding to each propositional variable and clause, we illustrate how we construct a non-stabilizing program $p$, its processes $\Pi_p$, its variables $V_p$, its read/write restrictions, its specification $\text{spec}$ and its invariant $I$. We shall use this mapping in Section 4.2.2 to demonstrate that the instance of 3-SAT is satisfiable iff a self-stabilizing version of $p$ exists.

Processes, variables and read/write restrictions. We consider four processes, $P_0, P_1, P_2, P_3$ in $p$. Each process $P_j$ ($j \in N_3$) has two variables $x_j$ and $y_j$, where the domain of $x_j$ is equal to $N_0$ and $y_j$ is a binary variable. The process $P_j$ can read both $x_j$ and $y_j$, but can write only $y_j$. Thus, the processes $P_0, P_1$ and $P_2$ cannot read each other’s variables. We also consider a fourth process $P_3$ that can read all variables and write to a binary variable $\text{sat} \in N_2$. The variable $\text{sat}$ can be read by processes $P_0, P_1$ and $P_2$, but cannot be written. Thus, we have $V_p = \{x_0, y_0, x_1, y_1, x_2, y_2, \text{sat}\}$, $\Pi_p = \{P_0, P_1, P_2, P_3\}$ and the topology of $p$ is identified by the read/write restrictions of processes as depicted in Fig. 2.

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


Corresponding to each clause \( C_i = (l_q \lor l_r \lor l_s) \), we construct a state predicate \( \text{Pred}C_i \equiv (x_0 = q \implies y_0 = b_0^i) \lor (x_1 = r \implies y_1 = b_1^i) \lor (x_2 = s \implies y_2 = b_2^i) \). In other words, we have \( \text{Pred}C_i \equiv ((x_0 = q) \land (x_1 = r) \land (x_2 = s)) \implies ((y_0 = b_0^i) \lor (y_1 = b_1^i) \lor (y_2 = b_2^i)) \). This way, we construct a state predicate \( \text{Clauses} \equiv (\forall i \in \mathbb{N}_n : \text{Pred}C_i) \). 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 \( l \) 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 \( v_i \in \mathbb{V} \) gets a unique truth-value in 3-SAT, the truth-value of \( l \) is independent from its position in the 3-SAT formula. Given the way we construct the state predicate \( \text{Clauses} \), it follows that, in the instance of Problem 3.3, whenever \( x_i = x_j \) we should have \( y_i = y_j \). Thus, we construct the state predicate \( \text{Iden} \equiv (\forall i, j \in \mathbb{N}_3 : (x_i = x_j \implies y_i = y_j)) \), which is conjoined with the predicate \( \text{Clauses} \).

In the instance of Problem 3.3, we require that \( \text{sat} = 1 \) holds in all invariant states.

Based on the aforementioned reasoning, the invariant of \( p \) is equal to the state predicate \( I_{ss} \), where

\[
I_{ss} \equiv \text{Iden} \land \text{Clauses} \land (\text{sat} = 1)
\]

Notice that the size of the state space of \( p \), denoted \( |S_p| \), is \( 2(2n)^3 \), which is polynomial in the size of the 3-SAT instance.

**Specification.** The safety of \( \text{spec} \) forbids any transition that starts in \( I_{ss} \). That is, the instance of Problem 3.3 is silent in its invariant (i.e., \( \delta_p I_{ss} = \emptyset \)).

**Example 4.2.** Consider the 3-SAT formula \( \phi \equiv (v_0 \lor v_1 \lor v_2) \land (\neg v_1 \lor \neg v_1 \lor \neg v_2) \land (\neg v_1 \lor \neg v_1 \lor v_2) \land (v_1 \lor \neg v_2 \lor \neg v_0) \). 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 \( C_0 \equiv (v_0 \lor v_1 \lor v_2), C_1 \equiv (\neg v_1 \lor \neg v_1 \lor \neg v_2), C_2 \equiv (\neg v_1 \lor \neg v_1 \lor v_2) \) and \( C_3 \equiv (v_1 \lor \neg v_2 \lor \neg v_0) \).

Thus, we have \( B^0 = \{1, 1, 1 \} \), \( B^1 = \{0, 0, 0 \} \), \( B^2 = \{0, 0, 1 \} \) and \( B^3 = \{1, 0, 0 \} \). The predicates \( \text{Pred}C_i (i \in \mathbb{N}_i) \) have the following form:

- \( \text{Pred}C_0 \equiv (x_0 = 0 \land x_1 = 1 \land x_2 = 2) \implies (y_0 = 1 \lor y_1 = 1 \lor y_2 = 1) \)
- \( \text{Pred}C_1 \equiv (x_0 = 1 \land x_1 = 1 \land x_2 = 2) \implies (y_0 = 0 \lor y_1 = 0 \lor y_2 = 0) \)
- \( \text{Pred}C_2 \equiv (x_0 = 1 \land x_1 = 1 \land x_2 = 2) \implies (y_0 = 0 \lor y_1 = 0 \lor y_2 = 1) \)
- \( \text{Pred}C_3 \equiv (x_0 = 1 \land x_1 = 2 \land x_2 = 0) \implies (y_0 = 1 \lor y_1 = 0 \lor y_2 = 0) \)

The state predicate \( \text{Iden} \) is as defined before.

**4.2.2 Correctness of Reduction**

In this section, 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 3.3, denoted \( p \).

**Lemma 4.3.** If the instance of 3-SAT has a satisfying valuation, then stabilization can be added to the instance of Problem 3.3.

Let there be a truth-value assignment to the propositional variables in \( V \) such that every clause evaluates to true; i.e., \( \forall i : i \in \mathbb{N}_i : C_i \). Let \( p_{ss} \) denote the self-stabilizing version of \( p \). Initially, \( \delta_p = \emptyset \) and \( p = p_{ss} \). Based on the value assignments to propositional variables, we include a set of transitions (represented as convergence actions) in \( p_{ss} \). Then, we show that the following three properties hold: the invariant \( I_{ss} \equiv \text{Clauses} \land \text{Iden} \land (\text{sat} = 1) \) remains closed, 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 \( P_r \) where \( j \in \mathbb{N}_3: x_j = r \land y_j = 0 \land \text{sat} = 0 \imp y_j = 1 \).
- If a propositional variable \( v_r \) (where \( r \in \mathbb{N}_n \)) is assigned false, then we include the following action in each process \( P_r \) where \( j \in \mathbb{N}_3: x_j = r \land y_j = 1 \land \text{sat} = 0 \imp y_j = 0 \).
- We include the following actions in \( P_3: (\text{Iden} \land \text{Clauses}) \land \text{sat} = 0 \imp \text{sat} = 1 \) and \( \neg (\text{Iden} \land \text{Clauses}) \land \text{sat} = 1 \imp \text{sat} = 0 \).

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

**Closure.** Since the first three processes can execute an action only in states where \( \text{sat} = 0 \), their actions are disabled where \( \text{sat} = 1 \). 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 in \( \neg I_{ss} \). Thus, no action violates the closure of \( I_{ss} \), and the second constraint of the output of Problem 3.1 holds.

**Livelock freedom.** To show livelock freedom, we prove that the included actions have no circular dependencies. Due to read/write restrictions, none of the first three processes 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, 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 $(\text{Iden} \land \text{Clauses})$ true when $sat = 0$. Due to write restrictions, once $P_3$ sets $sat$ to 1 from states $(\text{Iden} \land \text{Clauses}) \land (sat = 0)$, 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 $\neg I_{ss} \equiv (\neg (\text{Iden} \land \text{Clauses}) \lor (sat = 0))$, there is at least one action that is enabled.

- **Case 1.** $(\text{Iden} \land \text{Clauses}) \land (sat = 0))$ holds. In these states, the first action of $P_3$ is enabled. Thus, there are no deadlocks in this case.
- **Case 2.** $(\neg (\text{Iden} \land \text{Clauses}) \land (sat = 1))$ holds. In this case, the second action of $P_3$ is enabled. Thus, there are no deadlocks in this case.
- **Case 3.** $(\neg (\text{Iden} \land \text{Clauses}) \land (sat = 0))$ holds. None of the actions of $P_3$ are enabled in this case. Nonetheless, since $\neg (\text{Iden} \land \text{Clauses})$ holds, either $\neg \text{Iden}$ or $\neg \text{Clauses}$, or both are true. When $\neg \text{Clauses}$ holds, there must be some state predicate $\text{PredC}_i$ $(i \in \mathbb{N}_3)$ that is false. (Recall that, the invariant $I_{ss}$ includes a state predicate $\text{PredC}_i \equiv (x_0 = q \Rightarrow y_0 = b_0) \lor (x_1 = r \Rightarrow y_1 = b_1) \lor (x_2 = s \Rightarrow y_2 = b_2)$ corresponding to each clause $C_i \equiv (l_q \lor l_r \lor l_s)$ in the instance of 3-SAT.) This means that the following three state predicates are false: $(x_0 = q \Rightarrow y_0 = b_0)$, $(x_1 = r \Rightarrow y_1 = b_1)$ and $(x_2 = s \Rightarrow y_2 = b_2)$. Since the instance of 3-SAT is satisfiable, at least one of the literals $l_q, l_r, l_s$ 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 $p_{ss}$: $(x_0 = q \land y_0 = b_0 \land sat = 0) \Rightarrow y_0 = b_0$, $(x_1 = r \land y_1 = b_1 \land sat = 0) \Rightarrow y_1 = b_1$, and $(x_2 = s \land y_2 = b_2 \land sat = 0) \Rightarrow y_2 = b_2$. Thus, there is some action that is enabled when $\neg \text{Clauses}$ holds. A similar reasoning implies that there exists some action that is enabled when $\neg \text{Iden}$ holds; hence no deadlocks in Case 3.

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

**Example 4.4.** Example construction:

In the example discussed in this section, the formula $\phi$ has a satisfying assignment of $v_0 = 1$, $v_1 = 0$, $v_2 = 0$. Using this value assignment, we include the following actions in the first three processes $P_j$ where $j \in \mathbb{N}_3$:

- $x_j = 0 \land y_j = 0 \land sat = 0 \Rightarrow y_j := 1$
- $x_j = 1 \land y_j = 1 \land sat = 0 \Rightarrow y_j := 0$
- $x_j = 2 \land y_j = 1 \land sat = 0 \Rightarrow y_j := 0$

The actions of $P_3$ are as follows:

- $(\text{Iden} \land \text{Clauses}) \land sat = 0 \Rightarrow sat := 1$
- $(\neg (\text{Iden} \land \text{Clauses}) \land sat = 1 \Rightarrow sat := 0$

Figure 3 shows the set of states where $x_0 = 0$, $x_1 = 1$, $x_2 = 2$ and transitions of the stabilizing program $p_{ss}$. Each state is represented by four bits which signify the values of $(y_0, y_1, y_2, sat)$. Invariant states are depicted by ovals and the labels on the transitions denote which process executes that transition.

**Lemma 4.5.** If there is a self-stabilizing version of the instance of Problem 3.3 where $F$ is unfair, then the corresponding 3-SAT instance has a satisfying valuation.

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 3.3.

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 $\neg I_{ss} \equiv (\neg (\text{Iden} \land \text{Clauses}) \lor (sat = 0))$. Thus, $p_{ss}$ must include actions that correct $(\neg (\text{Iden} \land \text{Clauses}) \land (sat = 0))$. Since $p_{ss}$ must adhere to the read/write restrictions of $p$, only $P_3$ can correct $(sat = 0)$. For the same reason, $P_3$ cannot contribute to correcting $(\neg (\text{Iden} \land \text{Clauses})$; 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 $\delta_{p_{ss}}|I_{ss} \subset \delta_{p}|I_{ss}$ 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 $s_0 \in I_{ss}$ and $(sat(s_0) = 1)$ for some fixed values of $x_j$ and $y_j$. Since $P_0$ cannot read $x_j$ or $y_j$ of other processes $P_i$, where $(i \in \mathbb{N}_3)$ and $(i \neq j)$, the transition $(s_0, s_1)$ has a groupmate $(s_0', s_1')$, where $x_j(s_0') = x_j(s_0)$ and $y_j(s_0') = y_j(s_0')$ 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 $(l_q \lor l_r \lor l_s)$ 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 $\delta_{p_{ss}}|I_{ss} \subset \delta_{p}|I_{ss}$. 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 \((\text{Iden} \land \text{Clauses}) \land \neg \text{sat} = 1\) and may only assign $sat$ to 1 when \((\text{Iden} \land \text{Clauses}) \land \text{sat} = 0\). As shown above, $P_0$, $P_1$, and $P_2$ cannot act when $sat = 1$, forcing $P_3$ to execute from \((\text{Iden} \land \text{Clauses}) \land (\neg \text{sat} = 1)\). $P_3$ must therefore have the action \((\text{Iden} \land \text{Clauses}) \land \neg \text{sat} = 1 \rightarrow \text{sat} := 0\). Consequently, $P_3$ cannot assign $sat$ to 1 when \((\text{Iden} \land \text{Clauses}) \land \text{sat} = 0\); otherwise, it would create a livelock with the previous action. From states where \((\text{Iden} \land \text{Clauses}) \land \text{sat} = 0\) 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 \((\text{Iden} \land \text{Clauses}) \land \neg \text{sat} = 1 \rightarrow \text{sat} := 0\) and \((\text{Iden} \land \text{Clauses}) \land \neg \text{sat} = 0 \rightarrow \text{sat} := 1\).

Each $P_j$, for $j \in \mathbb{N}_3$, must have exactly one action for each unique value of $x_j$. When $sat = 0$, 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 $sat = 0$.) Thus, when $(x_j = a \land \neg \text{sat} = 0)$ holds, process $P_j$ has four 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 these 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 one action. To make $\text{Iden}$ 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 a state where $x_i = x_j$. There are two possibilities for the $y$ values in this non-invariant state, $y_i = 0 \land y_j = 1$ or $y_i = 1 \land y_j = 0$. $P_i$ can resolve either scenario with an action but cannot resolve both as this would require two 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 $x_j = a \land y_j = 0 \land \neg \text{sat} = 0 \rightarrow y_j := 1$, then we assign true to the propositional variable $v_a$. If $P_j$ includes the action $x_j = a \land y_j = 1 \land \neg \text{sat} = 0 \rightarrow y_j := 0$, then we assign false to $v_a$. Let $P_j$ include the action $x_j = a \land y_j = 0 \land \neg \text{sat} = 0 \rightarrow y_j := 1$. By contradiction, if another process $P_i$ where $i \in \mathbb{N}_3, i \neq j$, includes the action $x_i = a \land y_i = 1 \land \neg \text{sat} = 0 \rightarrow y_i := 0$, then $\text{Iden}$ would be violated and $p_3$ would never recover from the state $x_j = a \land x_i = a \land y_i = 1 \land y_j = 0 \land \neg \text{sat} = 0$; i.e., a deadlock state, which is a contradiction with $p_3$ being self-stabilizing. Thus, each propositional variable gets a unique truth-value assignment and these value assignments are logically consistent.

Satisfying the clauses. Since $p_3$ is self-stabilizing from $I_{ss}$, eventually $I_{ss}$ becomes true; i.e., every $\text{PredC}_i$ in the $\text{Clauses}$ predicate becomes true. The one-to-one correspondence created by the mapping between each state predicate $\text{PredC}_i$ and each clause $C_i$ implies that $\text{PredC}_i$ 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 $p_3$.

**Theorem 4.6.** Adding stabilization to low atomicity programs is NP-complete.

**Proof.** The NP-hardness of adding stabilization follows from Lemmas 4.5 and 4.3. The NP membership of adding stabilization has already been established in [6]; hence the NP-completeness.

**Corollary 4.7.** Adding nonmasking fault tolerance to low atomicity programs under no fairness is NP-complete.

Proof follows from Theorem 4.6 and the fact that Problem 3.3 is a special case of Problem 3.2.

### 4.3 Hardness under Weak Fairness

This section illustrates that, even under the assumption of weak fairness, addition of nonmasking fault tolerance in general, and self-stabilization in particular remain hard problems.

**Theorem 4.8.** Adding stabilization under weak fairness is NP-complete.

**Proof.** Consider the mapping from 3-SAT to Problem 3.3 presented in Section 4.2. We leverage the same mapping in order to create a mapping from an instance of 3-SAT to an instance of Problem 3.3 where $F = \text{weak}$. Let $p_{ss}$ denote the instance of Problem 3.3 with the invariant $I_{ss} \equiv \text{Clauses} \land \text{Iden} \land (\neg \text{sat} = 1)$ constructed corresponding to the 3-SAT formula. The instance of Problem 3.3 where $F = \text{weak}$ includes exactly the same processes and variables in $p_{ss}$. Moreover, we compose $p_{ss}$ with the token ring program introduced in Section 2. Since the state space of the TR program includes 27 states, the size of the state space of the instance of Problem 3.3 under weak fairness remains polynomial in the size of the 3-SAT formula. (The size of the state space of the instance of Problem 3.3 where $F = \text{weak}$ is $27 \times |S_{\text{TR}}|$.) Let the invariant of the resulting program be equal to the conjunction of the invariants of the two programs; i.e., $I_w \equiv I_{ss} \land I_{\text{TR}}$. Thus, the resulting composed program will converge to $I_w$ iff both $p_{ss}$ and the TR program stabilize to their corresponding invariants.

$\Rightarrow$ **Proof:** We show that if the 3-SAT instance is satisfiable then the composition of $p_{ss}$ and TR is self-stabilizing from $I_w$ under weak fairness. If the 3-SAT formula is satisfiable then $p_{ss}$ is strongly stabilizing from $I_{ss}$. Moreover, Dijkstra [1] has illustrated that the TR program is strongly stabilizing. Outside $I_w$, if $I_{ss}$ has become true and $I_{\text{TR}}$ is false, then the TR program will eventually recover to $I_{\text{TR}}$. If the TR program has recovered to its invariant, but $p_{ss}$ has not yet recovered to $I_{ss}$, then there must be some action $A$ of $p_{ss}$ that is enabled (because the 3-SAT formula is satisfiable). At the same time, TR’s computations in $I_{\text{TR}}$ are infinite. That is, the action $A$ is continuously enabled in a cycle formed in the state predicate $\neg I_w \land I_{\text{TR}}$. Such a cycle is a non-progress cycle only under no fairness assumption; i.e., under a weakly fair scheduler, the composed program will eventually stabilize to $I_w$.

$\Leftarrow$ **Proof:** Let there be a program $p_w$ composed of the variables of TR and $p_{ss}$, and actions that enable stabilization to $I_w$ from any state under weak fairness. That is, $I_{ss}$ and $I_{\text{TR}}$ must both become true eventually. Our proof strategy is two-fold. First, we make the following observations to enable compositional reasoning about the two components of $p_w$:
Observation 4.9. Only processes of TR can make $I_{TR}$ true and only processes of $p_{ss}$ can contribute to reaching $I_{ss}$ from any state.

Proof of Observation 4.9 is straightforward due to the read/write restrictions of processes and the independence of the two components in reading/writing the variables of each other. That is, even if the actions of the two components get interleaved, they will not impact the recovery of each component to its invariant under weak fairness. Second, since the TR component does not intervene the convergence of $p_{ss}$ to $I_{ss}$ (based on Observation 4.9), we can reason about $p_{ss}$ separately. We prove that, even under weak fairness, the $p_{ss}$ component of $p_{we}$ should be strongly self-stabilizing from $I_{ss}$. This way, we can reuse the proof of Lemma 4.5 to demonstrate how the instance of 3-SAT is satisfied. To this end, we prove that neither $P_3$ nor $P_0$, $P_1$ and $P_2$ can have any cycles in $\delta_{p_{ss}} | - I_{ss}$.

- **Actions of $P_3$ alone cannot create a cycle in $\delta_{p_{ss}} | - I_{ss}$.** From the proof of Lemma 4.5, we know that $P_0$, $P_1$ and $P_2$ can only execute if $sat = 0$; otherwise, the closure of $I_{ss}$ would be violated. The only way $P_3$ can have a cycle in $\delta_{p_{ss}} | - I_{ss}$ is to toggle the value of $sat$ (because only $P_3$ has the permission to write $sat$). Since $P_0$, $P_1$ and $P_2$ can only execute if $sat = 0$, no action would be continuously enabled in this cycle. This would constitute a non-progress cycle under weak fairness, which is a contradiction with $p_{we}$ having no non-progress cycles under weak fairness.

- **There is no cycle in $\delta_{p_{ss}} | - I_{ss}$ where multiple processes participate.** Since the actions of processes $P_0$, $P_1$ and $P_2$ in $p_{ss}$ are independent from each other, it is impossible that a cycle exists in which $P_0$, $P_1$ and $P_2$ participate. Moreover, there is no cycle that is formed by the interleaving of the actions of $P_0$, $P_1$ and $P_2$ with $P_3$'s actions since all processes of $p_{ss}$ would be participating in such a cycle; i.e., none of the actions of $p_{ss}$ would be continuously enabled. Such a cycle would constitute a non-progress cycle under weak fairness.

- **Processes $P_i$ (i \in \mathbb{N}_3)$ of $p_{ss}$ cannot form any cycles alone in $\delta_{p_{ss}} | - I_{ss}$.** By contradiction, consider a case where some $P_i$ contains a cycle including the actions $x_i = a \land y_i = 0 \land sat = 0 \rightarrow y_i := 1$ and $x_i = a \land y_i = 1 \land sat = 0 \rightarrow y_i := 0$. These two actions capture an equivalence class of cycles in the state space of $p_{ss}$. Each cycle in this equivalence class includes two global states where $x_i = a \land y_i = 0 \land sat = 0$ holds in one and $x_i = a \land y_i = 1 \land sat = 0$ holds in the other. Consider another process $P_j$ where $j \in \mathbb{N}_3$ and $j \neq i$. Either $P_j$ is not in any cycles, or $P_j$ is also trapped in a cycle similar to $P_i$'s. The former case means that, by weak fairness, $P_j$ will get to a state where all its actions are disabled. In this case, the cycles of $P_j$ become non-progress cycles under weak fairness. In the latter case, both $P_i$ and $P_j$ would be in a cycle in which no action is continuously enabled; hence a non-progress cycle under weak fairness. Now, we illustrate that $P_3$ cannot help $P_i$ to exit its cycle either. Toggling the value of $y_i$ would affect the truth-value of the predicates $PredC_m$ that depend on the state of $P_i$, where $m \in \mathbb{N}_3$. This in turn could change the truth value of the predicate $Iden \land \text{Clause}$. Since the actions of $P_3$ must include $Iden$ and $\text{Clause}$ in their guards, $P_3$ cannot be continuously enabled in the cycle of $P_i$. Thus, the cycle of $P_i$ forms a non-progress cycle under weak fairness, which is a contradiction with $p_{we}$ being self-stabilizing from $I_{ss}$ under weak fairness.

Since $p_{ss}$ must be a strongly stabilizing program, the proof of Lemma 4.5 can be reused to demonstrate that the instance of 3-SAT is satisfied.

\[\square\]

**Corollary 4.10.** Adding nonmasking fault tolerance under the assumption of weak fairness is NP-complete.

Proof of Corollary 4.10 follows from Theorem 4.8 and the fact that Problem 3.3 is a special case of Problem 3.2 where $F = weak$.

### 4.4 Hardness under Strong Fairness

In this section, we present a somewhat surprising result that adding nonmasking fault tolerance to low atomicity programs remains NP-hard even under strong fairness! This is surprising because adding self-stabilization under strong fairness (a.k.a. weak stabilization [11]) is known to be polynomial [11], [12], [13]. Our proof strategy is as follows. We first reuse the reduction presented in the proofs of Lemmas 4.3 and 4.5 to illustrate that adding nonmasking tolerance to low atomicity programs under no fairness is NP-hard. This may seem as a redundant result to Corollary 4.7, however, in the second step of our strategy, we reuse the mapping and reduction of this proof for showing the NP-hardness of adding nonmasking tolerance under strong fairness.

**An alternative proof for the NP-hardness of adding nonmasking fault tolerance under no fairness (i.e., Corollary 4.7).** First, we present a mapping from an arbitrary instance of 3-SAT to an instance of adding nonmasking fault tolerance (i.e., Problem 3.2). In Section 4.2.1, we augment the instance of Problem 3.3 where $F = unfair$ with an additional process and two new types of faults. (Fig. 2 depicts the structure of the instance of Problem 3.2.) The idea behind this mapping is that finding a fault-span and a new invariant $I \subseteq I$ for an intolerant program $p$ with its invariant $I$ is at least as hard as adding stabilization.

Fig. 4 illustrates the structure of our mapping for adding nonmasking fault tolerance. Processes $P_0$ to $P_3$ are taken from the system of Fig. 2. We add a new process $P_4$ that has a read-only binary variable failed used to mark unrecoverable states. The invariant of the intolerant program $p$ is $I \equiv I_{ss} \land (failed = 0)$ where $I_{ss} \equiv Iden \land \text{Clauses} \land (sat = 1)$ is the invariant of the system of $p_{ss}$ in Fig. 2. Any state where failed = 1 is unrecoverable since failed cannot be modified by any process. We consider two classes of faults $f_{ss}$ and $f_n$ denoted by $f \equiv f_{ss} \cup f_n$, where $f_{ss}$ and $f_n$ are defined as:

1. Otherwise, $P_3$ would include two actions $sat = 1 \rightarrow sat := 0$ and $sat = 0 \rightarrow sat := 1$ forming a cycle, whose impossibility we have already shown in the first item of our reasoning.
Fauls ensure that a nonmasking \( f \)-tolerant program \( p' \) from \( I \) exists if \( f \) is stabilizing from \( I_{ss} \). The fault \( f_{ss} \) may occur from states in \( I' \subseteq I' \) and perturb the program to any state where \( sat = 0 \) (failed = 0 is unchanged) and all \( x_i \) and \( y_i \) values are randomly chosen by the random function \( select \). The fault-class \( f_n \) transitions to an unrecoverable state (failed = 1) when \( I_{ss} \) does not hold but \( sat = 1 \). In effect, \( P_3 \) is forced to assign \( sat = 0 \) only when \( Iden \land Clauses \) holds. \( P_0 \) to \( P_2 \) must act to satisfy \( Iden \land Clauses \) when \( sat = 0 \), preserving our mapping between program actions and a 3-SAT truth-assignment.

The size of the state space \( |S_p| \) remains polynomial in the number of propositional variables \( n \) in the corresponding 3-SAT instance, specifically \( |S_p| = 2^3(2n)^3 \). It remains to show that a satisfying truth-assignment exists for the 3-SAT instance if \( f \) is a nonmasking \( f \)-tolerant version of the instance in Fig. 4 exists from \( I \equiv I_{ss} \land (failed = 0) \).

\[ P_0 \ldots P_3 \] : Unchanged from Figure 2.

\[ P_4 : \{ \text{failed} \in \mathbb{N}_2 \} \quad \text{(read-only)} \]

Invariant: \( I \equiv I_{ss} \land (failed = 0) \)

Faults: \( f \equiv f_{ss} \cup f_n \)

\[ \neg I_{ss} \land sat = 1 \rightarrow failed = 1 \]

Stabilizing program \( p_{ss} \) for the corresponding system in Fig. 2 can be constructed from \( p' \), and then we shall reuse the proof of Lemma 4.5 to satisfy the 3-SAT instance. Observe that all states where failed = 1 must be excluded because recovery is impossible from \( failed = 1 \) due to write restrictions. Moreover, states where \( \neg I_{ss} \land sat = 1 \) holds cannot be in \( T \) either, otherwise \( f_n \) could assign \( failed = 1 \). Thus, the weakest and strongest predicates that can be considered as \( T \) are respectively equal to \( (I \land sat = 0) \land failed = 0 \) (note \( I \), not \( I' \) and \( I' \land sat = 0 \land failed = 0 \). From \( I' \), the occurrence of \( f_{ss} \) can perturb the state of the program to any state where \( sat = 0 \land failed = 0 \) holds. Thus, recovery to \( I' \) should be provided from \( sat = 0 \). In such states, either \( (Iden \land Clauses) \) holds or not. If \( (Iden \land Clauses) \) holds in states where \( sat = 0 \), then \( p' \) can recover to \( I' \) only with an action of \( P_3 \) that sets \( sat = 1 \). If \( (Iden \land Clauses) \) does not hold when \( sat = 0 \), then \( P_3 \) must not set \( sat = 1 \) because then the state of \( p' \) will reach \( \neg I_{ss} \land sat = 1 \) from where fault \( f_n \) can occur and set \( failed = 1 \), which is an unrecoverable state. The only processes that have read/write permission to make \( (Iden \land Clauses) \) true are \( P_0, P_1 \) and \( P_2 \). Thus, \( p' \) must provide recovery from \( \neg (Iden \land Clauses) \) to \( (Iden \land Clauses) \) when \( sat = 0 \). We can use these actions, along with actions \( \neg (Iden \land Clauses) \land sat = 0 \rightarrow sat = 1 \) of \( P_3 \), to construct a program \( p_{ss} \) which is self-stabilizing for the corresponding instance given in Fig. 2. From this point, we use Lemma 4.5 on \( p_{ss} \) to find a truth-assignment which satisfies the 3-SAT instance.

**Theorem 4.11.** Adding nonmasking fault tolerance to low atomicty programs under strong fairness is NP-complete.

**Proof.** Our proof strategy is to augment the mapping presented in the alternative proof of Corollary 4.7 and then show that the instance of 3-SAT is satisfiable if \( f \) is nonmasking fault tolerance can be added to the instance of Problem 3.2 where \( F = \text{strong} \). The proposed polynomial-time mapping is as follows. We construct an intolerant program as demonstrated in Fig. 5. Processes \( P_0 \) to \( P_3 \) are the same as those in the program of Fig. 4. We include three new variables \( z_0, z_1 \) and \( z_2 \) in process \( P_3 \) which can be read and written only by \( P_3 \). The domain of each \( z_i \), where \( i \in \mathbb{N}_3 \), is equal to \( \{0, 1, 2\} \). We also consider a new fault-class \( f_{ss} \) for \( i \in \mathbb{N}_3 \). The invariant of the instance of Problem 3.2 is \( I \equiv I_{ss} \land (z_0 = z_1 = z_2 = 0) \land (failed = 0) \).
The classes of faults include \( f \equiv f_{ss} \cup f_0 \cup f_1 \cup f_{ss1} \cup f_{ss2} \), where \( f_{ss} \) and \( f_0 \) are taken from Fig. 4, and \( f_{ss1} \) is defined as follows (\( i \in \mathbb{N}_3 \)):

\[
\begin{align*}
    f_{ss1} & : \ y_i = 0 \land sat = 0 \land z_i = 0 \rightarrow z_i := 1 \\
    f_{ss} & : \ y_i = 1 \land sat = 0 \land z_i = 1 \rightarrow z_i := 2 \\
    f_{ss} & : \ y_i = 0 \land sat = 0 \land z_i = 2 \rightarrow failed := 1
\end{align*}
\]

The new fault-class \( f_{ss} \) ensures that the processes \( P_0, P_1 \) and \( P_2 \) of any \( f \)-tolerant program \( p' \) do not form nontrivial cycles in the state predicate \( sat = 0 \) if \( p' \) is nonmasking \( f \)-tolerant from \( I \subseteq I \) under strong fairness. Without \( f_{ss} \), it would be trivial to add fault tolerance under strong fairness by including the actions \( y_i = 0 \land sat = 0 \rightarrow y_i := 1 \) and \( y_i = 1 \land sat = 0 \rightarrow y_i := 0 \) in \( P_i \) for each specific value of \( x_i \) where \( i \in \mathbb{N}_3 \), and the action \( Iden \land \text{Clauses} \land sat = 0 \rightarrow sat := 1 \) in \( P_i \).

Observe that the size of the state space \( |S_I| \) remains polynomial in the number of propositional variables \( n \) from the corresponding 3-SAT instance as \( |S_I| = 2^n (6n)^3 \). Now, we illustrate that a satisfying truth-value assignment exists for the 3-SAT instance if and only if a nonmasking \( f \)-tolerant version of the instance of Problem 3.2 exists where \( \mathcal{F} = \text{strong} \).

\( \Rightarrow \) \textbf{Proof:} Given a satisfying valuation for the 3-SAT instance, we can create a nonmasking \( f \)-tolerant program \( p' \) with invariant \( I' = I \) as specified in Fig. 5, where \( f = f_{ss} \cup f_0 \cup f_{ss1} \cup f_{ss2} \). From \( I, f_{ss} \) can perturb the program to states where \( sat = 0 \land failed = 0 \). Thus, states in \( \neg(Iden \land \text{Clauses}) \land sat = 1 \) are unreachables in the \( f \)-span of \( p' \) from \( I \), thereby ensuring that \( f_{ss} \) cannot take the program to the unrecoverable state \( failed = 1 \). Moreover, the state \( y_i = 0 \land z_i = 2 \) must be excluded from the \( f \)-span; otherwise, fault \( f_{ss} \) could perturb the program state to \( failed = 1 \). Thus, the weakest predicate we can consider to be the \( f \)-span of \( p' \) from \( I \) is equal to \( T = (I_{ss} \land sat = 0) \land (failed = 0) \land (y_0 = 1 \lor z_0 = 2) \land (y_1 = 1 \lor z_1 = 2) \land (y_2 = 1 \lor z_2 = 2) \).

We include the actions of \( P_0, P_1 \) and \( P_2 \) in \( p' \) based on the method outlined in the proof of Lemma 4.3. Thus, only one of the actions \( x_i = a \land y_i = 0 \land sat = 0 \rightarrow y_i := 1 \) and \( x_i = a \land y_i = 1 \land sat = 0 \rightarrow y_i := 0 \) is included in each process \( P_i \), where \( i \in \mathbb{N}_3 \). Process \( P_3 \) includes the actions \( (Iden \land \text{Clauses}) \land sat = 0 \rightarrow sat := 1 \) and \( \neg(Iden \land \text{Clauses}) \land sat = 1 \rightarrow sat := 0 \). Finally, the process \( P_1 \) includes the actions \( z_i \neq 0 \rightarrow z_i := 0 \) for \( i \in \mathbb{N}_3 \).

We show that the program \( p' \) (with the aforementioned actions) is nonmasking \( f \)-tolerant from \( I \) under strong fairness. Once \( p' \) is perturbed to \( T - I \), recovery to \( I \) is achieved as follows. The processes \( P_0, P_1 \) and \( P_2 \) ensure that \( (Iden \land \text{Clauses}) \) is satisfied, and then \( P_3 \) sets \( sat = 1 \). Moreover, \( P_1 \) sets \( y_i = 0 \), thereby recovering to \( I \). Using Fig. 6, we show that no computation prefix of \( p' \mid f \) from invariant \( I \) reaches the state \( failed = 1 \) even if faults \( f_{ss} \) occur.

The two values in Fig. 6 respectively denote the values of \( y_i \) and \( z_i \), where \( i \in \mathbb{N}_3 \), \( sat = 0 \) and \( x_i = 1 \) is fixed. These variables are only affected by processes \( P_1 \) and \( P_2 \) and the fault-class \( f_{ss} \). Dashed arrows represent the two possible actions of \( P_f \) if \( P_f \) included both actions that change \( y_i \) (i.e., \( y_i = 0 \land sat = 0 \rightarrow y_i := 1 \) and \( y_i = 1 \land sat = 0 \rightarrow y_i := 0 \), of which exactly one is chosen in our construction of \( p' \) (for each unique \( x_i \) value). Since only one action is chosen, there exists no computation prefix of \( p' \mid f \) from invariant \( I \) to an unrecoverable state where \( failed = 1 \). Notice that without the fault-class \( f_{ss} \), the program that includes both actions that change \( y_i \) would have been nonmasking \( f \)-tolerant under strong fairness because the cycles formed in \( p' \mid (T - I) \) are not livelocks under strong fairness. Moreover, from every state in \( T - I \) there is an enabled action; i.e., deadlock freedom in \( T - I \). Thus, \( p' \) is nonmasking \( f \)-tolerant from \( I \) under strong fairness.

\( \square \)

\( \Leftarrow \) \textbf{Proof:} Given a program \( p' \) that is nonmasking \( f \)-tolerant under strong fairness and meets the constraints of Problem 3.2, we build a program \( p_f \) with invariant \( I_f \equiv I_{ss} \land failed = 0 \) that is nonmasking \( f_{ss} \) and \( f_{ss} \)-tolerant from \( I_f \) under no fairness (see Fig. 4). We note that, in the presence of faults \( f_{ss} \), \( P_i \) must have actions to eventually assign 0 to \( z_i \) (required by invariant) from any nonzero value of \( z_i \), which is reached in the fault-span. Thus, in Fig. 6, \( P_i \) transitions simply assign \( z_i := 0 \). Program \( p_f \) includes the actions of \( P_0 \) to \( P_3 \) which do not form self-loops and recover from states where \( sat = 0 \). Actions of \( p_f \) map to a satisfying truth-assignment for the instance of 3-SAT. Notice that, \( p_f \) does not tolerate \( f_{ss} \).

The fault-span \( T_f \) of \( p_f \) is a subset of the fault-span \( T \) of \( p' \) since program and fault transitions of \( p_f \) are a subset of those transitions of \( p' \). It follows that \( T_f \) does not contain unrecoverable states (\( failed = 1 \)) nor does it include states \( \neg I_{ss} \land sat = 1 \) from where \( f_{ss} \) could bring \( p_f \) to an unrecoverable state. Thus, \( T_f \equiv (I_{ss} \land sat = 0) \land failed = 0 \) due to the definition of \( I_f \), fault class \( f_{ss} \), and the excluded states which lead to \( failed = 1 \). Clearly a computation exists in \( p_f \) from every state in \( T_f \) to its invariant \( I_f \) and \( I_f \) (modulo \( z \) variables) and \( p_f \) eventually reaches \( I \) from all states in its fault-span \( T_f \). The only states where \( sat = 1 \) holds in \( T_f \) are also in \( I_f \). Moreover, we argue that actions of \( P_f \) from states where \( sat = 0 \) (and are not self-loops) bring the system to a state where \( I_{ss} \) holds. Otherwise, some action would exist to set \( sat = 1 \) while preserving \( \neg I_{ss} \). As a result, \( f_{ss} \) could be enabled and could take the program state to \( failed = 1 \), which would be a contradiction with \( p' \) being nonmasking \( f \)-tolerant from some non-empty subset of \( I \). Thus, \( P_3 \) actions only set \( sat = 1 \) when the resulting state is in \( I_{ss} \).

Let us now show that if \( p' \) is to be nonmasking \( f \)-tolerant under strong fairness, then none of the processes \( P_i \) \( (i \in \mathbb{N}_3) \) can form nontrivial cycles in states \( sat = 0 \). When \( sat = 0 \), \( y_i \), and \( z_i \) take values shown in Fig. 6. We
illustrate that, for a specific \( x \) value, each process \( P \) must have only one action that updates \( y \), to ensure no computation prefix of \( p'(f \) reaches failed = 1. By contradiction, assume \( P \) (\( i \in N_k \)) has a non-trivial cycle for some fixed \( x_j = a \) (\( a \in N_k \)). Since the cycle exists in \( p'(f) \) when \( sat = 0 \), therefore \( P \) must have actions \( x_i = a \land y_i = 0 \land sat = 0 \rightarrow y_i := 1 \) and \( x_i = a \land y_i = 1 \land sat = 0 \rightarrow y_i := 0 \). Now, we demonstrate the following computation prefix that reaches failed = 1.

1. Transitions of \( f_{ss} \) perturb \( p' \) to \( x_i = a \land y_i = 0 \land sat = 0 \), where \( z_i = 0 \) and failed = 0.
2. Then, transitions of \( f_{ss} \) can occur, setting \( z_i \) to 1.
3. \( P \) sets \( y_j \) to 1.
4. Transitions of \( f_{ss} \) occur again, setting \( z_i \) to 2.
5. \( P \) sets \( y_0 \) to 0.
6. From this state, fault \( f_{ss} \) can occur, setting failed to 1 from where no recovery is possible.

Thus, \( P \) (\( i \in N_k \)) cannot have cycles. Recall that when \( P_3 \) acts to change \( sat \) from 0 to 1, the resulting state must satisfy \( I_{ss} \). As a result, the program \( p_{ss} \) constructed from the actions of processes \( P_0 \ldots P_3 \) when \( sat = 0 \) is nonmasking \((f_{ss} \cup f_{ss})\)-tolerant from \( I_{ss} \). The actions of \( p_{ss} \) can be mapped to a satisfying truth-assignment for the instance of 3-SAT.

We now discuss the impact of our hardness results on failsafe and masking fault tolerance. Failsafe fault tolerance does not require recovery to invariant. Thus, the issue of fairness is irrelevant for failsafe fault tolerance. For masking fault tolerance, we observe that in the proof of NP-completeness of Problem 3.3 under no fairness in Section 4.2.2, one can consider the write restrictions of each process as part of a safety property where a process \( P \) is not allowed to write any \( x_j \), where \( 0 \leq j \leq 2 \). Thus, it follows that adding stabilization under no fairness would become an instance of the problem of adding masking fault tolerance. This way, we simply reuse the proof of NP-completeness of adding strong stabilization to prove the NP-completeness of adding masking fault tolerance under no fairness. (This result matches with Kulkarni and Arora’s results in [6].)

The hardness of adding masking fault tolerance under weak and strong fairness follow accordingly from the NP-completeness proofs of this section.

**Corollary 4.12.** Adding masking fault tolerance is NP-complete under weak or strong fairness.

## 5 Discussion

This section discusses algorithmic design of self-stabilization, complexity of algorithmic design and fairness assumptions. Existing methods for the algorithmic design of self-stabilization include constraint-based methods [26] and sound heuristics [13], [27]. Abujarad and Kulkarni [26] 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. 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 [13], [27], 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.

Most hardness results [6], [9], [28] presented for the addition of fault tolerance lack the additional constraint of recovery from any state, which we have in the addition of stabilization. The proof of NP-hardness of adding failsafe fault tolerance presented in [9] 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. The problem of adding masking fault tolerance relies on finding a subset of the state space from where recovery is possible; no need to provide recovery from every 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 iff 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 self-stabilization. The essence of the proof in [28] also relies on the same principle where Bonakdarpour and Kulkarni illustrate the NP-hardness of designing progress from one state predicate to another in low atomicity programs. Most existing algorithmic methods [6], [13], [26], [27], [28], [29] investigate the problem of adding fault tolerance under no fairness assumption. To the best of our knowledge, this paper is the first to investigate the impact of fairness on the addition of fault tolerance.

## 6 Conclusions and Future Work

This paper illustrates that adding nonmasking fault tolerance to low atomicity programs is an NP-hard problem under no fairness, weak, and strong fairness. In the low atomicity model, program processes have read/write restrictions with respect to the variables of other processes. The presented proof of hardness is from 3-SAT to the problem of adding stabilization to non-stabilizing programs, which is a special case of adding nonmasking fault tolerance. We first presented a proof for the NP-hardness of adding stabilization under no fairness. Then we showed that, even under weak fairness adding stabilization remains an NP-hard problem, which implies the NP-hardness of adding nonmasking tolerance under weak fairness. While it is known that adding stabilization under strong fairness (a.k.a. weak stabilization) can be done in polynomial time (in the size of state space), we showed that adding nonmasking tolerance under strong fairness remains NP-hard in general.

To extend this work, we will investigate special cases where the addition of stabilization in particular and nonmasking in general can be performed efficiently. That is, for what programs, classes of faults and invariants can the addition of nonmasking fault tolerance be done efficiently?

## Acknowledgments

The authors would like to thank the anonymous referees for their valuable comments and suggestions. This work was sponsored by the NSF grant CCF-1116546.


REFERENCES


